Chapter 5
Comparing Groups: Tables and Visualizations

Marketing analysts often investigate differences between groups of people. Do men or women subscribe to our service at a higher rate? Which demographic segment can best afford our product? Does the product appeal more to homeowners or renters? The answers help us to understand the market, to target customers effectively, and to evaluate the outcome of marketing activities such as promotions.

Such questions are not confined to differences among people; similar questions are asked of many other kinds of groups. One might be interested in grouping data geographically: does Region A perform better than Region B? Or by time period: did same-store sales increase after a promotion such as a mailer or a sale? In all such cases, we are comparing one group of data to another to identify an effect.

In this chapter, we examine the kinds of comparisons between groups that often arise in marketing, with data that illustrate a consumer segmentation project. We review Python procedures to find descriptive summaries by groups, and then visualize the data in several ways.

5.1 Simulating Consumer Segment Data

We begin by creating a dataset that exemplifies a consumer segmentation project. For this example, we are offering a subscription-based service (such as cable television or membership in a warehouse club) and have collected data from $N = 300$ respondents on *age*, *gender*, *income*, *number of children*, whether they *own or rent* their homes, and whether they currently *subscribe* to the offered service or not. We use these data in later chapters as well.

Questions around customer segments are common in marketing research. These segments might be produced via a clustering algorithm (which we look at in Chap. 10) or could be created by some other heuristic, such as geographic location combined with age. In these data, we have assigned each respondent to one of four consumer segments: "Suburb mix," "Urban hip," "Travelers," or "Moving up." In this chapter we do not address *how* such segments might be identified; we just presume to know them. We will then look at how we might determine how to form these groups based on other factors, such as age, gender, or subscription status. If you know the group assignments, as we presume here, the segments themselves may be viewed as arbitrary; the same methods may be used to compare groups based on region or any another factor instead.

Segmentation data are moderately complex and we separate our code into three parts:

1. Definition of the data structure: the demographic variables (age, gender, and so forth) plus the segment names and sizes.
2. Parameters for the distributions of demographic variables, such as the mean and variance of each.
3. Code that iterates over the segments and variables to draw random values according to those definitions and parameters.

By organizing the code this way, it becomes easy to change some aspect of the simulation to draw data again. For instance, if we wanted to add a segment or change the mean of one of the demographic variables, only minor change to the code would be required. We also use this structure to teach new Python commands that appear in the third step to generate the data.

If you wish to load the data directly, it is available from the book's web site:

```
In [1]: import pandas as pd
        segment_data = pd.read_csv('http://bit.ly/PMR-ch5')
        segment_data.head()
```

© Springer Nature Switzerland AG 2020

J. S. Schwarz et al., *Python for Marketing Research and Analytics*, https://doi.org/10.1007/978-3-030-49720-0_5

```
Out[1]:
         Segment        age  gender         income  kids  own_home  \
    0  travelers  60.794945    male   57014.537526     0      True
    1  travelers  61.764535  female   43796.941252     0     False
    ...
    4  travelers  60.594199  female  103020.070798     0      True

       subscribe
    0      False
    1      False
    ...
    4      False
```

```
In [2]: segment_data.describe()
```

```
Out[2]:
               age         income        kids
    count  300.000000    300.000000  300.000000
    mean   40.923350   50669.454237    1.273333
    ...
    max    79.650722  108830.388732    7.000000
```

However, we recommend that you at least read the data generation sections. We demonstrate important Python language skills on simulating a dataset given a few basic statistics we want the dataset to represent.

5.1.1 *Segment Data Definition*

Our first step is to define general characteristics of the dataset: the variable names and the type of distribution from which they are drawn:

```
In [3]: segment_variables = ['age', 'gender', 'income', 'kids', 'own_home',
                             'subscribe']
        segment_variables_distribution = dict(zip(segment_variables,
                                              ['normal', 'binomial',
                                               'normal','poisson',
                                               'binomial', 'binomial']))

        segment_variables_distribution['age']
```

```
Out[3]: 'normal'
```

We have defined six variables: age, gender, income, kids, own_home, and subscribe, defined in `segment_variables`. `segment_variables_distribution` defines what kind of data will be present in each of those variables: normal data (continuous), binomial (yes/no), or Poisson (counts). `segment_variables_distribution` is a dictionary keyed by the variable name. For example, we see that when we pass `'age'` into `segment_variables_distribution` we get `'normal'`, indicating that we want age drawn from a normal distribution.

Next we start defining the statistics for each variable in each segment:

```
In [4]: segment_means = {'suburb_mix': [40, 0.5, 55000, 2, 0.5, 0.1],
                         'urban_hip':  [24, 0.7, 21000, 1, 0.2, 0.2],
                         'travelers':  [58, 0.5, 64000, 0, 0.7, 0.05],
                         'moving_up':  [36, 0.3, 52000, 2, 0.3, 0.2]}
```

`segment_means` is a dictionary keyed by the segment names. Each segment name has the means associated with it in a list. The list is ordered based on the `segment_variables` list we defined before. So the first value is the mean age for

that segment, the second values is the mean `gender` (i.e. the gender ratio), the third value is the mean `income`, and so forth. We used lists here because it makes it easy to compare the means to each other. We can quickly see that the mean age of 'suburb_mix' is 40 whereas for travelers it is 58. When we draw the random data later in this section, our routine will look up values in this matrix and sample data from distributions with those parameters.

In the case of binomial and Poisson variables, we only need to specify the mean. In these data, `gender`, `own_home`, and `subscribe` will be simulated as binomial (yes/no) variables, which requires specifying the probability for each draw. `kids` is represented as a Poisson (count) variable, whose distribution is specified by its mean. Note that we use these distributions for simplicity and do not mean to imply that they are necessarily the *best* distributions to fit real observations of these variables. For example, real observations of income are better represented with a skewed distribution.

However, for normal variables—in this case, `age` and `income`, the first and third variables—we additionally need to specify the *variance* of the distribution, the degree of dispersion around the mean. So we create another dictionary that defines the standard deviation for the variables that require it:

```
In [5]: # standard deviations for each segment
        # None = not applicable for the variable)
        segment_stddev = {'suburb_mix': [5, None, 12000, None, None, None],
                          'urban_hip':  [2, None, 5000, None, None, None],
                          'travelers':  [8, None, 21000, None, None, None],
                          'moving_up':  [4, None, 10000, None, None, None]}
```

Our next step is somewhat optional, but is good practice. We now have nearly all we need to generate a simulated dataset. But we can make our process cleaner by getting all values keyed by exactly what they are. What do we mean by that? We used set the mean and standard deviation values in lists above, but those are keyed numerically, so if we changed the order of our variables, we would use the wrong value. Instead, it's best practice to key them by the variable name. So we will now create a dictionary that contains all the statistics for each segment in a resilient structure from which we could create the entire dataset without referencing any other variables.

There is one more statistic left to set, which is the segment sizes. Here, we set those, and then we iterate through all the segments and all the variables and create a dictionary to hold everything:

```
In [6]: segment_names = ['suburb_mix', 'urban_hip', 'travelers', 'moving_up']
        segment_sizes = dict(zip(segment_names,[100, 50, 80, 70]))

        segment_statistics = {}
        for name in segment_names:
          segment_statistics[name] = {'size': segment_sizes[name]}
          for i, variable in enumerate(segment_variables):
            segment_statistics[name][variable] = {
                'mean': segment_means[name][i],
                'stddev': segment_stddev[name][i]
            }
```

What does this give us? We can check the values we get for the moving_up segment:

```
In [7]: segment_statistics['moving_up']

Out[7]: {'age': {'mean': 36, 'stddev': 4},
         'gender': {'mean': 0.3, 'stddev': None},
         'income': {'mean': 52000, 'stddev': 10000},
         'kids': {'mean': 2, 'stddev': None},
         'own_home': {'mean': 0.3, 'stddev': None},
         'size': 70,
         'subscribe': {'mean': 0.2, 'stddev': None}}
```

We see all the statistics for each variable defined explicitly. We can see that the mean income for moving_up is $52,000 with a standard deviation of $10,000. And that the mean age is 36 and the segment will be 30% male. There is a similar dictionary for each segment. With this dictionary (called a *lookup table*), we can create our simulated dataset.

5.1.2 Final Segment Data Generation

To generate the segment data, the logic we follow is to use nested `for` loops, one for the segments and another within that for the set of variables.

To outline how this will work, consider the following *pseudocode* (sentences organized like code):

```
Set up dictionary "segment_constructor" and pseudorandom number sequence
For each SEGMENT i in "segment_names" {
  Set up a temporary dictionary "segment_data_subset" for this SEGMENT's data
  For each VARIABLE in "seg_variables" {
    Check "segment_variable_distribution[variable]" to find distribution type for VARIABLE

    Look up the segment size and variable mean and standard deviation in segment_statistics for
    that SEGMENT and VARIABLE to
    ... Draw random data for VARIABLE (within SEGMENT) with
    ... "size" observations
  }
  Add this SEGMENT's data ("segment_data_subset") to the overall data ("segment_constructor")

  Create a DataFrame "segment_data" from "segment_constructor"
}
```

Pseudocode is a good way to outline and debug code conceptually before you actually write it. In this case, you can compare the pseudocode to the actual Python code to see how we accomplish each step. Translating the outline into Python, we write:

```
In [8]: import numpy as np

        np.random.seed(seed=2554)
        segment_constructor = {}

        # Iterate over segments to create data for each
        for name in segment_names:
          segment_data_subset = {}
          print('segment: {0}'.format(name))
          # Within each segment, iterate over the variables and generate data
          for variable in segment_variables:
            print('\tvariable: {0}'.format(variable))
            if segment_variables_distribution[variable] == 'normal':
              # Draw random normals
              segment_data_subset[variable] = np.random.normal(
                  loc=segment_statistics[name][variable]['mean'],
                  scale=segment_statistics[name][variable]['stddev'],
                  size=segment_statistics[name]['size']
              )
            elif segment_variables_distribution[variable] == 'poisson':
              # Draw counts
              segment_data_subset[variable] = np.random.poisson(
                  lam=segment_statistics[name][variable]['mean'],
                  size=segment_statistics[name]['size']
              )
            elif segment_variables_distribution[variable] == 'binomial':
              # Draw binomials
              segment_data_subset[variable] = np.random.binomial(
                  n=1,
                  p=segment_statistics[name][variable]['mean'],
                  size=segment_statistics[name]['size']
              )
```

```
            else:
              # Data type unknown
              print('Bad segment data type: {0}'.format(
                  segment_variables_distribution[j])
                   )
              raise StopIteration
          segment_data_subset['Segment'] = np.repeat(
              name,
              repeats=segment_statistics[name]['size']
          )
          segment_constructor[name] = pd.DataFrame(segment_data_subset)
        segment_data = pd.concat(segment_constructor.values())
```

The core commands occur inside the `if` statements: according to the data type we want ("normal", "poisson", or "binomial"), use the appropriate pseudorandom function to draw data (the function `np.random.normal(loc, scale, size)`, `np.random.poisson(lam, size)`, or `np.random.binomial(n, size, p)`, respectively). We draw all of the values for a given variable within a given segment with a single command (drawing all the observations at once, with length specified by `segment_statistics[name]['size']`).

We can see an example of how this works by setting `name = 'suburb_mix'` and `variable = 'age'` and running one of the commands from the loop. We set `size=10` so we don't get too many values:

```
In [9]:name = 'suburb_mix'
       variable = 'age'
       print(segment_statistics[name][variable]['mean'])
       print(segment_statistics[name][variable]['stddev'])
       np.random.normal(
           loc=segment_statistics[name][variable]['mean'],
           scale=segment_statistics[name][variable]['stddev'],
           size=10
       )

40
5

Out[9]: array([37.16950666, 45.23743976, 44.23421807, 41.62070249, 30.66891058,
                44.86711234, 34.48936766, 42.63618686, 45.16799349, 42.61294136])
```

Note that the input code ends with the `)`. The numbers `40` and `5` are the result of the print statement, which will appear in the output block in a Colab notebook or as printed here in a Jupyter notebook.

On the last two lines of output, we see that this output has ten values. Those values are distributed around 40 and a standard deviation of 5 seems believable, although it is hard to really assess that with such a small sample.

For the `Segment` variable, we merely want a repetition of the segment name. To do this, we can use `np.repeat(a, repeats)`, which will repeat the input `a` `repeats` times:

```
In [10]: np.repeat(name, repeats=10)

Out[10]: array(['suburb_mix', 'suburb_mix', 'suburb_mix', 'suburb_mix',
                'suburb_mix', 'suburb_mix', 'suburb_mix', 'suburb_mix',
                'suburb_mix', 'suburb_mix'], dtype='|S10')
```

Back to the main simulation code, there are a few things to note. To see that the code is working and to show progress, we use `print()` to print out the segment and variable names as the loop iterates. That results in the following output as the code runs:

```
segment: suburb_mix
        variable: age
        variable: gender
        variable: income
        variable: kids
        variable: own_home
        variable: subscribe
```

```
segment: urban_hip
        variable: age
        variable: gender
        variable: income
        variable: kids
        variable: own_home
        variable: subscribe
segment: travelers
        variable: age
        variable: gender
        variable: income
        variable: kids
        variable: own_home
        variable: subscribe
segment: moving_up
        variable: age
        variable: gender
        variable: income
        variable: kids
        variable: own_home
        variable: subscribe
```

Inside the first loop (`name` loop), we define `segment_data_subset` as a dictionary. In vectorized programming languages, such as R or Matlab, it would be advisable to *preallocate* the data structures as in those languages, whenever an object grows in memory—such as adding a row—a copy is made of the object. This uses twice the memory and slows things down. Preallocating avoids that problem.

Python, however is extremely efficient at growing native iterable types, such as `lists` and `dicts`, in memory. For this reason, we generate the data in a `dict`, and then convert it to a Pandas `DataFrame` for analysis.

Exceptions to this preallocation rule would be when using non-native, vectorized objects such as Numpy `arrays` and Pandas `DataFrames`. Whenever there is a need to iteratively generate data in such types, rather than converting to them from a native type, it is advisable to preallocate the data arrays. Another benefit of preallocating is that it adds a bit of error checking: if a result doesn't fit into the dataframe where it *should* fit, we will get a warning or error.

We finish the `if` blocks in our code with a `StopIteration` error that is raised in the case that a proposed data type doesn't match what we expect. There are three `if` tests for the expected data types, and a final `else` block in case none of the `ifs` matches. This protects us in the case that we mistype a data type or if we try to use a distribution that hasn't been defined in the random draw code, such as a gamma distribution. This error condition would cause the code to exit immediately and print an error string.

Notice that we are doing a lot of thinking ahead about how our code might change and potentially break in the future to ensure that we would get a warning when something goes wrong. Our code also has another advantage that you may not notice right away: we call each random data function such as `np.random.normal` in exactly one place. If we discover that there was something wrong with that call—say we wanted to change one of the parameters of the call—we only need to make the correction in one place. This sort of planning is a hallmark of good programming in Python or any other language. While it might seem overly complex at first, many of these ideas will become habitual as you write more programs.

To finish up the dataset, we perform a few housekeeping tasks, converting each binomial variable to clearer values, booleans or strings:

```
In [11]: segment_data['gender'] = segment_data['gender'].apply(
             lambda x: 'male' if x else 'female'
         )
         segment_data['own_home'] = segment_data['own_home'].apply(
             lambda x: True if x else False
         )
         segment_data['subscribe'] = segment_data['subscribe'].apply(
             lambda x: True if x else False
         )
```

We may now inspect the data. As always, we recommend a data inspection plan as noted in Sect. 3.6, although we only show one of those steps here:

```
In [12]: segment_data.describe(include='all')
```

```
Out[12]:            Segment        age gender         income        kids  \
         count          300 300.000000    300     300.000000  300.000000
         unique           4        NaN      2            NaN         NaN
         top     suburb_mix        NaN   male            NaN         NaN
         freq           100        NaN    156            NaN         NaN
         mean           NaN  40.923350    NaN   50669.454237    1.273333
         std            NaN  12.827494    NaN   19336.497748    1.413725
         min            NaN  18.388730    NaN   11297.309231    0.000000
         25%            NaN  32.870035    NaN   41075.804389    0.000000
         50%            NaN  38.896711    NaN   51560.344807    1.000000
         75%            NaN  47.987569    NaN   62172.668698    2.000000
         max            NaN  79.650722    NaN  108830.388732    7.000000

                own_home subscribe
         count       300       300
         unique        2         2
         top       False     False
         freq        167       265
         mean        NaN       NaN
         std         NaN       NaN
         min         NaN       NaN
         25%         NaN       NaN
         50%         NaN       NaN
         75%         NaN       NaN
         max         NaN       NaN
```

The dataframe is now suitable for exploration. And we have reusable code: we could create data with more observations, different segment sizes, or segments with different distributions or means by simply adjusting the matrices that define the segments and running the code again.

As a final step we save the dataframe as a backup and to use again in later chapters (Sects. 10.2 and 11.1.2). Change the destination if you have created a folder for this book or prefer a different location:

```
In [13]: from google.colab import files
         with open('segment_dataframe_Python_intro_Ch5.csv', 'w') as f:
           segment_data.to_csv(f)

         files.download('segment_dataframe_Python_intro_Ch5.csv')
```

Note that if you are running Python locally, the `files.download()` command is unnecessary, as is importing the `files` module (which is Colab-specific).

5.2 Finding Descriptives by Group

For our consumer segmentation data, we are interested in how measures such as household income and gender vary for the different segments. With this insight, a firm might develop tailored offerings for the segments or engage in different ways to reach them.

An ad hoc way to do this is with dataframe indexing: find the rows that match some criterion, and then take the mean (or some other statistic) for the matching observations on a variable of interest. For example, to find the mean income for the "moving_up" segment:

```
In [14]: segment_data.loc[segment_data.Segment == 'moving_up']['income'].mean()
```

```
Out[14]: 51763.55266630597
```

This says "from the income observations, take all cases where the Segment column is 'moving_up' and calculate their mean." We could further narrow the cases to "moving_up" respondents who also do not subscribe using Boolean logic:

```
In [15]: segment_data.loc[
              (segment_data['Segment'] == 'moving_up') &
              (segment_data['subscribe'] == False)
         ]['income'].mean()

Out[15]: 52495.6820839035
```

This quickly becomes tedious when you wish to find values for multiple groups.

As we saw briefly in Sect. 3.2.1, a more general way to do this is with `data.groupby(INDICES)[COLUMN].FUNCTION`. The result of `groupby()` is to divide `data` into groups for each of the unique values in `INDICES` and then apply the `FUNCTION` function to the data in `COLUMN` for each group:

```
In [16]: segment_data.groupby('Segment')['income'].mean()

Out[16]: Segment
         moving_up     51763.552666
         suburb_mix    55552.282925
         travelers     62609.655328
         urban_hip     20267.737317
         Name: income, dtype: float64
```

With `groupby()`, keep in mind that it is a method on `data` and the splitting factors `INDICES` are the argument. The `FUNCTION`, `mean()` in this case, is applied to a single `COLUMN`, 'income' in this case. There are a subset of defined methods that can be applied to the columns, such as `mean()` and `sum()`, but any method can be applied using the `apply` method as described in Sect. 3.3.3.

You can break out the results by multiple factors if you supply factors in a `list`. For example, we can break out by segment and subscription status:

```
In [17]: segment_data.groupby(['Segment', 'subscribe'])['income'].mean()

Out[17]: Segment     subscribe
         moving_up   False          52495.682084
                     True           49079.078135
         suburb_mix  False          55332.038973
                     True           58478.381142
         travelers   False          62940.429960
                     True           49709.444658
         urban_hip   False          20496.375001
                     True           19457.112800
         Name: income, dtype: float64
```

Here, we can use the `unstack()` method on the output to get a nicer formatting of the output:

```
In [18]: segment_data.groupby(
              ['Segment', 'subscribe']
         )['income'].mean().unstack()

Out[18]: subscribe           False         True
         Segment
         moving_up    52495.682084  49079.078135
         suburb_mix   55332.038973  58478.381142
         travelers    62940.429960  49709.444658
         urban_hip    20496.375001  19457.112800
```

What does `unstack()` do? Since we grouped by two different columns, we wound up with a hierarchical index. We can "unstack," or pivot, that hierarchy, making one dimension a column and the other a row using `unstack()`. This can make the output easier to read and to work with.

Suppose we wish to add a "segment mean" column to our dataset, a new observation for each respondent that contains the mean income for their respective segment so we can compare respondents' incomes to those typical for their segments. We can do this by using `groupby()` to get the segment means, and then using `join()` to add the mean segment income as a column `income_seg`. We generally do not like adding derived columns to primary data because we like to separate data from subsequent computation, but we do so here for illustration:

```
In [19]: np.random.seed(4532)
         segment_income = segment_data.groupby('Segment')['income'].mean()
         segment_data = segment_data.join(segment_income,
                                          on='Segment',
                                          rsuffix='_segment')
         segment_data.head(5)

Out[19]:       age    gender        income     kids  own_home  subscribe  \
         0  44.057078  female  54312.575694     3     False      False
         1  34.284213  female  67057.192182     1     False      False
         2  45.159484  female  56306.492991     3      True      False
         3  41.032557    male  66329.337521     1     False       True
         4  41.781819  female  56500.410372     2     False      False

              Segment  income_segment
         0  suburb_mix    55552.282925
         1  suburb_mix    55552.282925
         2  suburb_mix    55552.282925
         3  suburb_mix    55552.282925
         4  suburb_mix    55552.282925
```

When we check the data, we see that each row has an observation that matches its segment mean.

It is worth thinking about how this works. In a `join()`, two `DataFrames`, two `Series`, or a `DataFrame` and a `Series` can be combined using a common column as an index, in this case `Segment`. Even though `segment_income` only had 4 rows, one for each segment, a value was added to every row of `seg` based on the shared value of the `Segment` column. The result is a dataframe in which each row of `segment_mean` occurs many times in the order requested.

Again, we don't want a derived column in our primary data, so we now remove that column by using the `drop()` method:

```
In [20]: segment_data.drop(labels='income_segment', axis=1, inplace=True)
         segment_data.head(5)

Out[20]:       age    gender        income      kids  own_home  subscribe  \
         0  44.057078  female  54312.575694     3     False      False
         ...
         4  41.781819  female  56500.410372     2     False      False

              Segment
         0  suburb_mix
         ...
         4  suburb_mix
```

As an aside, `drop()` removes an entire row or column from a dataframe. We specify whether we want it to be a row or column with the `axis` argument: `0` for row and `1` for column. Which column or row to remove is specified with the `label` argument, which can specify a single label or can be a list of labels to be removed. The `inplace=True` argument specifies that this should be done on the object itself. The default value for `inplace` is `False`, in which case `drop()` will return a copy of the input dataframe rather than modifying it.

Going back to our main point, which was being quickly able to compare an individual response to the segment mean, we see that `groupby()` exemplifies the power of Python and pandas to extract and manipulate data with simple and concise commands.

5.2.1 *Descriptives for Two-way Groups*

A common task in marketing is cross-tabulating, separating customers into groups according to two (or more) factors. We can use `groupby()` to aggregate across multiple factors. For example:

```
In [21]: segment_data.groupby(['Segment', 'own_home'])['income'].mean()

Out[21]: Segment      own_home
         moving_up    False       51430.222115
                      True        52363.547659
         suburb_mix   False       56764.508540
                      True        54239.038508
         travelers    False       62923.233941
                      True        62449.907732
         urban_hip    False       20139.092369
                      True        21057.984851
         Name: income, dtype: float64
```

We now have a separate group for each combination of `Segment` and `own_home` and can begin to see how `income` is related to both the `Segment` and the `own_home` variables.

The grouping can be extended to include as many grouping variables as needed:

```
In [22]: segment_data.groupby(
             ['Segment', 'own_home', 'subscribe']
         )['income'].mean()

Out[22]: Segment      own_home  subscribe
         moving_up    False     False        52380.092911
                                True         47630.738931
                      True      False        52714.693149
                                True         51251.586942
         suburb_mix   False     False        56478.645027
                                True         59451.625569
                      True      False        54160.506701
                                True         56045.270075
         travelers    False     False        62923.233941
                      True      False        62949.533735
                                True         49709.444658
         urban_hip    False     False        20171.798013
                                True         20031.163747
                      True      False        22281.548438
                                True         13716.603325
         Name: income, dtype: float64
```

And, again, we can use `unstack()` to make it more readable:

```
In [23]: segment_data.groupby(
             ['Segment', 'own_home', 'subscribe']
         )['income'].mean().unstack()

Out[23]: subscribe                    False         True
         Segment      own_home
         moving_up    False     52380.092911  47630.738931
                      True      52714.693149  51251.586942
         suburb_mix   False     56478.645027  59451.625569
                      True      54160.506701  56045.270075
         travelers    False     62923.233941           NaN
                      True      62949.533735  49709.444658
```

```
urban_hip   False     20171.798013  20031.163747
            True      22281.548438  13716.603325
```

The `groupby` method allows us to compute functions of continuous variables, such as the `mean` of `income` or `age`, for any combination of factors (`Segment`, `own_home` and so forth). This is such a common task in marketing research that there used to be entire companies who specialize in producing cross tabs. As we've just seen, these are not difficult to compute in Python.

We might also want to know the *frequency* with which different combinations of `Segment` and `own_home` occur. We can compute frequencies using `groupby()` along with the `count()` method to obtain one-way or multi-way counts:

```
In [24]: segment_data.groupby(
             ['Segment', 'own_home']
         )['subscribe'].count().unstack()
```

```
Out[24]: own_home     False  True
         Segment
         moving_up       45    25
         suburb_mix      52    48
         travelers       27    53
         urban_hip       43     7
```

There are 7 observed customers in the "Urban hip" segment who own their own homes, and 53 in the "Travelers" segment.

Suppose we want a breakdown of the number of kids in each household (`kids`) by `segment`:

```
In [25]: segment_data.groupby(
             ['kids', 'Segment']
         ).subscribe.count().unstack(level=1)
```

```
Out[25]: Segment  moving_up  suburb_mix  travelers  urban_hip
         kids
         0             13.0        15.0       80.0       14.0
         1             18.0        27.0        NaN       21.0
         2             21.0        21.0        NaN       12.0
         3              9.0        29.0        NaN        1.0
         4              5.0         3.0        NaN        1.0
         5              2.0         3.0        NaN        1.0
         6              1.0         2.0        NaN        NaN
         7              1.0         NaN        NaN        NaN
```

This tells us that we have 14 "Urban hip" respondents with 0 kids, 21 "Suburb mix" respondents with 2 kids, and so forth. It represents purely the count of incidence for each crossing point between the two factors, `kids` and `Segment`. In this case we are treating `kids` as a factor and not a number. Note that `NaN` indicates that there were no values for that combination of factors, i.e., the count is zero.

We can also use the `crosstabs()` function to get the same result:

```
In [26]: pd.crosstab(segment_data['kids'], segment_data['Segment'])
```

```
Out[26]: Segment  moving_up  suburb_mix  travelers  urban_hip
         kids
         0               13          15         80         14
         1               18          27          0         21
         2               21          21          0         12
         3                9          29          0          1
         4                5           3          0          1
         5                2           3          0          1
         6                1           2          0          0
         7                1           0          0          0
```

However, `kids` is actually a count variable; if a respondent reported 3 kids, that is a count of 3 and we could add together the counts to get the total number of children reported in each segment.:

```
In [27]: segment_data.groupby('Segment')['kids'].sum()
```

```
Out[27]: Segment
         moving_up     130
         suburb_mix    195
         travelers       0
         urban_hip      57
         Name: kids, dtype: int64
```

Python typically has many ways to arrive at the same result. This may seem overly complex yet it is a good thing. One reason is that there are multiple options to match your style and situation. Each method produces results in a different format, and one format might work better in some situation than another. Another reason is that you can do the same thing in two different ways and compare the answers, thus testing your analyses and uncovering potential errors.

5.2.2 *Visualization by Group: Frequencies and Proportions*

Tables are very valuable for exploring data and interactions between various factors. However, visualizations can rapidly reveal associations that may be less obvious when observed within a table.

The most commonly used plotting library in Python, and default pandas plotting library is `matplotlib` (Hunter 2007). Its integration with pandas makes plotting DataFrames straightforward.

Suppose we plot the count of subscribers for each segment to understand better which segments use the subscription service, as in Fig. 5.1:

```
In [28]: import matplotlib.pyplot as plt

         segments_groupby_segments = segment_data.groupby(['Segment'])
         segments_groupby_segments['subscribe'].value_counts().unstack().plot(
             kind='barh',
             figsize=(8, 8)
         )
         plt.xlabel('counts')
```

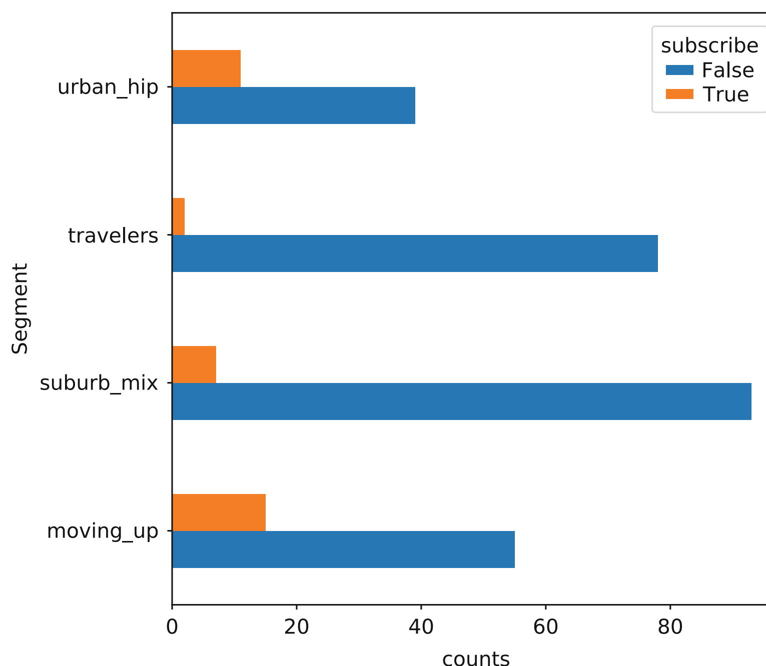

Fig. 5.1 Conditional histogram for count of subscribers within each segment

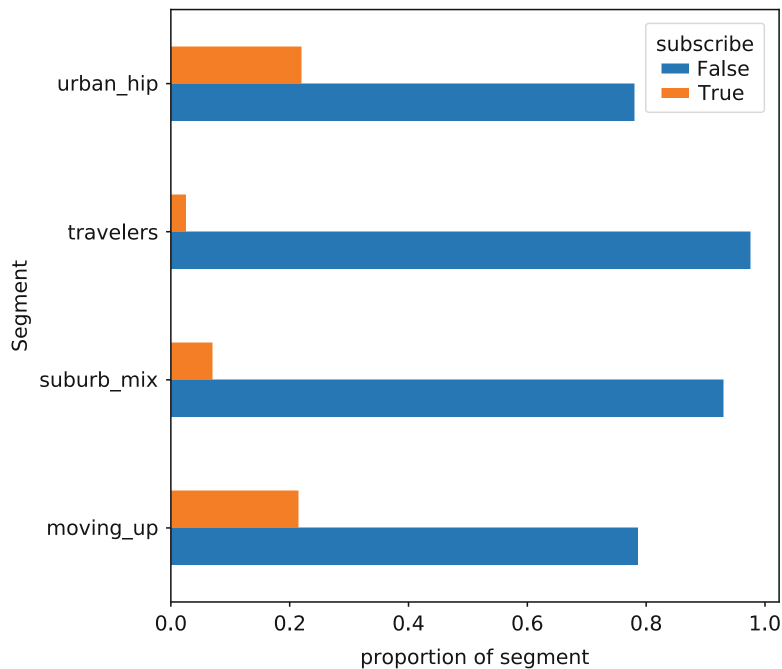

Fig. 5.2 Conditional histogram for proportion of subscribers within each segment

Here, we used the `value_counts()` function introduced in 3.2.2. `unstack()` *unstacks* the indices, turning the series object in to a dataframe that we can easily plot.

By passing `normalize=True` to `value_counts()` we can get proportions within each segment that subscribe, as in Fig. 5.2:

```
In [29]: segments_groupby_segments['subscribe'].value_counts(
             normalize=True
         ).unstack().plot(
             kind='barh',
             figsize=(8, 8)
         )
         plt.xlabel('proportion of segment')
```

And by aggregating by `subscribe` and running `value_count()` on `Segment` we can see breakdown of subscribers and non-subscribers by segment (Fig. 5.3):

```
In [30]: segment_data.groupby(['subscribe'])['Segment'].value_counts(
             normalize=True
         ).unstack().plot(kind='barh', figsize=(8, 8))
         plt.xlabel('proportion of subscribers')
```

Another popular packages is `seaborn`, which simplifies some of the aggregation steps and makes attractive figures with the default options. We can easily create something similar to Fig. 5.2 (not shown):

```
In [31]: import seaborn as sns
         sns.barplot(y='Segment', x='subscribe', data=segment_data,
                     orient='h', ci=None)
```

Seaborn also includes the `facetgrid()` function which allows the creation of multipanel figures, as in Fig. 5.4:

```
In [32]: g = sns.FacetGrid(segment_data, col='Segment')
         g.map(sns.barplot, 'subscribe', orient='v', ci=None)
```

This particular usage is not very interesting, but we can now separate out another factor, such as home ownership and have the respective bars in separate rows (Fig. 5.5):

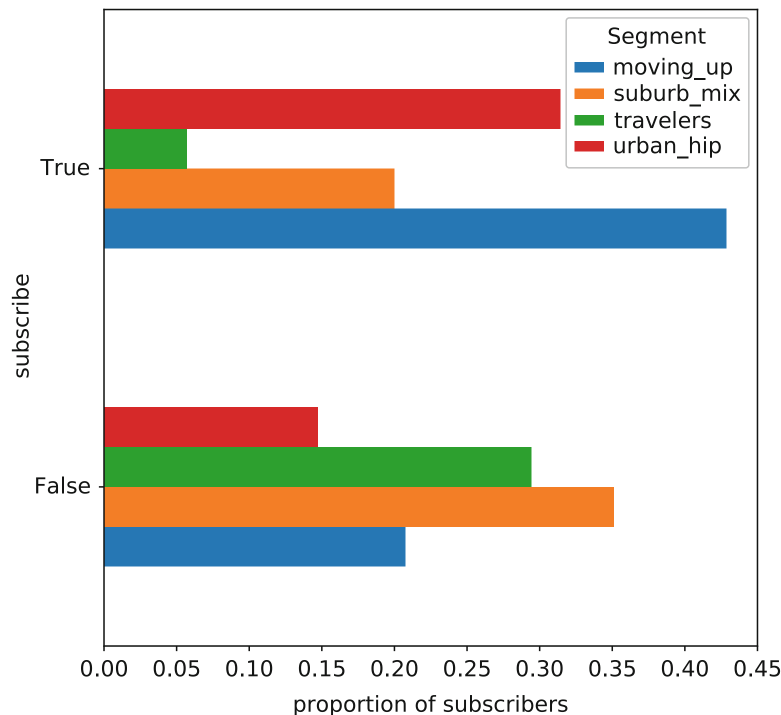

Fig. 5.3 Conditional histogram for proportion of segments within each subscription state

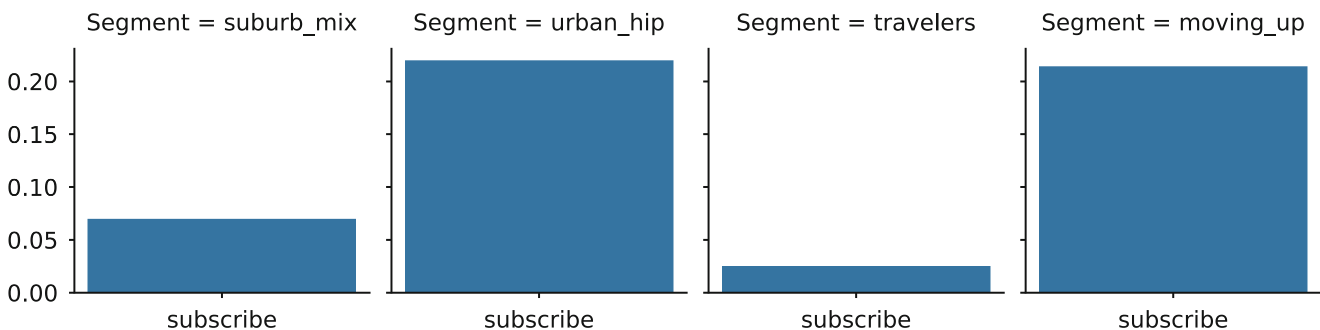

Fig. 5.4 Conditional histogram for proportion of segments within each subscription state generate using Seaborn facetgrid()

```
In [33]: g = sns.FacetGrid(segment_data, col='Segment', row='own_home')
         g.map(sns.barplot, 'subscribe', orient='v', ci=None)
```

5.2.3 *Visualization by Group: Continuous Data*

In the previous section we saw how to plot counts and proportions. What about continuous data? How would we plot `income` by `segment` in our data? A simple way is to use `groupby()` to find the mean income, and then use the `plot(kind='bar')` method to plot the computed values:

```
In [34]: segment_data.groupby(['Segment'])['income'].mean().plot.bar()
```

The result is in the left panel of Fig. 5.6. We can also use seaborn `barplot()` to produce a similar plot, shown in the right panel of Fig. 5.6:

```
In [35]: sns.barplot(x='Segment', y='income', data=segment_data, color='.6',
                     estimator=np.mean, ci=95)
```

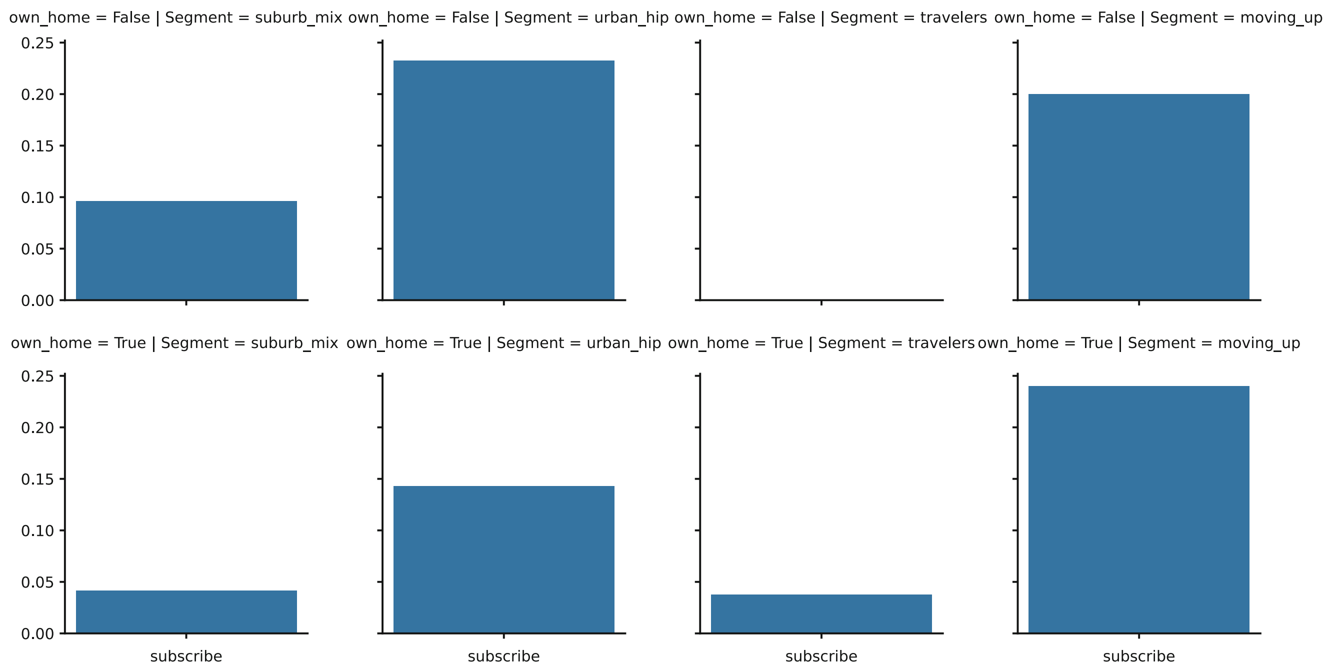

Fig. 5.5 Conditional histogram for proportion of segments within each subscription state generate using Seaborn facetgrid()

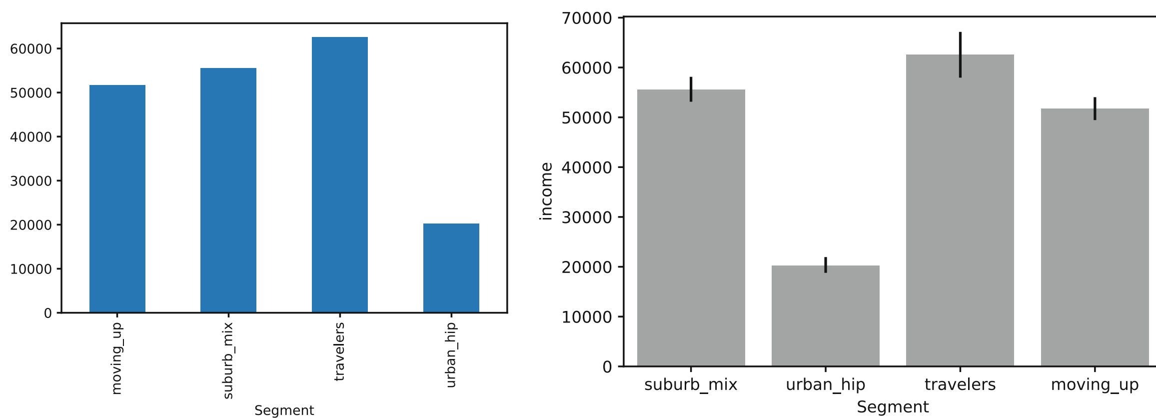

Fig. 5.6 Average income by segment using `groupby()` and `plot()` in the left panel and seaborn `barplot()` in the right panel

Note that the two plotting functions order the segments differently. Seaborn does more processing of the data and does things like sorting the columns. In general, Seaborn figures work better out of the box, but can be more difficult to customize.

Adding Another Factor

How do we split this out further by home ownership? Using matplotlib, we can add another `groupby` factor, `own_home`, shown in Fig. 5.7 left panel:

```
In [36]: segment_data.groupby(
             ['Segment', 'own_home']
         )['income'].mean().unstack().plot.bar()
```

Using Seaborn, an additional factor may be added in the form of a facet grid, as in Fig. 5.6, or by setting the `hue` parameter, as shown in the right panel of Fig. 5.7:

```
In [37]: sns.barplot(x='Segment', y='income', hue='own_home',
                     data=segment_data, estimator=np.mean, ci=95)
```

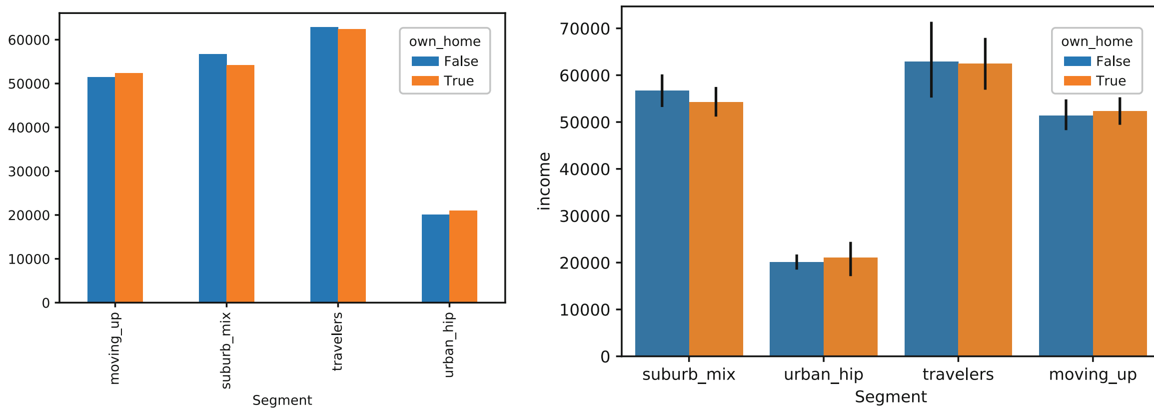

Fig. 5.7 Average income by segment and home ownership using `plot()` (left) or Seaborn `barplot()` (right)

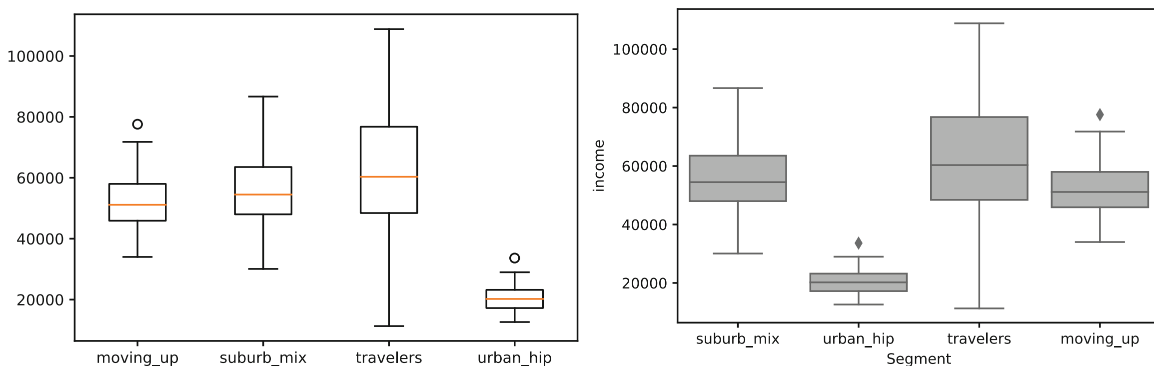

Fig. 5.8 Box-and-whiskers plot for income by segment using matplotlib (left) Seaborn (right) `boxplot()` functions

Box Plot

A more informative plot for comparing values of continuous data, like `income` for different groups is a *box-and-whiskers* plot (also known simply as a "boxplot"), which we first encountered in Sect. 3.4.2. A boxplot is better than a barchart because it shows more about the *distributions* of values.

We can create a boxplot using the matplotib `boxplot()` function:

```
In [38]: x = segment_data.groupby('Segment')['income'].apply(list)
         _ = plt.boxplot(x=x.values, labels=x.index)
```

Seaborn `boxplot()` works with a DataFrame and two factors (at least one of which must be numeric):

```
In [39]: sns.boxplot(x='Segment', y='income', data=segment_data,
                     color='0.7', orient='v')
```

Figure 5.8 shows that the income for "Travelers" is higher and also has a greater range, with a few "Travelers" reporting very low incomes. The range of income for "Urban hip" is much lower and tighter. Although box-and-whisker plots are not common in business reporting, we think they should be. They encode a lot more information than the averages shown in Fig. 5.6.

To break this down by more factors, we may add a `hue` argument. The Seaborn `facetgrid()` method allows us to condition on more factors. However, for two factors, such as comparing income by segment and home ownership, we might use `hue`:

```
In [40]: sns.boxplot(y='Segment', x='income', hue='own_home',
                     data=segment_data, color='0.7', orient='h')
```

In Fig. 5.9, it is clear that within segments there is no consistent relationship between income and home ownership.

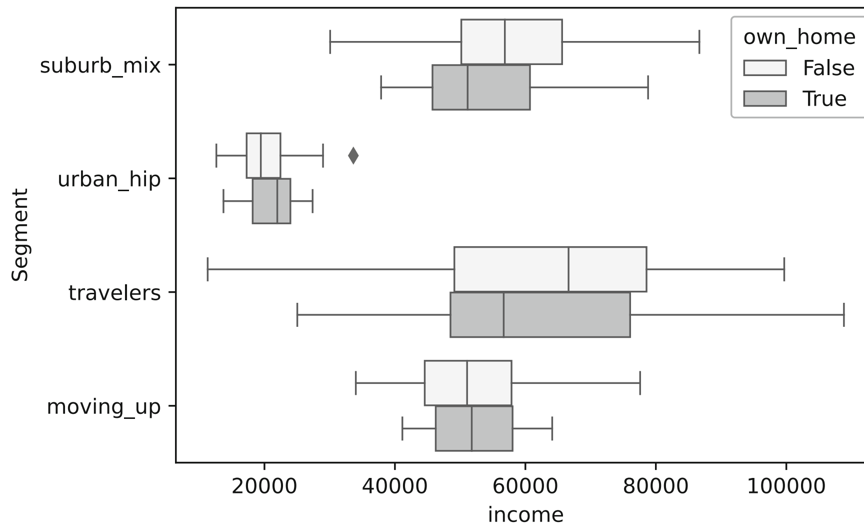

Fig. 5.9 Box-and-whiskers plot for income by segment and home ownership using `boxplot`

5.2.4 *Bringing It All Together*

We have learned how to approach comparing groups. How might we use this? As analysts, we explore data in order to learn new information that we can share and to inform marketing and product decisions. So how might we interpret what we have seen so far?

We have not yet done any statistical analysis, which we introduce in Chap. 6, so any conclusions must be tempered. But, directionally, we observe that the segments differ in several ways that may affect how we should market our subscription product. If our subscription is an expensive, luxury product, we might want to target only the wealthier segments. Perhaps those without children are more likely to have disposable income, and they may be members of the "travelers segment," which has a very low rate of subscription. On the other hand, if our product is intended for young urbanites (i.e., "urban hip"), who show a high subscription rate, we might take more care with pricing, as the average income is lower in that group.

The exact interpretation depends on what problem we are trying to solve. Are we trying to understand our current customers so we can get more similar customers? Or are we trying to expand our customer base into different groups?

The way we approach an analysis is driven by the questions we want to answer, not by the data we have. Sometimes our ability to answer those questions is hampered by the data we have available. In that case, we might think about new data sources, or apply cautious interpretation.

5.3 Learning More*

The topics in this chapter are foundational both for programming skills in Python and for applied statistics.

For categorical data analysis, the best starting place is—although not specific to Python—is Agresti's *An Introduction to Categorical Data Analysis* (Agresti 2012).

In Chap. 6 we continue our investigation with methods that formalize group comparisons and estimate the statistical strength of differences between groups.

5.4 Key Points

This was a crucial chapter for doing everyday analytics with Python. Here are some of the key points.

- We generated a very complicated dataset; to do so, we defined each segment variable, its distribution type, and the parameters of those distributions. We used those initializations in a set of for loops to generate the dataset (Sect. 5.1)

- The `groupby()` command can split up data and automatically apply functions such as `mean()` and `count()` (Sect. 5.2)
- Frequency of occurrence can be found with `groupby()` and the `count()` function or using the pandas `crosstabs()` function (Sect. 5.2.1)
- matplotlib and Seaborn both offer valuable plotting functions. Seaborn plots tend to look better in default settings, but are more complex to customize than matplotlib plots (Sect. 5.2.2)
- Charts of proportions and occurrence by a factor can be generated using `groupby()` along with the `plot()` method or by using the Seaborn `barplot()` function (Sect. 5.2.2)
- The Seaborn `FacetGrid()` class extends such plots to multiple factors (Sect. 5.2.2)
- Plots for continuous data by factor can also use `groupby()` along with `plot()` or the Seaborn `barplot()` function, or even better, box-and-whiskers plots with `boxplot()`, from either matplotlib or Seaborn. (Sect. 5.2.3)