1.4 Transforming Variables

May 8, 2019

1 1.4 Transforming Variables

In this section, we will transform and combine existing variables to obtain new variables. Our examples are drawn from a data set of house prices in Ames, Iowa. This data set is stored in a tab-separated values file. For more information about the variables in this data set, please refer to the data documentation.

```
In [1]: %matplotlib inline
        import pandas as pd
        pd.options.display.max_rows = 10

        df = pd.read_csv(
            "https://raw.githubusercontent.com/dlsun/data-science-book/master/data/AmesHousing
            sep="\t")
        df.head()
```

```
Out[1]:    Order        PID  MS SubClass MS Zoning  Lot Frontage  Lot Area Street  \
        0      1  526301100           20        RL         141.0     31770   Pave
        1      2  526350040           20        RH          80.0     11622   Pave
        2      3  526351010           20        RL          81.0     14267   Pave
        3      4  526353030           20        RL          93.0     11160   Pave
        4      5  527105010           60        RL          74.0     13830   Pave

          Alley Lot Shape Land Contour    ...      Pool Area Pool QC  Fence  \
        0   NaN       IR1          Lvl    ...              0     NaN    NaN
        1   NaN       Reg          Lvl    ...              0     NaN  MnPrv
        2   NaN       IR1          Lvl    ...              0     NaN    NaN
        3   NaN       Reg          Lvl    ...              0     NaN    NaN
        4   NaN       IR1          Lvl    ...              0     NaN  MnPrv

          Misc Feature Misc Val Mo Sold Yr Sold Sale Type  Sale Condition  SalePrice
        0          NaN        0       5    2010       WD           Normal     215000
        1          NaN        0       6    2010       WD           Normal     105000
        2         Gar2    12500       6    2010       WD           Normal     172000
        3          NaN        0       4    2010       WD           Normal     244000
        4          NaN        0       3    2010       WD           Normal     189900

        [5 rows x 82 columns]
```

1.1 Applying Transformations

1.1.1 Quantitative Variables

There are several reasons to transform quantitative variables, including:

1. to change the measurement units
2. to make the variable more amenable to analysis

As an example of the first reason, suppose we want the lot areas to be in acres instead of square feet. Since there are 43560 square feet in an acre, this requires dividing each lot area by 43560. We can **broadcast** the division over the entire `Series`.

```
In [2]: df["Lot Area"] / 43560

Out[2]: 0       0.729339
        1       0.266804
        2       0.327525
        3       0.256198
        4       0.317493
                  ...
        2925    0.182208
        2926    0.203972
        2927    0.239692
        2928    0.229798
        2929    0.221006
        Name: Lot Area, Length: 2930, dtype: float64
```

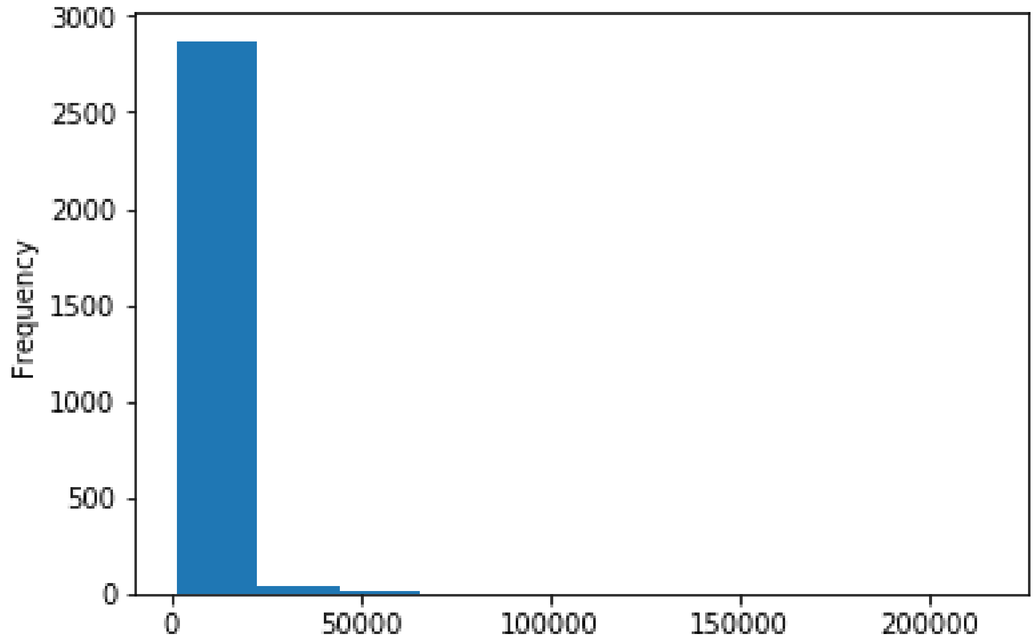
If we want to store the results as a new variable in the `DataFrame`, we simply assign the `Series` to a new column in the `DataFrame`. The command below does two things: creates a new column in the `DataFrame` called "Lot Area (acres)" *and* populates it with the values from the `Series` above.

```
In [3]: df["Lot Area (acres)"] = df["Lot Area"] / 43560
```

The second reason for transforming quantitative variables is to make them more amenable to analysis. To see why a variable might not be amenable to analysis, let's take a look at a histogram of lot areas.

```
In [4]: df["Lot Area"].plot.hist()

Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x7f587180be48>
```



There are a few homes with such extreme lot areas that we get virtually no resolution at the lower end of the distribution. Over 95% of the observations are in a single bin of this histogram. In other words, this variable is extremely **skewed**.

One way to improve this histogram is to use more bins. But this does not solve the fundamental problem: we need more resolution at the lower end of the scale and less resolution at the higher end. One way to spread out the values at the lower end of a distribution and to compress the values at the higher end is to take the logarithm (provided that the values are all positive). Log transformations are particularly effective at dealing with right-skewed data.

The log function is not built into Python or `pandas`. We have to import the log function from a library called `numpy`, which contains many functions and data structures for numerical computations. In fact, `pandas` is built on top of `numpy`. When we apply `numpy`'s `log` function to a `pandas` `Series`, the function is automatically broadcast over the elements of the `Series`, returning another `Series`. Let's save the results to a variable called "log(Lot Area)".

```
In [5]: import numpy as np
        df["log(Lot Area)"] = np.log(df["Lot Area"])
        df["log(Lot Area)"]
```

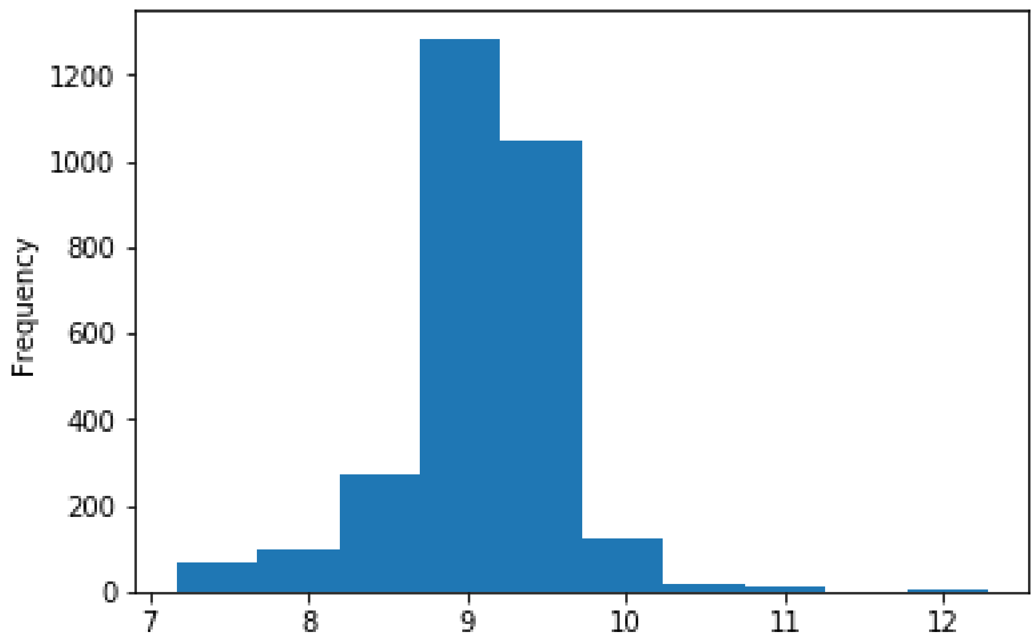
```
Out[5]: 0       10.366278
        1        9.360655
        2        9.565704
        3        9.320091
        4        9.534595
                   ...
        2925     8.979291
        2926     9.092120
        2927     9.253496
```

```
2928    9.211340
2929    9.172327
Name: log(Lot Area), Length: 2930, dtype: float64
```

These numbers are not very interpretable on their own, but if we make a histogram of these values, we see that the lower end of the distribution is now more spread out, and the extreme values are not so extreme anymore.

```
In [6]: df["log(Lot Area)"].plot.hist()
```

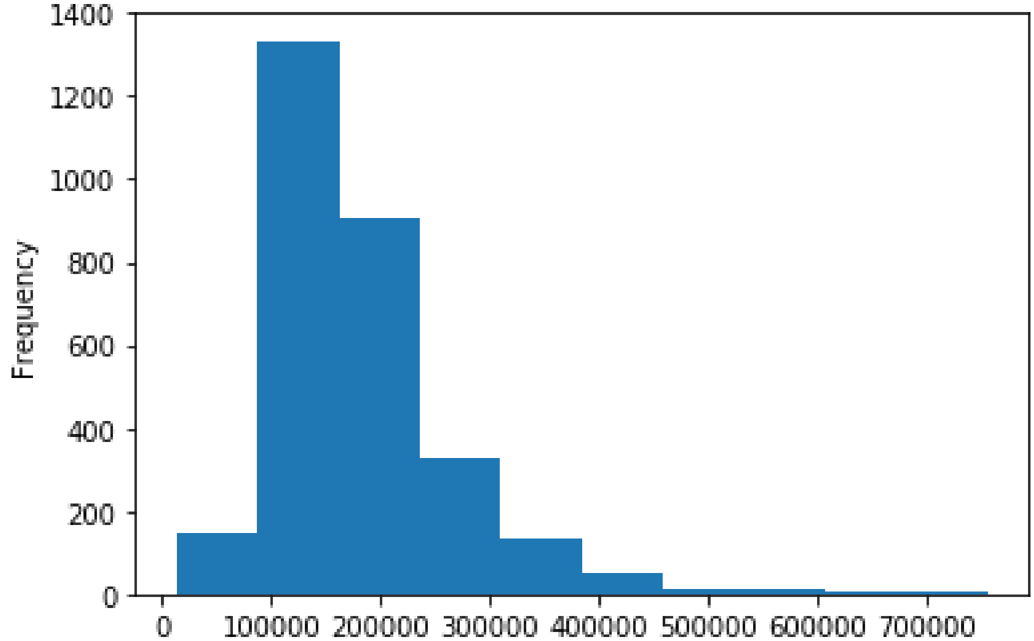
```
Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x7f586b768dd8>
```



It is possible for a log transformation to overcorrect for skew. For example, the "SalePrice" variable is also right-skewed.

```
In [7]: df["SalePrice"].plot.hist()
```

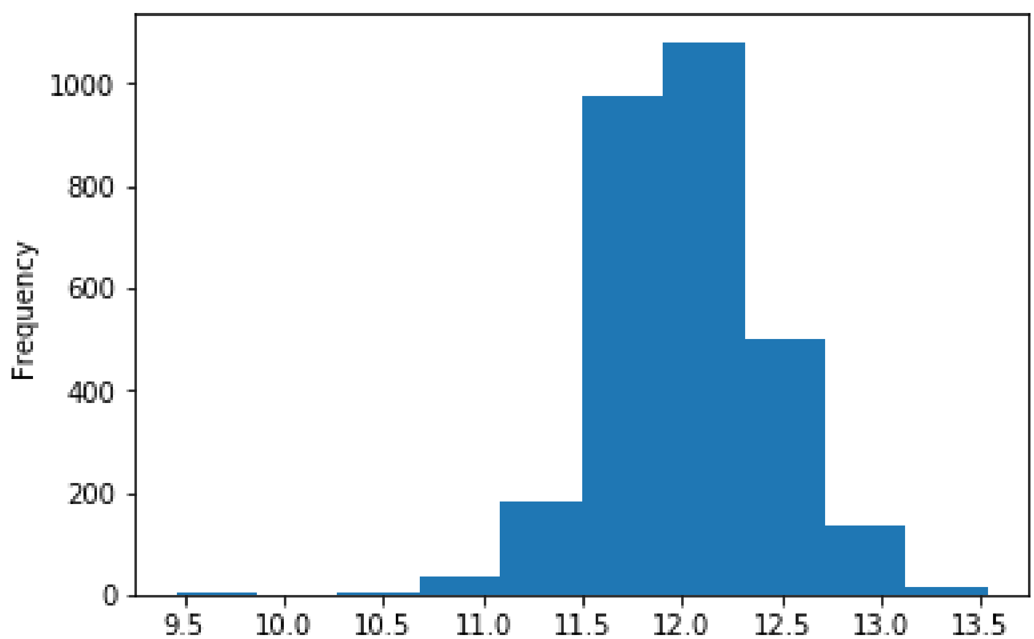
```
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x7f586b6c08d0>
```



But if we take logs, the distribution becomes somewhat left-skewed.

```
In [8]: np.log(df["SalePrice"]).plot.hist()

Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x7f586b69e278>
```



Is there a transformation that makes the resulting distribution more symmetric?

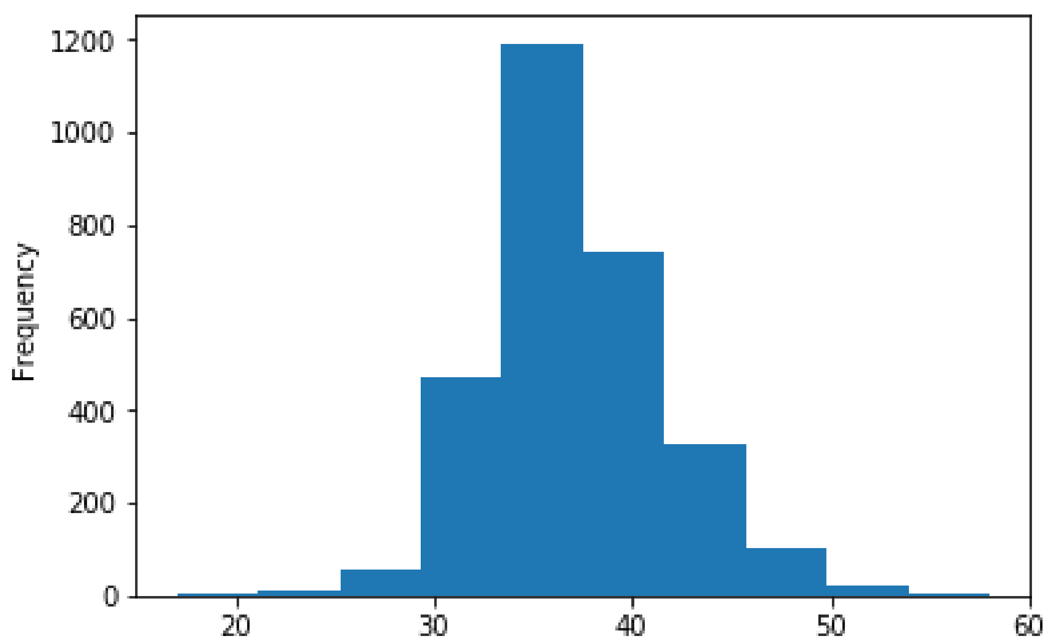
In fact, log is just one transformation in a whole family of transformations. Because the transformations in this family involve raising the values to some power, the statistician John Tukey called this the **ladder of powers**:

$$x(\lambda) = \begin{cases} x^\lambda & \lambda > 0 \\ \log(x) & \lambda = 0 \\ -x^\lambda & \lambda < 0 \end{cases}$$

$\lambda = 1$ corresponds to no transformation at all. As we decrease λ, the distribution becomes more left-skewed (which is useful if the original distribution was right-skewed). Since log ($\lambda = 0$) was an overcorrection, let's back off and increase λ:

```
In [9]: (df["SalePrice"] ** .3).plot.hist()
```

```
Out[9]: <matplotlib.axes._subplots.AxesSubplot at 0x7f586b794a90>
```



This seems to be better. We can move λ up and down the ladder until the distribution is the shape we want.

1.1.2 Why $\lambda = 0$ corresponds to log (Optional)

You might have noticed that it does not make sense to use the transformation x^0 for $\lambda = 0$, since anything raised to the zero power equals 1. But why is $\log(x)$ the right function to replace x^0?

The answer has to do with calculus. We want to understand the behavior of x^λ as λ approaches 0. To do this properly, we actually need to consider the function

$$\frac{x^\lambda - 1}{\lambda}.$$

Subtracting 1 and dividing by λ are just constants that shift and scale the distribution; they do not affect the overall shape of the distribution. Therefore, the histogram of x^λ will look the same as the histogram of $(x^\lambda - 1)/\lambda$; only the axes will be different.

Using calculus, you can show that the limit of the above function as λ approaches 0 is:

$$\lim_{\lambda \to 0} \frac{x^\lambda - 1}{\lambda} = \log(x).$$

(Challenge: prove this!) This is why it makes sense to slot $\log(x)$ in for x^0.

1.1.3 Other Mathematical Functions in Numpy (Optional)

You might wonder what other mathematical functions are available in `numpy` besides `log`. For one, there is `log10`, which implements the base-10 logarithm. (By default, `np.log` is the natural logarithm, base-e.)

Here is an exhaustive list of the mathematical functions. All of these functions are compatible with `pandas`.

1.1.4 Categorical Variables

Categorical variables sometimes also require transformation, although for different reasons than quantitative variables. With categorical variables, the values are usually labels, so it does not make sense to take logarithms or to raise them to powers. However, we might want to change the labels of the categories.

For example, according to the data documentation, the categorical variable "Heating QC" (heating quality and condition) in the Ames data set has five categories: excellent, good, average/typical, fair, and poor.

```
In [10]: df["Heating QC"]

Out[10]: 0       Fa
         1       TA
         2       TA
         3       Ex
         4       Gd
                 ..
         2925    TA
         2926    TA
         2927    TA
         2928    Gd
         2929    Ex
         Name: Heating QC, Length: 2930, dtype: object
```

The categories are currently labeled as "Ex", "Gd", "TA", "Fa", and "Po", which might be cryptic to a reader. We might want to replace them with more descriptive labels. This requires a transformation.

To do this, we can use the `.map()` method of `Series`. This method takes as input a dictionary that specifies the mapping between the current labels and the desired labels. So, for example, if we want all instances of "Ex" to be replaced by "Excellent", we would add the key "Ex" to this dictionary, with a value of "Excellent".

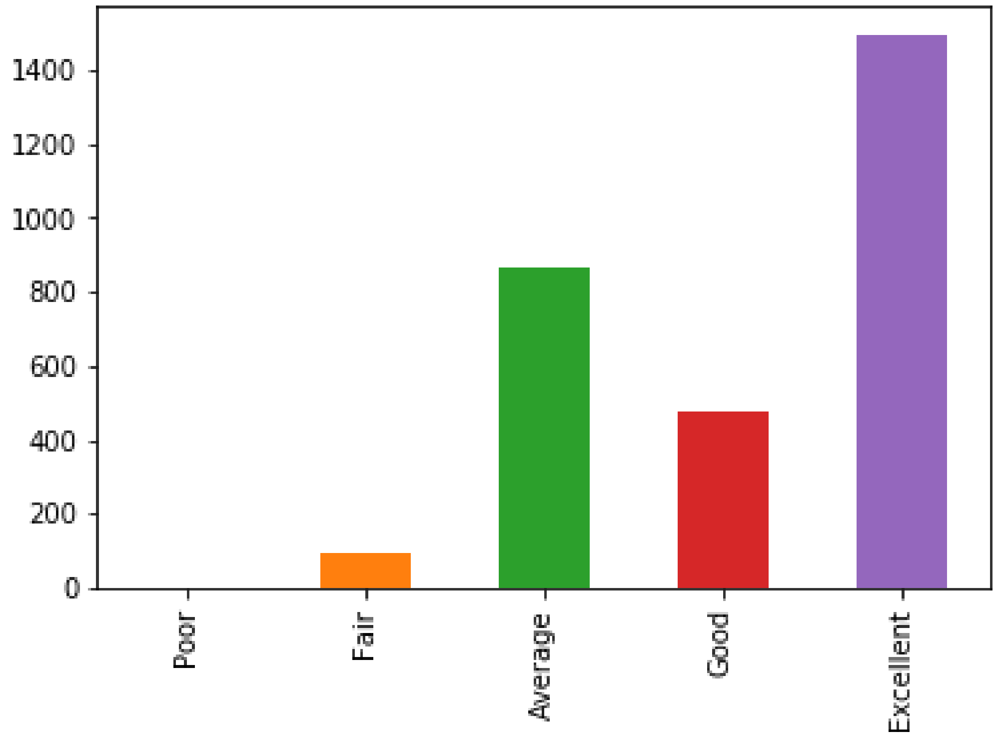
```
In [11]: df["Heating QC"].map({
             "Ex": "Excellent",
             "Gd": "Good",
             "TA": "Average",
             "Fa": "Fair",
             "Po": "Poor"
         })

Out[11]: 0            Fair
         1         Average
         2         Average
         3       Excellent
         4            Good
                    ...
         2925      Average
         2926      Average
         2927      Average
         2928         Good
         2929    Excellent
         Name: Heating QC, Length: 2930, dtype: object
```

Now when we make a bar chart, the labels will come out correctly. We just have to make sure they come out in the order we want.

```
In [12]: df["Heating QC"].map({
                 "Ex": "Excellent",
                 "Gd": "Good",
                 "TA": "Average",
                 "Fa": "Fair",
                 "Po": "Poor"
         }).value_counts()[["Poor", "Fair", "Average", "Good", "Excellent"]].plot.bar()

Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x7f586b5acf60>
```



Transformations of categorical variables are not always merely cosmetic. For example, we may want to combine several categories into one. The code below turns heating quality into a binary categorical variable (acceptable / unacceptable), according to whether the heating quality is at least average:

```
In [13]: df["Heating QC Binary"] = df["Heating QC"].map({
                "Ex": "Acceptable",
                "Gd": "Acceptable",
                "TA": "Acceptable",
                "Fa": "Unacceptable",
                "Po": "Unacceptable"
         })

         df["Heating QC Binary"]

Out[13]: 0         Unacceptable
         1           Acceptable
         2           Acceptable
         3           Acceptable
         4           Acceptable
                       ...
         2925        Acceptable
         2926        Acceptable
         2927        Acceptable
```

```
2928      Acceptable
2929      Acceptable
Name: Heating QC Binary, Length: 2930, dtype: object
```

The binary variable contains less information than the original variable, but we may not need the finer-grained detail about the heating, if all we want to know is whether it is acceptable or not.

```
In [14]: df["Heating QC Binary"].value_counts()
```

```
Out[14]: Acceptable      2835
         Unacceptable      95
         Name: Heating QC Binary, dtype: int64
```

1.2 Combining Variables

We can also create new variables out of multiple existing variables. For example, in the current data set, the information about when a house was sold is spread across two variables, "Yr Sold" and "Mo Sold" (1-12 indicating the month). We can combine these two variables into one, by dividing the month the house was sold by 12 and then adding that to the year. So for example, this new variable would equal 2010.5 if the house was sold in June 2010 and 2006.75 if it was sold in September 2006.

```
In [15]: df["Date Sold"] = df["Yr Sold"] + df["Mo Sold"] / 12
         df["Date Sold"]
```

```
Out[15]: 0       2010.416667
         1       2010.500000
         2       2010.500000
         3       2010.333333
         4       2010.250000
                    ...
         2925    2006.250000
         2926    2006.500000
         2927    2006.583333
         2928    2006.333333
         2929    2006.916667
         Name: Date Sold, Length: 2930, dtype: float64
```

Notice how the division by 12 is *broadcast* over the elements of the `Series`, and the addition of the two `Series` is elementwise.

Another example of a variable that can be derived from two existing variables is the *cost per square foot*, which is a common way to compare prices of different-sized homes. To calculate the cost per square foot of a home, we can simply divide the two `Series`, and the division will be elementwise.

```
In [16]: df["Cost per Sq Ft"] = df["SalePrice"] / df["Gr Liv Area"]
         df["Cost per Sq Ft"]
```

```
Out[16]: 0       129.830918
         1       117.187500
         2       129.420617
         3       115.639810
         4       116.574586
                    ...
         2925    142.073779
         2926    145.232816
         2927    136.082474
         2928    122.390209
         2929     94.000000
         Name: Cost per Sq Ft, Length: 2930, dtype: float64
```

2 Exercises

Exercise 1. What happens if you leave out a category in the dictionary that you pass to `.map()`?

```
In [17]: #If we leave out a category then it will be mapped to NaN
         df["Lot Shape"].map({
             "Reg":0,
             "IR2":2,
             "IR3":3
         }).value_counts()
```

```
Out[17]: 0.0    1859
         2.0      76
         3.0      16
         Name: Lot Shape, dtype: int64
```

```
In [24]: df["Lot Shape"]
```

```
Out[24]: 0       IR1
         1       Reg
         2       IR1
         3       Reg
         4       IR1
                ...
         2925    IR1
         2926    IR1
         2927    Reg
         2928    Reg
         2929    Reg
         Name: Lot Shape, Length: 2930, dtype: object
```

Exercises 2-4 deal with the Ames housing data set from earlier. Refer to the data documentation if you have any trouble finding or understanding a variable in this data set.

Exercise 2. The number of bathrooms is typically reported as a decimal to allow for half bathrooms (i.e., bathrooms without a shower). In this data set, the number of full bathrooms and the number of half bathrooms are separate variables. Create a new variable with the number of bathrooms in each home.

```
In [26]: df["Full Bath"], df["Half Bath"]
         df["Total Bathrooms"] = df["Half Bath"]*0.5 + df["Full Bath"] + df["Bsmt Half Bath"]*(
         df["Total Bathrooms"].head()

Out[26]: 0    2.0
         1    1.0
         2    1.5
         3    3.5
         4    2.5
         Name: Total Bathrooms, dtype: float64
```

Exercise 3. Create a categorical variable that indicates whether or not a home has a pool.

```
In [27]: df["Pool QC"]

         df["Pool QC Binary"] = df["Pool QC"].map({
                "Ex": "Yes",
                "Gd": "Yes",
                "TA": "Yes",
                "Fa": "Yes",
                #or say np.nan : "No"
         }).fillna("No")

         df["Pool QC Binary"].value_counts()

Out[27]: No     2917
         Yes      13
         Name: Pool QC Binary, dtype: int64
```

Exercise 4. There are four types of utilities: electricity, gas, water, and sewage. Right now, the combination of utilities in a home is encoded in a single variable called "Utilities". Convert this variable into four boolean variables, each one indicating whether or not a home has a particular utility.

```
In [20]: df["Utilities"]

         df["Electric"] = df["Utilities"].map({
                "AllPub": "True",
                "NoSewr": "True",
                "NoSeWa": "True",
                "ELO": "True",
         })

         df["Gas"] = df["Utilities"].map({
                "AllPub": "True",
                "NoSewr": "True",
```

```
        "NoSeWa": "True",
        "ELO": "False",
})

df["Water"] = df["Utilities"].map({
        "AllPub": "True",
        "NoSewr": "True",
        "NoSeWa": "False",
        "ELO": "False",
})

df["Sewage"] = df["Utilities"].map({
        "AllPub": "True",
        "NoSewr": "False",
        "NoSeWa": "False",
        "ELO": "False",
})

df["Electric"].value_counts(), df["Gas"].value_counts(), df["Water"].value_counts(), d
```

```
Out[20]: (True    2930
          Name: Electric, dtype: int64, True    2930
          Name: Gas, dtype: int64, True     2929
          False       1
          Name: Water, dtype: int64, True     2927
          False       3
          Name: Sewage, dtype: int64)
```

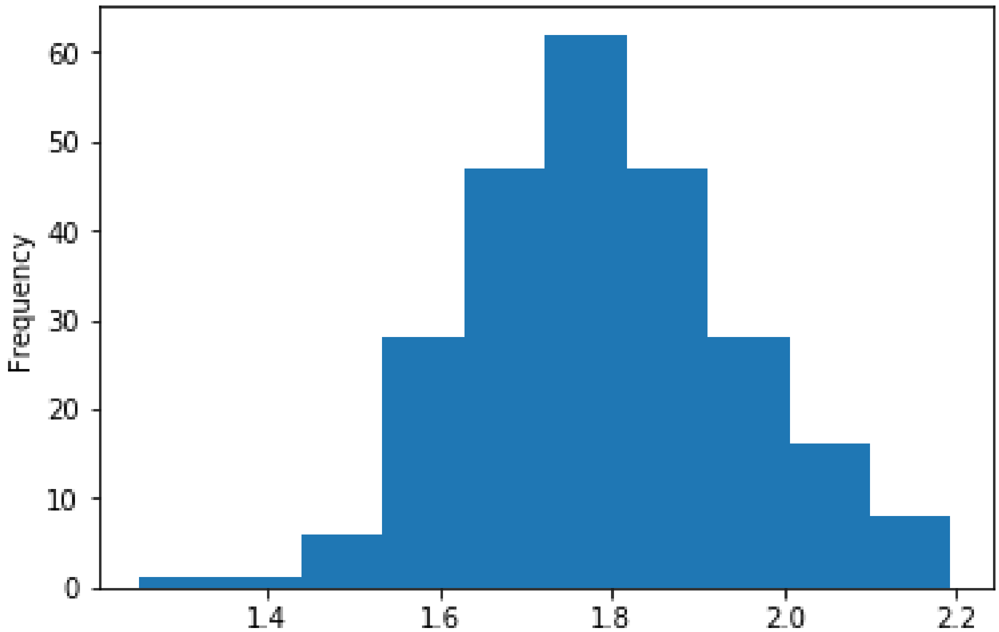
Exercises 5-7 deal with the Tips data set (https://raw.githubusercontent.com/dlsun/data-science-book/

Exercise 5. Make a visualization that shows the distribution of the total bills. Transform the variable first so that it is approximately symmetric.

```
In [29]: tips = pd.read_csv("/data301/data/tips.csv")

         #tips["total_bill"].plot.hist()
         #np.log(tips["total_bill"]).plot.hist()
         (tips["total_bill"] ** .2).plot.hist()
```

```
Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x7f586b486ac8>
```



Exercise 6. Suppose the total bill + tip are divided evenly among the people in each party. Which table paid the most *per person*?

```
In [30]: tips
         avg_payment = (tips["total_bill"] + tips["tip"]) / tips["size"]
         avg_payment.idxmax(), avg_payment.max()
         #would be better to use variable name "cost per person"

Out[30]: (184, 21.774999999999999)
```

Exercise 7. Make a visualization that shows how busy the restaurant is by day. Your visualization should display the full name of each day, i.e., "Thursday" instead of "Thur".

```
In [23]: tips
         tips["day"].value_counts()
         tips["full_day"] = tips["day"].map({
             "Sat":"Saturday",
             "Sun":"Sunday",
             "Fri":"Friday",
             "Thur":"Thursday",
         })
         full_day_counts = tips["full_day"].value_counts()
         full_day_order = full_day_counts[["Sunday", "Thursday", "Friday", "Saturday"]]
         full_day_order.plot.bar()
         full_day_order

Out[23]: Sunday      76
         Thursday    62
```

```
Friday      19
Saturday    87
Name: full_day, dtype: int64
```