Saved



55 Points

Let’s dive right into working with two really important tools: Matplotlib and Seaborn! In this exercise, you will learn how to use Matplotlib, a powerful Python data visualization library. Matplotlib provides the building blocks to create rich visualizations of many different kinds of datasets. You will learn how to create visualizations for different kinds of data and how to customize, automate, and share these visualizations.

**Course Description**



Visualizing data in plots and figures exposes the underlying patterns in the data and provides insights. Good visualizations also help you communicate your data to others, and are useful to data analysts and other consumers of the data. In this course, you will learn how to use Matplotlib, a powerful Python data visualization library. Matplotlib provides the building blocks to create rich visualizations of many different kinds of datasets. You will learn how to create visualizations for different kinds of data and how to customize, automate, and share these visualizations.

**Daily XP0**

# Introduction to data visualization with Matplotlib

**50 XP**

## 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.

## 2. 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.

## 3. 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-dot-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. Here, you can see a Figure with empty Axes. No data has been added yet.

## 4. 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.

## 5. Adding data to axes

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-dot-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.

## 6. 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.

## 7. 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.

## 8. Practice making a figure!

Now it's your turn. In the exercises, you will practice making a figure and axes and adding data into them.

# Using the matplotlib.pyplot interface

There are many ways to use Matplotlib. In this course, we will focus on the pyplot interface, which provides the most flexibility in creating and customizing data visualizations.

Initially, we will use the pyplot interface to create two kinds of objects: Figure objects and Axes objects.

This course introduces a lot of new concepts, so if you ever need a quick refresher, download the [*Matplotlib Cheat Sheet*](https://res.cloudinary.com/dyd911kmh/image/upload/v1676360378/Marketing/Blog/Matplotlib_Cheat_Sheet.pdf) and keep it handy!

##### Instructions

**100 XP**

* Import the matplotlib.pyplot API, using the conventional name plt.
* Create Figure and Axes objects using the plt.subplots function.
* Show the results, an empty set of axes, using the plt.show function.
* # Import the matplotlib.pyplot submodule and name it plt
* import \_\_\_\_ as \_\_\_\_
* # Create a Figure and an Axes with plt.subplots
* fig, ax = \_\_\_\_
* # Call the show function to show the result
* \_\_\_\_

# Import the matplotlib.pyplot submodule and name it plt

import matplotlib.pyplot as plt

# Create a Figure and an Axes with plt.subplots

fig, ax = plt.subplots()

# Call the show function to show the result

plt.show()

# Import the matplotlib.pyplot submodule and name it plt

import matplotlib.pyplot as plt

# Create a Figure and an Axes with plt.subplots

fig, ax = plt.subplots()

# Call the show function to show the result

plt.show

<function matplotlib.pyplot.show()>

# Import the matplotlib.pyplot submodule and name it plt

import matplotlib.pyplot as plt

# Create a Figure and an Axes with plt.subplots

fig, ax = plt.subplots()

# Call the show function to show the result

plt.show()

Nicely done! This script provides the basis for everything we'll do in this course.

# Adding data to an Axes object

Adding data to a figure is done by calling methods of the Axes object. In this exercise, we will use the plot method to add data about rainfall in two American cities: Seattle, WA and Austin, TX.

The data are stored in two pandas DataFrame objects that are already loaded into memory: seattle\_weather stores information about the weather in Seattle, and austin\_weather stores information about the weather in Austin. Each of the DataFrames has a "MONTH" column that stores the three-letter name of the months. Each also has a column named "MLY-PRCP-NORMAL" that stores the average rainfall in each month during a ten-year period.

In this exercise, you will create a visualization that will allow you to compare the rainfall in these two cities.

##### Instructions

**100 XP**

* Import the matplotlib.pyplot submodule as plt.
* Create a Figure and an Axes object by calling plt.subplots.
* Add data from the seattle\_weather DataFrame by calling the Axes plot method.
* Add data from the austin\_weather DataFrame in a similar manner and call plt.show to show the results.
* # Import the matplotlib.pyplot submodule and name it plt
* \_\_\_\_
* # Create a Figure and an Axes with plt.subplots
* fig, ax = \_\_\_\_
* # Plot MLY-PRCP-NORMAL from seattle\_weather against the MONTH
* ax.\_\_\_\_(seattle\_weather["MONTH"], \_\_\_\_)
* # Plot MLY-PRCP-NORMAL from austin\_weather against MONTH
* ax.\_\_\_\_(\_\_\_\_, \_\_\_\_)
* # Call the show function
* \_\_\_\_

Great! Next you will learn how to label the axes on this plot.

import matplotlib.pyplot as plt

# Create a Figure and an Axes with plt.subplots

fig, ax = plt.subplots()

# Plot MLY-PRCP-NORMAL from seattle\_weather against the MONTH

ax.plot(seattle\_weather["MONTH"], seattle\_weather["MLY-PRCP-NORMAL"])

# Plot MLY-PRCP-NORMAL from austin\_weather against MONTH

ax.plot(austin\_weather["MONTH"], austin\_weather["MLY-PRCP-NORMAL"])

# Call the show function

# Import the matplotlib.pyplot submodule and name it plt import matplotlib.pyplot as plt # Create a Figure and an Axes with plt.subplots fig, ax = plt.subplots() # Plot MLY-PRCP-NORMAL from seattle\_weather against the MONTH ax.plot(seattle\_weather["MONTH"], seattle\_weather["MLY-PRCP-NORMAL"]) # Plot MLY-PRCP-NORMAL from austin\_weather against MONTH ax.plot(austin\_weather["MONTH"], austin\_weather["MLY-PRCP-NORMAL"]) # Call the show function plt.show()

# Customizing data appearance

We can customize the appearance of data in our plots, while adding the data to the plot, using key-word arguments to the plot command.

In this exercise, you will customize the appearance of the markers, the linestyle that is used, and the color of the lines and markers for your data.

As before, the data is already provided in pandas DataFrame objects loaded into memory: seattle\_weather and austin\_weather. These each have a "MONTHS" column and a "MLY-PRCP-NORMAL" that you will plot against each other.

In addition, a Figure object named fig and an Axes object named ax have already been created for you.

##### Instructions

**100 XP**

* 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').
* Pass the linestyle key-word argument to use dashed lines for the data from both cities ('--').
* # Plot Seattle data, setting data appearance
* ax.plot(seattle\_weather["MONTH"], seattle\_weather["MLY-PRCP-NORMAL"], \_\_\_\_)
* # Plot Austin data, setting data appearance
* ax.plot(austin\_weather["MONTH"], austin\_weather["MLY-PRCP-NORMAL"], \_\_\_\_)
* # Call show to display the resulting plot
* plt.show()

# Plot Seattle data, setting data appearance ax.plot(seattle\_weather["MONTH"], seattle\_weather["MLY-PRCP-NORMAL"], marker='o', linestyle='--', color='b') # Plot Austin data, setting data appearance ax.plot(austin\_weather["MONTH"], austin\_weather["MLY-PRCP-NORMAL"], marker='v', linestyle='--', color='r') # Call show to display the resulting plot plt.show()

# Plot Seattle data, setting data appearance

ax.plot(seattle\_weather["MONTH"], seattle\_weather["MLY-PRCP-NORMAL"], marker='o', linestyle='--', color='b')

# Plot Austin data, setting data appearance

ax.plot(austin\_weather["MONTH"], austin\_weather["MLY-PRCP-NORMAL"], marker='v', linestyle='--', color='r')

# Call show to display the resulting plot

plt.show()

Beautiful! Now you can create plots that look exactly as you'd like them to appear.

# Customizing axis labels and adding titles

Customizing the axis labels requires using the set\_xlabel and set\_ylabel methods of the Axes object. Adding a title uses the set\_title method.

In this exercise, you will customize the content of the axis labels and add a title to a plot.

As before, the data is already provided in pandas DataFrame objects loaded into memory: seattle\_weather and austin\_weather. These each have a "MONTH" column and a "MLY-PRCP-NORMAL" column. These data are plotted against each other in the first two lines of the sample code provided.

In addition, a Figure object named fig and an Axes object named ax have already been created for you.

##### Instructions

**100 XP**

* Use the set\_xlabel method to add the label: "Time (months)".
* Use the set\_ylabel method to add the label: "Precipitation (inches)".
* Use the set\_title method to add the title: "Weather patterns in Austin and Seattle".
* ax.plot(seattle\_weather["MONTH"], seattle\_weather["MLY-PRCP-NORMAL"])
* ax.plot(austin\_weather["MONTH"], austin\_weather["MLY-PRCP-NORMAL"])
* # Customize the x-axis label
* \_\_\_\_
* # Customize the y-axis label
* \_\_\_\_
* # Add the title
* \_\_\_\_
* # Display the figure
* plt.show()

ax.plot(seattle\_weather["MONTH"], seattle\_weather["MLY-PRCP-NORMAL"]) ax.plot(austin\_weather["MONTH"], austin\_weather["MLY-PRCP-NORMAL"]) # Customize the x-axis label ax.set\_xlabel("Time (months)") # Customize the y-axis label ax.set\_ylabel("Precipitation (inches)") # Add the title ax.set\_title("Weather patterns in Austin and Seattle") # Display the figure plt.show()

ax.plot(seattle\_weather["MONTH"], seattle\_weather["MLY-PRCP-NORMAL"])

ax.plot(austin\_weather["MONTH"], austin\_weather["MLY-PRCP-NORMAL"])

# Customize the x-axis label

ax.set\_xlabel("Time (months)")

# Customize the y-axis label

ax.set\_ylabel("Precipitation (inches)")

# Add the title

ax.set\_title("Weather patterns in Austin and Seattle")

# Display the figure

plt.show()

Great! We'll learn more about annotating figures later in the course.

**Daily XP500**

# Small multiples

**50 XP**

## 1. Small multiples

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

## 2. 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.

## 3. Adding more data

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?

## 4. And more data

This code adds the data from Austin to the plot. When we display the plot,

## 5. Too much data!

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.

## 6. 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.

## 7. 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.

## 8. 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,

## 9. Subplots with data

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.

## 10. 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.

## 11. Practice making subplots!

Next, go ahead and practice creating visualizations with small multiples.

# Creating a grid of subplots

How would you create a Figure with 6 Axes objects organized in 3 rows and 2 columns?

##### Possible Answers

* fig, ax = plt.subplots((3, 2))
* fig, ax = plt.axes((2, 3))
* **fig, ax = plt.subplots(3, 2)**
* fig, ax = plt.subplots((2, 3))
* That's right! This gives you the output you need.

# Creating small multiples with plt.subplots

Small multiples are used to plot several datasets side-by-side. In Matplotlib, small multiples can be created using the plt.subplots() function. The first argument is the number of rows in the array of Axes objects generate and the second argument is the number of columns. In this exercise, you will use the Austin and Seattle data to practice creating and populating an array of subplots.

The data is given to you in DataFrames: seattle\_weather and austin\_weather. These each have a "MONTH" column and "MLY-PRCP-NORMAL" (for average precipitation), as well as "MLY-TAVG-NORMAL" (for average temperature) columns. In this exercise, you will plot in a separate subplot the monthly average precipitation and average temperatures in each city.

* Create a Figure and an array of subplots with 2 rows and 2 columns.
* Addressing the top left Axes as index 0, 0, plot the Seattle precipitation.
* In the top right (index 0,1), plot Seattle temperatures.
* In the bottom left (1, 0) and bottom right (1, 1) plot Austin precipitations and temperatures.
* # Create a Figure and an array of subplots with 2 rows and 2 columns
* fig, ax = plt.subplots(\_\_\_\_, \_\_\_\_)
* # Addressing the top left Axes as index 0, 0, plot month and Seattle precipitation
* ax[0, 0].plot(\_\_\_\_, \_\_\_\_)
* # In the top right (index 0,1), plot month and Seattle temperatures
* ax[0, 1].plot(\_\_\_\_, \_\_\_\_)
* # In the bottom left (1, 0) plot month and Austin precipitations
* ax[\_\_\_\_].plot(\_\_\_\_, \_\_\_\_)
* # In the bottom right (1, 1) plot month and Austin temperatures
* ax[\_\_\_\_].plot(\_\_\_\_, \_\_\_\_)
* plt.show()

That's great! Next, you will put together all the things you've learned so far.

# Create a Figure and an array of subplots with 2 rows and 2 columns fig, ax = plt.subplots(2, 2) # Addressing the top left Axes as index 0, 0, plot month and Seattle precipitation ax[0, 0].plot(seattle\_weather["MONTH"], seattle\_weather["MLY-PRCP-NORMAL"]) # In the top right (index 0,1), plot month and Seattle temperatures ax[0, 1].plot(seattle\_weather["MONTH"], seattle\_weather["MLY-TAVG-NORMAL"]) # In the bottom left (1, 0) plot month and Austin precipitations 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"]) plt.show()

# Create a Figure and an array of subplots with 2 rows and 2 columns

fig, ax = plt.subplots(2, 2)

# Addressing the top left Axes as index 0, 0, plot month and Seattle precipitation

ax[0, 0].plot(seattle\_weather["MONTH"], seattle\_weather["MLY-PRCP-NORMAL"])

# In the top right (index 0,1), plot month and Seattle temperatures

ax[0, 1].plot(seattle\_weather["MONTH"], seattle\_weather["MLY-TAVG-NORMAL"])

# In the bottom left (1, 0) plot month and Austin precipitations

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"])

plt.show()

# Small multiples with shared y axis

When creating small multiples, it is often preferable to make sure that the different plots are displayed with the same scale used on the y-axis. This can be configured by setting the sharey key-word to True.

In this exercise, you will create a Figure with two Axes objects that share their y-axis. As before, the data is provided in seattle\_weather and austin\_weather DataFrames.

##### Instructions

**100 XP**

* Create a Figure with an array of two Axes objects that share their y-axis range.
* 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.
* 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.
* # Create a figure and an array of axes: 2 rows, 1 column with shared y axis
* fig, ax = plt.subplots(2, 1, sharey=True)
* # Plot Seattle precipitation data in the top axes
* \_\_\_\_.plot(\_\_\_\_, \_\_\_\_, color = \_\_\_\_)
* \_\_\_\_.plot(\_\_\_\_, \_\_\_\_, color = \_\_\_\_, linestyle = \_\_\_\_)
* \_\_\_\_.plot(\_\_\_\_, \_\_\_\_, color = \_\_\_\_, linestyle = \_\_\_\_)
* # Plot Austin precipitation data in the bottom axes
* \_\_\_\_.plot(\_\_\_\_, \_\_\_\_, color = \_\_\_\_)
* \_\_\_\_.plot(\_\_\_\_, \_\_\_\_, color = \_\_\_\_, linestyle = \_\_\_\_)
* \_\_\_\_.plot(\_\_\_\_, \_\_\_\_, color = \_\_\_\_, linestyle = \_\_\_\_)
* plt.show()

Good job! Next, you will learn how to plot more complex time-series data.

# Create a figure and an array of axes: 2 rows, 1 column with shared y axis

fig, ax = plt.subplots(2, 1, sharey=True)

# Plot Seattle precipitation data in the top axes

ax[0].plot(seattle\_weather['MLY-PRCP-NORMAL'], color = 'b')

ax[0].plot(seattle\_weather['MLY-PRCP-NORMAL'], seattle\_weather['MLY-PRCP-25PCTL'], color = 'b', linestyle = '--')

ax[0].plot(seattle\_weather['MLY-PRCP-NORMAL'],seattle\_weather['MLY-PRCP-75PCTL'] , color = 'b', linestyle = '--')

# Plot Austin precipitation data in the bottom axes

ax[1].plot(austin\_weather['MLY-PRCP-NORMAL'], color = 'r')

ax[1].plot(austin\_weather['MLY-PRCP-NORMAL'], austin\_weather['MLY-PRCP-25PCTL'], color = 'r', linestyle = '--')

ax[1].plot(austin\_weather['MLY-PRCP-NORMAL'], austin\_weather['MLY-PRCP-75PCTL'], color = 'r', linestyle = '--')

plt.show()

# Create a figure and an array of axes: 2 rows, 1 column with shared y axis fig, ax = plt.subplots(2, 1, sharey=True) # Plot Seattle precipitation data in the top axes ax[0].plot(seattle\_weather['MLY-PRCP-NORMAL'], color = 'b') ax[0].plot(seattle\_weather['MLY-PRCP-NORMAL'], seattle\_weather['MLY-PRCP-25PCTL'], color = 'b', linestyle = '--') ax[0].plot(seattle\_weather['MLY-PRCP-NORMAL'],seattle\_weather['MLY-PRCP-75PCTL'] , color = 'b', linestyle = '--') # Plot Austin precipitation data in the bottom axes ax[1].plot(austin\_weather['MLY-PRCP-NORMAL'], color = 'r') ax[1].plot(austin\_weather['MLY-PRCP-NORMAL'], austin\_weather['MLY-PRCP-25PCTL'], color = 'r', linestyle = '--') ax[1].plot(austin\_weather['MLY-PRCP-NORMAL'], austin\_weather['MLY-PRCP-75PCTL'], color = 'r', linestyle = '--') plt.show()

**Daily XP800**

**Plotting time-series data**

**50 XP**

**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.

**2. 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.

**3. 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-underscore-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-underscore-col key-word argument.

**4. 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.

**5. 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.

**6. 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.

**7. 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.

**8. 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.

**9. Let's practice time-series plotting!**

Before moving on, let's practice indexing and plotting time-series data.

# Read data with a time index

pandas DataFrame objects can have an index that denotes time. This is useful because Matplotlib recognizes that these measurements represent time and labels the values on the axis accordingly.

In this exercise, you will read data from a CSV file called climate\_change.csv that contains measurements of CO2 levels and temperatures made on the 6th of every month from 1958 until 2016. You will use pandas' read\_csv function.

To designate the index as a DateTimeIndex, you will use the parse\_dates and index\_col key-word arguments both to parse this column as a variable that contains dates and also to designate it as the index for this DataFrame.

By the way, if you haven't downloaded it already, check out the [*Matplotlib Cheat Sheet*](https://datacamp-community-prod.s3.amazonaws.com/e1a8f39d-71ad-4d13-9a6b-618fe1b8c9e9). It includes an overview of the most important concepts, functions and methods and might come in handy if you ever need a quick refresher!

##### Instructions

**100 XP**

* Import the pandas library as pd.
* 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.
* # Import pandas as pd
* \_\_\_\_
* # Read the data from file using read\_csv
* climate\_change = pd.read\_csv(\_\_\_\_, \_\_\_\_, \_\_\_\_)

Nicely done! Next, we'll plot these data.

# Import pandas as pd

import pandas as pd

# Read the data from file using read\_csv

climate\_change = pd.read\_csv('climate\_change.csv', parse\_dates=["date"], index\_col=["date"])

# Import pandas as pd import pandas as pd # Read the data from file using read\_csv climate\_change = pd.read\_csv('climate\_change.csv', parse\_dates=["date"], index\_col=["date"])

# Plot time-series data

To plot time-series data, we use the Axes object plot command. The first argument to this method are the values for the x-axis and the second argument are the values for the y-axis.

This exercise provides data stored in a DataFrame called climate\_change. This variable has a time-index with the dates of measurements and two data columns: "co2" and "relative\_temp".

In this case, the index of the DataFrame would be used as the x-axis values and we will plot the values stored in the "relative\_temp" column as the y-axis values. We will also properly label the x-axis and y-axis.

##### Instructions

**100 XP**

* 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.
* Set the x-axis label to 'Time'.
* Set the y-axis label to 'Relative temperature (Celsius)'.
* Show the figure.
* Great! Next, you'll zoom in to select a part of the time-series.
* import matplotlib.pyplot as plt
* fig, ax = plt.subplots()
* # Add the time-series for "relative\_temp" to the plot
* ax.plot(climate\_change.index,climate\_change['relative\_temp'])
* # Set the x-axis label
* ax.set\_xlabel('Time')
* # Set the y-axis label
* ax.set\_ylabel('Relative temperature (Celsius)')
* # Show the figure
* plt.show()

import matplotlib.pyplot as plt fig, ax = plt.subplots() # Add the time-series for "relative\_temp" to the plot ax.plot(climate\_change.index,climate\_change['relative\_temp']) # Set the x-axis label ax.set\_xlabel('Time') # Set the y-axis label ax.set\_ylabel('Relative temperature (Celsius)') # Show the figure plt.show()

# Using a time index to zoom in

When a time-series is represented with a time index, we can use this index for the x-axis when plotting. We can also select a range of dates to zoom in on a particular period within the time-series using pandas' indexing facilities. In this exercise, you will select a portion of a time-series dataset and you will plot that period.

The data to use is stored in a DataFrame called climate\_change, which has a time-index with dates of measurements and two data columns: "co2" and "relative\_temp".

##### Instructions

**100 XP**

* Use plt.subplots to create a Figure with one Axes called fig and ax, respectively.
* Create a variable called seventies that includes all the data between "1970-01-01" and "1979-12-31".
* Add the data from seventies to the plot: use the DataFrame index for the x value and the "co2" column for the y values.
* import matplotlib.pyplot as plt
* # Use plt.subplots to create fig and ax
* \_\_\_\_
* # Create variable seventies with data from "1970-01-01" to "1979-12-31"
* seventies = climate\_change[\_\_\_\_]
* # Add the time-series for "co2" data from seventies to the plot
* \_\_\_\_(\_\_\_\_, \_\_\_\_["co2"])
* # Show the figure
* \_\_\_\_

Nice. That's what the data for the seventies looks like!

import matplotlib.pyplot as plt

# Use plt.subplots to create fig and ax

fig, ax = plt.subplots()

# Create variable seventies with data from "1970-01-01" to "1979-12-31"

seventies = climate\_change["1970-01-01":"1979-12-31"]

# Add the time-series for "co2" data from seventies to the plot

ax.plot(seventies.index, seventies["co2"])

# Show the figure

plt.show()

import matplotlib.pyplot as plt # Use plt.subplots to create fig and ax fig, ax = plt.subplots() # Create variable seventies with data from "1970-01-01" to "1979-12-31" seventies = climate\_change["1970-01-01":"1979-12-31"] # Add the time-series for "co2" data from seventies to the plot ax.plot(seventies.index, seventies["co2"]) # Show the figure plt.show()

# Plot

# Plotting time-series with different variables

**50 XP**

## 1. 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.

## 2. Plotting two time-series together

For example, consider the climate-underscore-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.

## 3. Plotting two time-series together

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.

## 4. 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 you can see 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.

## 5. 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-underscore-ylabel function. We repeat this in our calls to plot and set-underscore-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.

## 6. 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-underscore-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-underscore-params method from the twin Axes object, setting the colors for these ticks to red.

## 7. Coloring the ticks

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.

## 8. A function that plots time-series

We use the def key-word to indicate that we are defining a function called plot-underscore-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-underscore-xlabel, set-underscore-ylabel, and tick-underscore-params.

## 9. Using our function

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

## 10. Create your own function!

In the exercises, you will gradually implement your own function from scratch.

# Plotting two variables

If you want to plot two time-series variables that were recorded at the same times, you can add both of them to the same subplot.

If the variables have very different scales, you'll want to make sure that you plot them in different twin Axes objects. These objects can share one axis (for example, the time, or x-axis) while not sharing the other (the y-axis).

To create a twin Axes object that shares the x-axis, we use the twinx method.

In this exercise, you'll have access to a DataFrame that has the climate\_change data loaded into it. This DataFrame was loaded with the "date" column set as a DateTimeIndex, and it has a column called "co2" with carbon dioxide measurements and a column called "relative\_temp" with temperature measurements.

##### Instructions

**100 XP**

* Use plt.subplots to create a Figure and Axes objects called fig and ax, respectively.
* Plot the carbon dioxide variable in blue using the Axes plot method.
* Use the Axes twinx method to create a twin Axes that shares the x-axis.
* Plot the relative temperature variable in red on the twin Axes using its plot method.
* import matplotlib.pyplot as plt
* # Initalize a Figure and Axes
* \_\_\_\_
* # Plot the CO2 variable in blue
* ax.plot(\_\_\_\_, \_\_\_\_, color=\_\_\_\_)
* # Create a twin Axes that shares the x-axis
* ax2 = \_\_\_\_
* # Plot the relative temperature in red
* \_\_\_\_.plot(\_\_\_\_, \_\_\_\_, color=\_\_\_\_)
* plt.show()

Great work. Next, let's implement this as a function.

import matplotlib.pyplot as plt

# 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()

import matplotlib.pyplot as plt # 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()

# 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.

Here, we will define a function that takes inputs such as a time variable and some other variable and plots them as x and y inputs. Then, it sets the labels on the x- and y-axis and sets the colors of the y-axis label, the y-axis ticks and the tick labels.

##### Instructions

**100 XP**

* Define a function called plot\_timeseries that takes as input an Axes object (axes), data (x,y), a string with the name of a color and strings for x- and y-axis labels.
* Plot y as a function of in the color provided as the input color.
* Set the x- and y-axis labels using the provided input xlabel and ylabel, setting the y-axis label color using color.
* Set the y-axis tick parameters using the tick\_params method of the Axes object, setting the colors key-word to color.
* # Define a function called plot\_timeseries
* def \_\_\_\_(axes, x, y, color, xlabel, ylabel):
* # Plot the inputs x,y in the provided color
* axes.\_\_\_\_(\_\_\_\_, \_\_\_\_, color=\_\_\_\_)
* # Set the x-axis label
* \_\_\_\_.\_\_\_\_(\_\_\_\_)
* # Set the y-axis label
* \_\_\_\_.\_\_\_\_(\_\_\_\_, color=\_\_\_\_)
* # Set the colors tick params for y-axis
* \_\_\_\_.\_\_\_\_('y', colors=\_\_\_\_)
* # 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)

# Define a function called plot\_timeseries def plot\_timeseries(axes, x, y, color, xlabel, ylabel): # Plot the inputs x,y in the provided color axes.plot(x, y, color=color) # Set the x-axis label axes.set\_xlabel(xlabel) # Set the y-axis label axes.set\_ylabel(ylabel, color=color) # Set the colors tick params for y-axis axes.tick\_params('y', colors=color)

# Using a plotting function

Defining functions allows us to reuse the same code without having to repeat all of it. Programmers sometimes say ["Don't repeat yourself"](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself).

In the previous exercise, you defined a function called plot\_timeseries:

plot\_timeseries(axes, x, y, color, xlabel, ylabel)

that takes an Axes object (as the argument axes), time-series data (as x and y arguments) the name of a color (as a string, provided as the color argument) and x-axis and y-axis labels (as xlabel and ylabel arguments). In this exercise, the function plot\_timeseries is already defined and provided to you.

Use this function to plot the climate\_change time-series data, provided as a pandas DataFrame object that has a DateTimeIndex with the dates of the measurements and co2 and relative\_temp columns.

##### Instructions

* 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".
* Use the ax.twinx method to add an Axes object to the figure that shares the x-axis with ax.
* 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)".
* fig, ax = plt.subplots()
* # Plot the CO2 levels time-series in blue
* \_\_\_\_(\_\_\_\_, \_\_\_\_, \_\_\_\_, "blue", \_\_\_\_, \_\_\_\_)
* # Create a twin Axes object that shares the x-axis
* ax2 = \_\_\_\_
* # Plot the relative temperature data in red
* \_\_\_\_(\_\_\_\_, \_\_\_\_, \_\_\_\_, "red", \_\_\_\_, \_\_\_\_)
* plt.show()
* fig, ax = plt.subplots()
* # Plot the CO2 levels time-series in blue
* plot\_timeseries(ax, climate\_change.index, climate\_change['co2'], "blue", 'Time (years)', 'CO2 levels')
* # Create a twin Axes object that shares the x-axis
* ax2 = ax.twinx()
* # Plot the relative temperature data in red
* plot\_timeseries(ax2, climate\_change.index, climate\_change['relative\_temp'], "red", 'Time (years)', 'Relative temperature (Celsius)')
* plt.show()
* # Plot the relative temperature data in red
* plot\_timeseries(ax2, climate\_change.index, climate\_change['relative\_temp'], "red", 'Time (years)', 'Relative temperature (Celsius)')
* plt.show()
* fig, ax = plt.subplots()
* # Plot the CO2 levels time-series in blue
* plot\_timeseries(ax, climate\_change.index, climate\_change['co2'], "blue", 'Time (years)', 'CO2 levels')
* # Create a twin Axes object that shares the x-axis
* ax2 = ax.twinx()
* # Plot the relative temperature data in red
* plot\_timeseries(ax2, climate\_change.index, climate\_change['relative\_temp'], "red", 'Time (years)', 'Relative temperature (Celsius)')
* plt.show()

# Annotating time-series data

**50 XP**

## 1. 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.

## 2. 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.

## 3. 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?

## 4. 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-point-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.

## 5. 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. If we pass an empty dictionary into the key-word argument, the arrow will have the default properties, as shown here.

## 6. 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.

## 7. 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.

## 8. Practice annotating plots!

But for now, start by practicing what you have learned so far.

# Annotating a plot of time-series data

Annotating a plot allows us to highlight interesting information in the plot. For example, in describing the climate change dataset, we might want to point to the date at which the relative temperature first exceeded 1 degree Celsius.

For this, we will use the annotate method of the Axes object. In this exercise, you will have the DataFrame called climate\_change loaded into memory. Using the Axes methods, plot only the relative temperature column as a function of dates, and annotate the data.

##### Instructions

**100 XP**

* Use the ax.plot method to plot the DataFrame index against the relative\_temp column.
* Use the annotate method to add the text '>1 degree' in the location (pd.Timestamp('2015-10-06'), 1).
* fig, ax = plt.subplots()
* # Plot the relative temperature data
* \_\_\_\_
* # Annotate the date at which temperatures exceeded 1 degree
* ax.\_\_\_\_(\_\_\_\_, \_\_\_\_)
* plt.show()

Good job! Now let's put it all together to create a figure that tells the whole story.

fig, ax = plt.subplots()

# Plot the relative temperature data

ax.plot(climate\_change.index, climate\_change['relative\_temp'])

# Annotate the date at which temperatures exceeded 1 degree

ax.annotate('>1 degree', xy=(pd.Timestamp('2015-10-06'), 1))

plt.show()

fig, ax = plt.subplots() # Plot the relative temperature data ax.plot(climate\_change.index, climate\_change['relative\_temp']) # Annotate the date at which temperatures exceeded 1 degree ax.annotate('>1 degree', xy=(pd.Timestamp('2015-10-06'), 1)) plt.show()

**Daily XP800**

##### Exercise

##### Exercise

# Plotting time-series: putting it all together

In this exercise, you will plot two time-series with different scales on the same Axes, and annotate the data from one of these series.

The CO2/temperatures data is provided as a DataFrame called climate\_change. You should also use the function that we have defined before, called plot\_timeseries, which takes an Axes object (as the axes argument) plots a time-series (provided as x and y arguments), sets the labels for the x-axis and y-axis and sets the color for the data, and for the y tick/axis labels:

plot\_timeseries(axes, x, y, color, xlabel, ylabel)

Then, you will annotate with text an important time-point in the data: on 2015-10-06, when the temperature first rose to above 1 degree over the average.

##### Instructions

**100 XP**

* Use the plot\_timeseries function to plot CO2 levels against time. Set xlabel to "Time (years)" ylabel to "CO2 levels" and color to 'blue'.
* Create ax2, as a twin of the first Axes.
* In ax2, plot temperature against time, setting the color ylabel to "Relative temp (Celsius)" and color to 'red'.
* 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.
* fig, ax = plt.subplots()
* # Plot the CO2 levels time-series in blue
* plot\_timeseries(\_\_\_\_, \_\_\_\_, \_\_\_\_, 'blue', \_\_\_\_, \_\_\_\_)
* # Create an Axes object that shares the x-axis
* ax2 = \_\_\_\_
* # Plot the relative temperature data in red
* plot\_timeseries(\_\_\_\_, \_\_\_\_, \_\_\_\_, 'red', \_\_\_\_, \_\_\_\_)
* # Annotate point with relative temperature >1 degree
* ax2.\_\_\_\_(">1 degree", \_\_\_\_, \_\_\_\_, \_\_\_\_)
* plt.show()

Beautiful! In the next chapter, you will learn how make quantitative comparisons using plots

fig, ax = plt.subplots()

# Plot the CO2 levels time-series in blue

plot\_timeseries(ax, climate\_change.index, climate\_change['co2'], 'blue', 'Time (years)', 'CO2 levels')

# Create an Axes object that shares the x-axis

ax2 = ax.twinx()

# Plot the relative temperature data in red

plot\_timeseries(ax2, climate\_change.index, climate\_change['relative\_temp'], 'red',

'Time (years)', 'Relative temp (Celsius)')

# Annotate point with relative temperature >1 degree

ax2.annotate(">1 degree", xy=(pd.Timestamp('2015-10-06'),1), xytext=(pd.Timestamp('2008-10-06'), -0.2), arrowprops={'arrowstyle':'->', 'color':'gray'})

plt.show()

fig, ax = plt.subplots() # Plot the CO2 levels time-series in blue plot\_timeseries(ax, climate\_change.index, climate\_change['co2'], 'blue', 'Time (years)', 'CO2 levels') # Create an Axes object that shares the x-axis ax2 = ax.twinx() # Plot the relative temperature data in red plot\_timeseries(ax2, climate\_change.index, climate\_change['relative\_temp'], 'red', 'Time (years)', 'Relative temp (Celsius)') # Annotate point with relative temperature >1 degree ax2.annotate(">1 degree", xy=(pd.Timestamp('2015-10-06'),1), xytext=(pd.Timestamp('2008-10-06'), -0.2), arrowprops={'arrowstyle':'->', 'color':'gray'}) plt.show()

**Daily XP900**

# Quantitative comparisons: bar-charts

**50 XP**

## 1. Quantitative comparisons: bar-charts

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

## 2. 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.

## 3. 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.

## 4. Interlude: rotate the tick labels

To fix these labels, we can rotate them by 90 degrees. This is done by using the set-underscore-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.

## 5. 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-dot-show.

## 6. 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.

## 7. Stacked bar chart

This is what the full stacked bar chart looks like.

## 8. 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.

## 9. Adding a legend

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.

## 10. Stacked bar chart with legend

This is what the figure looks like with the legend.

## 11. Create a bar chart!

Now, you try!

# Bar chart

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.

For example, in this exercise, you will visualize the number of gold medals won by each country in the provided medals DataFrame. The DataFrame contains the countries as the index, and a column called "Gold" that contains the number of gold medals won by each country, according to their rows.

##### Instructions

**100 XP**

* Call the ax.bar method to plot the "Gold" column as a function of the country.
* Use the ax.set\_xticklabels to set the x-axis tick labels to be the country names.
* In the call to ax.set\_xticklabels rotate the x-axis tick labels by 90 degrees by using the rotation key-word argument.
* Set the y-axis label to "Number of medals".
* fig, ax = plt.subplots()
* # Plot a bar-chart of gold medals as a function of country
* \_\_\_\_
* # Set the x-axis tick labels to the country names
* \_\_\_\_.set\_xticklabels(\_\_\_\_, \_\_\_\_)
* # Set the y-axis label
* \_\_\_\_
* plt.show()

fig, ax = plt.subplots()

# Plot a bar-chart of gold medals as a function of country

ax.bar(medals.index, medals['Gold'])

# Set the x-axis tick labels to the country names

ax.set\_xticklabels(medals.index, rotation=90)

# Set the y-axis label

ax.set\_ylabel("Number of medals")

plt.show()

fig, ax = plt.subplots() # Plot a bar-chart of gold medals as a function of country ax.bar(medals.index, medals['Gold']) # Set the x-axis tick labels to the country names ax.set\_xticklabels(medals.index, rotation=90) # Set the y-axis label ax.set\_ylabel("Number of medals") plt.show()

# Stacked bar chart

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.

In this exercise, you will have access to a DataFrame called medals that contains an index that holds the names of different countries, and three columns: "Gold", "Silver" and "Bronze". You will also have a Figure, fig, and Axes, ax, that you can add data to.

You will create a stacked bar chart that shows the number of gold, silver, and bronze medals won by each country, and you will add labels and create a legend that indicates which bars represent which medals.

##### Instructions

**100 XP**

* Call the ax.bar method to add the "Gold" medals. Call it with the label set to "Gold".
* 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".
* Use ax.bar to add "Bronze" bars on top of that, using the bottom key-word and label it as "Bronze".
* # Add bars for "Gold" with the label "Gold"
* \_\_\_\_(\_\_\_\_, \_\_\_\_, label=\_\_\_\_)
* # Stack bars for "Silver" on top with label "Silver"
* \_\_\_\_(\_\_\_\_, \_\_\_\_, bottom=\_\_\_\_, \_\_\_\_)
* # Stack bars for "Bronze" on top of that with label "Bronze"
* \_\_\_\_
* # Display the legend
* ax.legend()
* plt.show()

# Add bars for "Gold" with the label "Gold"

\_\_\_\_(\_\_\_\_, \_\_\_\_, label=\_\_\_\_)

# Stack bars for "Silver" on top with label "Silver"

\_\_\_\_(\_\_\_\_, \_\_\_\_, bottom=\_\_\_\_, \_\_\_\_)

# Stack bars for "Bronze" on top of that with label "Bronze"

\_\_\_\_

# Display the legend

ax.legend()

plt.show()

# # Add bars for "Gold" with the label "Gold" ax.bar(medals.index, medals['Gold'], label='Gold') # Stack bars for "Silver" on top with label "Silver" ax.bar(medals.index, medals['Silver'], bottom=medals['Gold'], label='Silver') # Stack bars for "Bronze" on top of that with label "Bronze" ax.bar(medals.index, medals['Bronze'], bottom=medals['Gold']+medals['Silver'], label='Bronze') # Display the legend ax.legend() plt.show()antitative comparisons: histograms

**50 XP**

## 1. 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.

## 2. 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.

## 3. 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.

## 4. 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.

## 5. 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-dot-show, so that a legend appears in the figure.

## 6. 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.

## 7. 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.

## 8. 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,

## 9. Histogram with a histtype of step

exposing that yes: there are rowers with a height of less than 180 centimeters.

## 10. Create your own histogram!

In the exercises to follow, you will create your own histograms.

# 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.

You will have two DataFrames to use. The first is called mens\_rowing and includes information about the medalists in the men's rowing events. The other is called mens\_gymnastics and includes information about medalists in all of the Gymnastics events.

##### Instructions

**100 XP**

* Use the ax.hist method to add a histogram of the "Weight" column from the mens\_rowing DataFrame.
* Use ax.hist to add a histogram of "Weight" for the mens\_gymnastics DataFrame.
* Set the x-axis label to "Weight (kg)" and the y-axis label to "# of observations".
* fig, ax = plt.subplots()
* # Plot a histogram of "Weight" for mens\_rowing
* ax.hist(\_\_\_\_)
* # Compare to histogram of "Weight" for mens\_gymnastics
* \_\_\_\_
* # Set the x-axis label to "Weight (kg)"
* \_\_\_\_
* # Set the y-axis label to "# of observations"
* \_\_\_\_
* plt.show()

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 abservations') plt.show()

# 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()

Nice. Now let's customize these histograms a bit by changing their default settings.

# "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.

The data is stored in a pandas DataFrame object called summer\_2016\_medals that has a column "Height". In addition, you are provided a pandas GroupBy object that has been grouped by the sport.

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.

##### Instructions

**100 XP**

* Use the hist method to display a histogram of the "Weight" column from the mens\_rowing DataFrame, label this as "Rowing".
* Use hist to display a histogram of the "Weight" column from the mens\_gymnastics DataFrame, and label this as "Gymnastics".
* 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.
* Add a legend to the figure, before it is displayed.
* fig, ax = plt.subplots()
* # Plot a histogram of "Weight" for mens\_rowing
* \_\_\_\_
* # Compare to histogram of "Weight" for mens\_gymnastics
* \_\_\_\_
* ax.set\_xlabel("Weight (kg)")
* ax.set\_ylabel("# of observations")
* # Add the legend and show the Figure
* \_\_\_\_
* plt.show()
* fig, ax = plt.subplots()
* # Plot a histogram of "Weight" for mens\_rowing
* ax.hist(mens\_rowing['Weight'], label='Rowing',bins=5,histtype='step')
* # Compare to histogram of "Weight" for mens\_gymnastics
* ax.hist(mens\_gymnastics['Weight'], label='Gymnastics', bins=5, histtype='step')
* ax.set\_xlabel("Weight (kg)")
* ax.set\_ylabel("# of observations")
* # Add the legend and show the Figure
* ax.legend()
* plt.show()

fig, ax = plt.subplots() # Plot a histogram of "Weight" for mens\_rowing ax.hist(mens\_rowing['Weight'], label='Rowing',bins=5,histtype='step') # Compare to histogram of "Weight" for mens\_gymnastics ax.hist(mens\_gymnastics['Weight'], label='Gymnastics', bins=5, histtype='step') ax.set\_xlabel("Weight (kg)") ax.set\_ylabel("# of observations") # Add the legend and show the Figure ax.legend() plt.show()

Well done! Let's see how to use these distributions in statistical plots.

# Statistical plotting

**50 XP**

## 1. Statistical plotting

In the previous lesson, 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.

## 2. 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, 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-dot-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.

## 3. 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.

## 4. 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.

## 5. Error bars in plots

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

## 6. 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

## 7. Interpreting boxplots

like this. 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.

## 8. Try it yourself!

In the exercises, you will make your own statistical visualizations.

# Adding error-bars to a bar chart

Statistical plotting techniques add quantitative information for comparisons into the visualization. For example, in this exercise, we will add error bars that quantify not only the difference in the means of the height of medalists in the 2016 Olympic Games, but also the standard deviation of each of these groups, as a way to assess whether the difference is substantial relative to the variability within each group.

For the purpose of this exercise, you will have two DataFrames: mens\_rowing holds data about the medalists in the rowing events and mens\_gymnastics will hold information about the medalists in the gymnastics events.

##### Instructions

**100 XP**

* Add a bar with size equal to the mean of the "Height" column in the mens\_rowing DataFrame and an error-bar of its standard deviation.
* Add another bar for the mean of the "Height" column in mens\_gymnastics with an error-bar of its standard deviation.
* Add a label to the the y-axis: "Height (cm)".
* fig, ax = plt.subplots()
* # Add a bar for the rowing "Height" column mean/std
* ax.\_\_\_\_("Rowing", \_\_\_\_, yerr=\_\_\_\_)
* # Add a bar for the gymnastics "Height" column mean/std
* \_\_\_\_
* # Label the y-axis
* \_\_\_\_
* plt.show()

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()

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()

That's great! These error bars can help you see that the difference in heights is rather large in terms of the standard deviation within every group.

# Adding error-bars to a plot

Adding error-bars to a plot is done by using the errorbar method of the Axes object.

Here, you have two DataFrames loaded: seattle\_weather has data about the weather in Seattle and austin\_weather has data about the weather in Austin. Each DataFrame has a column "MONTH" that has the names of the months, a column "MLY-TAVG-NORMAL" that has the average temperature in each month and a column "MLY-TAVG-STDDEV" that has the standard deviation of the temperatures across years.

In the exercise, you will plot the mean temperature across months and add the standard deviation at each point as y errorbars.

##### Instructions

**100 XP**

* Use the ax.errorbar method to add the Seattle data: the "MONTH" column as x values, the "MLY-TAVG-NORMAL" as y values and "MLY-TAVG-STDDEV" as yerr values.
* Add the Austin data: the "MONTH" column as x values, the "MLY-TAVG-NORMAL" as y values and "MLY-TAVG-STDDEV" as yerr values.
* Set the y-axis label as "Temperature (Fahrenheit)".
* fig, ax = plt.subplots()
* # Add Seattle temperature data in each month with error bars
* ax.errorbar(\_\_\_\_, \_\_\_\_, \_\_\_\_)
* # Add Austin temperature data in each month with error bars
* \_\_\_\_
* # Set the y-axis label
* \_\_\_\_
* plt.show()

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['MLY-TAVG-STDDEV']) # Add Austin temperature data in each month with error bars ax.errorbar(austin\_weather['MONTH'], austin\_weather['MLY-TAVG-NORMAL'], yerr=austin\_weather['MLY-TAVG-STDDEV']) # Set the y-axis label ax.set\_ylabel('Temperature (Fahrenheit)') plt.show()

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['MLY-TAVG-STDDEV'])

# Add Austin temperature data in each month with error bars

ax.errorbar(austin\_weather['MONTH'], austin\_weather['MLY-TAVG-NORMAL'], yerr=austin\_weather['MLY-TAVG-STDDEV'])

# Set the y-axis label

ax.set\_ylabel('Temperature (Fahrenheit)')

plt.show()

# Creating boxplots

Boxplots provide additional information about the distribution of the data that they represent. They tell us what the median of the distribution is, what the inter-quartile range is and also what the expected range of approximately 99% of the data should be. Outliers beyond this range are particularly highlighted.

In this exercise, you will use the data about medalist heights that you previously visualized as histograms, and as bar charts with error bars, and you will visualize it as boxplots.

Again, you will have the mens\_rowing and mens\_gymnastics DataFrames available to you, and both of these DataFrames have columns called "Height" that you will compare.

##### Instructions

**100 XP**

* Create a boxplot that contains the "Height" column for mens\_rowing on the left and mens\_gymnastics on the right.
* Add x-axis tick labels: "Rowing" and "Gymnastics".
* Add a y-axis label: "Height (cm)".
* fig, ax = plt.subplots()
* # Add a boxplot for the "Height" column in the DataFrames
* \_\_\_\_
* # Add x-axis tick labels:
* \_\_\_\_
* # Add a y-axis label
* \_\_\_\_
* plt.show()

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()

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()

# Very good. This provides even more information. For example, we can see how many individuals are outliers within their group. Next, let's see how we compare two variables within a single sample. Quantitative comparisons: scatter plots

**50 XP**

## 1. 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.

## 2. 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-dot-show to display the figure.

## 3. 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-dot-show.

## 4. 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.

## 5. 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-dot-show.

## 6. 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.

## 7. Practice making your own scatter plots!

In the exercises, go ahead and practice making your own scatter plots.

# Simple scatter plot

Scatter are a bi-variate visualization technique. They plot each record in the data as a point. The location of each point is determined by the value of two variables: the first variable determines the distance along the x-axis and the second variable determines the height along the y-axis.

In this exercise, you will create a scatter plot of the climate\_change data. This DataFrame, which is already loaded, has a column "co2" that indicates the measurements of carbon dioxide every month and another column, "relative\_temp" that indicates the temperature measured at the same time.

##### Instructions

**100 XP**

* Using the ax.scatter method, add the data to the plot: "co2" on the x-axis and "relative\_temp" on the y-axis.
* Set the x-axis label to "CO2 (ppm)".
* Set the y-axis label to "Relative temperature (C)".
* fig, ax = plt.subplots()
* # Add data: "co2" on x-axis, "relative\_temp" on y-axis
* \_\_\_\_
* # Set the x-axis label to "CO2 (ppm)"
* \_\_\_\_
* # Set the y-axis label to "Relative temperature (C)"
* \_\_\_\_
* plt.show()
* 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()

Nicely done. Now, let's encode the time dimension with color.

fig, ax = plt.subplots() # Add data: "co2" on x-axis, "relative\_temp" on y-axis ax.scatter(climate\_change['co2'], climate\_change['relative\_temp']) # Set the x-axis label to "CO2 (ppm)" ax.set\_xlabel('CO2 (ppm)') # Set the y-axis label to "Relative temperature (C)" ax.set\_ylabel('Relative temperature (C)') plt.show()

# Encoding time by color

The screen only has two dimensions, but we can encode another dimension in the scatter plot using color. Here, we will visualize the climate\_change dataset, plotting a scatter plot of the "co2" column, on the x-axis, against the "relative\_temp" column, on the y-axis. We will encode time using the color dimension, with earlier times appearing as darker shades of blue and later times appearing as brighter shades of yellow.

##### Instructions

**100 XP**

* Using the ax.scatter method add a scatter plot of the "co2" column (x-axis) against the "relative\_temp" column.
* Use the c key-word argument to pass in the index of the DataFrame as input to color each point according to its date.
* Set the x-axis label to "CO2 (ppm)" and the y-axis label to "Relative temperature (C)".
* fig, ax = plt.subplots()
* # Add data: "co2", "relative\_temp" as x-y, index as color
* \_\_\_\_
* # Set the x-axis label to "CO2 (ppm)"
* \_\_\_\_
* # Set the y-axis label to "Relative temperature (C)"
* \_\_\_\_
* plt.show()
* fig, ax = plt.subplots()
* # Add data: "co2", "relative\_temp" as x-y, index as color
* ax.scatter(climate\_change['co2'], climate\_change['relative\_temp'], c=climate\_change.index)
* # Set the x-axis label to "CO2 (ppm)"
* ax.set\_xlabel('CO2 (ppm)')
* # Set the y-axis label to "Relative temperature (C)"
* ax.set\_ylabel('Relative temperature (C)')
* plt.show()

fig, ax = plt.subplots() # Add data: "co2", "relative\_temp" as x-y, index as color ax.scatter(climate\_change['co2'], climate\_change['relative\_temp'], c=climate\_change.index) # Set the x-axis label to "CO2 (ppm)" ax.set\_xlabel('CO2 (ppm)') # Set the y-axis label to "Relative temperature (C)" ax.set\_ylabel('Relative temperature (C)') plt.show()

This is beautiful! In the next chapter you will learn to share you figures with others and how to automate their creation.

# Preparing your figures to share with others

**50 XP**

## 1. 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.

## 2. 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.

## 3. 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.

## 4. Back to the default

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

## 5. 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.

## 6. The "bmh" style

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

## 7. 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.

## 8. 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.

## 9. Practice choosing the right style for you!

In the exercises, you'll practice selecting some of these styles for your own visualizations.

# Selecting a style for printing

You are creating a figure that will be included in a leaflet printed on a black-and-white printer. What style should you choose for your figures?

In the console, we have loaded the medals dataset. Before initializing Axes and Figure objects and plotting them, you can try setting a style to use.

##### Possible Answers

* 'seaborn-colorblind'
* **'grayscale'**
* 'tableau-colorblind10'
* 'bmh'

**Correct! This style would make sure that your figures appear the same on your screen and on the page.**

# Switching between styles

Selecting a style to use affects all of the visualizations that are created after this style is selected.

Here, you will practice plotting data in two different styles. The data you will use is the same weather data we used in the first lesson: you will have available to you the DataFrame seattle\_weather and the DataFrame austin\_weather, both with records of the average temperature in every month.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))

Select the 'ggplot' style, create a new Figure called fig, and a new Axes object called ax with plt.subplots.

 [2](javascript:void(0))

Select the 'Solarize\_Light2' style, create a new Figure called fig, and a new Axes object called ax with plt.subplots.

# Use the "ggplot" style and create new Figure/Axes

\_\_\_\_

\_\_\_\_

ax.plot(seattle\_weather["MONTH"], seattle\_weather["MLY-TAVG-NORMAL"])

plt.show()

# 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 "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 "Solarize\_Light2" style and create new Figure/Axes

\_\_\_\_

\_\_\_\_

ax.plot(austin\_weather["MONTH"], austin\_weather["MLY-TAVG-NORMAL"])

plt.show()

# 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()

Great! As you've seen, the style will only apply to a figure that is created after the style to use is selected.

# 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()

# Saving your visualizations

**50 XP**

## 1. 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.

## 2. A figure to share

Take for example this figure 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-dot-show method at the end of this code.

## 3. Saving the figure to file

Now, we replace the call to plt-dot-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-dot-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.

## 4. 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.

## 5. 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.

## 6. 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-underscore-size-underscore-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

## 7. Another aspect ratio

or long and narrow, like here.

## 8. Practice saving your visualizations!

In the exercises that follow, you'll practice saving your visualization as files.

# Saving a file several times

If you want to share your visualizations with others, you will need to save them into files. Matplotlib provides as way to do that, through the savefig method of the Figure object. In this exercise, you will save a figure several times. Each time setting the parameters to something slightly different. We have provided and already created Figure object.

##### Instructions 1/3

* [1](javascript:void(0))Examine the figure by calling the plt.show() function.

 [2](javascript:void(0))Save the figure into the file my\_figure.png, using the default resolution.

 [3](javascript:void(0))Save the figure into the file my\_figure\_300dpi.png and set the resolution to 300 dpi.

# Show the figure

\_\_\_\_

# Show the figure

plt.show()

# Show the figure plt.show()

# Save as a PNG file

\_\_\_\_

# Save as a PNG file

fig.savefig('my\_figure.png', dpi=300)

# Save as a PNG file with 300 dpi

\_\_\_\_

# Save as a PNG file with 300 dpi fig.savefig('my\_figure\_300dpi.png', dpi=300)

Nice! You are ready to share your visualizations in all kinds of formats! Don't forget that you can also use other formats. For example, JPG can be used to save images with lossy compression.

# Save a figure with different sizes

Before saving your visualization, you might want to also set the size that the figure will have on the page. To do so, you can use the Figure object's set\_size\_inches method. This method takes a sequence of two values. The first sets the width and the second sets the height of the figure.

Here, you will again have a Figure object called fig already provided (you can run plt.show if you want to see its contents). Use the Figure methods set\_size\_inches and savefig to change its size and save two different versions of this figure.

##### Instructions 1/2

[1](javascript:void(0))Set the figure size as width of 3 inches and height of 5 inches and save it as 'figure\_3\_5.png' with default resolution.

 [2](javascript:void(0))Set the figure size to width of 5 inches and height of 3 inches and save it as 'figure\_5\_3.png' with default settings.

# Set figure dimensions and save as a PNG

\_\_\_\_

\_\_\_\_

# Set figure dimensions and save as a PNG

fig.set\_size\_inches([3, 5])

fig.savefig('figure\_3\_5.png', dpi=100)

# Set figure dimensions and save as a PNG fig.set\_size\_inches([3, 5]) fig.savefig('figure\_3\_5.png', dpi=100)

# Set figure dimensions and save as a PNG

fig.set\_size\_inches([3, 5])

fig.savefig('figure\_3\_5.png', dpi=100)

# Set figure dimensions and save as a PNG

fig.set\_size\_inches([3, 5])

fig.savefig('figure\_3\_5.png', dpi=100)

# Set figure dimensions and save as a PNG

\_\_\_\_

\_\_\_\_

# Set figure dimensions and save as a PNG fig.set\_size\_inches([5,3]) fig.savefig('figure\_5\_3.png', dpi=100)

Great work! This gives you two versions of your figure to choose between.

**Daily XP1050**

# Automating figures from data

**50 XP**

## 1. 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.

## 2. 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.

## 3. 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.

## 4. 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 10 different branches of sport here.

## 5. 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-underscore-d-f, 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.

## 6. 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.

## 7. Practice automating visualizations!

In the exercises that follow, you will use this principle to create visualizations that adapt to the data provided.

# Unique values of a column

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.

In this exercise and the next, you will be visualizing the weight of athletes in the 2016 summer Olympic Games again, from a dataset that has some unknown number of branches of sports in it. This will be loaded into memory as a pandas DataFrame object called summer\_2016\_medals, which has a column called "Sport" that tells you to which branch of sport each row corresponds. There is also a "Weight" column that tells you the weight of each athlete.

In this exercise, we will extract the unique values of the "Sport" column

* Create a variable called sports\_column that holds the data from the "Sport" column of the DataFrame object.
* 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.
* Print the sports variable to the console.
* # Extract the "Sport" column
* sports\_column = \_\_\_\_
* # Find the unique values of the "Sport" column
* sports = \_\_\_\_
* # Print out the unique sports values
* \_\_\_\_

Well done! Now you are ready to use these unique values to visualize the data.

# Extract the "Sport" column

sports\_column = summer\_2016\_medals['Sport']

# Find the unique values of the "Sport" column

sports = sports\_column.unique()

# Print out the unique sports values

print(sports)

# Extract the "Sport" column

sports\_column = summer\_2016\_medals['Sport']

# Find the unique values of the "Sport" column

sports = sports\_column.unique()

# Print out the unique sports values

print(sports)

['Rowing' 'Taekwondo' 'Handball' 'Wrestling' 'Gymnastics' 'Swimming'

'Basketball' 'Boxing' 'Volleyball' 'Athletics']

<script.py> output:

['Rowing' 'Taekwondo' 'Handball' 'Wrestling' 'Gymnastics' 'Swimming'

'Basketball' 'Boxing' 'Volleyball' 'Athletics']

# 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.

This is what you will do in this exercise. You will be visualizing data about medal winners in the 2016 summer Olympic Games again, but this time you will have a dataset that has some unknown number of branches of sports in it. This will be loaded into memory as a pandas DataFrame object called summer\_2016\_medals, which has a column called "Sport" that tells you to which branch of sport each row corresponds. There is also a "Weight" column that tells you the weight of each athlete.

##### Instructions

**100 XP**

* Iterate over the values of sports setting sport as your loop variable.
* In each iteration, extract the rows where the "Sport" column is equal to sport.
* 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.
* Save the figure into the file "sports\_weights.png".
* fig, ax = plt.subplots()
* # Loop over the different sports branches
* for \_\_\_\_ in \_\_\_\_:
* # Extract the rows only for this sport
* sport\_df = \_\_\_\_
* # Add a bar for the "Weight" mean with std y error bar
* \_\_\_\_
* ax.set\_ylabel("Weight")
* ax.set\_xticklabels(sports, rotation=90)
* # Save the figure to file
* \_\_\_\_
* fig, ax = plt.subplots()
* # Loop over the different sports branches
* for sport in sports:
* # Extract the rows only for this sport
* sport\_df = summer\_2016\_medals[summer\_2016\_medals['Sport']== sport]
* # Add a bar for the "Weight" mean with std y error bar
* ax.bar(sport, sport\_df['Weight'].mean(),
* yerr=sport\_df['Weight'].std())
* ax.set\_ylabel("Weight")
* ax.set\_xticklabels(sports, rotation=90)
* # Save the figure to file
* fig.savefig('sports\_weights.png')

fig, ax = plt.subplots() # Loop over the different sports branches for sport in sports: # Extract the rows only for this sport sport\_df = summer\_2016\_medals[summer\_2016\_medals['Sport']== sport] # Add a bar for the "Weight" mean with std y error bar ax.bar(sport, sport\_df['Weight'].mean(), yerr=sport\_df['Weight'].std()) ax.set\_ylabel("Weight") ax.set\_xticklabels(sports, rotation=90) # Save the figure to file fig.savefig('sports\_weights.png')

# Where to go next

**50 XP**

## 1. Where to go next

Congratulations! You have completed this introduction to Matplotlib. And yet, we have only scratched the surface in what Matplotlib can do.

## 2. The Matplotlib gallery

One way to learn about other kinds of visualizations that you can create with Matplotlib is to visit the online gallery of examples on the Matplotlib website, at this URL.

## 3. Gallery of examples

The gallery contains several dozen examples of figures that you can create with Matplotlib. If you click on one of the figures, you will land in a page

## 4. Example page with code

that contains not only a larger version of the example, but also the full Python code that would generate this example from scratch. If you are interested in creating a visualization that is a variation on this example, you can start by copying over this example code and editing it to fit your particular use-case. This is much better than starting from scratch!

## 5. Plotting data in 3D

Here are a few of the things that you might want to do next. In this course we always visualized data using the two dimensions of the page, but you can also extend your capability to visualize data, by adding perspective to your visualizations to make them appear three-dimensional. For example, here is a parametric curve through a three-dimensional space. In this web page, you can learn more about creating three-dimensional visualizations.

## 6. Visualizing images with pseudo-color

Another capability of Matplotlib is visualizing data from images. For example, here is an image visualized using pseudo-color, where each value in the image is translated into a color. You can learn more about working with images in this URL.

## 7. Animations

You might remember this visualization that I showed you in the very first lesson of this course. It used one more dimension, time, by varying the display through animation. You can create animations by creating multiple frames of the movie, each as its own visualization, and then stitching them together into a movie using tools such as Quicktime, but Matplotlib also has its own interface for creating animations. You can learn about this interface at this URL.

## 8. Using Matplotlib for geospatial data

There are multiple software packages that extend Matplotlib's capability to a variety of different kinds of data. For example, Cartopy extends Matplotlib to be used with geospatial data, such as maps.

## 9. pandas + Matplotlib = Seaborn

Another library that extends Matplotlib is Seaborn. This library creates very sophisticated statistical visualizations from pandas data structures, such as DataFrames. The nice thing about Seaborn is that you can create elegant and sophisticated visualizations of your data with very little code. For example, this code would create this visualization that encodes the fuel efficiency of cars as a function of their horsepower, but also encodes the country in which the car was manufactured, using the color of the bubbles, as well as their weight, using the size of each bubble.

## 10. Seaborn example gallery

Seaborn also has an extensive example gallery that you can visit in this URL.

## 11. Good luck visualizing your data!

So as you can see, there is a lot more to learn about data visualization. But thanks to this course, you have already taken your first step along the path to visualizing your data in Python, using Matplotlib. Good luck visualizing your data!