Chapter 3
Describing Data

In this chapter, we tackle our first marketing analytics problem: exploring a new dataset. The goals for this chapter are to learn how to:

- Simulate a dataset
- Summarize and explore a dataset with descriptive statistics (mean, standard deviation, and so forth)
- Explore simple visualization methods

Such investigation is the simplest analysis one can do yet also the most crucial. It is important to describe and explore any dataset before moving on to more complex analysis. This chapter will build your Python skills and provide a set of tools for exploring your own data.

3.1 Simulating Data

We start by creating data to be analyzed in later parts of the chapter. Why simulate data and not work entirely with real datasets? There are several reasons. The process of creating data lets us practice and deepen Python skills from Chap. 2. It makes the book less dependent on vagaries of finding and downloading online datasets. And it lets you manipulate the synthetic data, run analyses again, and examine how the results change.

Perhaps most importantly, data simulation highlights a strength of Python: because it is easy to simulate data, Python analysts often use simulated data to prove that their methods are working as expected. When we know what the data *should* say (because we created it), we can test our analyses to make sure they are working correctly before applying them to real data. If you have real datasets that you work with regularly, we encourage you to use those for the same analyses alongside our simulated data examples. (See Sect. 2.6.2 for more information on how to load data files.)

We encourage you to create data in this section step by step because we teach Python along the way. However, if you are in a hurry to learn how to compute means, standard deviations and other summary statistics, you could quickly run the commands in this section to generate the simulated data. Alternatively, the following command will load the data from the book's web site, and you can then go to Sect. 3.2:

```
In [0]: import pandas as pd
        store_sales = pd.read_csv('http://bit.ly/PMR-ch3')
```

But if you're new to Python data analysis, don't do that! Instead, work through the following section to create the data from scratch. If you accidentally ran the command above, you can use `del store_sales` to remove the data before proceeding.

3.1.1 Store Data: Setting the Structure

Our first dataset represents observations of total sales by week for two competing products at a chain of stores. We begin by creating a data structure that will hold the data, a simulation of sales for the two products in 20 stores over 2 years, with price

J. S. Schwarz et al., *Python for Marketing Research and Analytics*, https://doi.org/10.1007/978-3-030-49720-0_3

and promotion status. We remove most of the Python output here to focus on the input commands. Type the following lines, but feel free to omit the comments (following "#"):

```
In [0]: # import numpy and pandas
        import pandas as pd
        import numpy as np

        # Constants
        N_STORES = 20
        N_WEEKS = 104

        # create a dataframe of initially missing values to hold the data
        columns = ('store_num', 'year', 'week', 'p1_sales', 'p2_sales',
                   'p1_price', 'p2_price', 'p1_promo', 'p2_promo', 'country')
        n_rows = N_STORES * N_WEEKS
        store_sales = pd.DataFrame(np.empty(shape=(n_rows, 10)),
                                   columns=columns)
```

Here we first set a few constants, the number of stores and the number of weeks of data for each store. We then import the NumPy and pandas libraries, which we will use extensively throughout this book, and create the empty dataframe. We could also have generated all the columns first and then put them together into a dataframe at the end, as we did in 2.5.3.

We see the simplest summary of the dataframe by looking at the `shape` parameter:

```
In [1]: store_sales.shape

Out[1]: (2080, 10)
```

As expected, `store_sales` has 2080 rows and 10 columns.

We can use `head()` to inspect `store_sales`:

```
In [2]: store_sales.head()

Out[2]:    store_num  year  week  p1_sales  p2_sales  p1_price  p2_price  \
        0        0.0   0.0   0.0       0.0       0.0       0.0       0.0
        1        0.0   0.0   0.0       0.0       0.0       0.0       0.0
        2        0.0   0.0   0.0       0.0       0.0       0.0       0.0
        3        0.0   0.0   0.0       0.0       0.0       0.0       0.0
        4        0.0   0.0   0.0       0.0       0.0       0.0       0.0

           p1_promo  p2_promo  country
        0       0.0       0.0      0.0
        1       0.0       0.0      0.0
        2       0.0       0.0      0.0
        3       0.0       0.0      0.0
        4       0.0       0.0      0.0
```

As expected, it is empty; all values are set to zero.

First, we will create a set of store "numbers" or "ids," which will serve to identify each store:

```
In [3]: store_numbers = range(101, 101 + N_STORES)
        list(store_numbers)

Out[3]: [101,
         102,
         103,
         104,
         105,
         106,
         107,
         108,
```

```
          109,
          110,
          111,
          112,
          113,
          114,
          115,
          116,
          117,
          118,
          119,
          120]
```

We used `range` to create store numbers ranging from 101 to 120. Note that we cast the `Range` object to a `list` in order to print it out. See Sect. 2.4.4 for an overview of `Range` objects.

Next, we assign each store a country:

```
In [4]: store_country = dict(zip(store_numbers,
                                 ['USA', 'USA', 'USA', 'DEU', 'DEU', 'DEU',
                                  'DEU', 'DEU', 'GBR', 'GBR', 'GBR', 'BRA',
                                  'BRA', 'JPN', 'JPN', 'JPN', 'JPN', 'AUS',
                                  'CHN', 'CHN']))
        store_country

Out[4]: {101: 'USA',
         102: 'USA',
         103: 'USA',
         104: 'DEU',
         105: 'DEU',
         106: 'DEU',
         107: 'DEU',
         108: 'DEU',
         109: 'GBR',
         110: 'GBR',
         111: 'GBR',
         112: 'BRA',
         113: 'BRA',
         114: 'JPN',
         115: 'JPN',
         116: 'JPN',
         117: 'JPN',
         118: 'AUS',
         119: 'CHN',
         120: 'CHN'}
```

Here we created a dictionary that maps from store number to the country of that store. We used `zip()`, introduced in Sect. 2.4.10 to combine the store number with a list of countries. We passed that zip object to `dict` to create the `store_country` mapping.

Now we will start filling in the `store_sales` dataframe:

```
In [5]: i = 0
        for store_num in store_numbers:
          for year in [1, 2]:
            for week in range(1, 53):
              store_sales.loc[i, 'store_num'] = store_num
              store_sales.loc[i, 'year'] = year
              store_sales.loc[i, 'week'] = week
```

```
            store_sales.loc[i, 'country'] = store_country[store_num]
            i += 1
```

What did we do here? For every store, we went through both years and every week and set the store number, year, week, and country. We used a set of nested `for` loops to do this. We used `i` to track the row count, incrementing it each time we set the values for a single row. The operator `+=` may be new to you. It *increments* a variable by the amount given; in this case it adds 1 to `i` and stores the result back into `i`. This is a shorter, more readable way to write `i = i + 1`.

We can check the overall data structure with `head()`:

```
In [6]: store_sales.head()
```

```
Out[6]:    store_num  year  week  p1_sales  p2_sales  p1_price  p2_price  \
        0      101.0   1.0   1.0       0.0       0.0       0.0       0.0
        1      101.0   1.0   2.0       0.0       0.0       0.0       0.0
        2      101.0   1.0   3.0       0.0       0.0       0.0       0.0
        3      101.0   1.0   4.0       0.0       0.0       0.0       0.0
        4      101.0   1.0   5.0       0.0       0.0       0.0       0.0

           p1_promo  p2_promo country
        0       0.0       0.0     USA
        1       0.0       0.0     USA
        2       0.0       0.0     USA
        3       0.0       0.0     USA
        4       0.0       0.0     USA
```

All of the specific measures (sales, price, promotion) are shown as missing values (indicated by zeros) because we haven't assigned other values to them yet, while the store numbers, year counters, week counters and country assignments look good.

We can check the `type` of each column in the `dtypes` attribute:

```
In [7]: store_sales.dtypes
```

```
Out[7]: store_num    float64
        year         float64
        week         float64
        p1_sales     float64
        p2_sales     float64
        p1_price     float64
        p2_price     float64
        p1_promo     float64
        p2_promo     float64
        country       object
        dtype: object
```

The types for all of the variables in our dataframe were dictated by the input data. For example, the values assigned to `store_sales.country` were of type `str` and pandas by default stores strings as `object` type:

```
In [8]: type(store_sales.country[0])
```

```
Out[8]: str
```

However, country labels are actually discrete values and not just arbitrary text. So it is better to represent country explicitly as a categorical variable. Similarly, `store_num` is a label, not a number as such. By converting those variables to categorical types, they will be treated as a categorical in subsequent analyses such as regression models. It is good practice to set variable types correctly as they are created; this will help you to avoid errors later.

We redefine `store_sales.store_num` and `store_sales.country` as categorical using the `astype()` method:

```
In [9]: store_sales.country = store_sales.country.astype(
            pd.CategoricalDtype())
        store_sales.store_num = store_sales.store_num.astype(
```

```
            pd.CategoricalDtype())
        print(store_sales.store_num.head())
        print(store_sales.country.head())
```

```
0    101.0
1    101.0
2    101.0
3    101.0
4    101.0
Name: store_num, dtype: category
Categories (20, float64): [101.0, 102.0, 103.0, ..., 118.0, 119.0, 120.0]
0    USA
1    USA
2    USA
3    USA
4    USA
Name: country, dtype: category
Categories (7, object): [AUS, BRA, CHN, DEU, GBR, JPN, USA]
```

```
In [10]: store_sales.dtypes
```

```
Out[10]: store_num    category
         year          float64
         week          float64
         p1_sales      float64
         p2_sales      float64
         p1_price      float64
         p2_price      float64
         p1_promo      float64
         p2_promo      float64
         country      category
         dtype: object
```

`store_num` and `country` are now defined as categories with 20 and 7 levels, respectively.

It is a good idea to inspect dataframes in the first and last rows because mistakes often surface there. You can use `head()` and `tail()` commands to inspect the beginning and end of the dataframe and `sample()` to inspect a random sample (we omit long output from the these commands):

```
In [11]: # Not shown
         store_sales.head(60) # 60 rows can be displayed without truncation;
         store_sales.tail(60) # make sure end looks OK too;
         store_sales.sample(60) # inspecting a random sample is also helpful;
```

It's always useful to debug small steps like this as you go.

3.1.2 Store Data: Simulating Data Points

We complete `store_sales` with random data for *store-by-week* observations of the sales, price, and promotional status (e.g. advertisement, endcap display, etc.) of these two competing products.

Before simulating random data, it is important to set the random number generation *seed* to make the process replicable. After setting a seed, when you draw random samples in the same sequence again, you get exactly the same (*pseudo-*)random numbers. Pseudorandom number generators (PRNGs) are a complex topic whose issues are out of scope here. If you are

using PRNGs for something important you should review the literature; it has been said that whole shelves of journals could be thrown away due to poor usage of random numbers. A starting point to learn more about PRNGs is Knuth (1997).

If you don't set a PRNG seed, numpy.random will select one for you, but you will get different random numbers each time you repeat the process. If you set the seed and execute commands in the order shown in this book, you will get the results that we show.

```
In [12]: np.random.seed(37204)
```

Now we can draw the random data. For each store in each week, we want to randomly determine whether each product was promoted or not. We can do this by drawing from the *binomial distribution*: this counts the number of "heads" in a collection of coin tosses. The coin can be "weighted", meaning it can have any proportion of heads, not just 50%.

To detail that process: we use the `np.random.binomial(n, p, size)` function to draw from the binomial distribution. For every row of the store data, as noted by `size=n_rows`, we draw from a distribution representing the number of heads in a single coin toss (`n=1`) with a coin that has probability `p=0.1` for product 1 and `p=0.15` for product 2.

We will use this distribution to represent promotional status. In other words, we randomly assign 10% likelihood of promotion for product 1, and 15% likelihood for product 2.

```
In [13]: # 10% promoted
         store_sales.p1_promo = np.random.binomial(n=1, p=0.1, size=n_rows)
         # 15% promoted
         store_sales.p2_promo = np.random.binomial(n=1, p=0.15, size=n_rows)
         store_sales.head(10) # how does it look so far? (not shown)
```

We can look at the count of promotions for product 1 to confirm that the values are realistic:

```
In [14]: store_sales.p1_promo.value_counts()
```

```
Out[14]: 0    1871
         1     209
         Name: p1_prom, dtype: int64
```

Next we set a price for each product in each row of the data. We suppose that each product is sold at one of five distinct price points ranging from $2.19 to $3.19 overall. We randomly draw a price for each week by defining a vector with the five price points and using `np.random.choice(a, size, replace)` to draw from it as many times as we have rows of data (`size=n_rows`). The five prices are sampled many times, so we sample with replacement (`replace=True`, which is the default so we don't write it):

```
In [15]: store_sales.p1_price = np.random.choice([2.19, 2.29, 2.49, 2.79,
                                                  2.99],
                                                 size=n_rows)
         store_sales.p2_price = np.random.choice([2.29, 2.49, 2.59, 2.99,
                                                  3.19],
                                                 size=n_rows)
         store_sales.sample(5) # now how does it look?
```

```
Out[15]:      store_num  year  week       p1_sales       p2_sales  p1_price  \
         61        101.0   2.0  10.0   0.000000e+00   0.000000e+00      2.49
         1259      113.0   1.0  12.0   0.000000e+00   0.000000e+00      2.79
         1784      118.0   1.0  17.0  3.326077e-316  3.326077e-316      2.79
         20        101.0   1.0  21.0   0.000000e+00   0.000000e+00      2.49
         1815      118.0   1.0  48.0  3.326128e-316  3.326128e-316      2.79

               p2_price  p1_promo  p2_promo country
         61        2.49         0         0     USA
         1259      2.29         0         0     BRA
         1784      2.49         0         0     AUS
         20        3.19         1         0     USA
         1815      2.99         0         0     AUS
```

We can see that all columns appear correct, except for the sales columns, which we have yet to set. Note that depending on your environment, some of the unset values may not be zero, but rather very small numbers such as 3.326128e–316 (which is $3.32 * 10^{-316}$, or effectively zero). That is because these values are uninitialized and may show *floating point error* (tiny variations due to how the infinite range of real numbers is stored in the finite space allocated to a variable in memory). If you are running this in Colab, these values will display as zeroes.

Question: if *price* occurs at five discrete levels, does that make it a categorical variable? That depends on the analytic question, but in general probably not. We often perform math on price, such as subtracting cost in order to find gross margin, multiplying by units to find total sales, and so forth. Thus, even though it may have only a few unique values, price is a number, not a factor.

Our last step is to simulate the sales figures for each week. We calculate sales as a function of the relative prices of the two products along with the promotional status of each.

Item sales are in unit counts, so we use the Poisson distribution to generate count data: `np.random.poisson(lam, size)`, where `size` is the number of draws and `lam` represents *lambda*, the defining parameter of the Poisson distribution. Lambda represents the *expected*, or mean, value of units per week.

We draw a random Poisson count for each row (`size=n_rows`), and set the mean sales (`lam`) of Product 1 to be higher than that of Product 2:

```
In [16]: # sales data, using poisson (counts) distribution, np.random.poisson()
         # first, the default sales in the absence of promotion
         sales_p1 = np.random.poisson(lam=120, size=n_rows)
         sales_p2 = np.random.poisson(lam=100, size=n_rows)
```

Now we scale those counts up or down according to the relative prices. Price effects often follow a logarithmic function rather than a linear function (Rao 2009), so we use `np.log(price)` here:

```
In [17]: # scale sales according to the ratio of log(price)
        log_p1_price = np.log(store_sales.p1_price)
        log_p2_price = np.log(store_sales.p2_price)

        sales_p1 = sales_p1 * log_p2_price/log_p1_price
        sales_p2 = sales_p2 * log_p1_price/log_p2_price
```

We have assumed that sales vary as the *inverse* ratio of prices. That is, sales of Product 1 go up to the degree that the `log(price)` of Product 1 is lower than the `log(price)` of Product 2.

Finally, we assume that sales get a 30 or 40% lift when each product is promoted in store. We simply multiply the promotional status vector (which comprises all {0, 1} values) by 0.3 or 0.4 respectively, and then multiply the sales vector by that. We use the `floor()` function to drop fractional values and ensure integer counts for weekly unit sales, and put those values into the dataframe:

```
In [18]: # final sales get a 30% or 40% lift when promoted
         store_sales.p1_sales = np.floor(sales_p1 *
                                         (1 + store_sales.p1_promo * 0.3))
         store_sales.p2_sales = np.floor(sales_p2 *
                                         (1 + store_sales.p2_promo * 0.4))
         store_sales.sample(10)
```

Out[18]:

	store_num	year	week	p1_sales	p2_sales	p1_price	p2_price	\
2001	120.0	1.0	26.0	187.0	79.0	2.29	3.19	
1076	111.0	1.0	37.0	114.0	111.0	2.79	2.49	
...								
233	103.0	1.0	26.0	195.0	66.0	2.19	2.99	
1990	120.0	1.0	15.0	146.0	98.0	2.79	2.99	

	p1_promo	p2_promo	country
2001	0	0	CHN
1076	0	0	GBR
...			
233	0	0	USA
1990	0	0	CHN

Inspecting the dataframe, we see that the data look plausible on the surface. Note that we truncated the response, as indicated by the "..."

Thanks to the power of Python, we have created a simulated dataset with 20800 values (2080 rows × 10 columns) using a total of 29 assignment commands. In the next section we explore the data that we created.

3.2 Functions to Summarize a Variable

Observations may comprise either *discrete* data that occurs at specific levels or *continuous* data with many possible values. We look at each type in turn. But first, let's consider an important tool for aggregation: the `groupby()` method.

3.2.1 Language Brief: `groupby()`

What should we do if we want to break out data by factors and summarize it, a process you might know as "cross-tabs" or "pivot tables"? For example, how can we compute the mean sales by store? We have voluminous data (every store by every week by each product) but many marketing purposes only need an aggregate figure such as a total or mean. We will see how to summarize by a factor within the data itself using the `groupby()` command.

`groupby()` is a method on pandas dataframes. The `by` argument specifies the column by which to group, for example `store_num`:

```
In [19]: store_sales.groupby('store_num')

Out[19]: <pandas.core.groupby.SeriesGroupBy object at 0x7f98df746668>
```

That function returns a `SeriesGroupBy` object, which can be saved to a variable or acted upon directly. That object contains each other column within the dataframe, which can be accessed via dot notation. Pandas analytical methods can then be applied to that group, such as `mean()`, `sum()`, etc. Also, `apply()` can be used, allowing any function to be used on the grouping. We can easily calculate the per-store mean:

```
In [20]: store_sales.groupby('store_num').p1_sales.mean()

Out[20]: store_num
         101.0    133.500000
         102.0    138.807692
         103.0    132.682692
         ...
         Name: p1_sales, dtype: float64
```

To group it by more than one factor, use a list of factors. For instance, we can obtain the mean of `p1_sales` by store and by year:

```
In [21]: store_sales.groupby(['store_num', 'year']).p1_sales.mean()

Out[21]: store_num  year
         101.0      1.0         132.538462
                    2.0         134.461538
         102.0      1.0         139.692308
                    2.0         137.923077
         103.0      1.0         130.557692
                    2.0         134.807692
         ...
         Name: p1_sales, dtype: float64
```

A limitation of `groupby()` is that the result is not always easy to read and not structured for reuse. How can we save the results as data to use for other purposes such as plotting?

For a single-level grouping, the output can be cast into a dataframe with the `pandas.DataFrame()` function. But for multi-level groupings, one option is to use the `unstack()` method which pivots the indices and returns a nicely formatted dataframe. "Pivoting" means what had been a set of outputs that look like rows become columns in the new dataframe:

```
In [22]: store_sales.groupby(['store_num', 'year']).p1_sales.mean().unstack()

Out[22]: year                1.0         2.0
         store_num
         101.0        132.538462  134.461538
         102.0        139.692308  137.923077
         103.0        130.557692  134.807692
         ...
```

Another example is computing the total (`sum()`) sales of P1 by country:

```
In [23]: p1_sales_by_country = store_sales.groupby(['country']).p1_sales.sum()
         p1_sales_by_country

Out[23]: country
         AUS    13980.0
         BRA    27857.0
         CHN    27642.0
         DEU    70323.0
         GBR    41915.0
         JPN    54817.0
         USA    42119.0
         Name: p1_sales, dtype: float64
```

We saved the result to `p1_sales_by_country` because we will use it in Sect. 3.4.5 to make a map. `groupby()` is the primary tool for aggregation of data that we will use throughout the book.

3.2.2 *Discrete Variables*

A basic way to describe discrete data is with frequency counts. The `value_counts()` method will count the observed prevalence of each value that occurs in a variable (i.e., a vector or a column in a dataframe). In `store_sales`, we may count how many times product 1 was observed to be on sale at each price point:

```
In [25]: store_sales.p1_price.value_counts()

Out[25]: 2.29    420
         2.19    417
         2.49    416
         2.99    415
         2.79    412
         Name: p1_price, dtype: int64
```

Note, `value_counts()` by default sorts by the descending count. If `sort=False` is passed as an argument, it will not be sorted. Another useful argument is `normalize=True`, which will return proportions rather than counts.

If your counts vary, that may be due to running commands in a different order or setting a different random number seed. The counts shown here assume that the commands have been run in the exact sequence shown in this chapter. There is no problem if your data are modestly different; just remember that it won't match the output here, or try Sect. 3.1.1 again.

One of the most useful features of Python is that most functions produce an object that you can store and use for further commands. So, for example, if you want to store the table that was created by `value_counts()`, you can just assign the same command to a named object:

```
In [26]: p1_table_0 = store_sales.p1_price.value_counts()
         p1_table_0
```

```
Out[26]: 2.29    420
         2.19    417
         2.49    416
         2.99    415
         2.79    412
         Name: p1_price, dtype: int64

In [27]: type(p1_table_0)

Out[27]: pandas.core.series.Series
```

The `type()` command shows us that the object produced by `value_counts()` is a pandas series, as introduced in 2.5.3, which is the pandas *vector* type. A pandas dataframe is composed of an indexed set of series.

We can use the `plot()` method on `p1_table0` to produce a quick plot:

```
In [28]: p1_table_0.plot.bar()
```

You can see from the resulting bar plot in Fig. 3.1 that the product was on sale at each price point roughly the same number of times. However, it is fairly ugly and the labels could be clearer. Later in this chapter we show how to modify a plot to get better results.

An analyst might want to know how often each product was promoted at each price point. The `pandas.crosstab()` function produces counts of observations at each level for two variables, i.e. a two-way *cross tab*:

```
In [29]: pd.crosstab(store_sales.p1_promo, store_sales.p1_price)

Out[29]: p1_price  2.19  2.29  2.49  2.79  2.99
         p1_promo
         0          371   380   378   371   371
         1           46    40    38    41    44
```

However, as discussed in Sect. 3.2.1 a more general approach is using the `groupby()` method:

```
In [30]: store_sales.groupby('p1_promo').p1_price.value_counts().unstack()

Out[30]: p1_price  2.19  2.29  2.49  2.79  2.99
         p1_promo
         0          371   380   378   371   371
         1           46    40    38    41    44
```

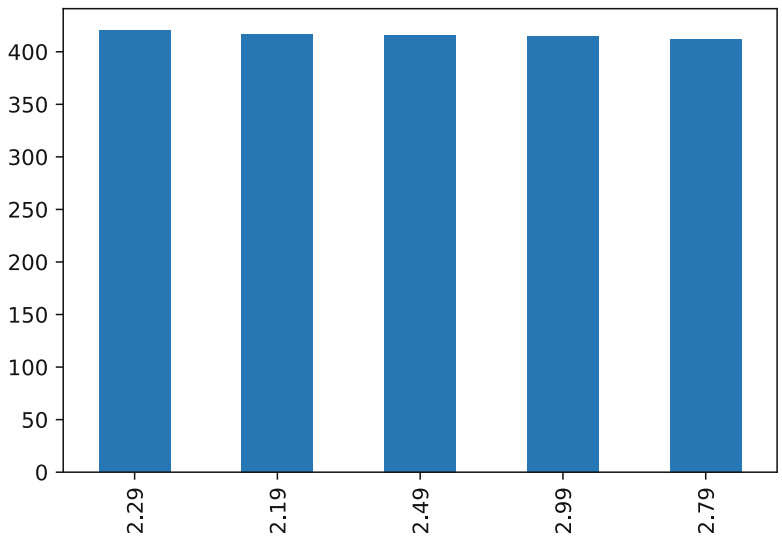

Fig. 3.1 A simple bar plot produced using the `plot()` method on the series containing sales counts. Default charts are sometimes unattractive, but there are many options to make them more attractive and useful

Again, the `unstack()` command is not crucial, but pivots the multi-index series returned by the `groupby()` into a dataframe, which offers a more intuitive display of the data and simplifies downstream analysis.

`groupby()` allows calculation of not just counts, but other functions as well, such as the arithmetic mean. Additionally, the data can be grouped by an arbitrary set of columns. For example, we can calculate the mean price by product, promotion status, and country:

```
In [31]: store_sales.groupby(['p1_promo', 'country']).p1_price.mean().unstack()

Out[31]: country         AUS       BRA       CHN       DEU       GBR      \
         p1_promo
         0          2.515843  2.554398  2.543093  2.553956  2.570212
         1          2.550000  2.454706  2.647143  2.530000  2.586552

         country         JPN       USA
         p1_promo
         0          2.544521  2.533463
         1          2.570000  2.538276
```

Returning to the initial cross tab, at each price level Product 1 is observed to have been promoted approximately 10% of the time (as expected, given how we created the data in Sect. 3.1.1). In fact, we can compute the exact fraction of times product 1 is on promotion at each price point if we assign the result to a variable and then divide it by the total sales by price in `p1_table_0`:

```
In [32]: p1_table_1 = store_sales.groupby('p1_promo').p1_price.value_counts()
         p1_table_1 = p1_table_1.unstack()
         p1_table_1.div(p1_table_0)

Out[32]:                 2.19      2.29      2.49      2.79      2.99
         p1_promo
         0          0.889688  0.904762  0.908654  0.900485  0.893976
         1          0.110312  0.095238  0.091346  0.099515  0.106024
```

The `div()` method applies element-wise division between the series of counts in `p1_table_0` and each row of `p1_table_1` (each of which is a series). Pandas uses the series indices, in this case the price, to match the division.

By combining operating results in this way, you can produce exactly the results you want along with code that can repeat the analysis on demand. This is very helpful to marketing analysts who produce weekly or monthly reports for sales, web traffic, and similar data.

3.2.3 *Continuous Variables*

Counts are useful when we have a small number of categories, but with continuous data it is more helpful to summarize the data in terms of its distribution. The most common way to do that is with mathematical functions that describe the range of the data, its center, the degree to which it is concentrated or dispersed, and specific points that may be of interest (such as the 90th percentile). Table 3.1 lists some pandas functions to calculate statistics for numeric vector data, such as numeric columns in a dataframe.

Table 3.1 Distribution functions that operate on a numeric vector

Describe	Function	Value
Extremes	`min(x)`	Minimum value
	`max(x)`	Maximum value
Central tendency	`mean(x)`	Arithmetic mean
	`median(x)`	Median
Dispersion	`var(x)`	Variance around the mean
	`std(x)`	Standard deviation (`sqrt(var(x))`)
	`mad(x)`	Median absolute deviation (a robust variance estimator)
Points	`quantile(x, q=[...])`	Percentiles

Following are examples of those common functions:

```
In [33]: store_sales.p2_sales.min()

Out[33]: 51.0

In [34]: store_sales.p1_sales.max()

Out[34]: 265.0

In [35]: store_sales.p1_promo.mean()

Out[35]: 0.10048076923076923

In [36]: store_sales.p2_sales.median()

Out[36]: 96.0

In [37]: store_sales.p1_sales.var()

Out[37]: 861.7204626392133

In [38]: store_sales.p1_sales.std()

Out[38]: 29.355075585649807

In [39]: store_sales.p1_sales.mad()

Out[39]: 23.253990384615314

In [40]: store_sales.p1_sales.quantile(q=[0.25, 0.5, 0.75])

Out[40]: 0.25    113.0
         0.50    130.0
         0.75    151.0
         Name: p1_sales, dtype: float64
```

In the case of `quantile()` we have asked for the 25th, 50th, and 75th percentiles using the argument `q=[0.25, 0.5, 0.75]`, which are also known as the *median* (50th percentile, same as the `median()` function) and the edges of the *interquartile range*, the 25th and 75th percentiles.

Change the `q=` argument in `quantile()` to find other quantiles:

```
In [41]: store_sales.p1_sales.quantile(q=[0.05, 0.95])

Out[41]: 0.05     93.0
         0.95    187.0
         Name: p1_sales, dtype: float64

In [42]: store_sales.p1_sales.quantile(q=np.arange(0, 1.1, 0.1))

Out[42]: 0.0     68.0
         0.1    100.0
         0.2    109.0
         0.3    116.0
         0.4    123.0
         0.5    130.0
         0.6    138.0
         0.7    146.0
         0.8    158.0
         0.9    174.0
         1.0    265.0
         Name: p1_sales, dtype: float64
```

The second example here shows that we may use sequences in many places in Python; in this case, we find every 10th percentile by creating a sequence using `numpy.arange(start, stop, step)` to yield the vector `0, 0.1, 0.2 ... 1.0`. Note that `numpy.arange()` is used here rather than the built-in `range()` function since we want decimal values; `range()` only supports integer values. Note also that `numpy.arange()` follows the Python iteration convention (see 2.4.4 that the `start` argument is *inclusive* whereas the `stop` argument is *exclusive*, so to include `1.0` in the vector, we must set the `stop` argument to be equal to the maximum value we desire plus the step (`1.1` in this case).

For skewed and asymmetric distributions that are common in marketing, such as unit sales or household income, the arithmetic `mean()` and standard deviation `std()` may be misleading; in those cases, the `median()` and interquartile range (IQR, the range of the middle 50% of data) are often more useful to summarize a distribution. Pandas does not have a built-in IQR function, but we can create one and apply it to our data:

```
In [43]: def iqr(x):
           return x.quantile(0.75) - x.quantile(0.25)
         iqr(store_sales.p1_sales)

Out[43]: 38.0
```

Notice that when we use the `iqr()` function that we wrote, the syntax is `iqr(store_sales.p1_sales)`. For a built-in method associated with the `DataFrame` class like `mean()`, the syntax is `store_sales.p1_sales.mean()`.

Suppose we wanted a summary of the sales for product 1 and product 2 based on their median and interquartile range. We might assemble these summary statistics into a dataframe that is easier to read than the one-line-at-a-time output above. We create a dataframe to hold our summary statistics and then populate it using functions from 3.1. We name the columns and rows, and fill in the cells with function values:

```
In [44]: pd.DataFrame([[store_sales.p1_sales.median(),
                         store_sales.p2_sales.median()],
                        [iqr(store_sales.p1_sales),
                         iqr(store_sales.p2_sales)]],
                       index=['Median sales', 'IQR'],
                       columns=['p1_sales', 'p2_sales'])

Out[44]:                p1_sales  p2_sales
         Median sales      130.0      96.0
         IQR                38.0      33.0
```

With this custom summary we see that median sales are higher for product 1 (130 versus 96) and that the variation in sales of product 1 (the IQR across observations by week) is also higher. Once we have this code, we can run it again the next time we have new sales data to produce a revised version of our table of summary statistics. Such code might be a good candidate for a custom function you can reuse (see Sects. 2.4.8 and 10.3.1). We'll see a shorter way to create this summary in Sect. 3.3.3.

3.3 Summarizing Dataframes

As useful as functions such as `mean()` and `quantile()` are, it is tedious to apply them one at a time to columns of a large dataframe, as we did with the summary table above. Pandas provides a variety of ways to summarize dataframes without writing extensive code. We describe two approaches: the basic `describe()` command and the Pandas approach to iterating over variables with `apply()`.

3.3.1 `describe()`

As we saw in Sect. 2.5.3, `describe()` is a good way to do a preliminary inspection of a dataframe or other object. When you use `describe()` on a dataframe, it reports a few descriptive statistics for every variable:

```
In [45]: store_sales.describe()
```

```
Out[45]:              year        week     p1_sales     p2_sales  \
         count  2080.00000  2080.00000  2080.000000  2080.000000
         mean      1.50000    26.50000   133.967788    99.911058
         std       0.50012    15.01194    29.355076    24.453788
         min       1.00000     1.00000    68.000000    51.000000
         25%       1.00000    13.75000   113.000000    82.000000
         50%       1.50000    26.50000   130.000000    96.000000
         75%       2.00000    39.25000   151.000000   115.000000
         max       2.00000    52.00000   265.000000   210.000000

                   p1_price     p2_price     p1_promo     p2_promo
         count  2080.000000  2080.000000  2080.000000  2080.000000
         mean      2.548654     2.716106     0.100481     0.145673
         std       0.300716     0.333559     0.300712     0.352863
         min       2.190000     2.290000     0.000000     0.000000
         25%       2.290000     2.490000     0.000000     0.000000
         50%       2.490000     2.590000     0.000000     0.000000
         75%       2.790000     2.990000     0.000000     0.000000
         max       2.990000     3.190000     1.000000     1.000000
```

`describe()` works similarly for single vectors:

```
In [46]: store_sales.p1_price.describe()
```

```
Out[46]: count    2080.000000
         mean        2.548654
         std         0.300716
         min         2.190000
         25%         2.290000
         50%         2.490000
         75%         2.790000
         max         2.990000
         Name: p1_price, dtype: float64
```

Perhaps the most important use for `describe()` is this: *after importing data, use* `describe()` *to do a quick quality check*. Check the `min` and `max` for outliers or miskeyed data, and check to see that the `mean` and `50%` (median) are reasonable and similar to one another (if you expect them to be similar, of course). This simple inspection often turns up errors in the data!

3.3.2 Recommended Approach to Inspecting Data

We can now recommend a general approach to inspecting a dataset after compiling or importing it; replace "`my_data`" and "`DATA`" with the names of your objects:

1. Import your data with `pandas.read_csv()` or another appropriate function and check that the importation process gives no errors.
2. Convert it to a dataframe if needed (`my_data=pd.DataFrame(DATA)`) and set column names (`my_data.columns = [...]`) if needed.
3. Examine `shape` to check that the dataframe has the expected number of rows and columns.
4. Use `head()` and `tail()` to check the first few and last few rows; make sure that header rows at the beginning and blank rows at the end were not included accidentally. Also check that no good rows were skipped at the beginning.
5. Use `sample()` to examine a few sets of random rows.
6. Check the dataframe structure with `dtypes` to ensure that variable types are appropriate. Change the type of variables—especially to categorical types—as necessary.
7. Run `describe()` and look for unexpected values, especially `min`, `max`, `count` that are unexpected.

3.3.3 `apply()` *

As we've seen, it is very useful to employ operations on each column of a dataframe, such as finding the mean on all numeric columns (columns 3–8):

```
In [47]: store_sales.iloc[:, 3:9].mean()

Out[47]: p1_sales    133.967788
         p2_sales     99.911058
         p1_price      2.548654
         p2_price      2.716106
         p1_promo      0.100481
         p2_promo      0.145673
         dtype: float64
```

We can also use the `axis` argument to run the function across rows rather than columns (the default is columns: `axis=0`):

```
In [48]: store_sales.iloc[:, 3:9].mean(axis=1).head()

Out[48]: 0    39.830000
         1    40.780000
         2    42.363333
         3    37.663333
         4    43.846667
         dtype: float64
```

That isn't very useful in this dataset, but can certainly be invaluable.

But what if we want to make a calculation that isn't a default pandas dataframe method, like our `iqr()` function? Let's try the syntax we've been using:

```
In [49]: store_sales.iloc[:, 3:9].iqr()


         ---------------------------------------------------------------------------

         AttributeError                            Traceback (most recent call last)

         <ipython-input-46-1ff1629f4f16> in <module>()
     ----> 1 store_sales.iloc[:, 3:9].iqr()
         ...
         AttributeError: 'DataFrame' object has no attribute 'iqr'
```

We get an AttributeError because we were trying to use `iqr()` as a method, which is effectively an attribute on the dataframe object that acts as a function on itself (see 2.4.8 for more information). But `iqr()` is a function that we defined. How do we apply that to the dataframe columns?

An advanced and powerful tool in pandas is the `apply()` method. `apply(function, axis, ...)` runs any function that you specify on each of the rows (when `axis=1`) and/or columns (the default, or when `axis=0`) of a dataframe. This allows any function to be applied to all columns or rows of the dataframe:

```
In [50]: store_sales.iloc[:, 3:9].apply(iqr)

Out[50]: p1_sales    38.0
         p2_sales    33.0
         p1_price     0.5
         p2_price     0.5
         p1_promo     0.0
         p2_promo     0.0
         dtype: float64
```

```
In [51]: store_sales.iloc[:, 3:9].apply(iqr, axis=1).head()

Out[51]: 0    81.750
         1    59.425
         2    60.200
         3    76.500
         4    84.775
         dtype: float64
```

What if we want to know something more complex? We can define an ad hoc *anonymous function*, known in Python as a *lambda* function. Imagine that we are checking data and wish to know the difference between the mean and median of each variable, perhaps to flag skew in the data. Lambda function to the rescue! We can `apply()` that calculation to multiple columns using an lambda function:

```
In [52]: store_sales.iloc[:, 3:9].apply(lambda x: x.mean() - x.median())

Out[52]: p1_sales    3.967788
         p2_sales    3.911058
         p1_price    0.058654
         p2_price    0.126106
         p1_promo    0.100481
         p2_promo    0.145673
         dtype: float64
```

This analysis shows that the mean of `p1_sales` and the mean of `p2_sales` are larger than the median by about three sales per week, which suggests there is a right-hand tail to the distribution. That is, there are some weeks with very high sales that pull the mean up. (Note that we only use this to illustrate an anonymous function; there are better, more specialized tests of skew, such as the `skew()` method.)

Experienced programmers: your first instinct, based on experience with procedural programming languages, might be to solve the preceding problem with a `for()` loop that iterates the calculation across columns. That is possible in Python, of course, but less efficient and less Pythonic in this case. Instead, try to think in terms of functions that are applied across data as we do here.

`apply()` also works on series objects, where the function is applied element-wise rather than to the series as a whole:

```
In [53]: store_sales.p1_sales.apply(lambda x: 'high' if x > 130 else 'low')[:5]

Out[53]: 0     low
         1    high
         2    high
         3     low
         4     low
         Name: p1_sales, dtype: object
```

All of these functions, including `apply()` and `describe()` return values that can be assigned to an object. For example, using `apply`, we can produce our customized summary dataframe from Sect. 3.2.3 more efficiently:

```
In [54]: pd.DataFrame([store_sales[['p1_sales', 'p2_sales']].median(),
                       store_sales[['p1_sales', 'p2_sales']].apply(iqr)],
                      index=['Median sales', 'IQR'])

Out[54]:                p1_sales  p2_sales
         Median sales      130.0      96.0
         IQR                38.0      33.0
```

If there were many products instead of just two, the code would still work if we changed the number of allocated rows, and `apply()` would run automatically across all of them.

Now that we know how to summarize data with statistics, it is time to visualize it.

3.4 Single Variable Visualization

The standard plotting library in Python is matplotlib. In concert with NumPy (Sect. 2.5.1), it produces a MATLAB-like plotting interface and is a dependency of most other plotting libraries (such as seaborn). There are many other plotting libraries to explore beyond matplotlib, such as seaborn, ggplot, Bokeh, Altair, and many other.

Pandas dataframes and series can be plotted using matplotlib-based methods. Here we examine histograms, density plots, and box plots, and take an initial look at more complex graphics including maps. Later chapters build on these foundational plots and introduce more that are available in other packages.

A quick note about plotting in Python. If you are using a Colab or Jupyter notebook, the plot will be shown inline (for Jupyter, the magic `%matplotlib inline` is required; for Colab this is default behavior). If you are using the Python command line interface (CLI) or running it as a script, the command `matplotlib.pyplot.show()` must follow the plot for it to appear (by convention, matplotlib.pyplot is imported as plt, in which case the command is `plt.show()`). Throughout this book, we omit `plt.show()`, but be sure to add it if you are not using a notebook interface.

3.4.1 Histograms

A fundamental plot for a single continuous variable is the *histogram*. Such a plot can be produced using the `hist()` method on a series (i.e. a column):

```
In [55]: store_sales.p1_sales.hist()
```

The result is shown in Fig. 3.2. It is not a bad start. We see that the weekly sales for product 1 range from a little less than 100 to a bit more than 250. But there are no axis labels present!

That plot was easy to make but the visual elements are less than pleasing, so we will improve it. For future charts, we will show either the basic chart or the final one, and will not demonstrate the successive steps to build one up. However, we go through the intermediate steps here so you can see the process of how to evolve a graphic in Python.

As you work through these steps, there are four things you should understand about graphics in Python:

- Python graphics are produced through commands that often seem tedious and require trial and iteration.
- Notebooks with inline plotting enabled are very convenient for developing figures, as they allow for rapid iteration.
- Despite the difficulties, Python graphics can be very high quality, portable in format, and even beautiful.
- Once you have code for a useful graphic, you can reuse it with new data. It is often helpful to tinker with previous plotting code when building a new plot, rather than recreating it.

Our first improvement to Fig. 3.2 is to change the title and axis labels. We do that by importing matplotlib.pyplot and using several functions:

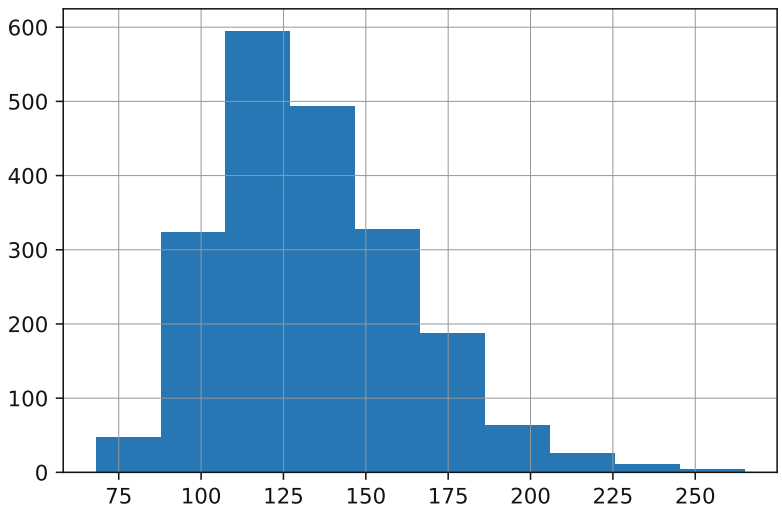

Fig. 3.2 A basic histogram using `hist()`

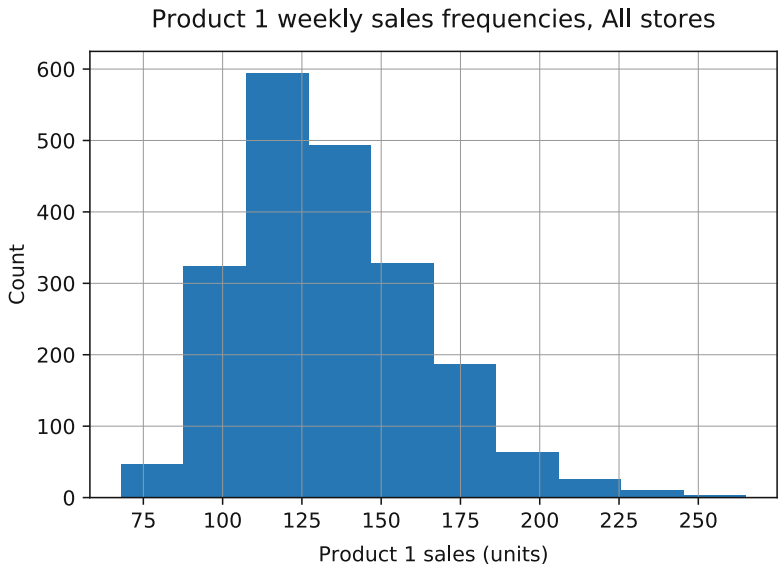

Fig. 3.3 The same histogram, with improved labels

`plt.title('...')` : sets the main title
`plt.xlabel('...')` : sets the x-axis label
`plt.ylabel('...')` : sets the y-axis label

We add the title and axis labels to our plot command:

```
In [56]: import matplotlib.pyplot as plt
         store_sales.p1_sales.hist()
         plt.title('Product 1 weekly sales frequencies, All stores')
         plt.xlabel('Product 1 sales (units)')
         plt.ylabel('Count')
```

The result is shown in Fig. 3.3 and is improved but not perfect; it would be nice to have more granularity (more bars) in the histogram. While we're at it, let's tweak the appearance by removing the background as well as coloring and adding borders to the bars. Here are a few additional arguments we can use with the `hist()` method:

`bins=NUM` : call for NUM bars in the result
`facecolor="..."` : color the bars
`edgecolor="..."` : color the bar borders

And the function `plt.box(False)` removes the plot background and `plt.grid(False)` removes the grid.

Additionally, the font is a bit small. We can set the font in the `rcParams` module, which will persist throughout the rest of the notebook:

```
In [57]: plt.rcParams.update({'font.size': 12})
```

There are many different parameters that can be set in rcParams, which are defined in the matplotlibrc file. A sample file can be found at https://matplotlib.org/3.1.1/tutorials/introductory/customizing.html#matplotlibrc-sample.

When specifying colors, matplotlib knows many by name, including the most common ones in English ("red", "blue", "green", etc.) and less common (such as "coral" and "papayawhip"). Many of these can be modified by adding the prefix "light" or "dark" (thus "lightgray", "darkred", and so forth). For a list of built-in color names, run the `matplotlib.colors.get_named_colors_mapping()` command.

For a set of common colors, there are single character representations, such as "k" for black, "r" for red, "g" for green, etc.

Colors can also be specified as RGB tuples, RGBA tuples, hex RGB or RGBA strings, and other ways. See https://matplotlib.org/tutorials/colors/colors.html for more information.

We add these modifications to our code, with the result shown in Fig. 3.4:

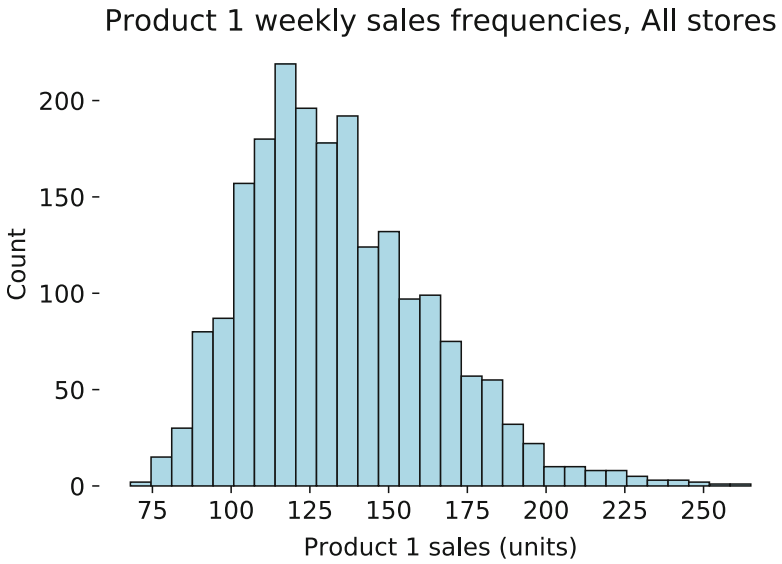

Fig. 3.4 The histogram after adjusting color and dividing the counts into a larger number of bins

```
In [58]: store_sales.p1_sales.hist(bins=30,
                                   edgecolor='k',
                                   facecolor='lightblue')
         plt.title('Product 1 weekly sales frequencies, All stores')
         plt.xlabel('Product 1 sales (units)')
         plt.ylabel('Count')
         plt.grid(False)
         plt.box(False)
```

Comparing Fig. 3.4 with 3.3 we notice a new problem: the y-axis value for the height of the bars changes according to count. The count depends on the number of bins and on the sample size. We can make it absolute by using *relative frequencies* (technically, the *density* estimate) instead of counts for each point. This makes the y-axis comparable across different sized samples.

Figure 3.4 has reasonable x-axis tick marks, but imagine we want more or fewer. Instead of using the default *tick marks* (axis numbers) for `hist()`, we can specify the x-axis number explicitly. The argument for relative frequency is `density=True` and the x-axis numbers are specified using the `plt.xticks()` function:

```
In [59]: store_sales.p1_sales.hist(bins=30,
                                   edgecolor='k',
                                   facecolor='lightblue',
                                   density=True)
         plt.title('Product 1 weekly sales frequencies, All stores')
         plt.xlabel('Product 1 sales (units)')
         plt.ylabel('Relative frequency')
         plt.xticks(range(60, 300, 20))
         plt.grid(False)
         plt.box(False)
```

With `plt.xticks()`, we have to tell it where to put the labels, which may be made with the `range()` function to generate a sequence of numbers. We could also specify a label to appear at each of those points rather than a number (see Fig. 3.9). The updated histogram is shown in Fig. 3.5. It is looking good now!

Finally, we add a smoothed estimation line. To do this, we use the `density()` plotting method on the `p1_sales` series. The density plot disrupted the axis autoscaling, so we also use `plt.xlim()` to specify the x-axis range.

```
In [60]: store_sales.p1_sales.hist(bins=30,
                                   edgecolor='k',
                                   facecolor='lightblue',
```

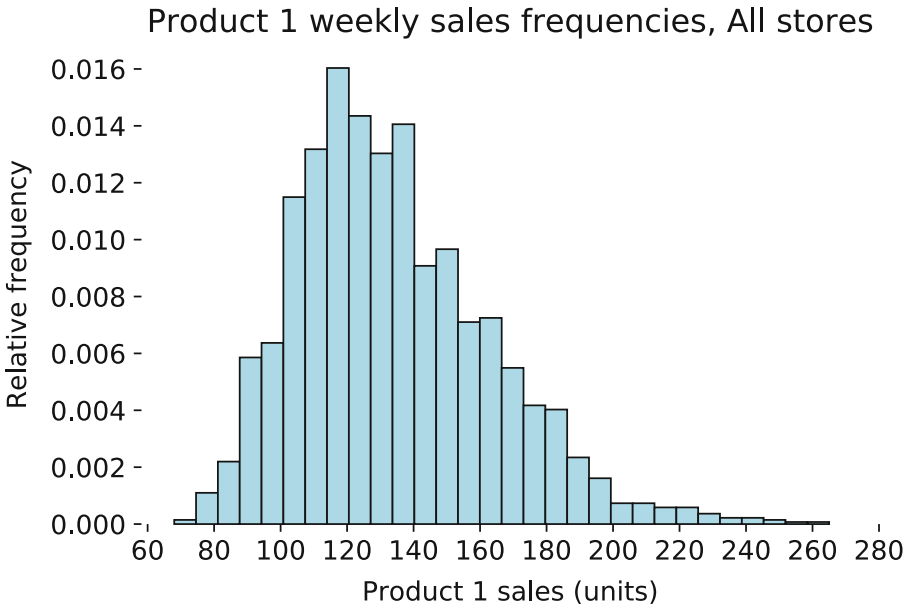

Fig. 3.5 Histogram with relative frequencies (density estimates) and improved axis tick mark labels

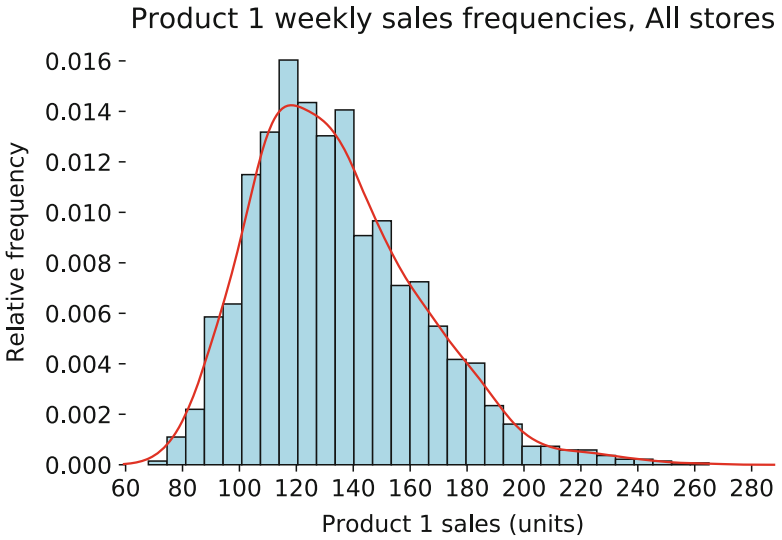

Fig. 3.6 Final histogram with density curve

```
                                density=True)
store_sales.p1_sales.plot.density(color='red')
plt.title('Product 1 weekly sales frequencies, All stores')
plt.xlabel('Product 1 sales (units)')
plt.ylabel('Relative frequency')
plt.xticks(range(60, 300, 20))
plt.xlim((60, 290))
plt.box(False)
```

Figure 3.6 is now very informative. Even someone who is unfamiliar with the data can see that this plot describes weekly sales for product 1 and that the typical sales range from about 80 to 200.

The process we have shown to produce this graphic is representative of how analysts use Python for visualization. You start with a default plot, change some of the options, and use functions like `plt.title()` and `density()` to alter features of the plot with complete control. Although at first this will seem cumbersome compared to the drag-and-drop methods of

other visualization tools, it really isn't much more time consuming if you use a code editor and become familiar with the plotting functions' examples and help files. It has the great advantage that once you've written the code, you can reuse it with different data.

Exercise Modify the code to create the same histogram for product 2. It requires only minor change to the code whereas with a drag-and-drop tool, you would start all over. If you produce a plot often, you could even write it as a custom function.

3.4.2 Boxplots

Boxplots are a compact way to represent a distribution. The pandas `box()` method is straightforward; we add labels, use the argument `vert=False` to rotate the plot 90° to look better, and use `sym='k.'` to specify the outlier marker:

```
In [61]: p = store_sales.p2_sales.plot.box(vert=False, sym='k.')
         plt.title('Weekly sales of P2, All stores')
         plt.xlabel('Weekly sales')
         p.set_facecolor('w')
```

Figure 3.7 shows the resulting graphic. The boxplot presents the distribution more compactly than a histogram. The median is the center line while the 25th and 75th percentiles define the *box*. The outer lines are *whiskers* at the points of the most extreme values that are no more than 1.5 times the width of the box away from the box. Points beyond the whiskers are outliers drawn as individual points. This is also known as a *Tukey boxplot* (after the statistician, Tukey) or as a *box-and-whiskers* plot.

Boxplots are even more useful when you compare distributions by some other factor. How do different stores compare on sales of product 2? The `boxplot()` method makes it easy to compare these with the `by` argument, which specifies the column by which to group. The `column` argument indicates the column represented by the boxplot distribution, `p2_sales` in this case. These correspond to the response variable `p2_sales` which we plot with regards to the explanatory variable `store_num`:

```
In [62]: store_sales.boxplot(column='p2_sales', by='store_num', vert=False,
                              sym='k.')
         plt.suptitle('')
         plt.title('Weekly sales of p2 by store')
         plt.xlabel('Weekly unit sales')
         plt.ylabel('Store')
         plt.box(False)
```

Fig. 3.7 A simple example of `boxplot()`

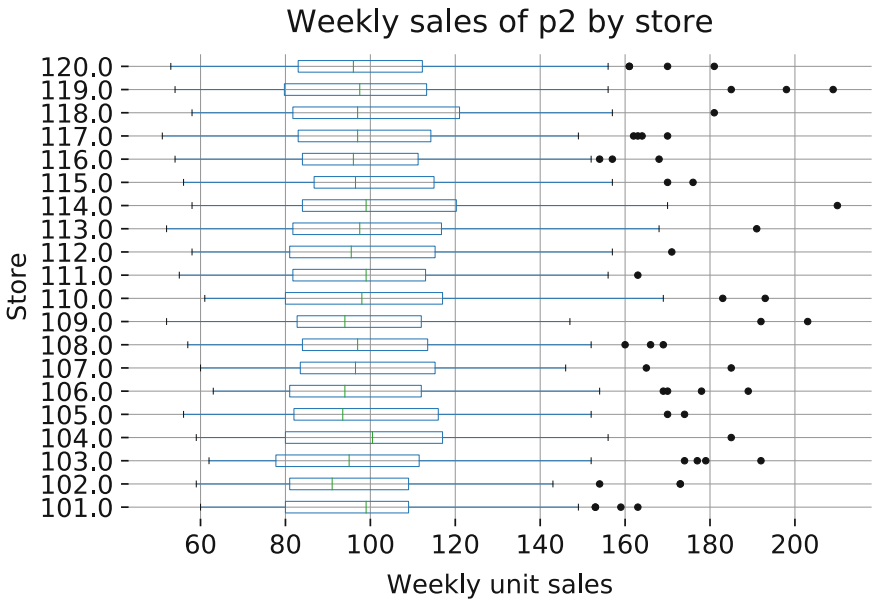

Fig. 3.8 `boxplot()` of sales by store

The result is Fig. 3.8, where stores are roughly similar in sales of product 2 (this is not a statistical test of difference, just a visualization). Note that `plt.suptitle()` removes the default title that the `boxplot()` method adds, as we'd prefer to specify a more informative title.

We see in Fig. 3.8 that the stores are similar in unit sales of P2, but do P2 sales differ in relation to in-store *promotion*? In this case, our explanatory variable would be the promotion variable for P2, so we use `boxplot()` now replacing `store_num` with the promotion variable `p2_promo`.

```
In [63]: store_sales.boxplot(column='p2_sales', by='p2_promo', vert=False,
                            sym='k.')
         plt.suptitle('')
         plt.title('Weekly sales of p2 with and without promotion')
         plt.xlabel('Weekly unit sales')
         plt.ylabel('P2 promo in store?')
         plt.yticks([1, 2], ['No', 'Yes'])
         plt.box(False)
```

The result is shown in Fig. 3.9. There is a clear visual difference in sales on the basis of in-store promotion!

To wrap up: boxplots are powerful tools to visualize a distribution and make it easy to explore how an outcome variable is related to another factor. In Chaps. 4 and 5 we explore many more ways to examine data association and statistical tests of relationships.

*3.4.3 QQ Plot to Check Normality**

This is an optional section on a graphical method to evaluate a distribution more formally. You may wish to skip to Sect. 3.4.4 on cumulative distributions or Sect. 3.4.5 that describes how to create maps in Python.

Quantile-quantile (QQ) plots are a good way to check one's data against a distribution that you think it should come from. Some common statistics such as the correlation coefficient r (to be precise, the *Pearson product-moment correlation coefficient*) are interpreted under an assumption that data are normally distributed. A QQ plot can confirm that the distribution is, in fact, normal by plotting the *observed* quantiles of your data against the quantiles that would be *expected* for a normal distribution.

To do this, we can use the `probplot()` function from the scipy.stats library, which compares data vs. a specified distribution, for example the normal distribution. We check `p1_sales` to see whether it is normally distributed:

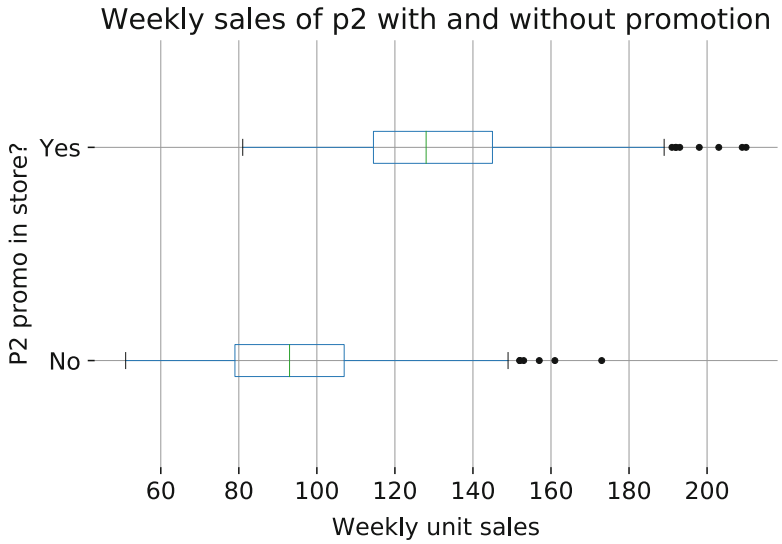

Fig. 3.9 Boxplot of product sales by promotion status

```
In [64]: from scipy import stats
         plt.figure(figsize=(7,7))
         stats.probplot(store_sales.p1_sales, dist='norm', plot=plt)
```

The QQ plot is shown in Fig. 3.10. The distribution of `p1_sales` is far from the line at the ends, suggesting that the data are not normally distributed. The upward curving shape is typical of data with high positive skew.

What should you do in this case? If you are using models or statistical functions that assume normally distributed data, you might wish to transform your data. As we've already noted, a common pattern in marketing data is a logarithmic distribution. We examine whether `p1_sales` is more approximately normal after a `log()` transform:

```
In [65]: plt.figure(figsize=(7,7))
         stats.probplot(np.log(store_sales.p1_sales), dist='norm', plot=plt)
```

The QQ plot for `log(p1_sales)` is shown in Fig. 3.11. The points are much closer to the solid line, indicating that the distribution of `log(store_sales.p1_sales)` is more approximately normal than the untransformed variable.

We recommend that you use `scipy.stats.probplot()` regularly to test assumptions about your data distribution. Web search will reveal further examples of common patterns that appear in QQ plots and how to interpret them.

*3.4.4 Cumulative Distribution**

This is another optional section, but one that can be quite useful. If you wish to skip ahead to cover just the fundamentals, you should continue with Sect. 3.4.5.

Another useful univariate plot involves the impressively named *empirical cumulative distribution function* (ECDF). It is less complex than it sounds and is simply a plot that shows the cumulative proportion of data values in your sample. This is an easy way to inspect a distribution and to read off percentile values.

We plot the ECDF of `p1_sales` by combining a few steps. We can use the `ECDF()` function from the statsmodels library to find the ECDF of the data. Then we put the results into `plot()`, adding options such as titles.

Suppose we also want to know the value for which 90% of weekly sales of P1will be lower than that value, i.e., the 90th percentile for weekly sales of P1. We can use `plot()` to add vertical and horizontal lines at the 90th percentile. We do not have to specify the exact value at which to draw a line for the 90th percentile; instead we use `quantile( , pr=0.9)` to find it. The `'k-'` positional argument indicates that we want the lines to be black and dashed and the `alpha=0.5` sets the transparency level of the lines to 50%:

```
In [66]: from statsmodels.distributions.empirical_distribution import ECDF
         e = ECDF(store_sales.p1_sales)
```

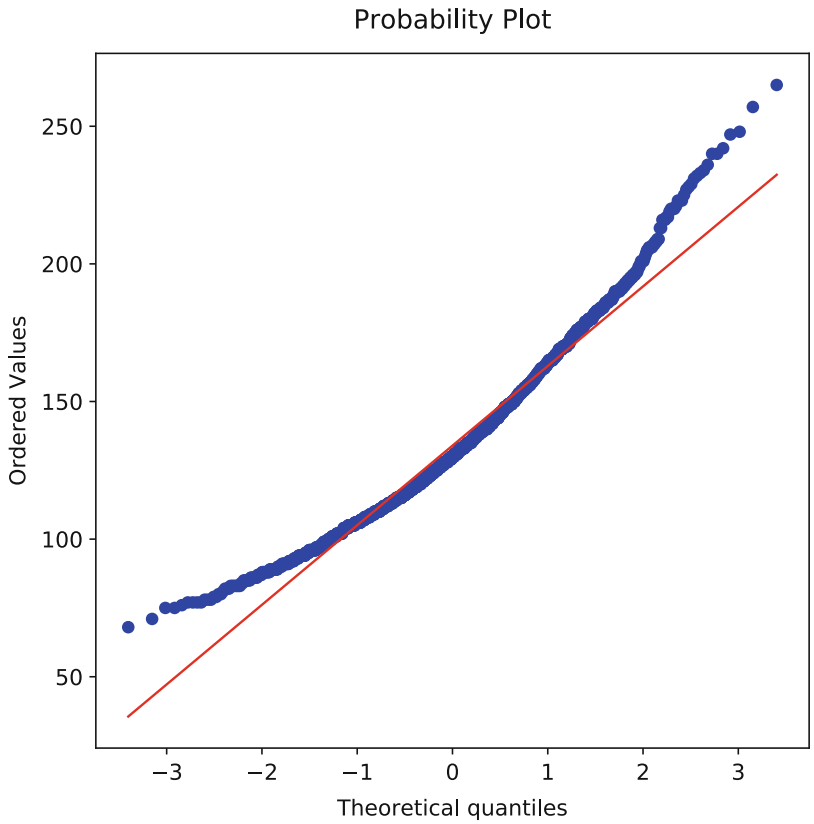

Fig. 3.10 QQ plot to check distribution. The tails of the distribution bow away from the line that represents an exact normal distribution, showing that the distribution of `p1_sales` is skewed

```
plt.subplot(2,1,1)
plt.plot(e.x, e.y)
plt.title('Cumulative distribution of p1 weekly sales')
plt.ylabel('Cumulative proportion')
plt.plot([60, 270], [0.9, 0.9], 'k--', alpha=0.5)
plt.plot([store_sales.p1_sales.quantile(.9),
          store_sales.p1_sales.quantile(.9)],
         [0, 1], 'k--', alpha=0.5)
plt.box(False)
```

We can see the resulting plot in Fig. 3.12.

`ECDF()` calculation is fairly simple. We can replicate the plot through a manual calculation (figure not shown):

```
In [67]: ecdf_x = store_sales.p1_sales.sort_values()
         ecdf_y = np.arange(0, 1, 1/len(store_sales.p1_sales))
         plt.subplot(2,1,2)
         plt.plot(ecdf_x, ecdf_y)
         plt.xlabel('P1 weekly sales, all stores')
         plt.ylabel('Cumulative proportion')
         plt.plot([60, 270], [0.9, 0.9], 'k--', alpha=0.5)
         plt.plot([store_sales.p1_sales.quantile(.9),
                   store_sales.p1_sales.quantile(.9)],
                  [0, 1], 'k--', alpha=0.5)
         plt.box(False)
```

If you run the code block above, you will get a plot just like that Fig. 3.12.

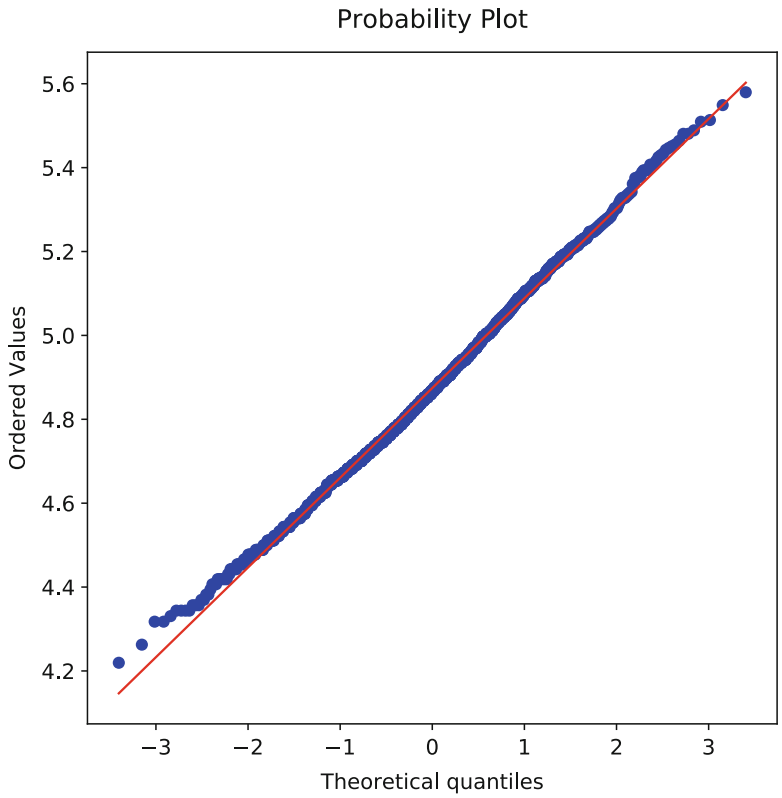

Fig. 3.11 QQ plot for the data after `log()` transformation. The sales figures are now much better aligned with the solid line that represents an exact normal distribution

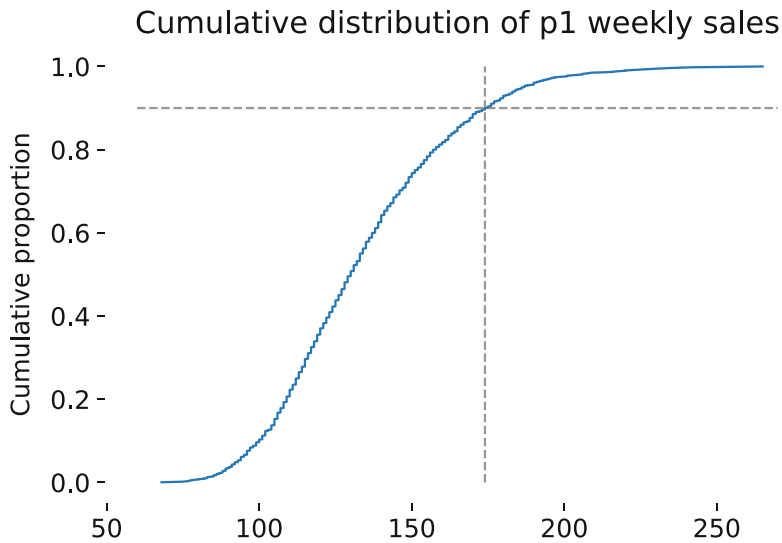

Fig. 3.12 Cumulative distribution plot with lines to emphasize the 90th percentile. The chart identifies that 90% of weekly sales are lower than or equal to 171 units. Other values are easy to read off the chart. For instance, in roughly 10% of weeks fewer than 100 units are sold, and in the upper 5% more than 200 units are sold

We often use cumulative distribution plots both for data exploration and for presenting data to others. They are a good way to highlight data features such as discontinuities in the data, long tails, and specific points of interest.

3.4.5 *Maps*

We often need to plot marketing data on a map. A common variety is a *choropleth* map, which uses graphics or color to indicate values of a variable such as income or sales. We consider how to do this for a world map using the `cartopy` library (Met Office 2010–2015).

cartopy is not a standard numerical Python library, so we may need to install it (see Sect. 2.4.9. In Colab, we can do this using the `!` operator to access the shell, allowing installation of the package with `pip` (output not shown):

```
In [68]: !apt-get -qq install python-cartopy python3-cartopy
         !pip uninstall -y shapely
         !pip install shapely --no-binary shapely
```

Here is a routine example. Suppose that we want to chart the total sales by country. We use `aggregate()` as in Sect. 3.2.1 to find the total sales of P1 by country:

We can then use cartopy functions to overlay sales data on a map. Note that this requires code more advanced than we've seen up to this point. We're not going to go through it in detail, but include it as a demonstration of the power of Python to analyze and visualize complex data.

```
In [69]: from cartopy.io import shapereader
         from cartopy import crs

         plt.figure(figsize=(16,6))
         ax = plt.axes(projection=crs.PlateCarree())

         shpfile = shapereader.natural_earth(resolution='110m',
                                             category='cultural',
                                             name='admin_0_countries')
         reader = shapereader.Reader(shpfile)
         countries = reader.records()
         max_sales = p1_sales_by_country.max()
         for country in countries:
           country_name = country.attributes['ADM0_A3']
           if country_name in p1_sales_by_country:
             ax.add_geometries(country.geometry, crs.PlateCarree(),
               facecolor=plt.cm.Greens(p1_sales_by_country[country_name]
                                       /max_sales),
               edgecolor='k')
           else:
             ax.add_geometries(country.geometry, crs.PlateCarree(),
               facecolor='w',
               edgecolor='k')
```

The result is shown in Fig. 3.13, known as a *choropleth* chart.

Although such maps are popular, they can be misleading. In *The Wall Street Journal Guide to Information Graphics*, Wong explains that choropleth charts are problematic because they confuse geographic area with scaled quantities (Wong (2013), p. 90). For instance, in Fig. 3.13, China is more prominent than Japan not because it has a higher value but because it is larger in size. We acknowledge the need for caution despite the popularity of such maps.

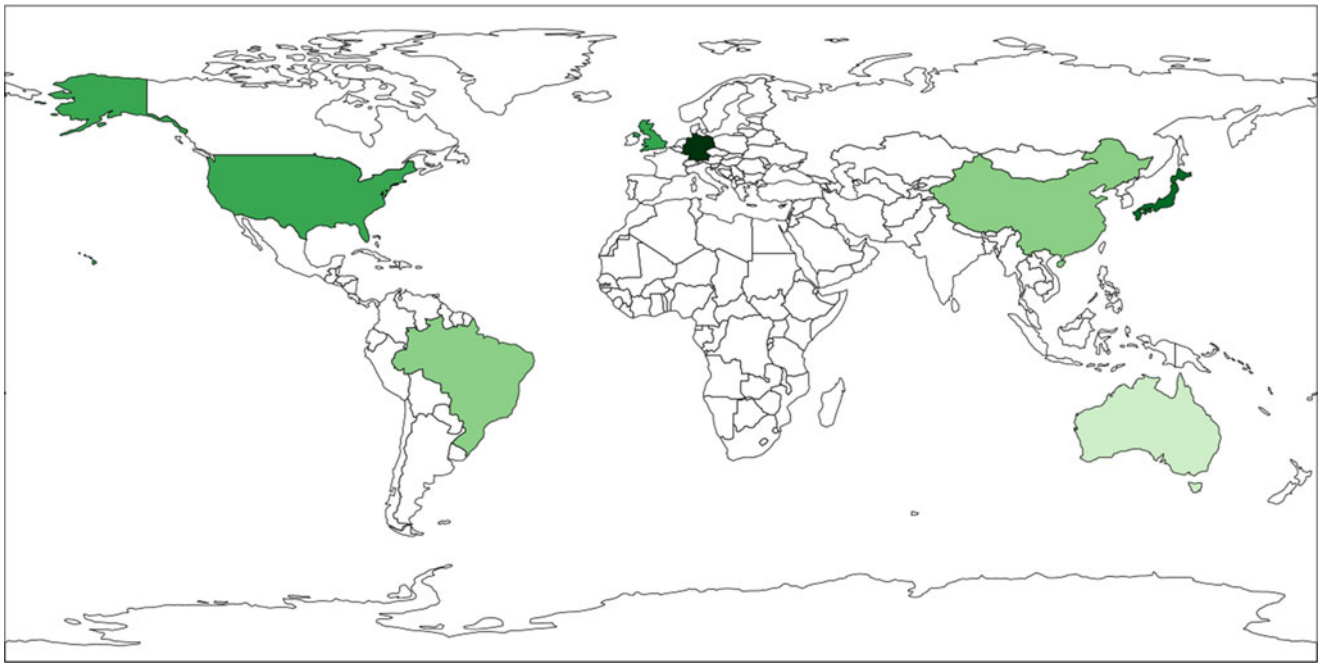

Fig. 3.13 World map for P1 sales by country, using `cartopy`

3.5 Learning More*

Plotting We demonstrate plotting in Python throughout this book. Python has multiple, often disjoint solutions for plotting and in this text we use plots as appropriate without going deeply into their details. The *base* plotting system is `matplotlib.pyplot`, which is leveraged by most other plotting libraries and is integrated into pandas.

Wong's *The Wall Street Journal Guide to Information Graphics* (Wong 2013) presents fundamentals of good style for effective graphics in any business context (not specific to Python).

Maps Producing maps in Python is an especially complex topic. Maps require three essential components: *shape files* that define the borders of areas (such as country or city boundaries); *spatial translation* of one's data (for instance, a database to match Zip codes in your data to the relevant areas on a map); and *plotting software* to perform the actual plotting. Python libraries such as `cartopy` usually provide access to all three of those elements.

3.6 Key Points

The following guidelines and pointers will help you to describe data accurately and quickly:

- Consider simulating data before collecting it, in order to test your assumptions and develop initial analysis code (Sect. 3.1).
- Always check your data for proper structure and data quality using `dtypes`, `head()`, `describe()`, and other basic inspection commands (Sect. 3.3.2).
- Describe discrete (categorical) data with `value_counts()` (Sect. 3.2.2) and inspect continuous data with `describe()` (Sect. 3.3).
- Histograms (Sect. 3.4.1) and boxplots (Sect. 3.4.2) are good for initial data visualization.
- Use `groupby()` to break out your data by grouping variables (Sect. 3.2.1).
- Advanced visualization methods include cumulative distribution (Sect. 3.4.4), normality checks (Sect. 3.4.3), and mapping (Sect. 3.4.5).