Do the genders commit different violations?

ANALYZING POLICE ACTIVITY WITH PANDAS

Kevin Markham
Founder, Data School

Counting unique values (1)

- `.value_counts()` : Counts the unique values in a Series
- Best suited for categorical data

```
ri.stop_outcome.value_counts()
```

```
Citation            77091
Warning              5136
Arrest Driver        2735
No Action             624
N/D                   607
Arrest Passenger      343
Name: stop_outcome, dtype: int64
```

2. Counting unique values (1)
Let's start by discussing a few methods that will help you with your analysis. The first method is value_counts(), which counts the unique values in a Series. It's best suited for a column that contains categorical rather than numerical data. For example, we can apply value_counts() to the stop_outcome column, which contains the outcome of each traffic stop. The results are displayed in descending order, so you can see that the most common outcome is a citation, also known as a ticket, and the second most common outcome is a warning.

Counting unique values (2)

```
ri.stop_outcome.value_counts().sum()
```

```
86536
```

```
ri.shape
```

```
(86536, 13)
```

3. Counting unique values (2)
Because value_counts() outputs a pandas Series, you can take the sum of this Series by simply adding the sum() method on the end. This is known as method chaining, a powerful technique we'll use throughout the course. The sum() of the value_counts() is actually equal to the number of rows in the DataFrame, which will be the case for any Series that has no missing values.

Expressing counts as proportions

```
ri.stop_outcome.value_counts()
```

```
77091/86536
```

```
0.89085467319959 32
```

```
ri.stop_outcome.value_counts(
  normalize=True)
```

```
Citation              77091
Warning                5136
Arrest Driver          2735
No Action               624
N/D                     607
Arrest Passenger        343
```

```
Citation              0.890855
Warning               0.059351
Arrest Driver         0.031605
No Action             0.007211
N/D                   0.007014
Arrest Passenger      0.003964
```

4. Expressing counts as proportions
Rather than examining the raw counts, you might prefer to see the stop outcomes as proportions of the total. So if you wanted to know what percentage of traffic stops ended in a citation, you would divide the number of citations by the total number of outcomes and get 0.89, or 89%. Rather than doing these calculations manually, you can instead set the normalize parameter of value_counts() to be True, and it will output proportions instead of counts. Citations are 89%, warnings are 6%, driver arrests are 3%, and so on.

Filtering DataFrame rows

```
ri.driver_race.value_counts()
```

```
White       61870
Black       12285
Hispanic     9727
Asian        2389
Other         265
```

```
white = ri[ri.driver_race == 'White']
white.shape
```

```
(61870, 13)
```

5. Filtering DataFrame rows
Let's now take a look at the value_counts() for a different column, driver_race. You can see that there are five unique categories present. If you wanted to filter the DataFrame to only include drivers of a particular race, such as White, you would write that as a condition and put it inside brackets, as you've seen previously. We'll save the result in a new object. The shape of the new DataFrame is 61,870 rows, because that's the number of White drivers in the dataset, and 13 columns. You can now analyze this smaller DataFrame separately.

Comparing stop outcomes for two groups

```
white.stop_outcome.value_counts(
  normalize=True)
```

```
Citation            0.902263
Warning             0.057508
Arrest Driver       0.024018
No Action           0.007031
N/D                 0.006433
Arrest Passenger    0.002748
```

```
asian = ri[ri.driver_race ==
          'Asian']
asian.stop_outcome.value_counts(
  normalize=True)
```

```
Citation            0.922980
Warning             0.045207
Arrest Driver       0.017581
No Action           0.008372
N/D                 0.004186
Arrest Passenger    0.001674
```

6. Comparing stop outcomes for two groups
For example, you could repeat the analysis of stop outcomes, but focus on White drivers only. Like before, you select the stop_outcome column and then chain the value_counts() method on the end. You could compare these results with the outcomes for another race, such as Asian, simply by changing the condition inside the brackets and then repeating the calculation. If you compare these two sets of numbers, you can see that the stop outcomes are fairly similar for these two groups.

Let's practice!

ANALYZING POLICE ACTIVITY WITH PANDAS

Does gender affect who gets a ticket for speeding?

ANALYZING POLICE ACTIVITY WITH PANDAS

Kevin Markham
Founder, Data School

Filtering by multiple conditions (1)

```
female = ri[ri.driver_gender == 'F']
female.shape
```

```
(23774, 13)
```

2. Filtering by multiple conditions (1)
We'll need to use one additional technique for this analysis, namely filtering a DataFrame by multiple conditions. You may remember this technique from previous courses, but we'll review it here. In the last exercise, you used a single condition, driver_gender equals F, to create a DataFrame of female drivers. It has 23,774 rows because that's the number of rows in the ri DataFrame that satisfy this condition.

Filtering by multiple conditions (2)

```
female_and_arrested = ri[(ri.driver_gender == 'F') &
                         (ri.is_arrested == True)]
```

- Each condition is surrounded by parentheses
- Ampersand (`&`) represents the `and` operator

```
female_and_arrested.shape
```

```
(669, 13)
```

- Only includes female drivers who were arrested

3. Filtering by multiple conditions (2)
What if we wanted to create a second DataFrame of female drivers, but only those who were arrested? We simply add a second condition to the filter, namely that the is_arrested column equals True. Notice that each condition is surrounded by parentheses, and there is an ampersand between the two conditions, which represents the logical AND operator. The second DataFrame is much smaller because it only includes rows that satisfy both conditions, meaning that it only includes female drivers who were also arrested.

Filtering by multiple conditions (3)

```
female_or_arrested = ri[(ri.driver_gender == 'F') |
                        (ri.is_arrested == True)]
```

- Pipe (`|`) represents the `or` operator

```
female_or_arrested.shape
```

```
(26183, 13)
```

- Includes all females
- Includes all drivers who were arrested

4. Filtering by multiple conditions (3)
When filtering a DataFrame by multiple conditions, another option is to use the vertical pipe character between the two conditions. The pipe represents the logical OR operator, which indicates that a row should be included in the DataFrame if it meets either condition. This DataFrame is larger than the last one because it includes all females regardless of whether they were arrested, as well as all drivers who were arrested, regardless of whether they are female.

Rules for filtering by multiple conditions

- Ampersand (`&`): only include rows that satisfy both conditions
- Pipe (`|`): include rows that satisfy either condition
- Each condition must be surrounded by parentheses
- Conditions can check for equality (`==`), inequality (`!=`), etc.
- Can use more than two conditions

5. Rules for filtering by multiple conditions
Here's a quick summary of the rules for filtering DataFrames by multiple conditions. Use the ampersand to only include rows that satisfy both conditions. Use the pipe to include rows that satisfy either condition. Each condition must be surrounded by parentheses. Conditions can check for equality, inequality, greater than, less than, and so on. And you can use more than two conditions to create a filter.

Correlation, not causation

- Analyze the relationship between gender and stop outcome
  - Assess whether there is a correlation
- Not going to draw any conclusions about causation
  - Would need additional data and expertise
  - Exploring relationships only

6. Correlation, not causation
In the upcoming exercises, you'll analyze the relationship between gender and stop outcome when a driver is pulled over for speeding. In other words, you're examining the data to assess whether there is a correlation between these two attributes. However, it's important to note that we're not going to draw any conclusions about causation during this course, since we don't have the data or the expertise required to do so. Instead, we're simply exploring the relationships between different attributes in the dataset.

Let's practice!

ANALYZING POLICE ACTIVITY WITH PANDAS

Does gender affect whose vehicle is searched?

ANALYZING POLICE ACTIVITY WITH PANDAS

Kevin Markham
Founder, Data School

datacamp

Math with Boolean values

```
ri.isnull().sum()
```

```
stop_date            0
stop_time            0
driver_gender        0
driver_race          0
violation_raw        0
...
```

- `True` = 1, `False` = 0

```
import numpy as np
np.mean([0, 1, 0, 0])
```

```
0.25
```

```
np.mean([False, True,
         False, False])
```

```
0.25
```

- Mean of Boolean Series represents percentage of `True` values

2. Math with Boolean values
Recall that you can perform mathematical operations on Boolean values. For example, you previously used the isnull() method to generate a DataFrame of True and False values, and then took the sum() to count the missing values in each column. This worked because True values were treated as ones and False values were treated as zeros. Now we'll use the NumPy library to demonstrate a different operation, namely the mean. If you take the mean() of the list 0 1 0 0 you'll get 0.25, calculated as 1 divided by 4. Similarly, if you take the mean() of the list False True False False, you'll also get 0.25. Thus, the mean of a Boolean Series represents the percentage of values that are True.

Taking the mean of a Boolean Series

```
ri.is_arrested.value_counts(normalize=True)
```

```
False    0.964431
True     0.035569
```

```
ri.is_arrested.mean()
```

```
0.0355690117407784
```

```
ri.is_arrested.dtype
```

```
dtype('bool')
```

3. Taking the mean of a Boolean Series
Now, let's see a real example of why it's useful to be able to take the mean of a Boolean Series. We'll first calculate the percentage of stops that result in an arrest using the value_counts() method. The arrest rate is around 3.6% since that's the percentage of True values. Note that this would work on an object column or a Boolean column. But we can get the same result more easily by taking the mean() of the is_arrested Series. This method only works because the data type is Boolean. This is exactly why you changed the data type of this Series from object to Boolean back in the first chapter.

Comparing groups using groupby (1)

- Study the arrest rate by police district

```
ri.district.unique()
```

```
array(['Zone X4', 'Zone K3', 'Zone X1', 'Zone X3',
       'Zone K1', 'Zone K2'], dtype=object)
```

```
ri[ri.district == 'Zone K1'].is_arrested.mean()
```

```
0.02434908389585342 3
```

4. Comparing groups using groupby (1)
The second technique we'll review is groupby(), which you've used in previous courses. Let's pretend that you wanted to study the arrest rate by police district. You can see that there are six districts by using the Series method unique(). One approach we've used to compare groups is to filter the DataFrame by each group, and then perform a calculation on each subset. So to calculate the arrest rate in Zone K1, we would filter by that district, select the is_arrested column, and then take the mean(). The arrest rate is about 2.4%, which is lower than the overall arrest rate of 3.6%.

Comparing groups using groupby (2)

```
ri[ri.district == 'Zone K2'].is_arrested.mean()
```

```
0.030800588834786546
```

```
ri.groupby('district').is_arrested.mean()
```

```
district
Zone K1    0.024349
Zone K2    0.030801
Zone K3    0.032311
Zone X1    0.023494
Zone X3    0.034871
Zone X4    0.048038
```

5. Comparing groups using groupby (2)
Next we calculate the arrest rate in Zone K2, which is about 3.1%. But rather than repeating this process for all six districts, we can instead group by the district column, which will perform the same calculation for all districts at once. You can see a noticeably higher arrest rate in Zone X4.

Grouping by multiple categories

```
ri.groupby(['district', 'driver_gender']).is_arrested.mean()
```

```
district  driver_gender
Zone K1   F                0.019169
          M                0.026588
Zone K2   F                0.022196
...       ...                   ...
```

```
ri.groupby(['driver_gender', 'district']).is_arrested.mean()
```

```
driver_gender  district
F              Zone K1     0.019169
               Zone K2     0.022196
...            ...              ...
```

6. Grouping by multiple categories
You can also group by multiple categories at once. For example, you can group by district and gender by passing it as a list of strings. This computes the arrest rate for every combination of district and gender. In other words, you can see the arrest rate for males and females in each district separately. Note that if you reverse the ordering of the items in the list, grouping first by gender and then by district, the calculations will be the same but the presentation of the results will be different. You can use whichever option makes it easier for you to understand the results.

Let's practice!

ANALYZING POLICE ACTIVITY WITH PANDAS

Does gender affect who is frisked during a search?

ANALYZING POLICE ACTIVITY WITH PANDAS

Kevin Markham
Founder, Data School

```
ri.search_conducted.value_counts()
```

```
False    83229
True      3307
```

```
ri.search_type.value_counts(dropna=False)
```

```
NaN                                83229
Incident to Arrest                  1290
Probable Cause                       924
Inventory                            219
Reasonable Suspicion                 214
Protective Frisk                     164
Incident to Arrest,Inventory         123
...
```

2. Examining the search types

As you've seen previously, the search_conducted field is True if there's a search during a traffic stop, and False otherwise. There's also a related field, search_type, that contains additional information about the search. Notice that the search_type field has 83,229 missing values, which is identical to the number of False values in the search_conducted field. That's because any time a search is not conducted, there's no information to record about a search, and thus the search_type will be missing. Note that the value_counts() method excludes missing values by default, and so we specified dropna equals False in order to see the missing values.

- `.value_counts()` excludes missing values by default
- `dropna=False` displays missing values

Examining the search types

```
ri.search_type.value_counts()
```

```
Incident to Arrest                          1290
Probable Cause                               924
Inventory                                    219
Reasonable Suspicion                         214
Protective Frisk                             164
Incident to Arrest,Inventory                 123
Incident to Arrest,Probable Cause            100
...
```

3. Examining the search types
There are only five possible values for search_type, which you can see at the top of the value_counts() output: Incident to Arrest, Probable Cause, Inventory, Reasonable Suspicion, and Protective Frisk. But sometimes, multiple values are relevant for a single traffic stop, in which case they're separated by commas. Let's focus on Inventory, meaning searches in which the police took an inventory of the vehicle. Looking at the third line of the value_counts() output, we see 219, which is the number of searches in which Inventory was the only search type. But what if we wanted to know the total number of times in which an inventory was done during a search? We'd also have to include any stops in which Inventory was one of multiple search types. To do this, we'll use a string method.

- Multiple values are separated by commas
- 219 searches in which "Inventory" was the only search type
- Locate "Inventory" among multiple search types

Searching for a string (1)

```
ri['inventory'] = ri.search_type.str.contains('Inventory', na=False)
```

- `str.contains()` returns `True` if string is found, `False` if not found
- `na=False` returns `False` when it finds a missing value

4. Searching for a string (1)
Back in chapter 1, you used a string method to concatenate two columns. This time, we'll use a string method called contains() that checks whether a string is present in each element of a given column. It returns True if the string is found, and False if it's not found. We also specify na equals False, which tells the contains() method to return False when it finds a missing value in the search_type column. We'll save the results in a new column called inventory.

Searching for a string (2)

```
ri.inventory.dtype
```

```
dtype('bool')
```

- `True` means inventory was done, `False` means it was not

```
ri.inventory.sum()
```

```
441
```

5. Searching for a string (2)
As expected, the data type of the column is Boolean. To be clear, a True value in this column means that an inventory was done during a search, and a False value means it was not. We can take the sum() of the inventory column to see that an inventory was done during 441 searches. This includes the 219 stops in which Inventory was the only search type, plus additional stops in which Inventory was one of multiple search types.

Calculating the inventory rate

```
ri.inventory.mean()
```

```
0.005096144957012106
```

- 0.5% of all traffic stops resulted in an inventory

```
searched = ri[ri.search_conducted == True]
searched.inventory.mean()
```

```
0.13335349259147264
```

- 13.3% of searches included an inventory

6. Calculating the inventory rate
What if we wanted to calculate the percentage of searches which included an inventory? You might think this would be as simple as taking the mean() of the inventory column, and the answer would be about 0.5%. But what's wrong with this calculation? 0.5% is the percentage of all traffic stops which resulted in an inventory, including those stops in which a search was not even done. Instead, we first need to filter the DataFrame to only include those rows in which a search was done, and then take the mean() of the inventory column. The correct answer is that 13.3% of searches included an inventory. This is a vastly different result, and it highlights the importance of carefully choosing which rows are relevant before doing a calculation.

Let's practice!

ANALYZING POLICE ACTIVITY WITH PANDAS