O'REILLY®

Advancing into Analytics

From Excel to Python and R

George Mount

CHAPTER 12

Data Manipulation and Visualization in Python

In Chapter 8 you learned how to manipulate and visualize data, with heavy help from the `tidyverse` suite of packages. Here we'll demonstrate similar techniques on the same *star* dataset, this time in Python. In particular, we'll use `pandas` and `seaborn` to manipulate and visualize data, respectively. This isn't a comprehensive guide to what these modules, or Python, can do with data analysis. Instead, it's enough to get you exploring on your own.

As much as possible, I'll mirror the steps and perform the same operations that we did in Chapter 8. Because of this familiarity, I'll focus less on the whys of manipulating and visualizing data than I will on hows of doing it in Python. Let's load the necessary modules and get started with *star*. The third module, `matplotlib`, is new for you and will be used to complement our work in `seaborn`. It comes installed with Anaconda. Specifically, we'll be using the `pyplot` submodule, aliasing it as `plt`.

```
In [1]: import pandas as pd
        import seaborn as sns
        import matplotlib.pyplot as plt

        star = pd.read_excel('datasets/star/star.xlsx')
        star.head()
Out[1]:
   tmathssk  treadssk            classk  totexpk   sex freelunk   race  \
0       473       447       small.class        7  girl       no  white
1       536       450       small.class       21  girl       no  black
2       463       439  regular.with.aide        0   boy      yes  black
3       559       448           regular       16   boy       no  white
4       489       447       small.class        5   boy      yes  white

   schidkn
```

```
0       63
1       20
2       19
3       69
4       79
```

Column-Wise Operations

In Chapter 11 you learned that `pandas` will attempt to convert one-dimensional data structures into Series. This seemingly trivial point will be quite important when selecting columns. Let's take a look at an example: say we *just* wanted to keep the *tmathssk* column from our DataFrame. We could do so using the familiar single-bracket notation, but this technically results in a Series, not a DataFrame:

```
In [2]:  math_scores = star['tmathssk']
         type(math_scores)

Out[2]: pandas.core.series.Series
```

It's probably better to keep this as a DataFrame if we aren't positive that we want *math_scores* to stay as a one-dimensional structure. To do so, we can use two sets of brackets instead of one:

```
In [3]: math_scores = star[['tmathssk']]
        type(math_scores)

Out[3]: pandas.core.frame.DataFrame
```

Following this pattern, we can keep only the desired columns in *star*. I'll use the `columns` attribute to confirm.

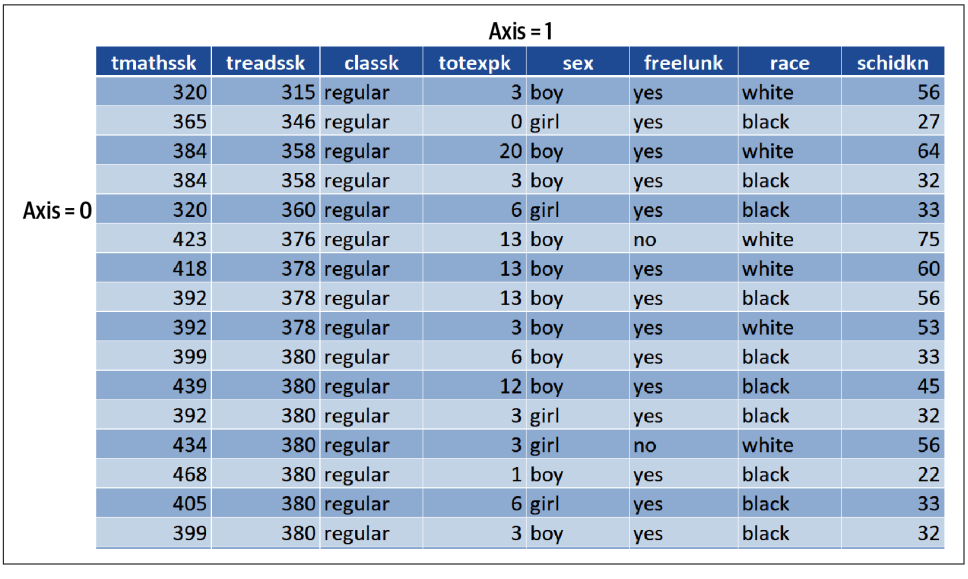
```
In [4]:  star = star[['tmathssk','treadssk','classk','totexpk','schidkn']]
         star.columns

Out[4]: Index(['tmathssk', 'treadssk', 'classk',
               'totexpk', 'schidkn'], dtype='object')
```

Object-Oriented Programming in Python

So far you've seen methods and functions in Python. These are things that objects can *do*. Attributes, on the other hand, represent some *state* of an object itself. These are affixed to an object's name with a period; unlike methods, no parentheses are used. Attributes, functions, and methods are all elements of *object-oriented programming* (OOP), a paradigm meant to structure work into simple and reusable pieces of code. To learn more about how OOP works in Python, check out *Python in a Nutshell*, 3rd edition by Alex Martelli et al. (O'Reilly).

To drop specific columns, use the `drop()` method. `drop()` can be used to drop columns *or* rows, so we'll need to specify which by using the `axis` argument. In `pandas`, rows are axis `0` and columns axis `1`, as Figure 12-1 demonstrates.



tmathssk	treadssk	classk	totexpk	sex	freelunk	race	schidkn
320	315	regular	3	boy	yes	white	56
365	346	regular	0	girl	yes	black	27
384	358	regular	20	boy	yes	white	64
384	358	regular	3	boy	yes	black	32
320	360	regular	6	girl	yes	black	33
423	376	regular	13	boy	no	white	75
418	378	regular	13	boy	yes	white	60
392	378	regular	13	boy	yes	black	56
392	378	regular	3	boy	yes	white	53
399	380	regular	6	boy	yes	black	33
439	380	regular	12	boy	yes	black	45
392	380	regular	3	girl	yes	black	32
434	380	regular	3	girl	no	white	56
468	380	regular	1	boy	yes	black	22
405	380	regular	6	girl	yes	black	33
399	380	regular	3	boy	yes	black	32

Figure 12-1. Axes of a `pandas` DataFrame

Here's how to drop the *schidkn* column:

```
In [5]: star = star.drop('schidkn', axis=1)
        star.columns

Out[5]: Index(['tmathssk', 'treadssk',
               'classk', 'totexpk'], dtype='object')
```

Let's now look at deriving new columns of a DataFrame. We can do that using bracket notation—this time, I *do* want the result to be a Series, as each column of a DataFrame is actually a Series (just as each column of an R data frame is actually a vector). Here I'll calculate combined math and reading scores:

```
In [6]: star['new_column'] = star['tmathssk'] + star['treadssk']
        star.head()

Out[6]:
   tmathssk  treadssk            classk  totexpk  new_column
0       473       447      small.class        7         920
1       536       450      small.class       21         986
2       463       439  regular.with.aide        0         902
3       559       448           regular       16        1007
4       489       447      small.class        5         936
```

Again, *new_column* isn't a terribly descriptive variable name. Let's fix that with the `rename()` function. We'll use the `columns` argument and pass data to it in a format you're likely unfamiliar with:

```
In [7]: star = star.rename(columns = {'new_column':'ttl_score'})
        star.columns

Out[7]: Index(['tmathssk', 'treadssk', 'classk', 'totexpk', 'ttl_score'],
           dtype='object')
```

The curly bracket notation used in the last example is a Python *dictionary*. Dictionaries are collections of *key-value* pairs, with the key and value of each element separated by a colon. This is a core Python data structure and one to check out as you continue learning the language.

Row-Wise Operations

Now let's move to common operations by row. We'll start with sorting, which can be done in `pandas` with the `sort_values()` method. We'll pass a list of columns we want to sort by in their respective order to the `by` argument:

```
In [8]: star.sort_values(by=['classk', 'tmathssk']).head()

Out[8]:
     tmathssk  treadssk   classk  totexpk  ttl_score
309       320       360  regular        6        680
1470      320       315  regular        3        635
2326      339       388  regular        6        727
2820      354       398  regular        6        752
4925      354       391  regular        8        745
```

By default, all columns are sorted ascendingly. To modify that, we can include another argument, `ascending`, which will contain a list of `True`/`False` flags. Let's sort *star* by class size (*classk*) ascending and math score (*treadssk*) descending. Because we're not assigning this output back to *star*, this sorting is not permanent to the dataset.

```
In [9]: # Sort by class size ascending and math score descending
        star.sort_values(by=['classk', 'tmathssk'],
         ascending=[True, False]).head()

Out[9]:
      tmathssk  treadssk   classk  totexpk  ttl_score
724        626       474  regular       15       1100
1466       626       554  regular       11       1180
1634       626       580  regular       15       1206
2476       626       538  regular       20       1164
2495       626       522  regular        7       1148
```

To filter a DataFrame, we'll first use conditional logic to create a Series of `True`/`False` flags indicating whether each row meets some criteria. We'll then keep only the rows in the DataFrame where records in the Series are flagged as `True`. For example, let's keep only the records where `classk` is equal to `small.class`.

```
In [10]: small_class = star['classk'] == 'small.class'
         small_class.head()

Out[10]:
        0     True
        1     True
        2    False
        3    False
        4     True
        Name: classk, dtype: bool
```

We can now filter by this resulting Series by using brackets. We can confirm the number of rows and columns in our new DataFrame with the `shape` attribute:

```
In [11]: star_filtered = star[small_class]
         star_filtered.shape

Out[11]: (1733, 5)
```

`star_filtered` will contain fewer rows than *star*, but the same number of columns:

```
In [12]: star.shape

Out[12]: (5748, 5)
```

Let's try one more: we'll find the records where `treadssk` is at least `500`:

```
In [13]: star_filtered = star[star['treadssk'] >= 500]
         star_filtered.shape

Out[13]: (233, 5)
```

It's also possible to filter by multiple conditions using and/or statements. Just like in R, `&` and `|` indicate "and" and "or" in Python, respectively. Let's pass both of the previous criteria into one statement by placing each in parentheses, connecting them with `&`:

```
In [14]: # Find all records with reading score at least 500 and in small class
         star_filtered = star[(star['treadssk'] >= 500) &
                              (star['classk'] == 'small.class')]
         star_filtered.shape

Out[14]: (84, 5)
```

Aggregating and Joining Data

To group observations in a DataFrame, we'll use the `groupby()` method. If we print `star_grouped`, you'll see it's a `DataFrameGroupBy` object:

```
In [15]: star_grouped = star.groupby('classk')
         star_grouped

Out[15]: <pandas.core.groupby.generic.DataFrameGroupBy
            object at 0x000001EFD8DFF388>
```

We can now choose other fields to aggregate this grouped DataFrame by. Table 12-1 lists some common aggregation methods.

Table 12-1. Helpful aggregation functions in pandas

Method	Aggregation type
`sum()`	Sum
`count()`	Count values
`mean()`	Average
`max()`	Highest value
`min()`	Lowest value
`std()`	Standard deviation

The following gives us the average math score for each class size:

```
In [16]: star_grouped[['tmathssk']].mean()

Out[16]:
                          tmathssk
    classk
    regular             483.261000
    regular.with.aide   483.009926
    small.class         491.470283
```

Now we'll find the highest total score for each year of teacher experience. Because this would return quite a few rows, I will include the `head()` method to get just a few. This practice of adding multiple methods to the same command is called method *chaining*:

```
In [17]: star.groupby('totexpk')[['ttl_score']].max().head()

Out[17]:
                ttl_score
        totexpk
        0             1171
        1             1133
        2             1091
        3             1203
        4             1229
```

Chapter 8 reviewed the similarities and differences between Excel's VLOOKUP() and a left outer join. I'll read in a fresh copy of *star* as well as *districts*; let's use pandas to join these datasets. We'll use the merge() method to "look up" data from *school-districts* into *star*. By setting the how argument to left, we'll specify a left outer join, which again is the join type most similar to VLOOKUP():

```
In [18]: star = pd.read_excel('datasets/star/star.xlsx')
         districts = pd.read_csv('datasets/star/districts.csv')
         star.merge(districts, how='left').head()

Out[18]:
   tmathssk  treadssk            classk  totexpk   sex freelunk   race  \
0       473       447       small.class        7  girl       no  white
1       536       450       small.class       21  girl       no  black
2       463       439  regular.with.aide        0   boy      yes  black
3       559       448           regular       16   boy       no  white
4       489       447       small.class        5   boy      yes  white

   schidkn    school_name          county
0       63     Ridgeville     New Liberty
1       20  South Heights         Selmont
2       19     Bunnlevel         Sattley
3       69          Hokah      Gallipolis
4       79   Lake Mathews  Sugar Mountain
```

Python, like R, is quite intuitive about joining data: it knew by default to merge on *schidkn* and brought in both *school_name* and *county*.

Reshaping Data

Let's take a look at widening and lengthening a dataset in Python, again using pandas. To start, we can use the melt() function to combine *tmathssk* and *treadssk* into one column. To do this, I'll specify the DataFrame to manipulate with the frame argument, which variable to use as a unique identifier with id_vars, and which variables to melt into a single column with value_vars. I'll also specify what to name the resulting value and label variables with value_name and var_name, respectively:

```
In [19]: star_pivot = pd.melt(frame=star, id_vars = 'schidkn',
            value_vars=['tmathssk', 'treadssk'], value_name='score',
            var_name='test_type')
         star_pivot.head()

Out[19]:
            schidkn test_type  score
         0       63  tmathssk    473
         1       20  tmathssk    536
         2       19  tmathssk    463
         3       69  tmathssk    559
         4       79  tmathssk    489
```

How about renaming *tmathssk* and *treadssk* as *math* and *reading*, respectively? To do this, I'll use a Python dictionary to set up an object called `mapping`, which serves as something like a "lookup table" to recode the values. I'll pass this into the `map()` method which will recode *test_type*. I'll also use the `unique()` method to confirm that only *math* and *reading* are now found in *test_type*:

```
In [20]: # Rename records in `test_type`
         mapping = {'tmathssk':'math','treadssk':'reading'}
         star_pivot['test_type'] = star_pivot['test_type'].map(mapping)

         # Find unique values in test_type
         star_pivot['test_type'].unique()

Out[20]: array(['math', 'reading'], dtype=object)
```

To widen *star_pivot* back into separate *math* and *reading* columns, I'll use the `pivot_table()` method. First I'll specify what variable to index by with the `index` argument, then which variables contain the labels and values to pivot from with the `columns` and `values` arguments, respectively.

It's possible in `pandas` to set unique index columns; by default, `pivot_table()` will set whatever variables you've included in the `index` argument there. To override this, I'll use the `reset_index()` method. To learn more about custom indexing in `pandas`, along with countless other data manipulation and analysis techniques we couldn't cover here, check out *Python for Data Analysis*, 2nd edition by Wes McKinney (O'Reilly).

```
In [21]: star_pivot.pivot_table(index='schidkn',
          columns='test_type', values='score').reset_index()

Out[21]:
         test_type  schidkn        math     reading
         0                1  492.272727  443.848485
         1                2  450.576923  407.153846
         2                3  491.452632  441.000000
         3                4  467.689655  421.620690
         4                5  460.084746  427.593220
         ..             ...         ...         ...
         74              75  504.329268  440.036585
         75              76  490.260417  431.666667
         76              78  468.457627  417.983051
         77              79  490.500000  434.451613
         78              80  490.037037  442.537037

         [79 rows x 3 columns]
```

Data Visualization

Let's now briefly touch on data visualization in Python, specifically using the `seaborn` package. `seaborn` works especially well for statistical analysis and with `pandas` DataFrames, so it's a great choice here. Similarly to how `pandas` is built on top of `numpy`, `seaborn` leverages features of another popular Python plotting package, `matplotlib`.

`seaborn` includes many functions to build different plot types. We'll modify the arguments within these functions to specify which dataset to plot, which variables go along the x- and y-axes, which colors to use, and so on. Let's get started by visualizing the count of observations for each level of *classk*, which we can do with the `countplot()` function.

Our dataset is *star*, which we'll specify with the `data` argument. To place our levels of *classk* along the x-axis we'll use the `x` argument. This results in the countplot shown in Figure 12-2:

```
In [22]: sns.countplot(x='classk', data=star)
```

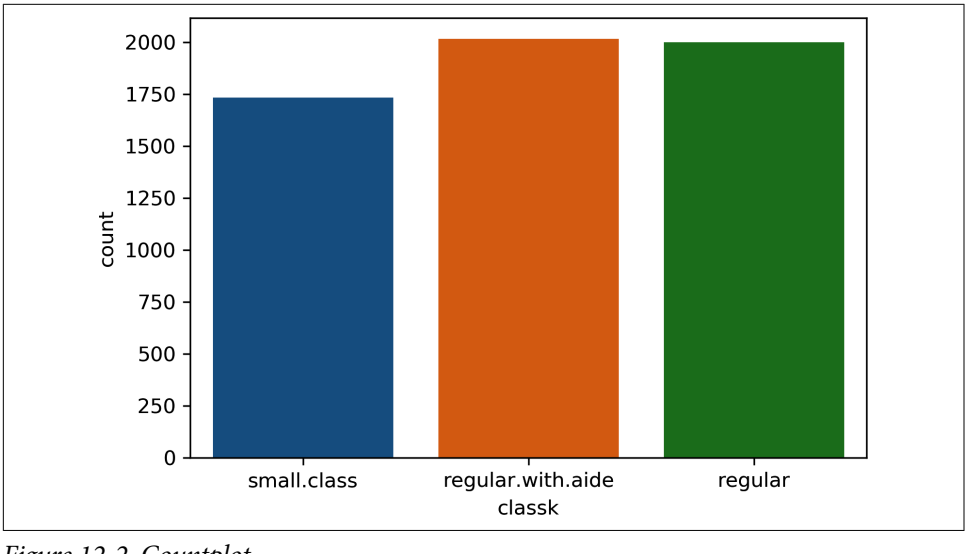


Figure 12-2. Countplot

Now for a histogram of *treadssk*, we'll use the `displot()` function. Again, we'll specify `x` and `data`. Figure 12-3 shows the result:

```
In [23]: sns.displot(x='treadssk', data=star)
```

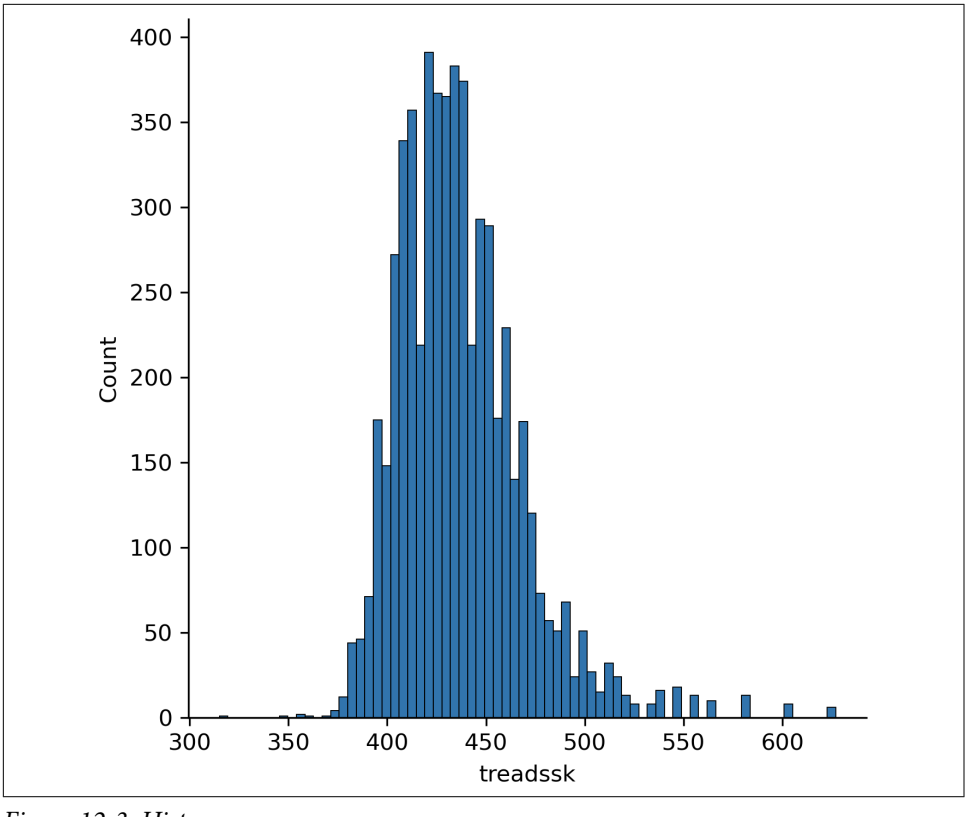


Figure 12-3. Histogram

`seaborn` functions include many optional arguments to customize a plot's appearance. For example, let's change the number of bins to 25 and the plot color to pink. This results in Figure 12-4:

```
In [24]: sns.displot(x='treadssk', data=star, bins=25, color='pink')
```

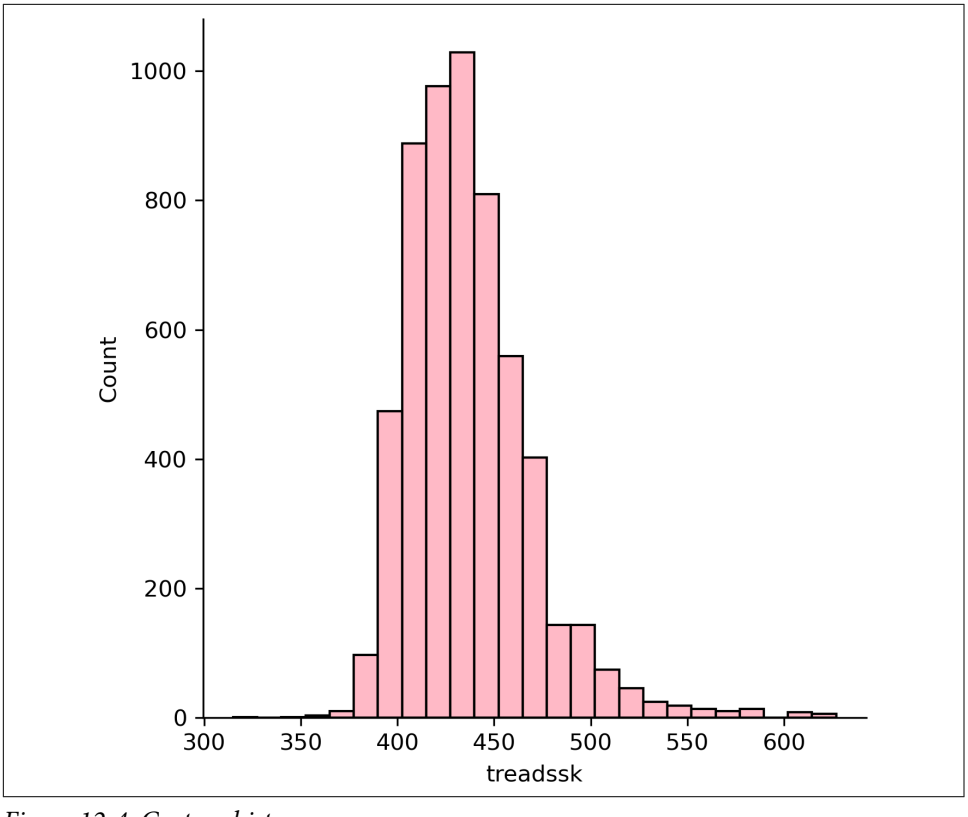


Figure 12-4. Custom histogram

To make a boxplot, use the `boxplot()` function as in Figure 12-5:

```
In [25]: sns.boxplot(x='treadssk', data=star)
```

In any of these cases so far, we could've "flipped" the plot by instead including the variable of interest in the `y` argument. Let's try it with our boxplot, and we'll get what's shown in Figure 12-6 as output:

```
In [26]: sns.boxplot(y='treadssk', data=star)
```

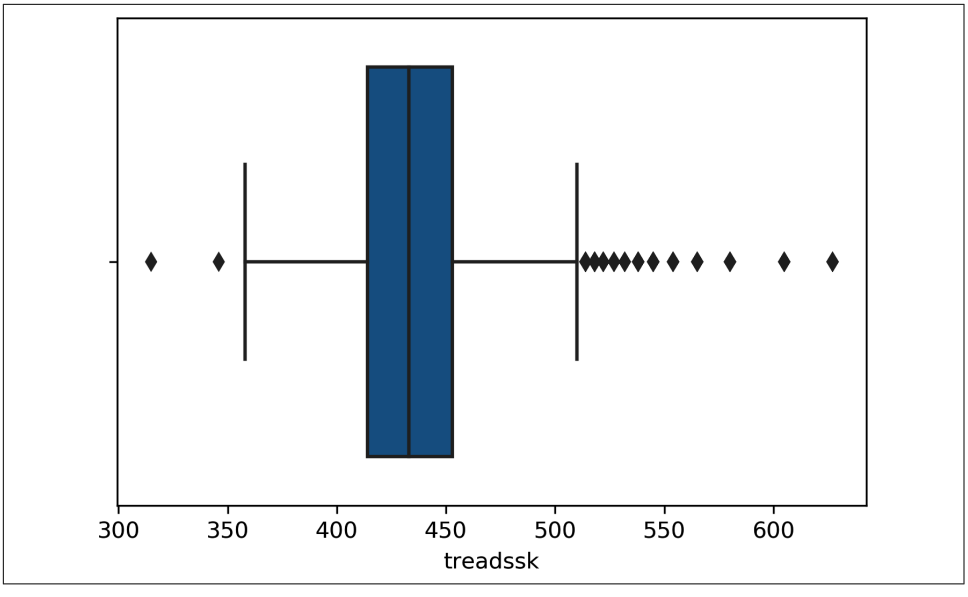


Figure 12-5. Boxplot

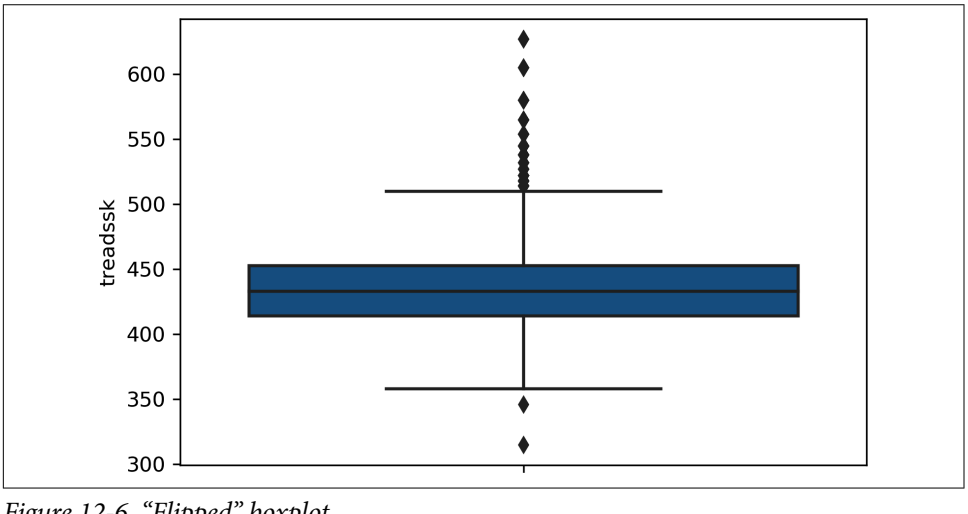


Figure 12-6. "Flipped" boxplot

To make a boxplot for each level of class size, we'll include an additional argument to plot *classk* along the x-axis, giving us the boxplot by group depicted in Figure 12-7:

```
In [27]: sns.boxplot(x='classk', y='treadssk', data=star)
```

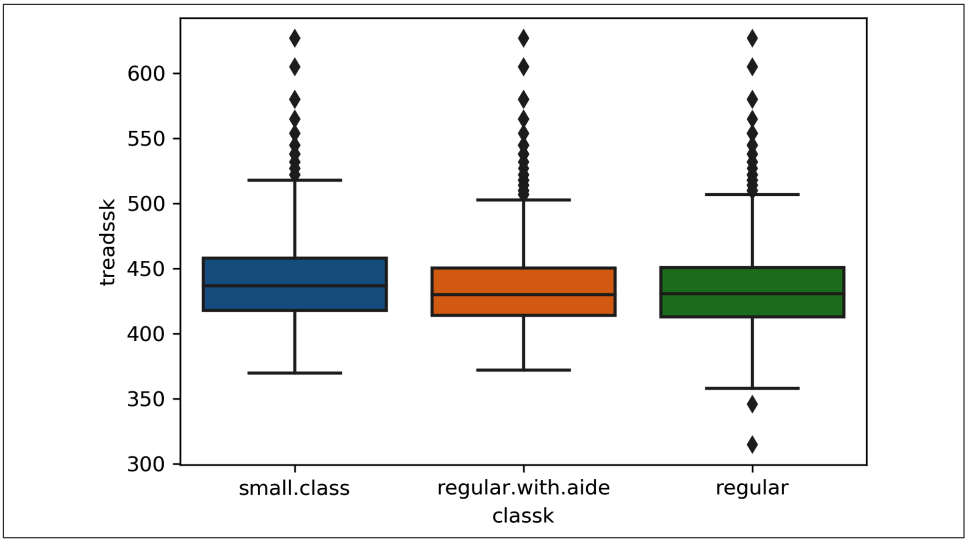


Figure 12-7. Boxplot by group

Now let's use the `scatterplot()` function to plot the relationship of *tmathssk* on the x-axis and *treadssk* on the y. Figure 12-8 is the result:

```
In [28]: sns.scatterplot(x='tmathssk', y='treadssk', data=star)
```

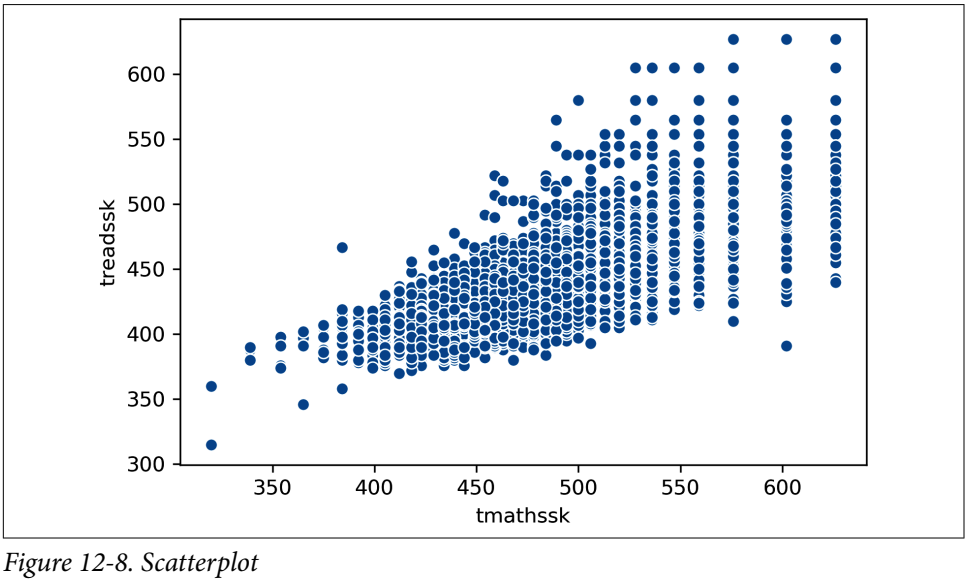


Figure 12-8. Scatterplot

Let's say we wanted to share this plot with an outside audience, who may not be familiar with what *treadssk* and *tmathssk* are. We can add more helpful labels to this chart by borrowing features from `matplotlib.pyplot`. We'll run the same `scatterplot()` function as before, but this time we'll also call in functions from `pyplot` to add custom x- and y-axis labels, as well as a chart title. This results in Figure 12-9:

```
In [29]: sns.scatterplot(x='tmathssk', y='treadssk', data=star)
         plt.xlabel('Math score')
         plt.ylabel('Reading score')
         plt.title('Math score versus reading score')
```

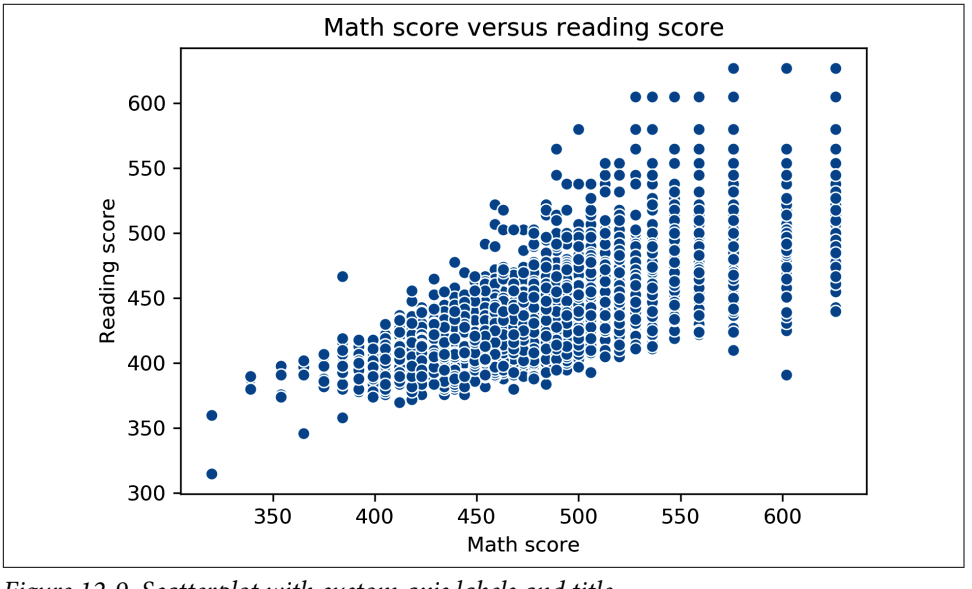


Figure 12-9. Scatterplot with custom axis labels and title

`seaborn` includes many more features for building visually appealing data visualizations. To learn more, check out the official documentation.

Conclusion

There's so much more that `pandas` and `seaborn` can do, but this is enough to get you started with the true task at hand: to explore and test relationships in data. That will be the focus of Chapter 13.

Exercises

The book repository has two files in the *census* subfolder of *datasets*, *census.csv* and *census-divisions.csv*. Read these into Python and do the following:

1. Sort the data by region ascending, division ascending and population descending. (You will need to combine datasets to do this.) Write the results to an Excel worksheet.
2. Drop the postal code field from your merged dataset.
3. Create a new column, *density*, that is a calculation of population divided by land area.
4. Visualize the relationship between land area and population for all observations in 2015.
5. Find the total population for each region in 2015.
6. Create a table containing state names and populations, with the population for each year 2010–2015 kept in an individual column.

CHAPTER 13
Capstone: Python for Data Analytics

At the end of Chapter 8 you extended what you learned about R to explore and test relationships in the *mpg* dataset. We'll do the same in this chapter, using Python. We've conducted the same work in Excel and R, so I'll focus less on the whys of our analysis in favor of the hows of doing it in Python.

To get started, let's call in all the necessary modules. Some of these are new: from `scipy`, we'll import the `stats` submodule. To do this, we'll use the `from` keyword to tell Python what module to look for, then the usual `import` keyword to choose a submodule. As the name suggests, we'll use the `stats` submodule of `scipy` to conduct our statistical analysis. We'll also be using a new package called `sklearn`, or *scikit-learn*, to validate our model on a train/test split. This package has become a dominant resource for machine learning and also comes installed with Anaconda.

```
In [1]: import pandas as pd
        import seaborn as sns
        import matplotlib.pyplot as plt
        from scipy import stats
        from sklearn import linear_model
        from sklearn import model_selection
        from sklearn import metrics
```

With the `usecols` argument of `read_csv()` we can specify which columns to read into the DataFrame:

```
In [2]: mpg = pd.read_csv('datasets/mpg/mpg.csv',usecols=
           ['mpg','weight','horsepower','origin','cylinders'])
        mpg.head()

Out[2]:
     mpg  cylinders  horsepower  weight origin
 0  18.0          8         130    3504    USA
 1  15.0          8         165    3693    USA
```

```
2  18.0          8          150    3436    USA
3  16.0          8          150    3433    USA
4  17.0          8          140    3449    USA
```

Exploratory Data Analysis

Let's start with the descriptive statistics:

```
In[3]: mpg.describe()

Out[3]:
              mpg   cylinders  horsepower       weight
count  392.000000  392.000000  392.000000   392.000000
mean    23.445918    5.471939  104.469388  2977.584184
std      7.805007    1.705783   38.491160   849.402560
min      9.000000    3.000000   46.000000  1613.000000
25%     17.000000    4.000000   75.000000  2225.250000
50%     22.750000    4.000000   93.500000  2803.500000
75%     29.000000    8.000000  126.000000  3614.750000
max     46.600000    8.000000  230.000000  5140.000000
```

Because *origin* is a categorical variable, by default it doesn't show up as part of `describe()`. Let's explore this variable instead with a frequency table. This can be done in `pandas` with the `crosstab()` function. First, we'll specify what data to place on the index: *origin*. We'll get a count for each level by setting the `columns` argument to `count`:

```
In [4]: pd.crosstab(index=mpg['origin'], columns='count')

Out[4]:
col_0   count
origin
Asia       79
Europe     68
USA       245
```

To make a two-way frequency table, we can instead set `columns` to another categorical variable, such as `cylinders`:

```
In [5]: pd.crosstab(index=mpg['origin'], columns=mpg['cylinders'])

Out[5]:
cylinders  3   4  5   6    8
origin
Asia       4  69  0   6    0
Europe     0  61  3   4    0
USA        0  69  0  73  103
```

Next, let's retrieve descriptive statistics for *mpg* by each level of *origin*. I'll do this by chaining together two methods, then subsetting the results:

```
In[6]: mpg.groupby('origin').describe()['mpg']

Out[6]:
        count       mean       std   min    25%   50%     75%   max
origin
Asia     79.0  30.450633  6.090048  18.0  25.70  31.6  34.050  46.6
Europe   68.0  27.602941  6.580182  16.2  23.75  26.0  30.125  44.3
USA     245.0  20.033469  6.440384   9.0  15.00  18.5  24.000  39.0
```

We can also visualize the overall distribution of *mpg*, as in Figure 13-1:

```
In[7]: sns.displot(data=mpg, x='mpg')
```

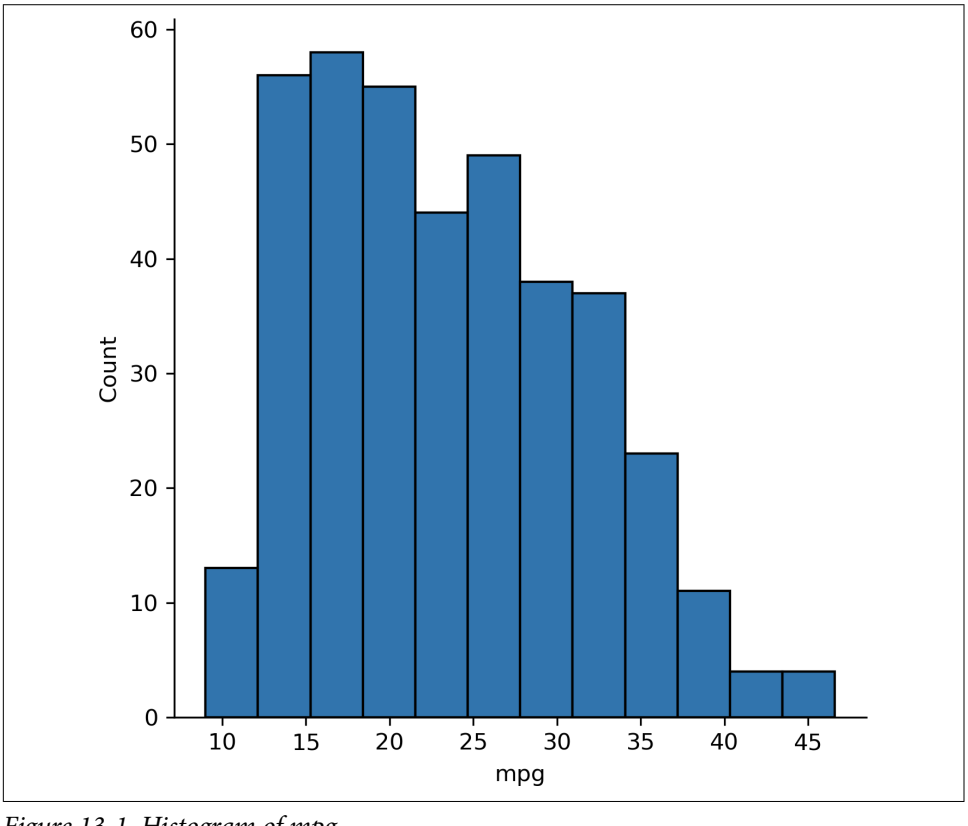


Figure 13-1. Histogram of mpg

Now let's make a boxplot as in Figure 13-2 comparing the distribution of *mpg* across each level of *origin*:

```
In[8]: sns.boxplot(x='origin', y='mpg', data=mpg, color='pink')
```

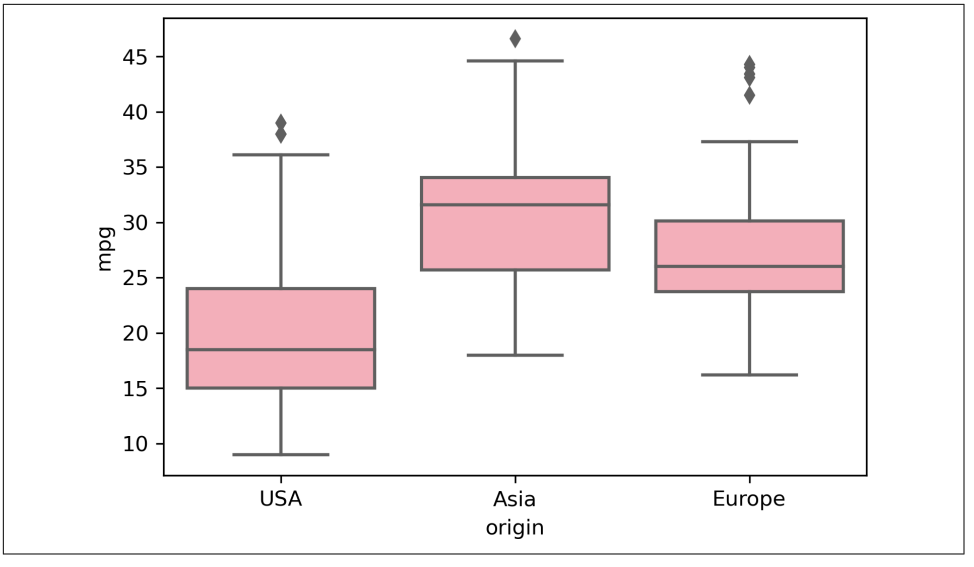


Figure 13-2. Boxplot of mpg by origin

Alternatively, we can set the `col` argument of `displot()` to `origin` to create faceted histograms, such as in Figure 13-3:

```
In[9]: sns.displot(data=mpg, x="mpg", col="origin")
```

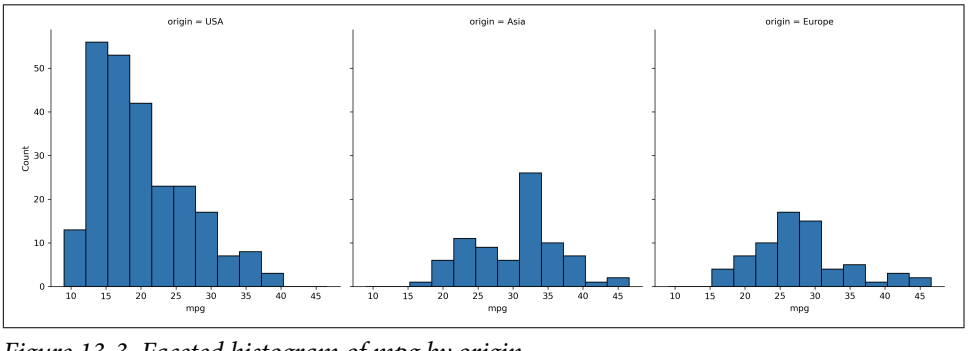


Figure 13-3. Faceted histogram of mpg by origin

Hypothesis Testing

Let's again test for a difference in mileage between American and European cars. For ease of analysis, we'll split the observations in each group into their own DataFrames.

```
In[10]: usa_cars = mpg[mpg['origin']=='USA']
        europe_cars = mpg[mpg['origin']=='Europe']
```

Independent Samples T-test

We can now use the `ttest_ind()` function from `scipy.stats` to conduct the t-test. This function expects two `numpy` arrays as arguments; `pandas` Series also work:

```
In[11]: stats.ttest_ind(usa_cars['mpg'], europe_cars['mpg'])

Out[11]: Ttest_indResult(statistic=-8.534455914399228,
             pvalue=6.306531719750568e-16)
```

Unfortunately, the output here is rather scarce: while it does include the p-value, it doesn't include the confidence interval. To run a t-test with more output, check out the `researchpy` module.

Let's move on to analyzing our continuous variables. We'll start with a correlation matrix. We can use the `corr()` method from `pandas`, including only the relevant variables:

```
In[12]: mpg[['mpg','horsepower','weight']].corr()

Out[12]:
                   mpg  horsepower    weight
mpg           1.000000   -0.778427 -0.832244
horsepower   -0.778427    1.000000  0.864538
weight       -0.832244    0.864538  1.000000
```

Next, let's visualize the relationship between *weight* and *mpg* with a scatterplot as shown in Figure 13-4:

```
In[13]: sns.scatterplot(x='weight', y='mpg', data=mpg)
        plt.title('Relationship between weight and mileage')
```

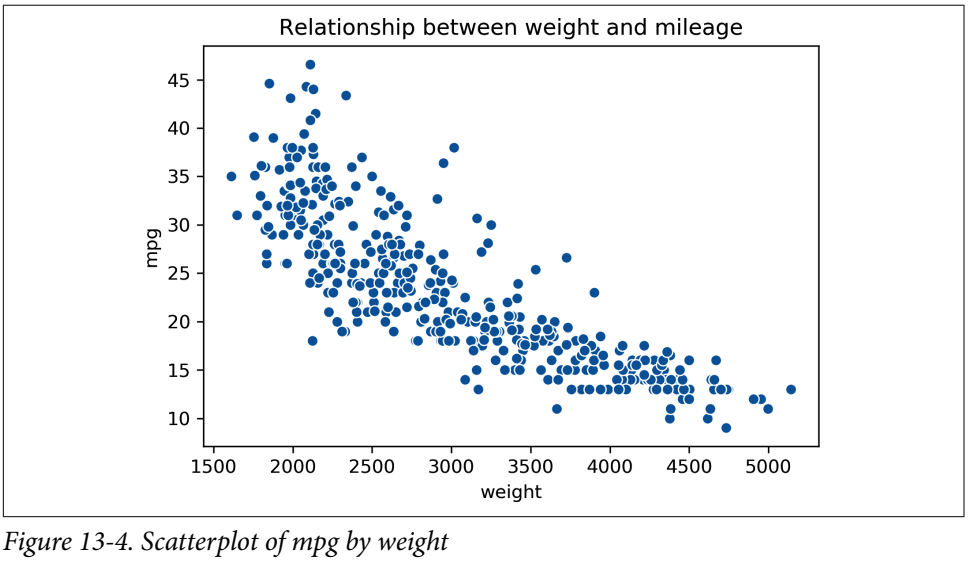


Figure 13-4. Scatterplot of mpg by weight

Alternatively, we could produce scatterplots across all pairs of our dataset with the `pairplot()` function from `seaborn`. Histograms of each variable are included along the diagonal, as seen in Figure 13-5:

```
In[14]: sns.pairplot(mpg[['mpg','horsepower','weight']])
```

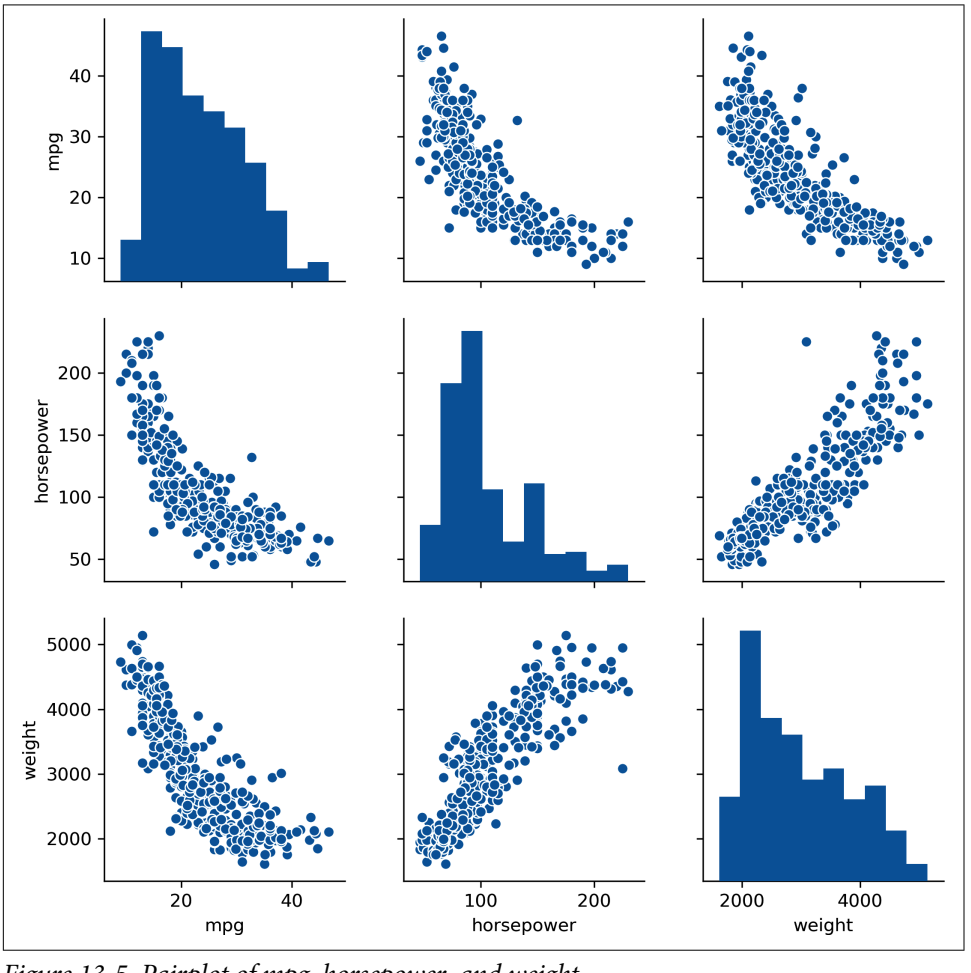


Figure 13-5. Pairplot of mpg, horsepower, and weight

Linear Regression

Now it's time for a linear regression. To do this, we'll use `linregress()` from `scipy`, which also looks for two `numpy` arrays or `pandas` Series. We'll specify which variable is our independent and dependent variable with the `x` and `y` arguments, respectively:

```
In[15]: # Linear regression of weight on mpg
        stats.linregress(x=mpg['weight'], y=mpg['mpg'])
```

```
Out[15]: LinregressResult(slope=-0.007647342535779578,
   intercept=46.21652454901758, rvalue=-0.8322442148315754,
   pvalue=6.015296051435726e-102, stderr=0.0002579632782734318)
```

Again, you'll see that some of the output you may be used to is missing here. *Be careful:* the `rvalue` included is the *correlation coefficient*, not R-square. For a richer linear regression output, check out the `statsmodels` module.

Last but not least, let's overlay our regression line to a scatterplot. `seaborn` has a separate function to do just that: `regplot()`. As usual, we'll specify our independent and dependent variables, and where to get the data. This results in Figure 13-6:

```
In[16]: # Fit regression line to scatterplot
        sns.regplot(x="weight", y="mpg", data=mpg)
        plt.xlabel('Weight (lbs)')
        plt.ylabel('Mileage (mpg)')
        plt.title('Relationship between weight and mileage')
```

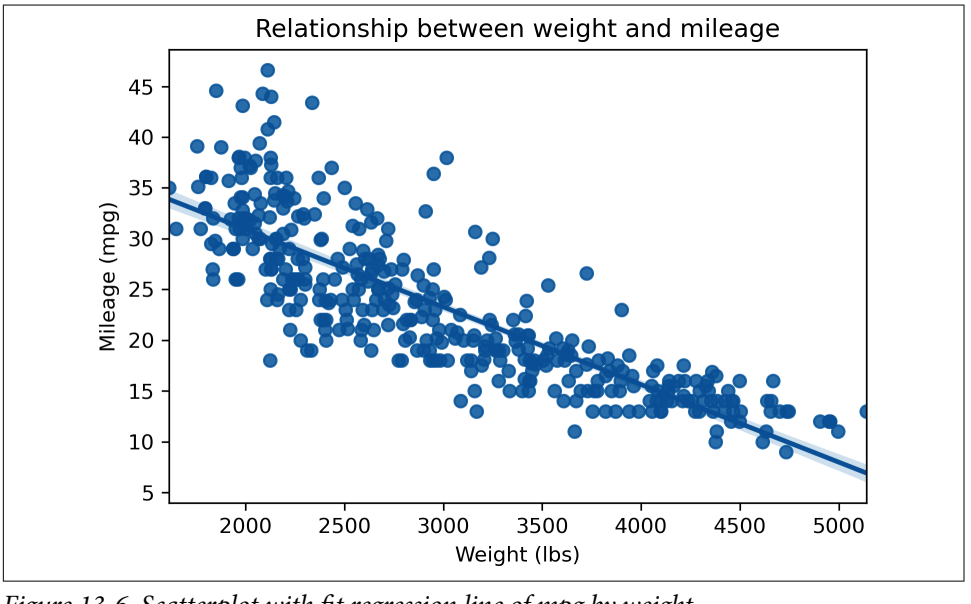


Figure 13-6. Scatterplot with fit regression line of mpg by weight

Train/Test Split and Validation

At the end of Chapter 9 you learned how to apply a train/test split when building a linear regression model in R.

We will use the `train_test_split()` function to split our dataset into *four* DataFrames: not just by training and testing but also independent and dependent variables. We'll pass in a DataFrame containing our independent variable first, then one

containing the dependent variable. Using the `random_state` argument, we'll seed the random number generator so the results remain consistent for this example:

```
In[17]: X_train, X_test, y_train, y_test =
        model_selection.train_test_split(mpg[['weight']], mpg[['mpg']],
        random_state=1234)
```

By default, the data is split 75/25 between training and testing subsets:

```
In[18]:  y_train.shape

Out[18]: (294, 1)


In[19]:  y_test.shape

Out[19]: (98, 1)
```

Now, let's fit the model to the training data. First we'll specify the linear model with `LinearRegression()`, then we'll train the model with `regr.fit()`. To get the predicted values for the test dataset, we can use `predict()`. This results in a `numpy` array, not a `pandas` DataFrame, so the `head()` method won't work to print the first few rows. We can, however, slice it:

```
In[20]:  # Create linear regression object
         regr = linear_model.LinearRegression()

         # Train the model using the training sets
         regr.fit(X_train, y_train)

         # Make predictions using the testing set
         y_pred = regr.predict(X_test)

         # Print first five observations
         y_pred[:5]

Out[20]:  array([[14.86634263],
         [23.48793632],
         [26.2781699 ],
         [27.69989655],
         [29.05319785]])
```

The `coef_` attribute returns the coefficient of our test model:

```
In[21]:  regr.coef_

Out[21]: array([[-0.00760282]])
```

To get more information about the model, such as the coefficient p-values or R-squared, try fitting it with the `statsmodels` package.

For now, we'll evaluate the performance of the model on our test data, this time using the `metrics` submodule of `sklearn`. We'll pass in our actual and predicted values to

the `r2_score()` and `mean_squared_error()` functions, which will return the R-squared and RMSE, respectively.

```
In[22]:  metrics.r2_score(y_test, y_pred)

Out[22]: 0.6811923996681357


In[23]:  metrics.mean_squared_error(y_test, y_pred)

Out[23]: 21.63348076436662
```

Conclusion

The usual caveat applies to this chapter: we've just scratched the surface of what analysis is possible on this or any other dataset. But I hope you feel you've hit your stride on working with data in Python.

Exercises

Take another look at the *ais* dataset, this time using Python. Read the Excel workbook in from the book repository and complete the following. You should be pretty comfortable with this analysis by now.

1. Visualize the distribution of red blood cell count (*rcc*) by sex (*sex*).
2. Is there a significant difference in red blood cell count between the two groups of sex?
3. Produce a correlation matrix of the relevant variables in this dataset.
4. Visualize the relationship of height (*ht*) and weight (*wt*).
5. Regress *ht* on *wt*. Find the equation of the fit regression line. Is there a significant relationship?
6. Split your regression model into training and testing subsets. What is the R-squared and RMSE on your test model?