Chapter 8:
Introduction to Python Data Science Modules

With the fast advancement in artificial intelligence, the landscape of the Predictive Analytics has changed dramatically. In this chapter, we'll look at some state-of-the-art Python modules used in the field of data science.

The content in this chapter is adapted from the *Data Science with Python* course on SOLOLEARN.COM: https://www.sololearn.com/learning/1093 and you can sign up for free.

Artificial intelligence, machine learning, and deep learning are used in many fields nowadays for predictive analytics purposes. Retailers such as Amazon.com are using these techniques to predict what customers will buy; online streaming companies such as Netlifx and Hulu use these algorithms to predict what movies subscribers will watch; doctors are using them to predict cancer and other diseases.

There are plenty of free online resources you can use to learn about this topic. One example is a course on Udacity on artificial intelligence https://classroom.udacity.com/courses/ud187.

Before you start this chapter, install the modules you need by running the following three lines of command in Anaconda prompt (Windows) or a terminal (Mac or Linux), with your virtual environment activated:

```
pip install numpy
pip install pandas
pip install matplotlib
```

Follow the on-screen instructions to finish installations.

What is Data Science?

There are many applications in business for data science including finding a better housing price prediction algorithm for Zillow, finding key attributes associated with wine quality, and building a recommendation system to increase the click-through-rate for Amazon.

Extracting insights from seemingly random data, data science normally involves collecting data, cleaning data, performing exploratory data analysis, building and evaluating machine learning models, and communicating insights to stakeholders.

Data science is a multidisciplinary field that unifies statistics, data analysis, machine learning and their related methods to extract knowledge and provide insights.

As a general-purpose programming language, Python is now the most popular programming language in data science. It's easy to use, has great community support, and integrates well with other frameworks (e.g., web applications) in an engineering environment. This chapter focuses on exploratory data analysis with three fundamental Python libraries: *numpy*, *pandas* and *matplotlib*.

We'll use datasets from a wide range of sources and formats: it could be collections of numerical measurements, text corpus, images, audio clips, or anything really. No matter the format, the first step in data science is to transform it into arrays of numbers.

For example, we collected 45 U.S. president heights in centimeters in chronological order and stored them in a list, a built-in data type in python (go back to Chapter 2 to refresh your memory about Python lists if you need to).

```
heights = [189, 170, 189, 163, 183, 171, 185,
           168, 173, 183, 173, 173, 175, 178,
           183, 193, 178, 173, 174, 183, 183,
           180, 168, 180, 170, 178, 182, 180,
           183, 178, 182, 188, 175, 179, 183,
           193, 182, 183, 177, 185, 188, 188,
           182, 185, 191]
```

In this example, George Washington was the first president, and his height was 189 cm.

If we wanted to know how many presidents are taller than 188cm, we could iterate through the list, compare each element against 188, and increase the count by 1 as the criteria is met. The script *heights.py* below accomplishes that:

```
cnt = 0
for height in heights:
    if height > 188:
        cnt +=1
print("the number of presidents taller than 188cm is", cnt)
```

After running the script in Spyder, you should see the following output:

```
the number of presidents taller than 188cm is 5
```

This shows that there are five presidents who are taller than 188 cm. You can download the script *heights.py* from Canvas and try it yourself.

No matter the format of the data, the first step in data science is to transform it into arrays of numbers. We'll learn how to use the Python *numpy* module to accomplish that.

The *numpy* Module

Numpy (short for Numerical Python) is a module that supports high-level mathematical operations on large, multi-dimensional datasets (which the module calls arrays and matrices). For starters, *numpy* allows us to find the answer to how many presidents are taller than 188cm with ease.

The script below, *heights_numpy.py*, shows us how to use the library and start with the basic object in *numpy*.

```
import numpy as np    #1

heights = [189, 170, 189, 163, 183, 171, 185,
           168, 173, 183, 173, 173, 175, 178,
           183, 193, 178, 173, 174, 183, 183,
           180, 168, 180, 170, 178, 182, 180,
           183, 178, 182, 188, 175, 179, 183,
           193, 182, 183, 177, 185, 188, 188,
           182, 185, 191]   #2
heights_arr = np.array(heights)    #3
print((heights_arr > 188).sum())    #4
print(heights_arr.size)
print(heights_arr.shape)
```

The import statement allows us to access the functions and modules inside the *numpy* library 1. The library will be used frequently, so by convention *numpy* is imported under a shorter name, *np*. We then define the list *heights*, which contains the heights of 45 presidents 2. The third line is to convert the list into a *numpy* array object, via the function `np.array()` 3. The last line provides a simple and natural solution, enabled by *numpy*, to the original question 4. We'll explain this line of code in great detail below.

After running the script *heights_numpy.py* in Spyder, you should see the following output:

```
5
45
(45,)
```

We'll explain in detail why you get those outputs in the subsection below.

As our datasets grow larger and more complicated, *numpy* allows us the use of a more efficient and for-loop-free method to manipulate and analyze our data. Our dataset example in this module will include the US Presidents' heights, ages and parties.

Size and Shape

An array class in Numpy is called an *ndarray* or n-dimensional array. We can use this to count the number of presidents in *heights_arr*, use attribute `numpy.ndarray.size`:

```
heights_arr.size
```

The above line of command produces an output of 45, meaning there are 45 elements in the array. Note that once an array is created in *numpy*, its size cannot be changed.

Size tells us how big the array is, shape tells us the dimension. To get current shape of an array, use attribute `shape`:

```
heights_arr.shape
```

The output is a tuple, (45,). Recall from Chapter 2 of this class that the built-in data type tuple is immutable whereas a list is mutable, containing a single value, indicating that there is only one dimension, i.e., axis 0. Along axis 0, there are 45 elements (one for each president) Here, `heights_arr` is a 1d array.

Attribute `size` in numpy is similar to the built-in method `len` in Python that is used to compute the length of iterable python objects like str, list, dict, etc.

Reshape

Other data we have collected include the ages of the presidents.

The script below, *heights_ages.py*, adds ages to the dataset and reshape it. The script is available in Canvas as well.

```
import numpy as np

heights = [189, 170, 189, 163, 183, 171, 185,
            168, 173, 183, 173, 173, 175, 178,
            183, 193, 178, 173, 174, 183, 183,
            180, 168, 180, 170, 178, 182, 180,
            183, 178, 182, 188, 175, 179, 183,
            193, 182, 183, 177, 185, 188, 188,
            182, 185, 191]

ages = [57, 61, 57, 57, 58, 57, 61, 54, 68, 51,
        49, 64, 50, 48, 65, 52, 56, 46, 54, 49,
        51, 47, 55, 55, 54, 42, 51, 56, 55, 51,
        54, 51, 60, 62, 43, 55, 56, 61, 52, 69,
        64, 46, 54, 47, 70]    #1

heights_and_ages = heights + ages    #2

heights_and_ages_arr = np.array(heights_and_ages)
print(heights_and_ages_arr.shape)   #3

heights_and_ages_arr=heights_and_ages_arr.reshape((2,45))    #4
print(heights_and_ages_arr.shape)
```

As mentioned before, *numpy* allows us to use a more efficient and for-loop-free method to manipulate and analyze our data. The age is a 45-element Python list 1. Since both heights and ages are all about the same presidents, we can combine them by using + to combine two lists 2. This produces one long array, with a dimension of (90,) 3. It would be clearer if we could align height and age for each president and reorganize the data into a 2 by 45 matrix where the first row contains all heights and the second row contains ages. To achieve this, a new array can be created by calling the `numpy.ndarray.reshape` method with new dimensions specified in a tuple: `heights_and_ages_arr.reshape((2,45))`.

The reshaped array is now a 2darray 4. We can reshape an array in multiple ways, as long as the size of the reshaped array matches that of the original. If you run the script *heights_ages.py*, you'll get an output of

```
(90,)
(2, 45)
```

Numpy can calculate the shape for us if we indicate the unknown dimension as -1. For example, given a 2darray `arr` of shape (3,4), `arr.reshape(-1)` would output a 1darray of shape (12,), while `arr.reshape((-1,2))` would generate a 2darray of shape (6,2).

Indexing

We can use array indexing to select individual elements from arrays. Like Python lists, numpy index starts from 0.

To access the height of the 3rd president Thomas Jefferson in the 1darray 'heights_arr':

```
heights_arr[2]
```

In a 2darray, there are two axes, axis 0 and 1. Axis 0 runs downward down the rows whereas axis 1 runs horizontally across the columns.

In the 2darrary heights_and_ages_arr, recall that its dimensions are (2, 45). To find Thomas Jefferson's age at the beginning of his presidency you would need to access the second row where ages are stored:

```
heights_and_ages_arr[1,2]
```

In 2darray, the row is axis 0 and the column is axis 1, therefore, to access a 2darray, numpy first looks for the position in rows, then in columns. So in our example, `heights_and_ages_arr[1,2]`, we are accessing row 2 (ages), column 3 (third president) to find Thomas Jefferson's age. Again, the numpy module uses 0 indexing, so `[1,2]` means the second row and third column.

The script below, *operations.py*, shows how indexing on array works, among other things. The script is available in Canvas as well.

```
--snip--
```

```
heights_arr = np.array(heights)    #1
heights_and_ages = heights + ages
heights_and_ages_arr = np.array(heights_and_ages)
heights_and_ages_arr=heights_and_ages_arr.reshape((2,45))

print("Thomas Jefferson's height is", heights_arr[2], "cm")    #2
print("Thomas Jefferson was", heights_and_ages_arr[1,2], "years old")  #3
print("the heights of the first 3 presidents are", heights_and_ages_arr[0,
0:3], "cm")

print("the heights of the first 3 presidents are",
heights_and_ages_arr[0,:3]*0.0328084, "feet")      #4
```

We first convert Python lists into *numpy* arrays and matrices 1. We then print out Thomas Jefferson's height as the third element in the array `heights_arr` 2. After that, we print out the age of Thomas Jefferson when he took office 3. We also print out the heights of the first three presidents. Finally, we convert the heights from centimeters to feet 4. We'll explain the details on slicing and mathematical operations in the next couple of sub-sections. If you run the script *operations.py*, you'll get an output of

```
Thomas Jefferson's height is 189 cm
Thomas Jefferson was 57 years old
the heights of the first 3 presidents are [189 170 189] cm
the heights of the first 3 presidents are [6.2007876 5.577428  6.2007876]
feet
```

We'll explain the outputs in the next two subsections.

Slicing

What if we want to inspect the first three elements from the first row in a 2darray? We use ":" to select all the elements from the index up to but not including the ending index. This is called *slicing*.

```
heights_and_ages_arr[0, 0:3]
```

When the starting index is 0, we can omit it as shown below:

```
heights_and_ages_arr[0, :3]
```

As a result, you have seen the heights of the first three presidents in the output from the script *operations.py*.

What if we'd like to see the entire third column? Specify this by using a ":" as follows

```
heights_and_ages_arr[:, 3]
```

Mathematical Operations on Arrays

Performing mathematical operations on arrays is straightforward. For instance, to convert the heights from centimeters to feet, knowing that 1 centimeter is equal to 0.0328084 feet, we can use multiplication:

```
heights_and_ages_arr[0,:]*0.0328084
```

The above line of command converts the heights of all 45 presidents from centimeters to feet.

Now we have all heights in feet. Note that this operation won't change the original array, it returns a new 1darray where 0.0328084 has been multiplied to each element in the first column of `heights_and_ages_arr`.

As a result, you have seen the heights of the first three presidents in feet instead of centimeters in the last line of output from the script *operations.py*.

Other mathematical operations for addition, subtraction, division and power (+, -, /, **) work the same way on arrays.

Numpy Array Method

In addition, there are several methods in numpy to perform more complex calculations on arrays. For example, the `sum()` method finds the sum of all the elements in an array:

```
heights_and_ages_arr.sum()
```

The sum of all heights and ages is 10575.

In order to sum all heights and sum all ages separately, we can specify axis=1 to calculate the sum inside each row:

```
heights_and_ages_arr.sum(axis=1)
```

WARNING: Here we are trying to sum up all values inside a row, but we put `axis=1` inside `sum()`. Normally, rows are associated with `axis=0` and columns are associated with `axis=1`. Here we are trying to keep the row (i.e., `axis=0`) and collapse the columns (i.e., `axis=1`), hence the command `sum(axis=1)`.

The output is the row sums: heights of all presidents (i.e., the first row) add up to 8100, and the sum of ages (i.e., the second row) is 2475.

On the other hand, if you want to obtain the sum inside each column, specify `axis=0`.

Other operations, such as `.min()`, `.max()`, `.mean()`, work in a similar way to `.sum()`.

The script *arr_methods.py* below shows how different array methods work. You can download the script in Canvas.

```
--snip--
heights_arr = np.array(heights)
heights_and_ages = heights + ages
heights_and_ages_arr = np.array(heights_and_ages)
heights_and_ages_arr=heights_and_ages_arr.reshape((2,45))

print(heights_and_ages_arr.sum())     #1
print("the sum of heights and ages are", heights_and_ages_arr.sum(axis=1))
# Obtain a list of minimum height and age
min_height_age=heights_and_ages_arr.min(axis=1)    #2
print(min_height_age)
# Extract the minimum height only
min_height=min_height_age[0]     #3
print(f"the minimum height is {min_height} cm")
# Obtain a list of maximu height and age
max_height_age=heights_and_ages_arr.max(axis=1)
print(max_height_age)
# Extract the maximum age only
max_age=max_height_age[1]
print(f"the maximum age is {max_age} years old")
# Find out who is the oldest president
oldest_president=heights_and_ages_arr[1,:].argmax()
print(f"the oldest president is number {oldest_president+1}")
```

```
# Print out the average height and age
print(f"the average height and age are {heights_and_ages_arr.mean(axis=1)}")#4
```

We first print out the sum of all heights and ages 1. We then print out the sum of heights and ages separately. At 2, we obtain the minimum height and age using the `min()` method and save the result in a list *min_height_age*. After that, we extract the first element from the list as the minimum height and print it out 3. Similarly, we obtain the maximum height and age using the `max()` method and extract the second element from the list as the maximum age and print it out.

We can also use the `argmax()` method to find out which president is the oldest.

Finally, we use the `mean()` method to obtain the average height and age and print them out 4. If you run the script *arr_methods.py*, you'll get the following output:

```
10575
the sum of heights and ages are [8100 2475]
[163  42]
the minimum height is 163 cm
[193  70]
the maximum age is 70 years old
the oldest president is number 45
the average height and age are [180.  55.]
```

Comparisons

In practicing data science, we often encounter comparisons to identify rows that match certain values. We can use operations including "<", ">", ">=", "<=", and "==" to do so. For example, in the heights_and_ages_arr dataset, we might be interested in only those presidents who started their presidency younger than 55 years old.

```
heights_and_ages_arr[:, 1] < 55
```

The output is a 1darray with boolean values that indicates which presidents meet the criteria. If we are only interested in which presidents started their presidency at 51 years of age, we can use "==" instead.

```
heights_and_ages_arr[:, 1] == 51
```

To find out how many rows satisfy the condition, use `.sum()` on the resultant 1d boolean array, e.g., `(heights_and_ages_arr[:, 1] == 51).sum()`, to see that there were exactly five presidents who started the presidency at age 51. True is treated as 1 and False as 0 in the sum.

The script *comparison.py* below shows how to find out the number of presidents started their presidency younger than 55, as well as the number at age 51. You can download the script in Canvas.

```
--snip--
heights_arr = np.array(heights)
heights_and_ages = heights + ages
heights_and_ages_arr = np.array(heights_and_ages)
heights_and_ages_arr=heights_and_ages_arr.reshape((2,45))

print(heights_and_ages_arr[1, :] < 55)
print("number of presidents started presidency younger than 55 is",
(heights_and_ages_arr[1, :] < 55).sum())
print("number of presidents started presidency at 51 is",
(heights_and_ages_arr[1, :] == 51).sum())
print("number of presidents taller than 180cm is",
(heights_and_ages_arr[0,:]>180).sum())
```

If you run the script *comparison.py*, you'll get the following output:

```
[False False False False False False False  True False  True  True False
  True  True False  True False  True  True  True  True  True False False
  True  True  True False False  True  True  True False False  True False
 False False  True False False  True  True  True False]
number of presidents started presidency younger than 55 is 22
number of presidents started presidency at 51 is 5
number of presidents taller than 180cm is 23
```

As you can see, `heights_and_ages_arr[:, 1]<55` produces a list of Boolean values, `True` or `False`. The sum of the list is 22, which means 22 presidents started their presidency younger than 55. We also find that five presidents started their presidency at the age of 51, and 23 presidents are taller than 180cm.

The *pandas* Module

What if we want to inspect the data on Abraham Lincoln in heights_and_ages_arr but cannot remember his integer position. Is there a convenient way to access the data by indexing the name of the president like

```
print(heights_and_ages_arr['Abraham Lincoln'])
```

Unfortunately, we will receive an error message. However, it is possible to do this in *pandas*. The *pandas* library is built on top of *numpy*, meaning a lot of features, methods, and functions are shared.

By convention, import the library under a short name "pd":

```
import pandas as pd
```

In later lessons, we will see that *pandas* allows us to access data by indexing the name directly. As *numpy* ndarrays are homogeneous, *pandas* relaxes this requirement and allows for various `dtypes` in its data structures.

Series

The Series is one building block in *pandas*. Pandas Series is a one-dimensional labeled array that can hold data of any type (integer, string, float, python objects, etc.), similar to a column in an excel spreadsheet. The axis labels are collectively called index.

If we are given a bag of letters a, b, and c, and count how many of each we have, we find that there are 1 a, 2 b's, and 3 c's. We could create a Series by supplying a list of counts and their corresponding labels:

```
pd.Series([1, 2, 3], index=['a', 'b', 'c'])
```

Alternatively, the values can be a *numpy* array:

```
pd.Series(np.array([1, 2, 3]), index=['a', 'b', 'c'])
```

Or, we could use a dictionary to specify the index with keys:

```
pd.Series({'a': 1, 'b': 2, 'c':3})
```

If we don't specify the index, by default, the index would be the integer positions starting from 0.

In a Series, we can access the value by its index directly. For example, if you run the following script *access_by_index.py*:

```
import pandas as pd
series = pd.Series({'a': 1, 'b': 2, 'c':3})
print(series['a'])
```

you'll get a result of 1.

Accessing the value by its index, rather than the integer position comes in handy when the dataset is of thousands, if not millions, of rows. Series is the building block for the DataFrame we will introduce next.

Think of Series as numpy 1darray with index or row names.

DataFrames

In data science, data is usually more than one-dimensional, and of different data types; thus Series is not sufficient. DataFrames are 2darrays with both row and column labels. One way to create a DataFrame from scratch is to pass in a dictionary. For example, this week, we sold 3 bottles of red wine to Adam, 6 to Bob, and 5 to Charles. We sold 5 bottles of white wine to Adam, 0 to Bob and 10 to Charles. We can organize the data into a DataFrame by creating a dict 'wine_dict' with the number of bottles of each wine type we sold, then pass it along with the customer names as index to create a DataFrame 'sales'.

Run the following script *create_dataframe.py*:

```
wine_dict = {
'red_wine': [3, 6, 5],
'white_wine':[5, 0, 10]
}
sales = pd.DataFrame(wine_dict, index=["adam", "bob", "charles"])

print(sales)
```

Think of DataFrame as a collection of the Series. Here, sales consists of two Series, one named under "red_wine", the other "white_wine", thus, we can access each series by calling its name:

```
sales['white_wine']
```

The DataFrame sales looks like this:

```
         red_wine  white_wine
adam            3           5
bob             6           0
charles         5          10
```

We will see other ways to index into DataFrames in later parts.

If we don't supply index, the DataFrame will generate an integer index starting from 0.

Inspect a DataFrame - Shape and Size

Let's take a look at a new DataFrame, in addition to heights and ages of the presidents, there is information on the order, names and parties and the data is in a csv file named *presidents.csv* that you can download on Canvas. The DataFrame *presidents_df* is read from a CSV file as follows.

Note that index is set to be the names of presidents. Run the script *shape_size.py* below, and the script is available on Canvas:

```
# Put file presidents.csv in the same folder as this script
import pandas as pd
presidents_df = pd.read_csv('presidents.csv', index_col='name')
print(presidents_df.shape)
print(presidents_df.shape[0])
print(presidents_df.size)
print(presidents_df.tail(n=3))
print(presidents_df.info())
print(presidents_df["height"].shape)
print("the median height and age of the 45 presidents are",
presidents_df[['height','age']].median())
print(presidents_df['age'].min())
```

There are 45 rows and 4 columns in this DataFrame. To get the number of rows we can access the first element in the tuple.

```
presidents_df.shape[0]
```

Size also works on DataFrame to return an integer representing the number of elements in this object.

```
presidents_df.size
```

Here both methods, `.shape` and `.size`, work in the same way as with `numpy ndarrays`.

Inspect a DataFrame - Head and Tail

Instead of looking at the entire dataset, we can just take a peep. To see the first few lines in a DataFrame, use `.head()`; if we don't specify n (the number of lines), by default, it displays the first five rows. Here we want to see the top 3 rows.

```
presidents_df.head(n=3)
```

In presidents_df, the index is the name of the president, there are four columns: order, age, height, and party. Similarly, if we want to see the last few rows, we can use `.tail()`, the default is also five rows.

Inspect a DataFrame - Info

Use `.info()` to get an overview of the DataFrame. Its output includes index, column names, count of non-null values, dtypes, and memory usage.

```
presidents_df.info()
```

The dtype for order, age, and height is integers, while party is an object. The count of non-null values in each column is the same as the number of rows, indicating no missing values. In addition to shape and size as seen in *numpy*, *pandas* features allow additional functionality to examine the data.

If you run the script *shape_size.py*, you'll get an output of

```
(45, 4)
```

```
45
180
                   order  age  height                 party
name
George Washington      1   57     189                  none
John Adams             2   61     170            federalist
Thomas Jefferson       3   57     189  democratic-republican
<class 'pandas.core.frame.DataFrame'>
Index: 45 entries, George Washington to Donald J. Trump
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   order   45 non-null     int64
 1   age     45 non-null     int64
 2   height  45 non-null     int64
 3   party   45 non-null     object
dtypes: int64(3), object(1)
memory usage: 1.8+ KB
None
```

Rows with .loc

Instead of memorizing the integer positions to locate the order, age, height, and party information of Abraham Lincoln, with DataFrame, we can access it by the name using `.loc[]`:

```
presidents_df.loc['Abraham Lincoln']
```

The result is a pandas Series of shape (4,).

```
type(presidents_df.loc['Abraham Lincoln'])
presidents_df.loc['Abraham Lincoln'].shape
```

We can also slice by index. Say we are interested in gathering information on all of the presidents between Abraham Lincoln and Ulysses S. Grant:

```
presidents_df.loc['Abraham Lincoln':'Ulysses S. Grant']
```

The result is a new DataFrame, a subset of 'presidents_df'.

`.loc[]` allows us to select data by label or by a conditional statement.

The script below, *pd_loc.py*, shows us how to use `.loc[]` in *pandas*.

```
# Put file presidents.csv in the same folder as this script
import pandas as pd
presidents_df = pd.read_csv('presidents.csv', index_col='name')

print(presidents_df.loc['Abraham Lincoln'])
print(type(presidents_df.loc['Abraham Lincoln']))
print(presidents_df.loc['Abraham Lincoln'].shape)

print(presidents_df.loc['Abraham Lincoln':'Ulysses S. Grant'])
```

After running the script *pd_loc.py* in Spyder, you should see the following output:

```
order                 16
age                   52
height               193
party         republican
Name: Abraham Lincoln, dtype: object
<class 'pandas.core.series.Series'>
(4,)
                  order  age  height          party
name
Abraham Lincoln      16   52     193     republican
Andrew Johnson       17   56     178  national union
Ulysses S. Grant     18   46     173     republican
```

The output shows that Lincoln was the 16th president, started his presidency at age 52, and he was 193cm tall. The data type for `presidents_df.loc['Abraham Lincoln']` is `pandas.core.series.Series`.

The output also shows all information for the three presidents between Lincoln and Grant.

Rows with .iloc

Alternatively, if we do know the integer position(s), we can use `.iloc[]` to access the row(s).

```
presidents_df.iloc[15]
```

To gather information from the 16th to 18th presidents, we can then:

```
presidents_df.iloc[15:18]
```

Try it yourself in Spyder on how `.iloc[]` works.

Columns

We can retrieve an entire column from presidents_df by name. First we access all the column names:

```
presidents_df.columns
```

The above command line returns an index object containing all column names as follows

```
Index(['order', 'age', 'height', 'party'], dtype='object')
```

From that you know presidents_df has four columns of data, with column names *order*, *age*, *height*, and *party*, respectively. Then we can access the column *height* by:

```
presidents_df['height']
presidents_df['height'].shape
```

Which returns a Series containing heights from all U.S. presidents.

To select multiple columns, we pass the names in a list, resulting in a DataFrame. Remember, we can use .head() to access the first 3 rows as shown below:

```
presidents_df[['height','age']].head(n=3)
```

In such a way, we focus on only the columns of interest.

When accessing a single column, one bracket results in a Series (single dimension) and double brackets results in a DataFrame (multi dimensional).

The script below, *pd_columns.py*, shows us how to access columns information in *pandas*.

```
# Put file presidents.csv in the same folder as this script
import pandas as pd
presidents_df = pd.read_csv('presidents.csv', index_col='name')

print(presidents_df.columns)
print(presidents_df['height'])
print(presidents_df['height'].shape)
print(presidents_df[['height','age']].head(n=3))
```

After running the script *pd_columns.py* in Spyder, you should see the following output:

```
Index(['order', 'age', 'height', 'party'], dtype='object')
name
George Washington          189
John Adams                 170
Thomas Jefferson           189
…
--snip--
…
George W. Bush             182
Barack Obama               185
Donald J. Trump            191
Name: height, dtype: int64
(45,)
                   height  age
name
George Washington     189   57
John Adams            170   61
Thomas Jefferson      189   57
```

Pandas Methods

It's not practical to print out an entire dataset with a large sample size. Instead, we want to summarize and characterize sample data using only a few values. For that, we can use several common *pandas* methods.

The simplest summary statistics, which are measures of location, include the minimum, the smallest number:

```
presidents_df.min()
```

maximum, the largest number:

```
presidents_df.max()
```

and mean, the average:

```
presidents_df.mean()
```

These methods work on Series as well. For example, `presidents_df['age'].mean()` results in 54.71, which is the average age of the presidents.

Other methods include `.median()`, `.var()`, and `.std()`.

The script below, *pd_methods.py*, shows how various methods work in *pandas*.

```
# Put file presidents.csv in the same folder as this script
import pandas as pd
presidents_df = pd.read_csv('presidents.csv', index_col='name')

print(presidents_df.min())
print(presidents_df.max())
print(presidents_df.mean())
print("when starting office, the average U.S president age is",
presidents_df['age'].mean())
```

The output from running the above script is:

```
order                1
age                 42
height             163
party       democratic
dtype: object
order        45
age          70
height      193
party       whig
dtype: object
order       23.022222
age         55.000000
height     180.000000
dtype: float64
when starting office, the average U.S president age is 55.0
```

Data Visualization with matplotlib

We'll learn how to visualize data using the *matplotlib* module.

Single Plot

You can use the *matplotlib* module to plot a graph to see the data.

One critical component of every figure is the figure title. The job of the title is to accurately communicate what the figure is about. In addition, axes need titles, or more commonly referred to as axis labels. The axis labels explain what the plotted data values are. We can specify the x and y axis labels and a title using `plt.xlabel()`, `plt.ylabel()` and `plt.title()`.

Run the following script *plot_sin.py*:

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

x = np.linspace(0,10,1000)
y = np.sin(x)

plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('y')
plt.title('function sin(x)')
plt.show()
```

Here we use `np.linspace()` to create a one-dimensional numpy array with 1000 values, evenly distributed between 0 and 10. The graph is shown in Figure 1. Note: you may need to go to the plots pane in the top right panel of your Spyder App to see the plot.

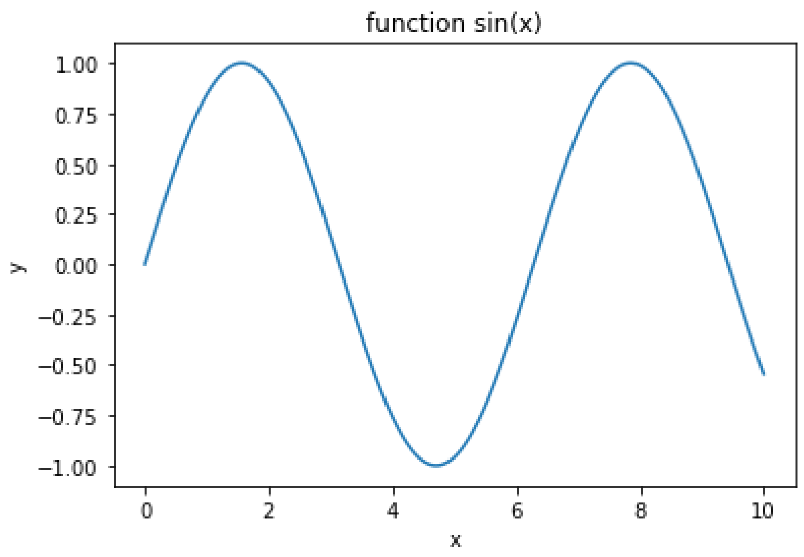

Figure 1: A plot with title and axis labels

Multiple Lines

Usually there are various datasets of similar nature, and we would like to compare them and observe the differences. We can plot multiple lines on the same figure. Say, the sin() function capture the tides on the east coast and cos() function capture the tides on the west coast at the same time, we can plot them both on the same figure by calling the .plot() function multiple times.

Colors and line styles can be specified to differentiate lines. Run the following script *two_lines.py*:

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

x = np.linspace(0,10,1000)
plt.plot(x, np.sin(x), color='k')
plt.plot(x, np.cos(x), color='r', linestyle ='--')
plt.show()
```

Note that we specified basic colors using a single letter, that is, k for black and r for red. More examples include b for blue, g for green, c for cyan, etc. For more details on the use of colors in matplotlib, refer to its documentation.

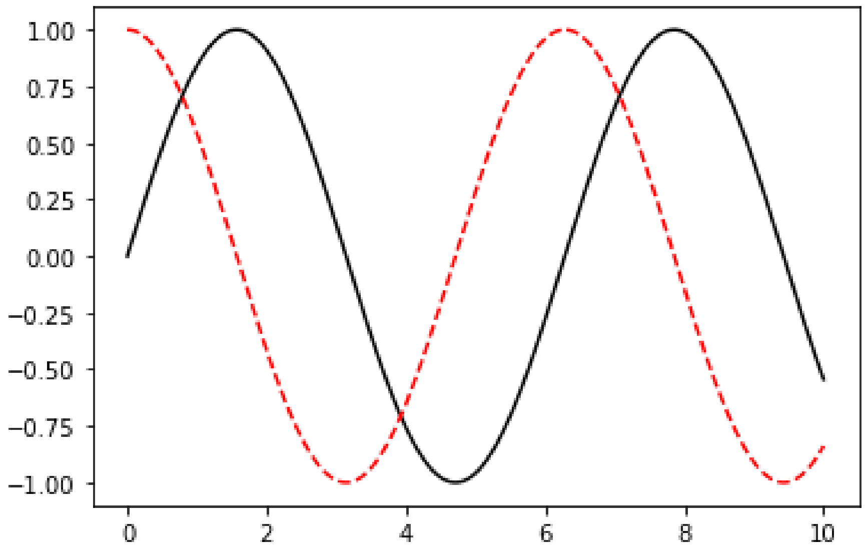

Figure 2: Plot multiple lines

It is sometimes helpful to compare different datasets visually on the same figure and matplotlib makes it easy!

Scatter Plot

Another simple plot type is a scatter plot, instead of points being joined by line segments, points are represented individually with a dot, or another shape. Use plt.scatter() to show the relationship of heights and ages. Run the script *scatter_plot.py*:

```
import matplotlib.pyplot as plt
import pandas as pd
presidents_df = pd.read_csv('presidents.csv', index_col='name')
plt.scatter(presidents_df['height'], presidents_df['age'])
plt.show()
```

So we created a scatter plot demonstrating the correlation between heights and ages of the presidents. Note that Matplotlib infers the appropriate ranges for us from the data along both x- and y-axis.

The scatter plot is shown in Figure 3.

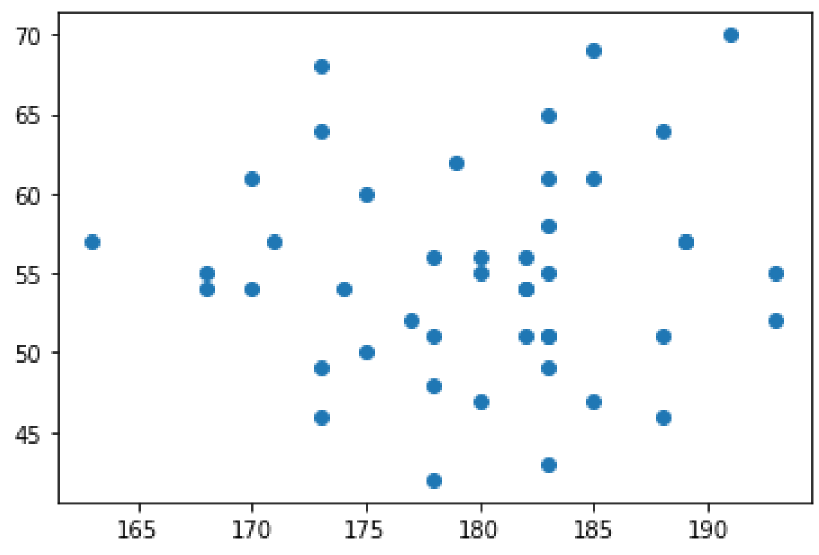

Figure 3: A scatter plot generated by the *matplotlib* module

Plotting with Pandas

A great thing about pandas is that it integrates well with *matplotlib*, so we can plot directly from DataFrames and Series. We specify the kind of plot as 'scatter', 'height' along x-axis, and 'age' along y-axis, and then give it a title.

Run the following script *scatter_pd.py*.

```
import pandas as pd
presidents_df = pd.read_csv('presidents.csv', index_col='name')
presidents_df.plot(kind = 'scatter',
                    x = 'height',
                    y = 'age',
                    title = 'U.S. presidents')
```

We created a scatter plot from the DataFrame, with both axis labels and title supplied. As we specified the x-axis and y-axis with column names from the DataFrame, the labels were also annotated on the axes.

The scatter plot generated by *pandas* is shown in Figure 4.

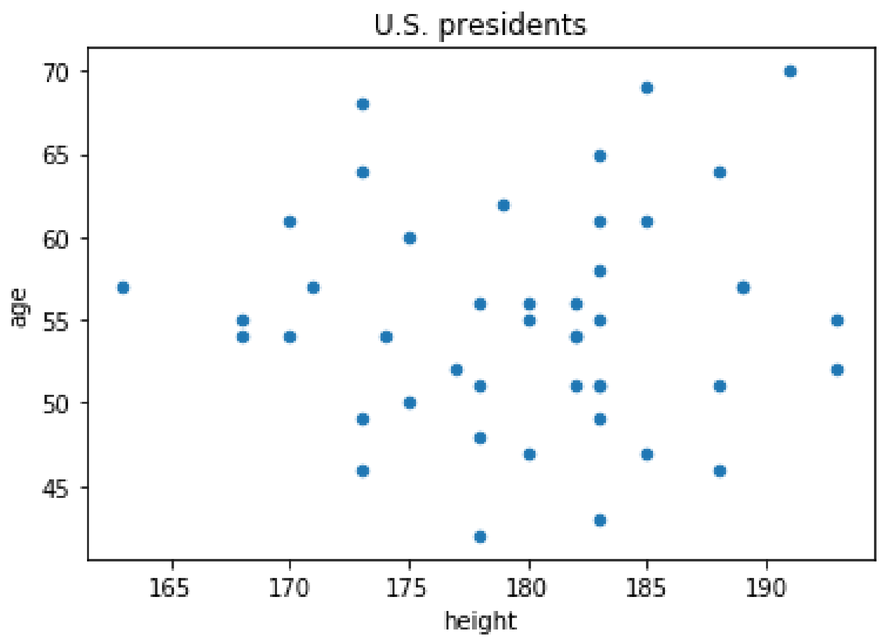

Figure 4: A scatter plot generated using the *pandas* module

We have created a scatter plot without directly importing the *matplotlib* module. The plot looks similar to the one we have seen in Figure 3, but with a title and axis labels.

Histogram

A histogram is a diagram that consists of rectangles with width equal to the interval and area proportional to the frequency of a variable. For example, in the histogram below (see Figure 5), there are five bins with equal length (i.e., [163, 169), [169, 175), [175, 181), [181, 187), and [187, 193)). The histogram tells us that there are 3 presidents whose height is between 163 cm and 169 cm.

Run the script *histogram.py*. It uses two different ways of creating the histogram, one with the *matplotlib* module, and one using *pandas* without directly importing the *matplotlib* module.

```
import matplotlib.pyplot as plt
import pandas as pd
presidents_df = pd.read_csv('presidents.csv', index_col='name')
presidents_df['height'].plot(kind='hist',
title = 'height',
bins=5)
plt.show()
plt.hist(presidents_df['height'], bins=15)
plt.show()
```

Here we see it is a left skewed distribution (the tail of the distribution on the left hand side is longer than on the right hand side) indicating that the mean (180.0 cm) is slightly lower than the median (182.0 cm). The distribution isn't quite symmetric.

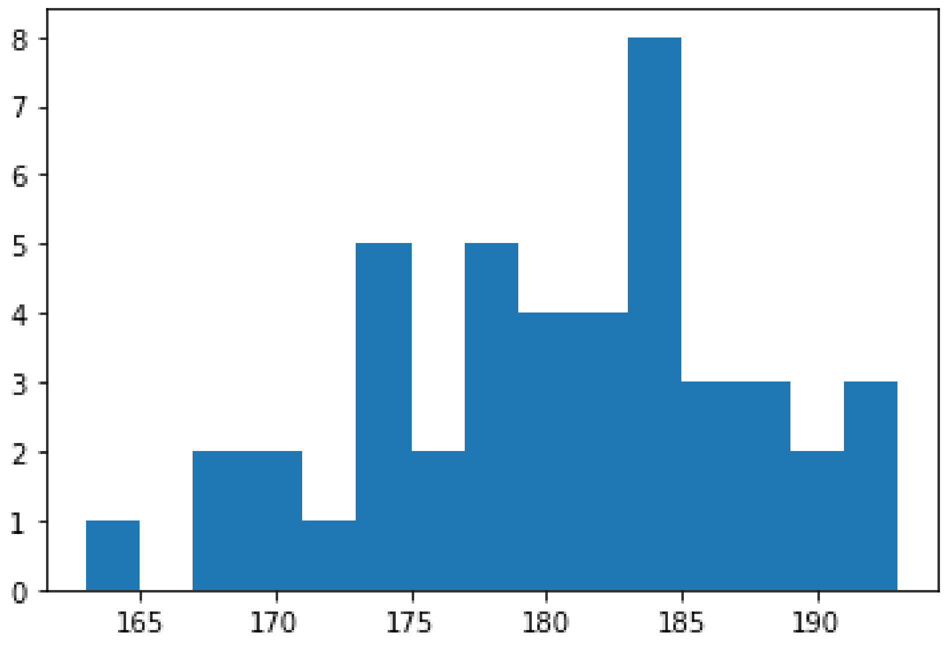

Figure 5: A histogram of presidents' heights

A histogram shows the underlying frequency distribution of a set of continuous data, allowing for inspection of the data for its shape, outliers, skewness, etc.

Bar Plot

Bar plots show the distribution of data over several groups. For example, a bar plot can depict the distribution of presidents by party.

Run the script *barplot.py*.

```
import matplotlib.pyplot as plt
import pandas as pd
presidents_df = pd.read_csv('presidents.csv', index_col='name')
party_cnt = presidents_df['party'].value_counts()
```

```
plt.style.use('ggplot')
party_cnt.plot(kind ='bar')
plt.show()
```

Bar plots are commonly confused with a histogram. Histogram presents numerical data whereas bar plot shows categorical data. The histogram is drawn in such a way that there is no gap between the bars, unlike in bar plots.

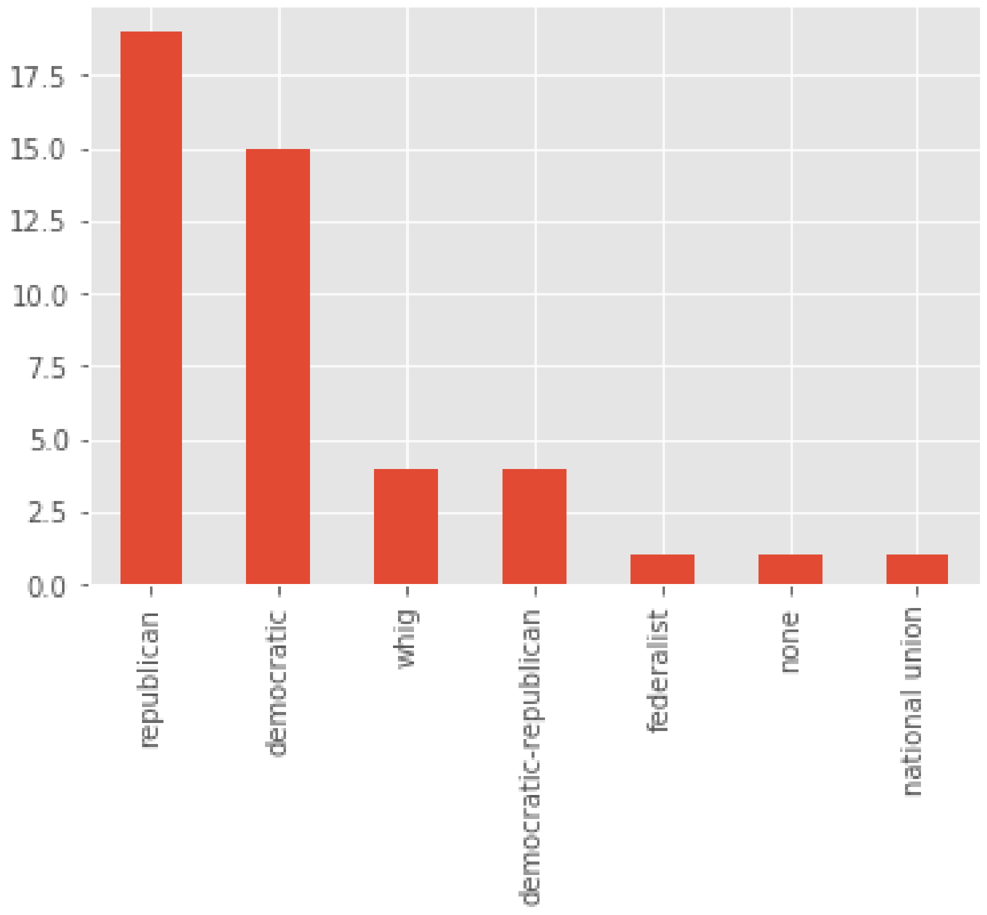

Figure 6: A bar plot of presidents' party affiliations

Making plots directly off pandas Series or DataFrame is powerful. It comes in handy when we explore data analysis in data science.

End of Chapter Exercises

1: Which of the following would correctly select row 5 and column 1 in the 2darray 'arr'?

```
arr[1,5]
arr[4,0]
arr[0,4]
arr[5,1]
```

2: Which of the following is the correct syntax to select the second column of the 2darray heights_and_ages_arr of shape (2, 45)?

```
heights_and_ages_arr[1, :]
heights_and_ages_arr[:, 1]
heights_and_ages_arr[, 1]
```

3: Suppose you have a Python list lst = [1,-1,1,-1], write a Python script to create a numpy array from the list lst.

4: What is the correct shape of arr = np.array([1,2,3])?

5: To compute the column sums of the 2darray `arr`, which sum() method would we use?

```
arr.sum(axis = 1)
arr.sum()
arr.sum(axis = 0)
```

6: Add a line of code at the end of the following script to find the column minimums.

```
import numpy as np
arr = np.array([[ 1, 2, 3], [2, 4, 6]])
```

7: Which of the following commands is to multiply the first row of a 2darray arr by 3:

```
arr[:,0]*3
arr[0] * 3
arr[0,:]*3
```

8: Add a line of code at the end of the script *operations.py* to print out the age of the 44th president (Obama) when he took office.

9: Add a line of code at the end of the script *comparison.py* to find out how many presidents are taller than 188cm.

10: Add a line of code at the end of the script *comparison.py* to find out how many presidents are exactly 183cm tall.

11: Add a line of code at the end of the script *shape_size.py* to print out the last six rows of observations of the DataFrame presidents_df.

12: Which of the following code would find the number of rows in a dataframe df?

```
df.shape[0]
df.nrow()
df.shape[1]
```

13: Add a line of code at the end of the script *shape_size.py* to print out the median height and median age of the 45 presidents.

14: Change the script *two_lines.py* so that the sin curve is blue and the cos curve is green.

15: Modify the script *histogram.py* to create a histogram of the presidents' age with ten bins. Use *pandas* without directly importing the *matplotlib* module.