Does time of day affect arrest rate?

ANALYZING POLICE ACTIVITY WITH PANDAS

Kevin Markham
Founder, Data School

Analyzing datetime data

```
apple
```

```
    price    volume       date_and_time
0  174.35  20567800 2018-01-08 16:00:00
1  174.33  21584000 2018-01-09 16:00:00
2  155.15  54390500 2018-02-08 16:00:00
3  156.41  70672600 2018-02-09 16:00:00
4  176.94  23774100 2018-03-08 16:00:00
5  179.98  32185200 2018-03-09 16:00:00
```

2. Analyzing datetime data
Back in chapter 1, we worked with a small DataFrame of Apple stock prices. We're going to use it here again, but this time it includes two days each from the first three months of 2018. There's also a new column, volume, that displays the number of Apple shares traded that day.

Accessing datetime attributes (1)

```
apple.dtypes
```

```
price                    float64
volume                     int64
date_and_time     datetime64[ns]
```

```
apple.date_and_time.dt.month
```

```
0    1
1    1
2    2
3    2
...
```

3. Accessing datetime attributes (1)
You might recall that we converted the date_and_time column to pandas datetime format. Because of datetime format, you actually have access to special date-based attributes via the dt accessor. For example, you can access the month as an integer by using the dt dot month attribute. There are many other similar attributes available, such as week, dayofweek, hour, minute, and so on.

Accessing datetime attributes (2)

```
apple.set_index('date_and_time', inplace=True)
apple.index
```

```
DatetimeIndex(['2018-01-08 16:00:00', '2018-01-09 16:00:00',
               '2018-02-08 16:00:00', '2018-02-09 16:00:00',
               '2018-03-08 16:00:00', '2018-03-09 16:00:00'],
              dtype='datetime64[ns]', name='date_and_time', freq=None)
```

```
apple.index.month
```

```
Int64Index([1, 1, 2, 2, 3, 3], dtype='int64', name='date_and_time')
```

- `dt` accessor is not used with a DatetimeIndex

4. Accessing datetime attributes (2)
Similar to our traffic stops dataset, we can set the date_and_time column as the DataFrame index. Because of its data type, it is now a DatetimeIndex. We can still access the same datetime attributes, such as month, and we get the same result as before, but we no longer have to use the dt accessor.

Calculating the monthly mean price

```
apple.price.mean()
```

```
169.52666666666667
```

```
apple.groupby(apple.index.month).price.mean()
```

```
date_and_time
1    174.34
2    155.78
3    178.46
Name: price, dtype: float64
```

```
monthly_price = apple.groupby(apple.index.month).price.mean()
```

5. Calculating the monthly mean price

Let's examine the price column of the apple DataFrame. If we wanted to calculate the mean price for all rows, we would simply use the mean() method. But what if we wanted to calculate the mean price for each month? One idea would be to use a groupby() operation, but we can't group by month as a string since it's not a column in the DataFrame. Instead, we would group by apple dot index dot month, and then take the mean() of the price column. This operation outputs a Series, in which the index is the month number and the values are the mean prices. We'll go ahead and save this Series as an object called monthly_price.

Plotting the monthly mean price

```
import matplotlib.pyplot as plt
```

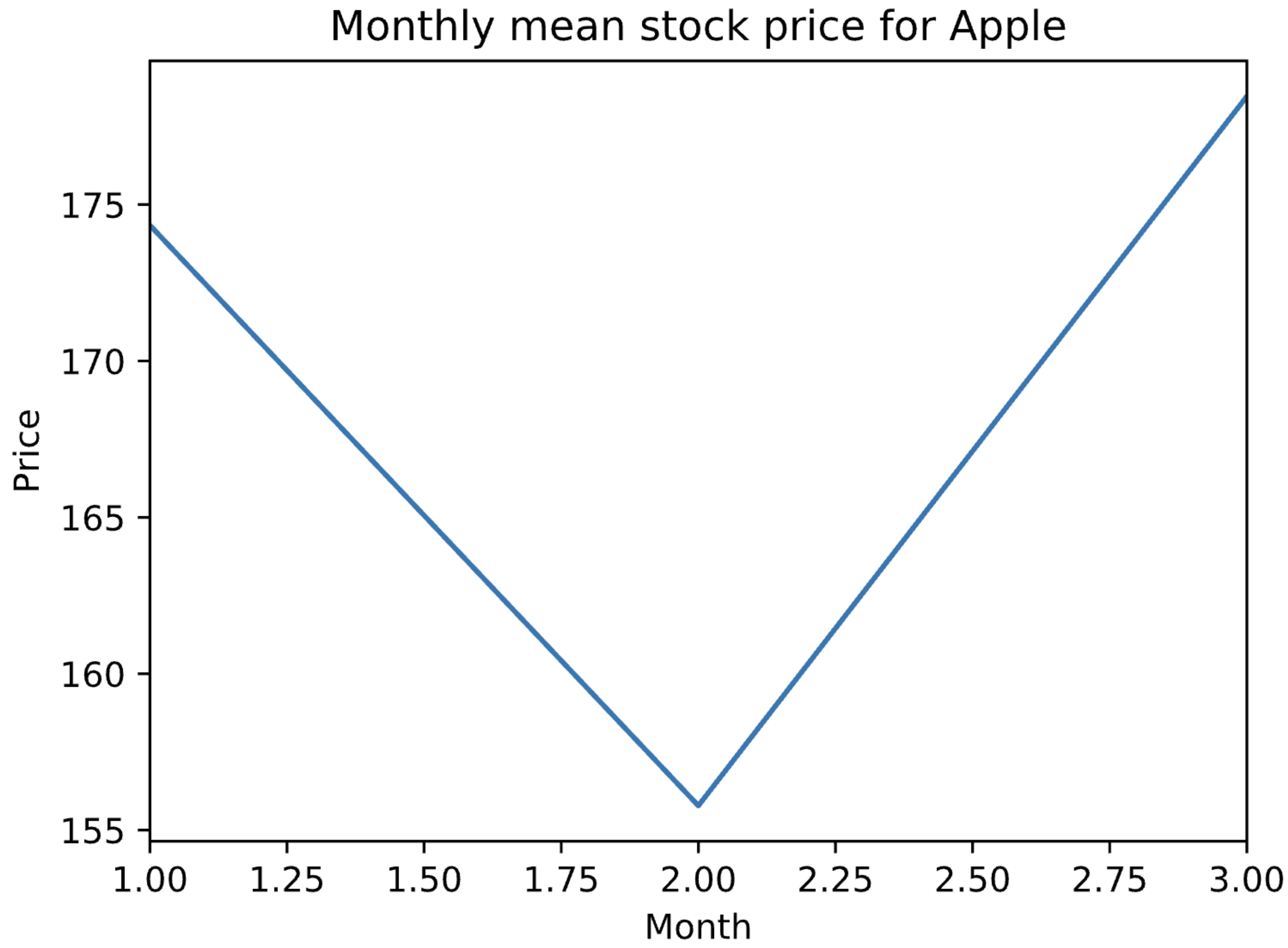
```
monthly_price.plot()
```

- Line plot: Series index on x-axis, Series values on y-axis

```
plt.xlabel('Month')
plt.ylabel('Price')
plt.title('Monthly mean stock price for Apple')
```

```
plt.show()
```

6. Plotting the monthly mean price
Let's say that we wanted to plot this data in order to visually examine the monthly price trends. We would start by importing matplotlib dot pyplot as plt. Then, we call the plot() method on the monthly_price Series. The default plot for a Series is a line plot, which uses the Series index on the x-axis and the Series values on the y-axis. Finally, we'll label the axes and provide a title for the plot, and then use the show() function to display the plot.



7. Line plot

It's a very simple plot in this case, but you can imagine that with a much larger dataset, **this plot could help you to understand the price trends in a way that examining the raw data could not.**

Let's practice!

ANALYZING POLICE ACTIVITY WITH PANDAS

Are drug-related stops on the rise?

ANALYZING POLICE ACTIVITY WITH PANDAS

Kevin Markham
Founder, Data School

Resampling the price

```
apple.groupby(apple.index.month).price.mean()
```

```
date_and_time
1     174.34
2     155.78
3     178.46
```

```
apple.price.resample('M').mean()
```

```
date_and_time
2018-01-31    174.34
2018-02-28    155.78
2018-03-31    178.46
```

Resampling the volume

```
apple
```

```
date_and_time          price    volume
2018-01-08 16:00:00   174.35   20567800
2018-01-09 16:00:00   174.33   21584000
2018-02-08 16:00:00   155.15   54390500
...                      ...        ...
```

```
apple.volume.resample('M').mean()
```

```
date_and_time
2018-01-31    21075900
2018-02-28    62531550
2018-03-31    27979650
```

Concatenating price and volume

```
monthly_price = apple.price.resample('M').mean()
monthly_volume = apple.volume.resample('M').mean()
```

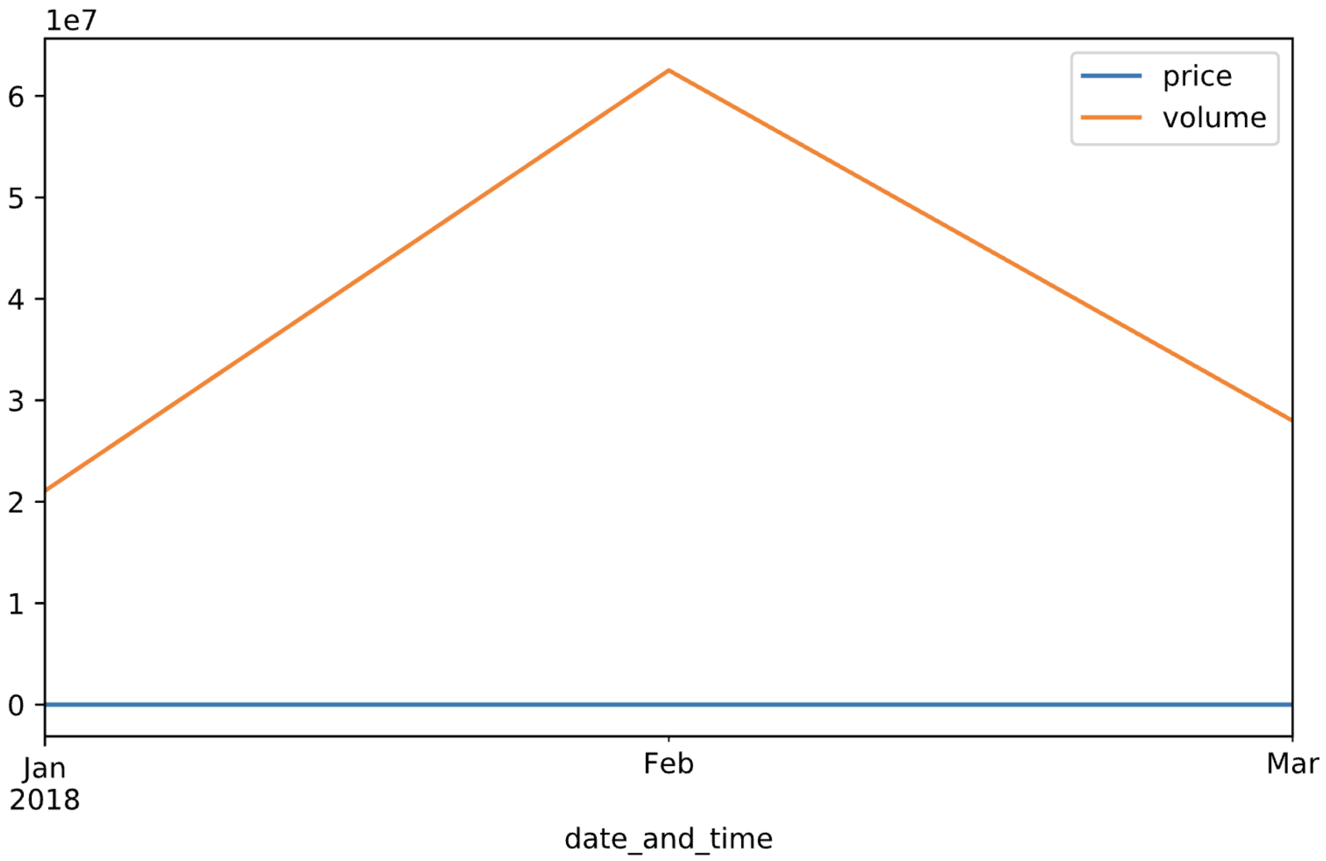
```
pd.concat([monthly_price, monthly_volume], axis='columns')
```

```
date_and_time   price    volume
2018-01-31     174.34  21075900
2018-02-28     155.78  62531550
2018-03-31     178.46  27979650
```

```
monthly = pd.concat([monthly_price, monthly_volume],
                    axis='columns')
```

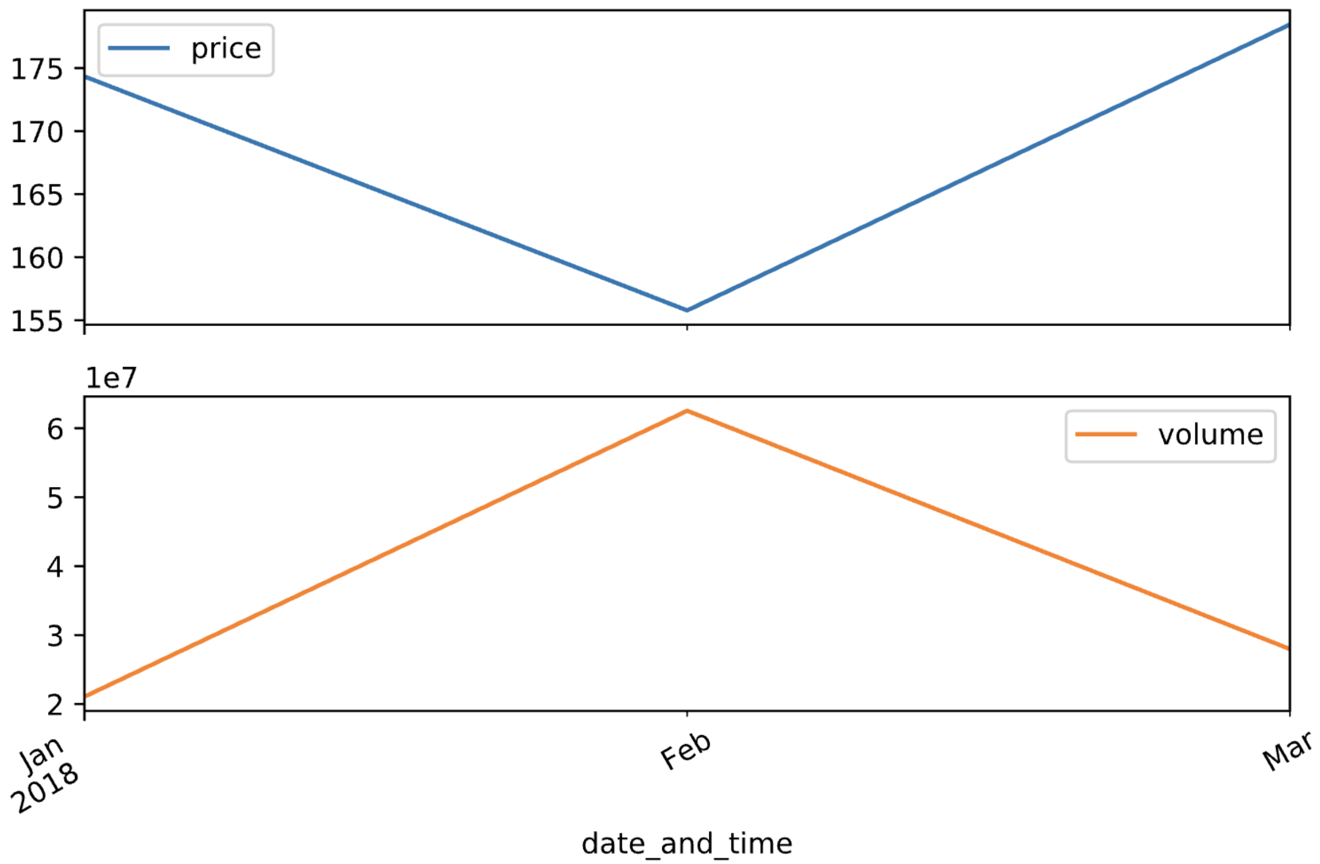
Plotting price and volume (1)

```
monthly.plot()
plt.show()
```



Plotting price and volume (2)

```
monthly.plot(subplots=True)
plt.show()
```



Let's practice!

ANALYZING POLICE ACTIVITY WITH PANDAS

What violations are caught in each district?

ANALYZING POLICE ACTIVITY WITH PANDAS

Kevin Markham
Founder, Data School

Computing a frequency table

```
pd.crosstab(ri.driver_race,
            ri.driver_gender)
```

```
driver_gender      F      M
driver_race
Asian            551   1838
Black           2681   9604
Hispanic        1953   7774
Other             53    212
White          18536  43334
```

- Frequency table: Tally of how many times each combination of values occurs

```
ri[(ri.driver_race == 'Asian') &
   (ri.driver_gender == 'F')
  ].shape
```

```
(551, 14)
```

- `driver_race` is along the index, `driver_gender` is along the columns

```
table = pd.crosstab(
  ri.driver_race,
  ri.driver_gender)
```

2. Computing a frequency table
One pandas function that might be new to you is crosstab(), short for cross-tabulation. To use crosstab(), you pass it two pandas Series that represent categories, and it outputs a frequency table in the form of a DataFrame. You can think of a frequency table as a tally of how many times each combination of values occurs in the dataset. ... Notice that race is along the index of the DataFrame and gender is along the columns, though you could transpose the DataFrame by reversing the order in which race and gender are passed to crosstab(). Let's go ahead and save the frequency table as an object called table.

Selecting a DataFrame slice

- `.loc[]` accessor: Select from a DataFrame by label

```
table
```

```
driver_gender      F      M
driver_race
Asian            551   1838
Black           2681   9604
Hispanic        1953   7774
Other             53    212
White          18536  43334
```

```
table.loc['Asian':'Hispanic']
```

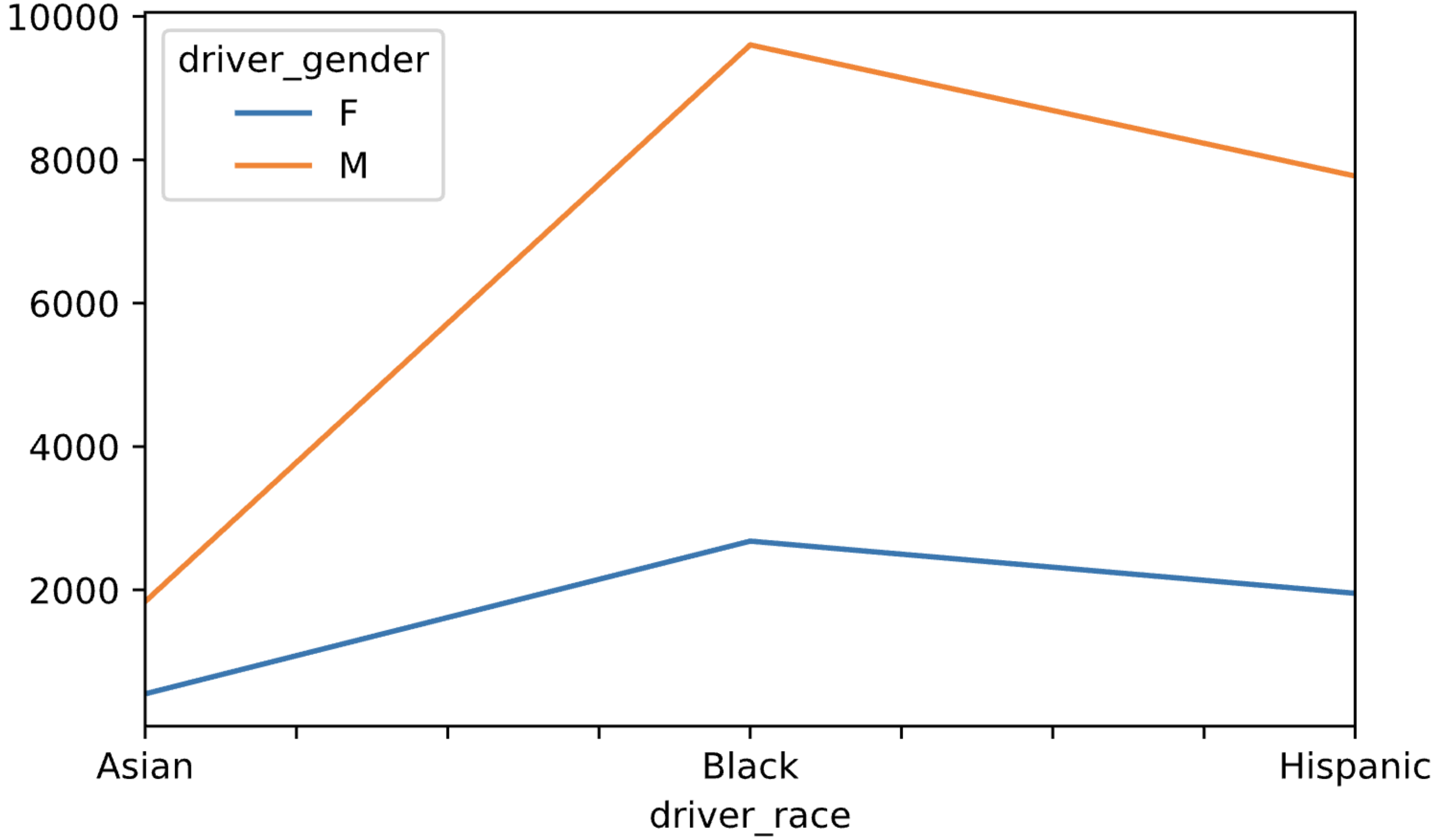
```
driver_gender      F      M
driver_race
Asian            551   1838
Black           2681   9604
Hispanic        1953   7774
```

```
table =
    table.loc['Asian':'Hispanic']
```

3. Selecting a DataFrame slice
As you might recall from previous courses, the loc accessor allows you to select portions of a DataFrame by label. Given our frequency table, let's pretend we wanted to select the Asian through Hispanic rows only. Using loc, we can extract this slice of the DataFrame by specifying the starting and ending labels, separated by a colon. Let's overwrite our existing table object with this smaller DataFrame.

Creating a line plot

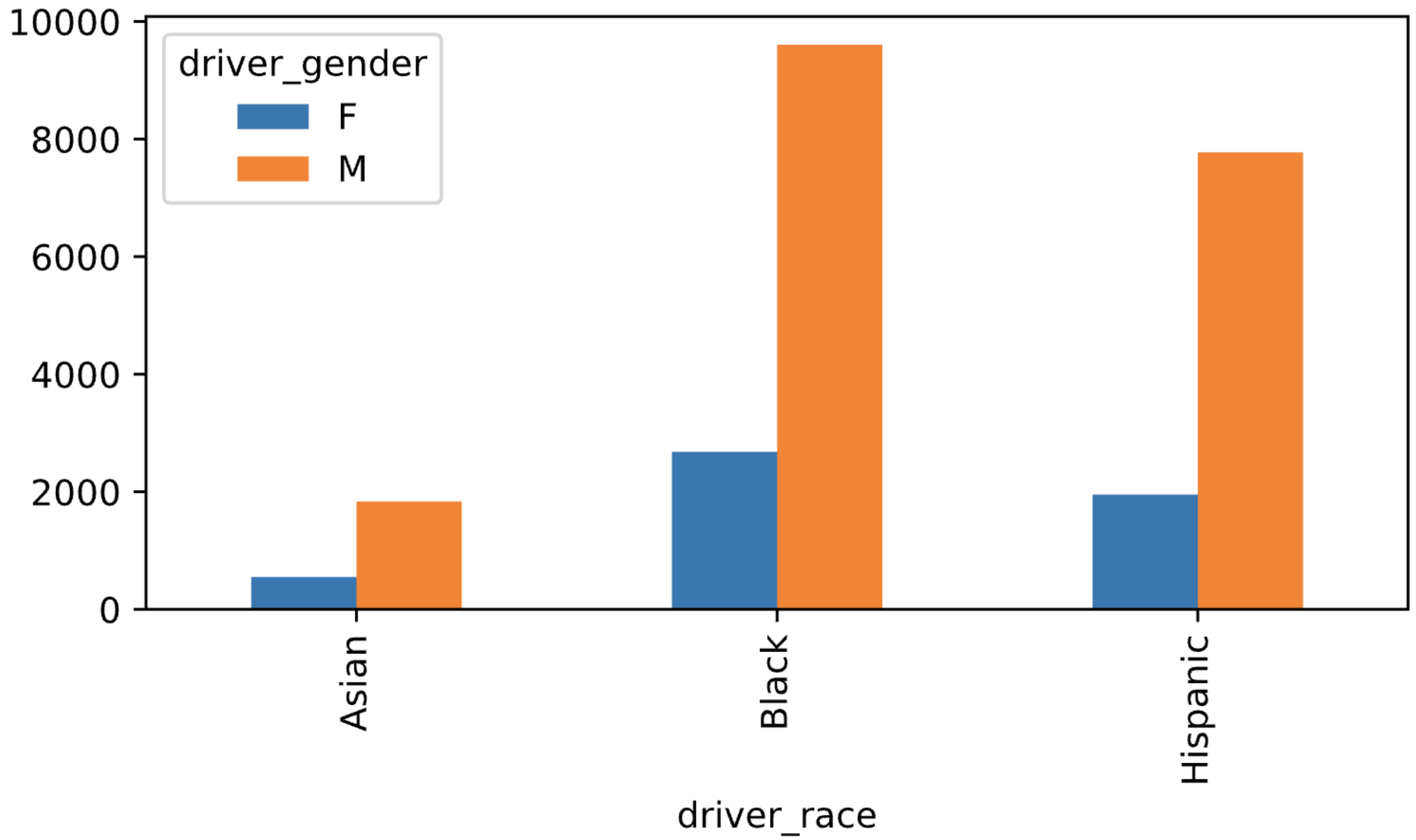
```
table.plot()
plt.show()
```



4. Creating a line plot
If we plot the table object, we'll get a line plot by default, in which the index is along the x-axis and each column becomes a line. However, a line plot is not appropriate in this case because it implies a change in time along the x-axis, whereas the x-axis actually represents three distinct categories.

Creating a bar plot

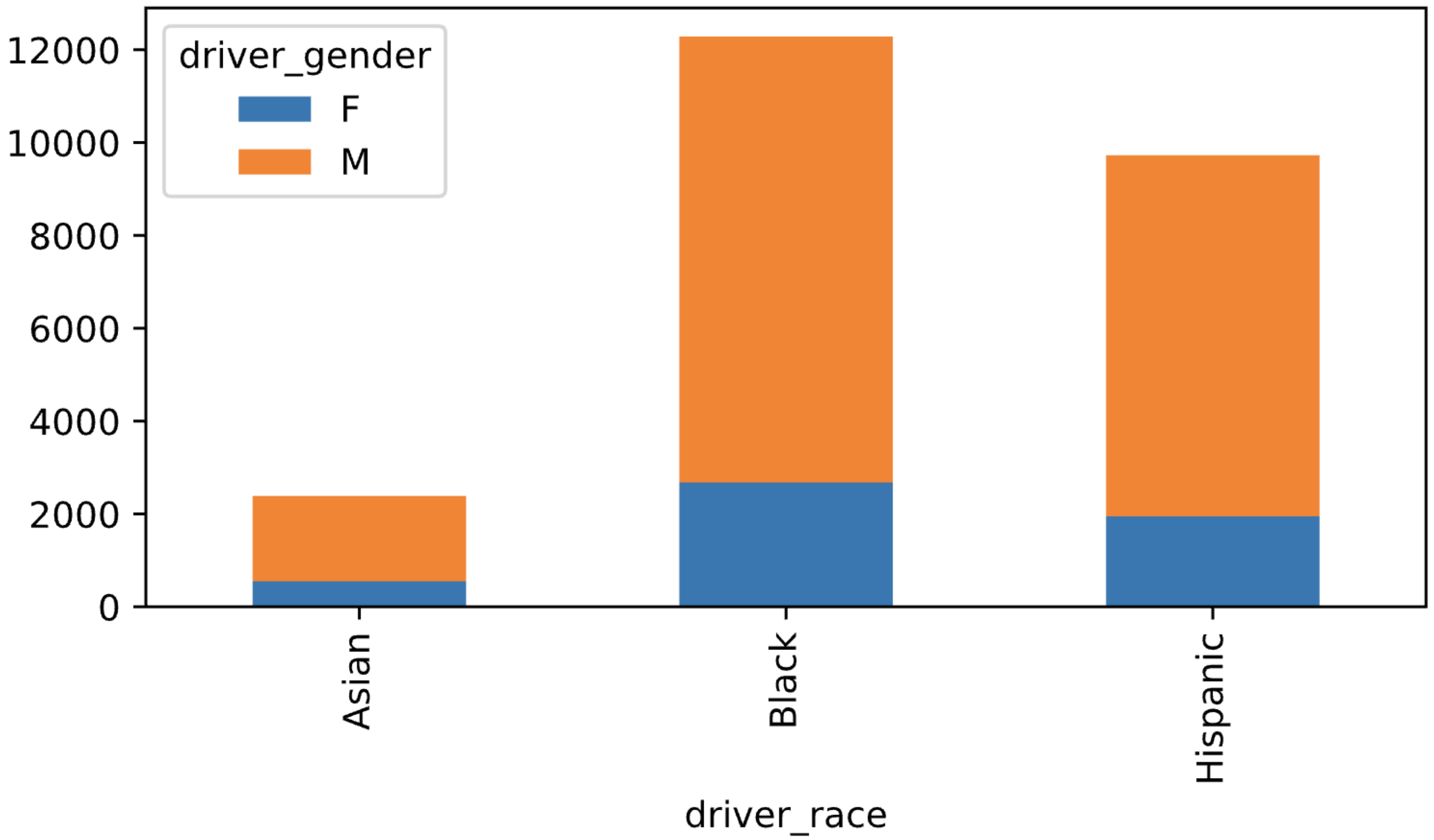
```
table.plot(kind='bar')
plt.show()
```



5. Creating a bar plot
By specifying kind equals bar, you can create a bar plot, which is much more appropriate than a line plot for comparing categorical data. With this plot, the numbers in our frequency table have been converted to bars for which the height represents the magnitude. Each gender has been assigned a color, and the two gender bars for each race are placed next to one another. The bar plot makes it especially easy to see the gender difference within each race. For all three races, we see that the number of males stopped is far greater than the number of females stopped.

Stacking the bars

```
table.plot(kind='bar', stacked=True)
plt.show()
```



6. Stacking the bars
A variation of the bar plot is the stacked bar plot, which you can generate by adding the argument stacked equals True. For each race, the two gender bars are now stacked on top of one another. The strength of this plot is that it helps you to see the total stops for each race, which was not as obvious when the bars were side-by-side. By emphasizing the totals, however, this plot slightly deemphasizes the individual components of each bar, and makes those components harder to compare against one another. Neither type of bar plot is right or wrong, rather you should choose the plot that best helps to answer the question you're asking.

Let's practice!

ANALYZING POLICE ACTIVITY WITH PANDAS

How long might you be stopped for a violation?

ANALYZING POLICE ACTIVITY WITH PANDAS

Kevin Markham
Founder, Data School

Analyzing an object column

```
apple
```

```
date_and_time          price    volume  change
2018-01-08 16:00:00   174.35  20567800    down
...                      ...       ...     ...
2018-03-09 16:00:00   179.98  32185200      up
```

- Create a Boolean column: `True` if the price went up, and `False` otherwise
- Calculate how often the price went up by taking the column mean

```
apple.change.dtype
```

```
dtype('O')
```

- `.astype()` can't be used in this case

2. Analyzing an object column
Let's return again to our DataFrame of Apple stock prices. A new column called change has been added to the DataFrame. It indicates whether the stock price went up or down compared to the previous trading day. Let's pretend we wanted to calculate how often the price went up. One way to do this would be to create a Boolean column that is True if the price went up, and False otherwise. Then we could easily calculate how often the price went up by taking the mean of the Boolean column. But how would we create this column? The change column has the object data type because it contains strings, and previously we've used the astype() method to convert strings to numbers or Booleans. However, astype() only works when pandas can infer how the conversion should be done, and that's not the case here. We'll need to find a different

Mapping one set of values to another

- Dictionary maps the values you have to the values you want

```
mapping = {'up':True, 'down':False}
apple['is_up'] = apple.change.map(mapping)
apple
```

```
date_and_time          price    volume change  is_up
2018-01-08 16:00:00   174.35  20567800   down  False
...                      ...       ...    ...    ...
2018-03-09 16:00:00   179.98  32185200     up   True
```

```
apple.is_up.mean()
```

```
0.5
```

3. Mapping one set of values to another
When you need to map one set of values to another, you can use the Series map() method. You provide it with a dictionary that maps the values you currently have to the values that you want. In this case, we want to map "up" to True and "down" to False, so we'll create a dictionary called mapping that specifies this. Then, we'll use the map() method on the change column, pass it the mapping object, and store the result in a new column called is_up. When we print the DataFrame, you'll see that the is_up column contains True when the change column says up, and False when the change column says down. Now that we have a Boolean column, we can calculate how often the price went up by taking the mean() of that column. The answer is that it went up 50% of the time.

Calculating the search rate

- Visualize how often searches were done after each violation type

```
ri.groupby('violation').search_conducted.mean()
```

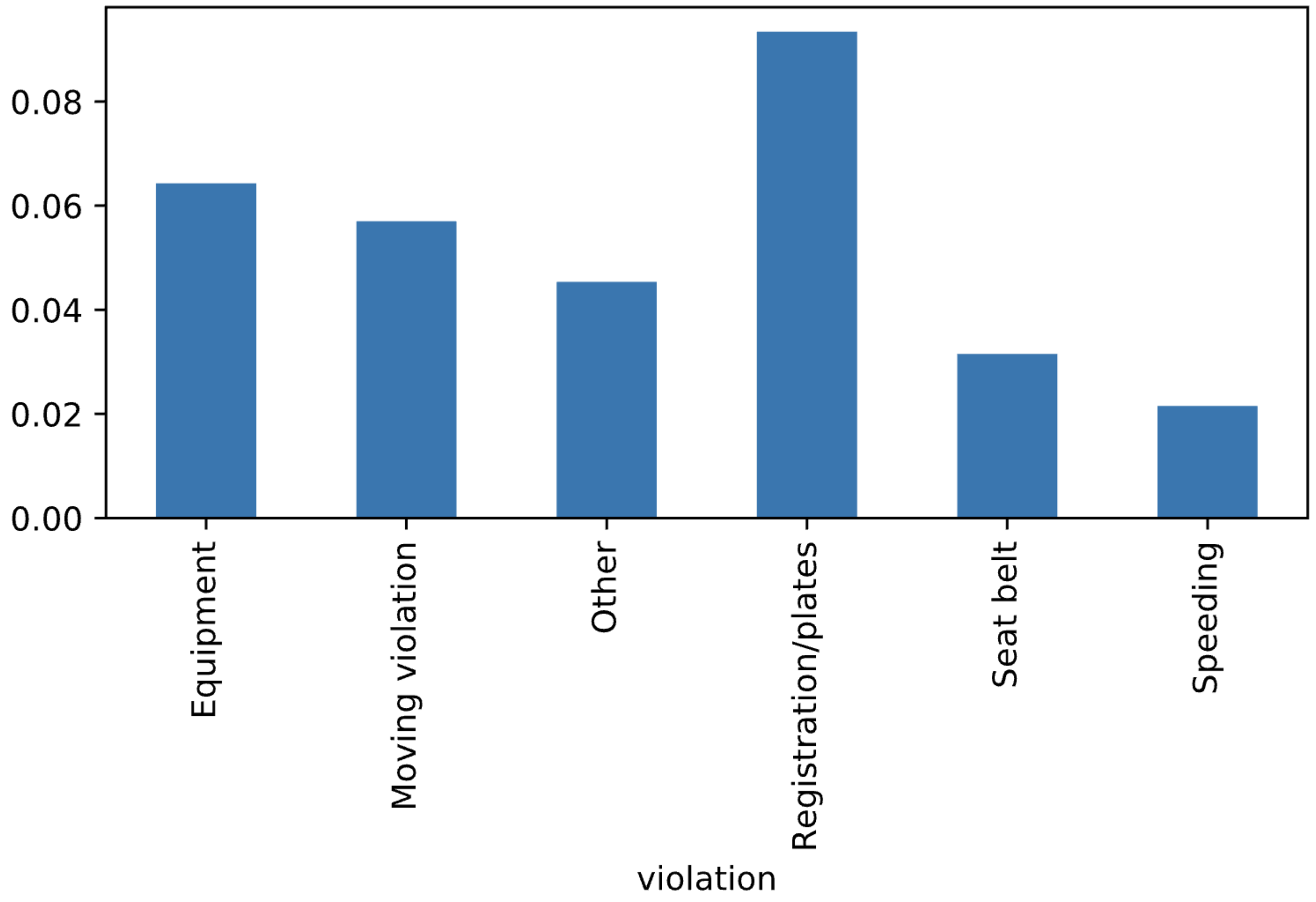
```
violation
Equipment              0.064280
Moving violation       0.057014
Other                  0.045362
Registration/plates    0.093438
Seat belt              0.031513
Speeding               0.021560
```

4. Calculating the search rate
Now we're going to return to our DataFrame of traffic stops, and shift to a completely separate topic. Let's say that we wanted to visualize how often searches were performed after each type of violation. We would group by violation, and then take the mean() of search_conducted. This calculates the search_rate for each of the six violation types, and returns a Series that is sorted in alphabetical order by violation. We'll save this as an object named search_rate.

```
search_rate = ri.groupby('violation').search_conducted.mean()
```

Creating a bar plot

```
search_rate.plot(kind='bar')
plt.show()
```



5. Creating a bar plot
To visualize the search rate, we'll create a bar plot since we're comparing the search rate across categories. The violations are displayed on the x-axis, and the search rate is on the y-axis. This plot looks okay, but there are two simple changes we can make that will make this plot more effective.

Ordering the bars (1)

- Order the bars from left to right by size

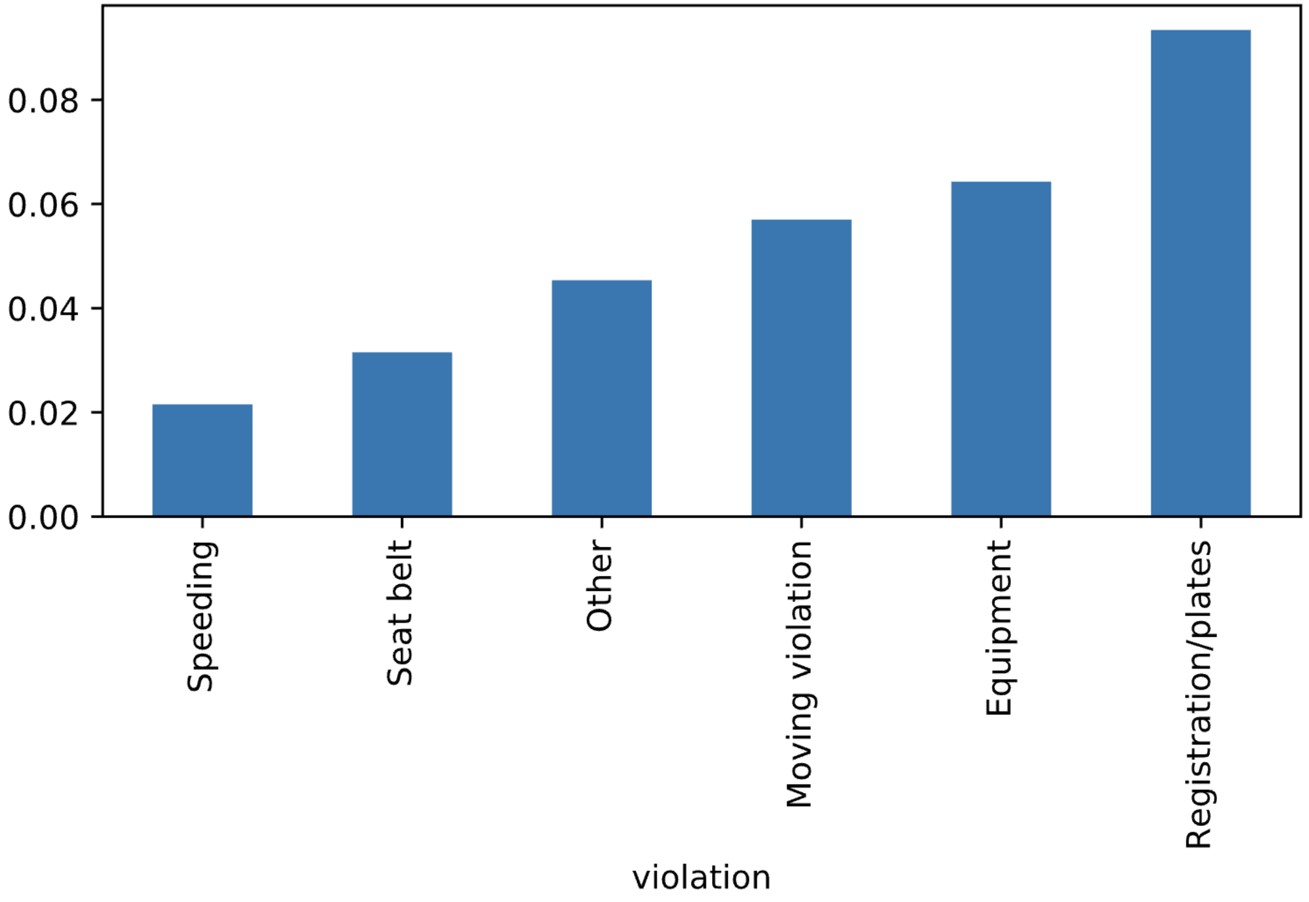
```
search_rate.sort_values()
```

```
violation
Speeding              0.021560
Seat belt             0.031513
Other                 0.045362
Moving violation      0.057014
Equipment             0.064280
Registration/plates   0.093438
Name: search_conducted, dtype: float64
```

6. Ordering the bars (1)
The first improvement we can make is to order the bars from left to right by size, which will make the plot easier to understand. All we need to do is to use the sort_values() method to sort the search_rate Series in ascending order.

Ordering the bars (2)

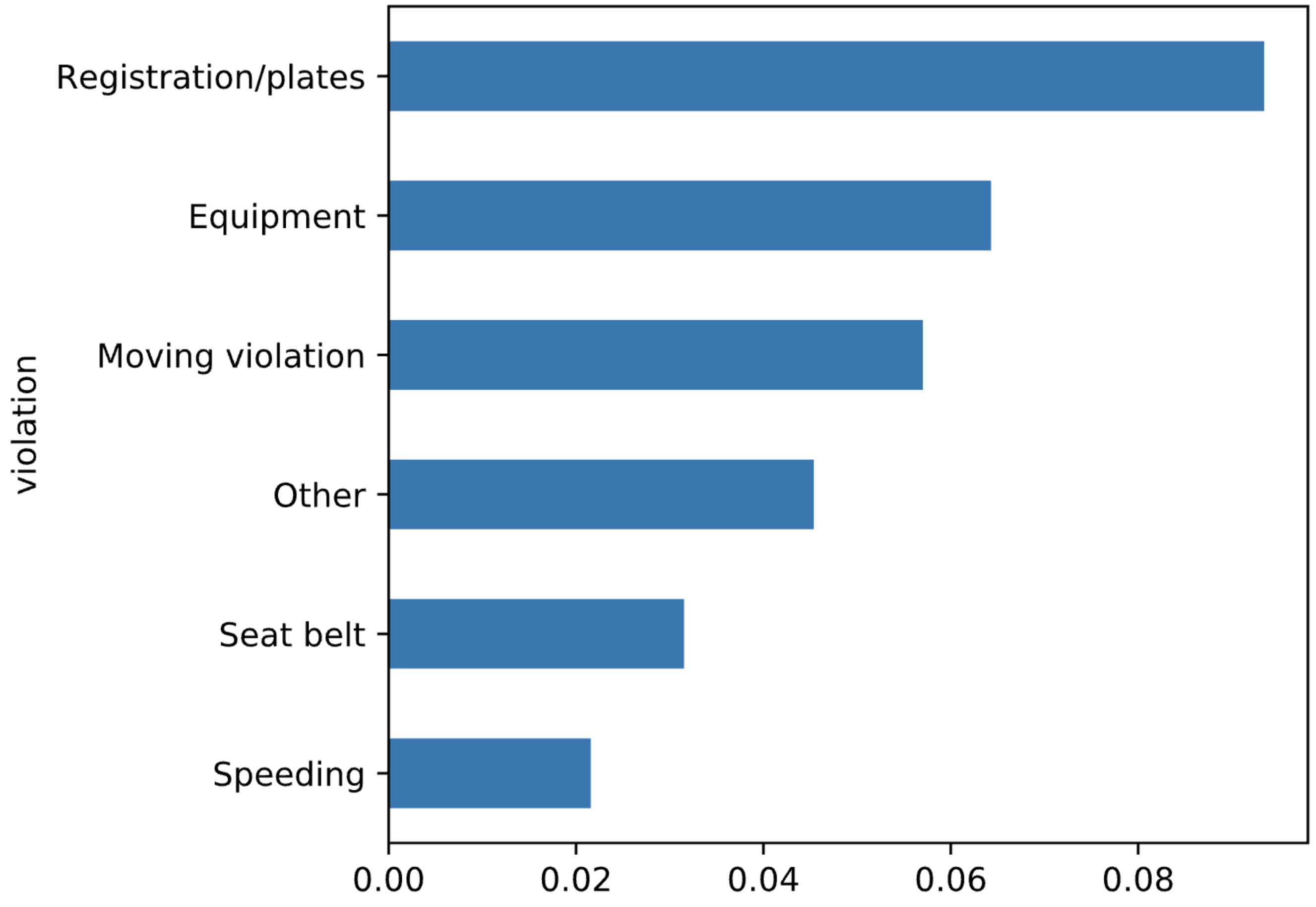
```
search_rate.sort_values().plot(kind='bar')
plt.show()
```



7. Ordering the bars (2)
Then, when we call the plot method on the sorted data, the bars are now ordered. This makes it easy to see which violations have the highest and the lowest search rates.

Rotating the bars

```
search_rate.sort_values().plot(kind='barh')
plt.show()
```



8. Rotating the bars
The second improvement we can make is to change the kind argument from bar to barh, which will rotate the bars so that they're horizontal. This makes it much easier to read the labels for each bar.

Let's practice!

ANALYZING POLICE ACTIVITY WITH PANDAS