Chapter 2. Interactive Data Analysis with pandas

In this chapter, we will cover the following topics:

- Exploring a dataset in the Notebook
- Manipulating data
- Complex operations

We'll see how to load, explore, and visualize a real-world dataset with pandas, matplotlib, and seaborn, all in the Notebook. We will also perform data manipulations efficiently.

Exploring a dataset in the Notebook

Here, we will explore a dataset containing the taxi trips made in New York City in 2013. Maintained by the New York City Taxi and Limousine Commission, this 50GB dataset contains the date, time, geographical coordinates of pickup and dropoff locations, fare, and other information for 170 million taxi trips.

To keep the analysis times reasonable, we will analyze a subset of this dataset containing 0.5% of all trips (about 850,000 rides). Compressed, this subset data represents a little less than 100MB. You are free to download and analyze the full dataset (or a larger subset), as explained below.

Provenance of the data

You will find the data subset we will be using in this chapter at https://github.com/ipython-books/minibook-2nd-data.

If you are interested in the original dataset containing all trips, you can refer to https://github.com/ipython-books/minibook-2nd-code/tree/master/chapter2/cleaning. This page contains the code to download the original dataset and create the data subset we'll be using in this chapter. It is recommended to have **100GB of free space** on your hard drive before you proceed with the full dataset (this requirement doesn't apply to the data subset we will be using in this chapter, of course).

The original 50GB dataset contained 24 zipped CSV files (a *data* and a *fare* file for every month). We created a Python script going through all of these files and extracting one row out of 200 rows.

Then, we ordered the rows by chronological order (using the pickup time).

Next, we removed all rows with inconsistent coordinates. We defined the coordinates of a rectangle surrounding Manhattan (to restrict ourselves to this area) and we only kept the rows where both pickup and dropoff locations were within this rectangle.

Finally, we ended up with two cleaned `nyc_data.csv` and `nyc_fare.csv` files.

We used some features of pandas to perform these steps efficiently. We will cover them later in this chapter.

Here are a few references:

- History of the dataset at http://chriswhong.com/open-data/foil_nyc_taxi/
- More recent data at http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml
- An interactive web application based on this dataset at http://hubcab.org
- pandas documentation at http://pandas.pydata.org/pandas-docs/stable/
- *Python for Data Analysis*, *O'Reilly Media*, by *Wes McKinney*, the creator of pandas, at http://

shop.oreilly.com/product/0636920023784.do

Tip

Public datasets

There is now a large variety of public datasets available online as part of the *open data* movement. Here are a few references:

- Open data on Wikipedia at https://en.wikipedia.org/wiki/Open_data
- Curated list of public datasets at https://github.com/caesar0301/awesome-public-datasets
- The home of the U.S. Government's open data at http://www.data.gov
- The open platform for French public data at https://www.data.gouv.fr/en/

Downloading and loading a dataset

Let's import a few packages we will need here.

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        %matplotlib inline
```

It is a common practice to import NumPy and assign it the `np` alias. Same for pandas with `pd`, and matplotlib's high-level interface named `pyplot` with `plt`. The `%matplotlib inline` magic command tells matplotlib to render figures as static images in the Notebook.

We now move to the `chapter2` subdirectory in the minibook's directory:

```
In [2]: %cd ~/minibook/chapter2/
```

Next, let's download the data subset, available on the book's data repository at https://github.com/ipython-books/minibook-2nd-data. **If you are on Windows, the following two commands won't work**. Instead, you can download the *NYC Taxi dataset* from the URL above and extract it in the current directory with a right-click.

```
In [3]: !wget https://raw.githubusercontent.com/ipython-books/minibook-2nd-data/
master/nyc_taxi.zip
        !unzip nyc_taxi.zip
In [4]: %ls data
Out[4]: nyc_data.csv  nyc_fare.csv  [...]
```

We are now in `~/minibook/chapter2/`, and we should have a `data/` subdirectory containing two CSV files. The `nyc_data.csv` file contains information about the rides, whereas `nyc_fare.csv` contains information about the fares.

```
In [5]: data_filename = 'data/nyc_data.csv'
        fare_filename = 'data/nyc_fare.csv'
```

Now, let's load the data. pandas provides a powerful `read_csv()` function that can read virtually any CSV file. This function accepts many options, as you can see in pandas' documentation page at http://pandas.pydata.org/pandas-docs/stable/generated/pandas.io.parsers.read_csv.html. Here, we just need to specify which columns contain the dates, so that pandas can parse them correctly.

Tip

Trial and error

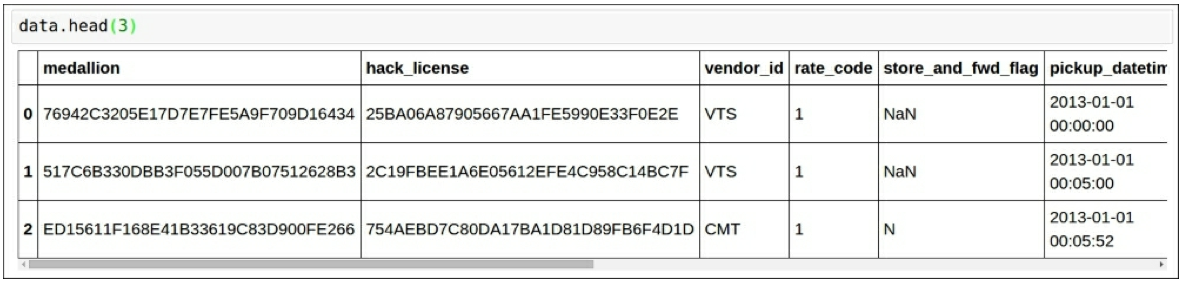
Typically, you should first try to open a dataset with `read_csv(filename)` with no special argument. If an error occurs, or if some columns are parsed incorrectly, you can fix the problem by passing extra parameters to the function, like we did here with `parse_dates`. You will find more information in pandas' documentation.

```
In [6]: data = pd.read_csv(data_filename,
                           parse_dates=['pickup_datetime',
                                        'dropoff_datetime'])
        fare = pd.read_csv(fare_filename,
                           parse_dates=['pickup_datetime'])
```

The `data` and `fare` variables are **DataFrame** objects. A DataFrame is a table containing rows (**observations** or **samples**) and columns (**features** or **variables**). DataFrames can contain text, numbers, dates, and other types of data. pandas provides Notebook-friendly display facilities for DataFrames, as we can see here:

```
In [7]: data.head(3)
```

The `head()` method of DataFrames displays the first few lines (here, three) of the table. Here is a screenshot:



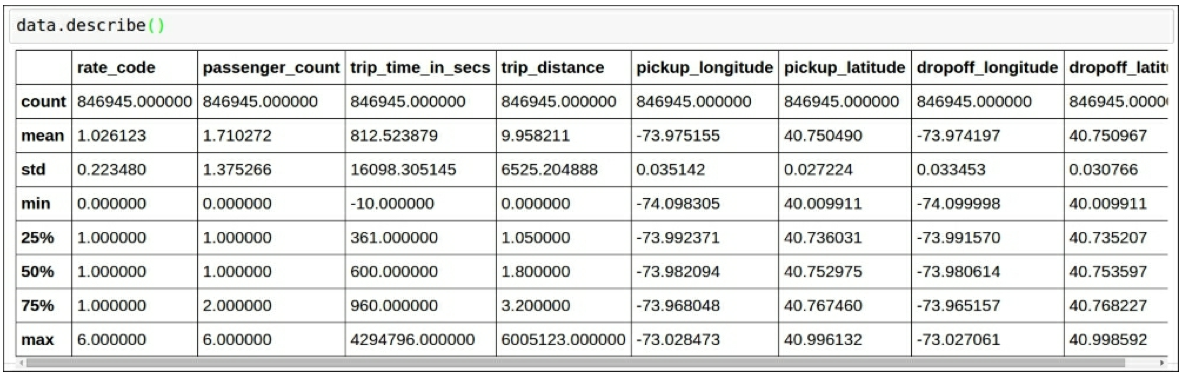
`data.head(3)`

	medallion	hack_license	vendor_id	rate_code	store_and_fwd_flag	pickup_datetim
0	76942C3205E17D7E7FE5A9F709D16434	25BA06A87905667AA1FE5990E33F0E2E	VTS	1	NaN	2013-01-01 00:00:00
1	517C6B330DBB3F055D007B07512628B3	2C19FBEE1A6E05612EFE4C958C14BC7F	VTS	1	NaN	2013-01-01 00:05:00
2	ED15611F168E41B33619C83D900FE266	754AEBD7C80DA17BA1D81D89FB6F4D1D	CMT	1	N	2013-01-01 00:05:52

A DataFrame in the Notebook

Similarly, the `tail()` method displays the last few lines of a DataFrame.

The `describe()` method shows basic statistics of all columns, as shown in the following screenshot:



`data.describe()`

	rate_code	passenger_count	trip_time_in_secs	trip_distance	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latit
count	846945.000000	846945.000000	846945.000000	846945.000000	846945.000000	846945.000000	846945.000000	846945.0000
mean	1.026123	1.710272	812.523879	9.958211	-73.975155	40.750490	-73.974197	40.750967
std	0.223480	1.375266	16098.305145	6525.204888	0.035142	0.027224	0.033453	0.030766
min	0.000000	0.000000	-10.000000	0.000000	-74.098305	40.009911	-74.099998	40.009911
25%	1.000000	1.000000	361.000000	1.050000	-73.992371	40.736031	-73.991570	40.735207
50%	1.000000	1.000000	600.000000	1.800000	-73.982094	40.752975	-73.980614	40.753597
75%	1.000000	2.000000	960.000000	3.200000	-73.968048	40.767460	-73.965157	40.768227
max	6.000000	6.000000	4294796.000000	6005123.000000	-73.028473	40.996132	-73.027061	40.998592

Describing a dataset

Making plots with matplotlib

Visualizing *raw* data, as opposed to aggregated statistics, often allows us to get a general idea about a dataset. Here, we will display the pickup and dropoff locations of all trips.

The first step is to get the actual coordinates from the DataFrame. We can find the list of columns as follows:

```
In [8]: data.columns
Out[8]: Index(['medallion',
               ...
               'pickup_datetime',
               'dropoff_datetime',
               'passenger_count',
               'trip_time_in_secs',
               'trip_distance',
               'pickup_longitude',
               'pickup_latitude',
               'dropoff_longitude',
               'dropoff_latitude'], dtype='object')
```

Four columns mention `latitude` and `longitude`. Let's load these columns:

```
In [9]: p_lng = data.pickup_longitude
        p_lat = data.pickup_latitude
        d_lng = data.dropoff_longitude
        d_lat = data.dropoff_latitude
```

With pandas, every column of a DataFrame can be obtained with the `mydataframe.columnname` syntax. An alternative syntax is `mydataframe['columnname']`.

Here, we created four variables with the coordinates of the pickup and dropoff locations. These variables are all `Series` objects:

```
In [10]: p_lng
Out[10]: 0        -73.955925
         1        -74.005501
         ...
         846943   -73.978477
         846944   -73.987206
         Name: pickup_longitude, Length: 846945, dtype: float64
```

A Series is an indexed list of values. Therefore, a DataFrame is simply a collection of Series columns.

Before we can make a plot, we need to get the coordinates of points in pixels instead of geographical coordinates. We can use the following function that performs a Mercator projection:

```
In [11]: def lat_lng_to_pixels(lat, lng):
             lat_rad = lat * np.pi / 180.0
             lat_rad = np.log(np.tan((lat_rad + np.pi / 2.0) / 2.0))
             x = 100 * (lng + 180.0) / 360.0
             y = 100 * (lat_rad - np.pi) / (2.0 * np.pi)
```

```
        return (x, y)
```

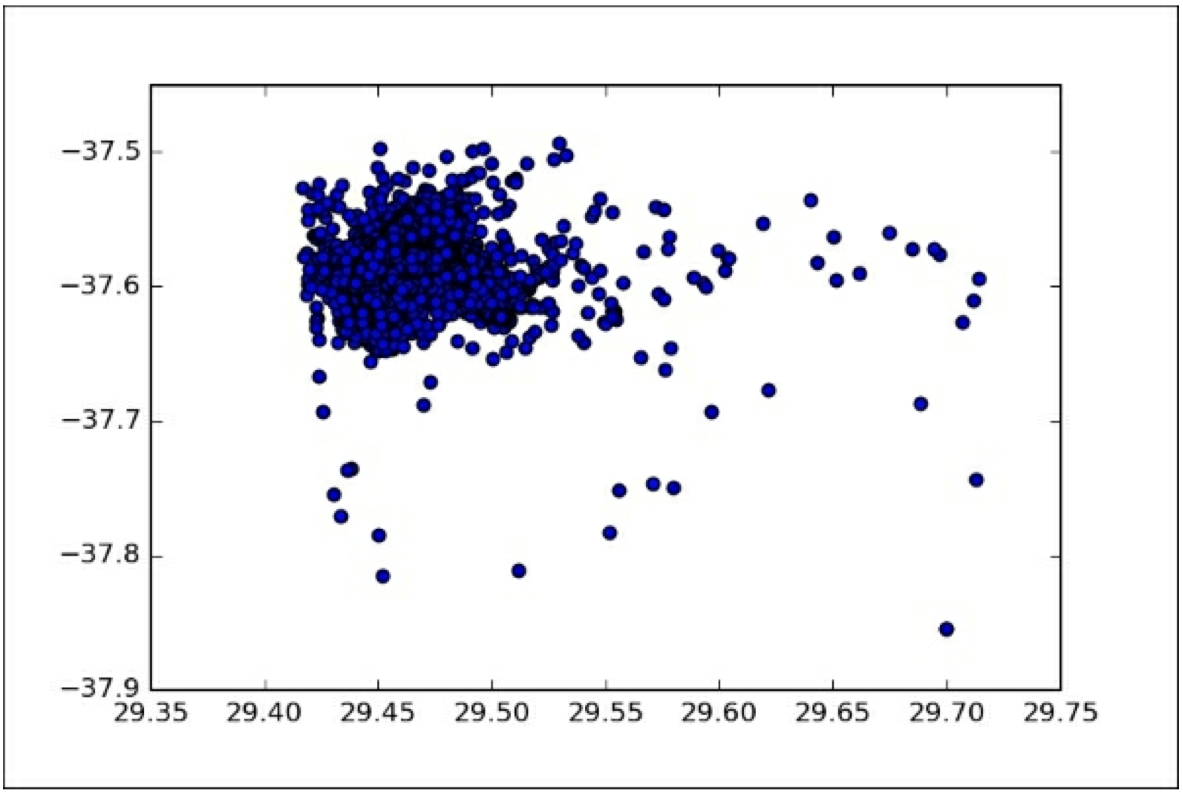
NumPy implements many mathematical functions like `np.log()` and `np.tan()`. These functions work on scalar numbers and also on pandas objects such as Series. Here, the following function call returns two new Series `px` and `py`:

```
In [12]: px, py = lat_lng_to_pixels(p_lat, p_lng)
In [13]: px
Out[13]: 0         29.456688
         1         29.442916
         ...
         846943    29.450423
         846944    29.447998
         Name: pickup_longitude, dtype: float64
```

We will give more details about mathematical operations on pandas objects later in this chapter.

The matplotlib `scatter()` function takes two arrays with *x* and *y* coordinates as inputs. A **scatter plot** is a common 2D figure showing points with various positions, sizes, colors, and marker shapes. The following command displays all pickup locations:

```
In [14]: plt.scatter(px, py)
```



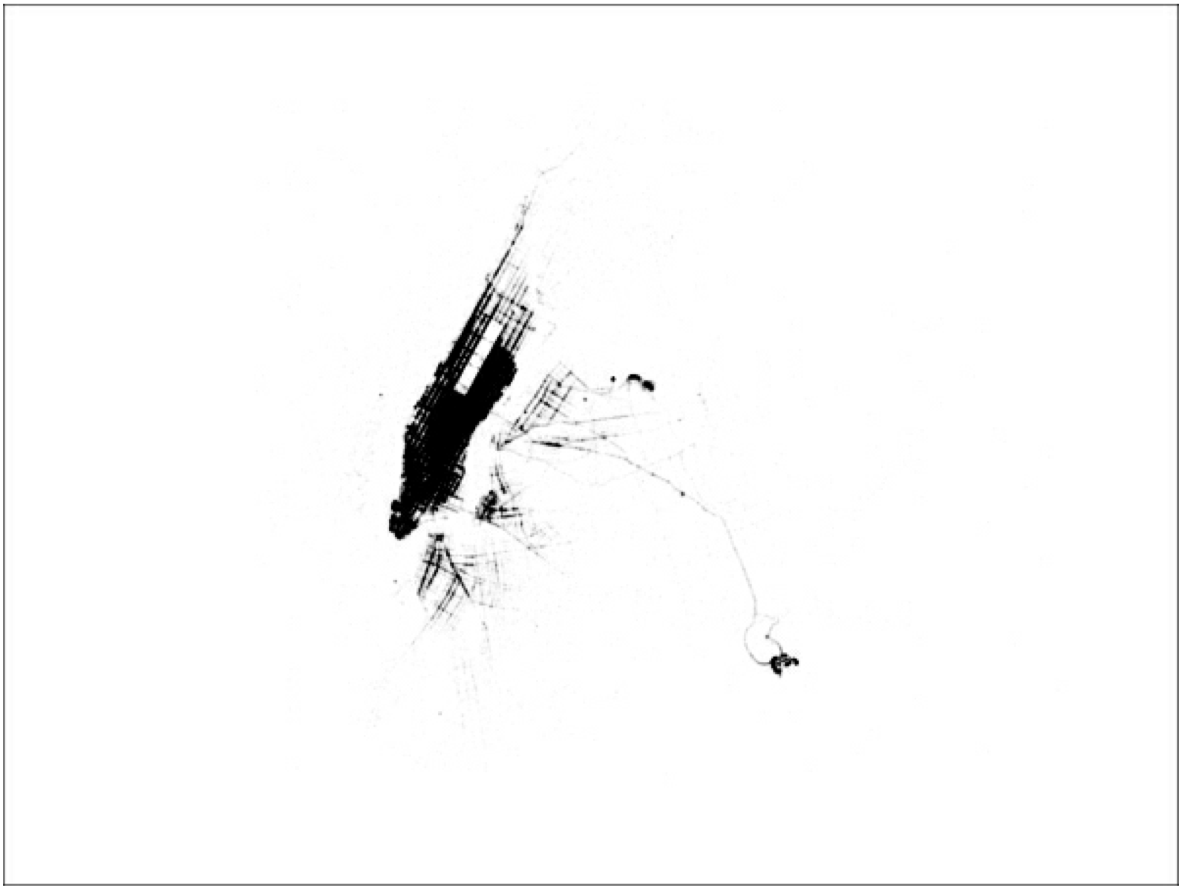
A scatter plot

Congratulations! You've made your first matplotlib plot. But it is not particularly appealing. First, the markers are too big. Second, there are too many points; we could make them a bit transparent to have a better idea of the distribution of the points. Third, we may want to zoom a bit more around Manhattan.

Fourth, could we make this figure bigger? And finally, we don't necessarily need the axes here.

Fortunately, matplotlib is highly customizable, and all aspects of the plot can be changed, as shown here:

```
In [15]: plt.figure(figsize=(8, 6))
         plt.scatter(px, py, s=.1, alpha=.03)
         plt.axis('equal')
         plt.xlim(29.40, 29.55)
         plt.ylim(-37.63, -37.54)
         plt.axis('off')
```



A better scatter plot

That's already better! Let's explain these commands in more detail:

- The `figure()` function lets us specify the figure size (in inches).
- The `scatter()` function accepts many keyword arguments to customize the aspect of the scatter plot. Here:
  - We use a small marker size with the `s` keyword argument.
  - We use a small `alpha` opacity value: the points become nearly transparent, which emphasizes the regions with high density.
- We use an equal aspect ratio with `axis('equal')`.
- We zoom in by specifying the limits of the *x* and *y* axes with `xlim()` and `ylim()`.
- We remove the axes with `axes('off')`.

You will find the full list of options of `scatter()` in matplotlib's documentation at http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.scatter.

Descriptive statistics with pandas and seaborn

Common statistical quantities are one function call away in pandas. Here are a few examples:

```
In [16]: px.count(), px.min(), px.max()
Out[16]: (846945, 29.4171, 29.7143)
In [17]: px.mean(), px.median(), px.std()
Out[17]: (29.451345, 29.44941, 0.00976)
```

pandas also provides facilities for common statistical plots. These facilities leverage the matplotlib and seaborn libraries.

matplotlib is the main plotting package in Python. Although highly powerful and flexible, it sometimes requires a significant amount of manual tuning in order to generate clean, high-quality, publication-ready figures. Several projects aim to offer higher-level, simpler user interfaces for high-quality plotting. Seaborn is one of them, and we will use it in this subsection.

First, we need to install seaborn, as it is not currently installed by default in the Anaconda distribution. Fortunately, installing it is easy with conda. We can even perform the installation from the Notebook, as shown here:

```
In [18]: !conda install seaborn -q -y
```

Tip

Conda optional arguments

The optional arguments `-q -y` tell conda not to display the progress bar and not to ask for confirmation, respectively. More information is available at http://conda.pydata.org/docs/commands/conda-install.html.

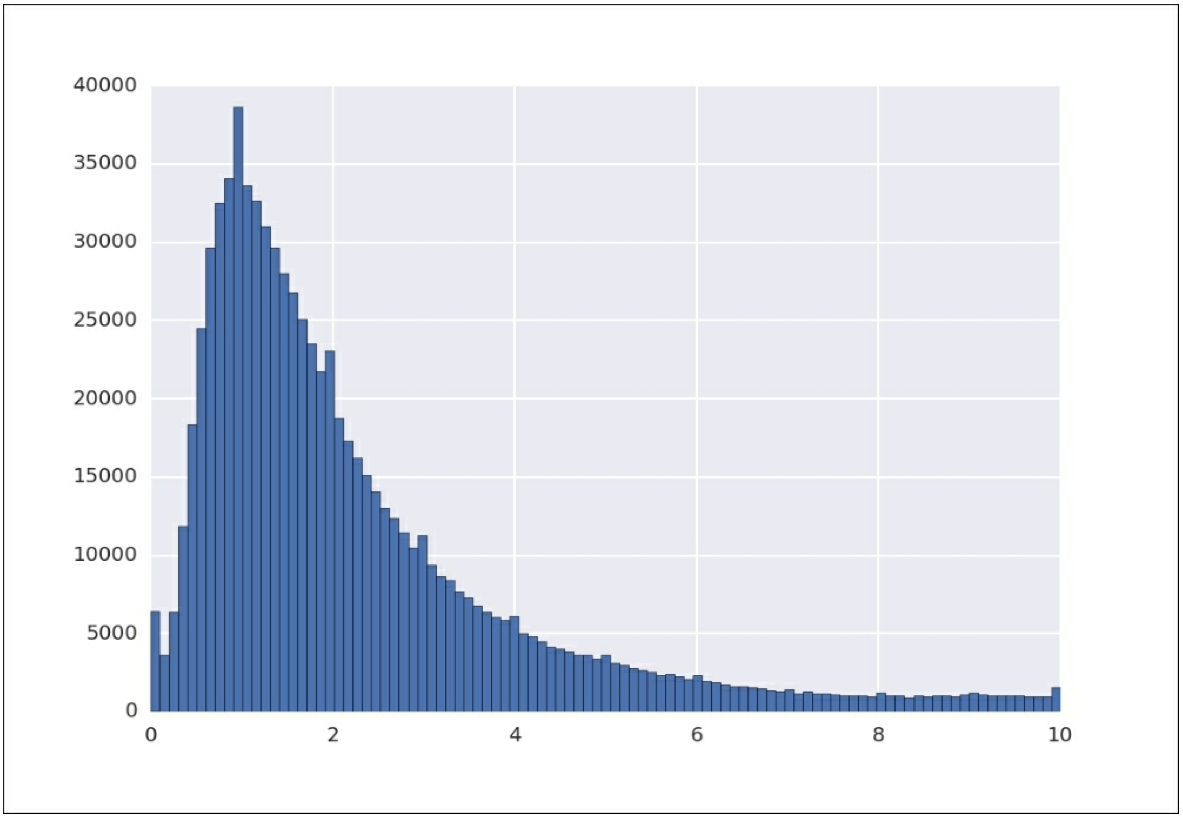
This command may take a while to complete, depending on your network connection. Let's check that seaborn has been correctly installed:

```
In [19]: import seaborn as sns
         sns.__version__
Out[19]: '0.6.0'
```

Importing seaborn automatically improves the aesthetics and color palettes of matplotlib figures. It also provides several easy-to-use statistical plotting functions.

Let's display a histogram of the trip distances. pandas provides a few simple plotting methods for `DataFrame` and `Series` objects. These methods are based on matplotlib, and benefit from the seaborn styling if seaborn has been imported. The `hist()` method displays a histogram of the values of a `Series` object. We can specify the histogram bins with the `bins` keyword argument. Here, we use NumPy's `linspace()` function to generate 100 linearly-spaced bins between 0 and 10:

```
In [20]: data.trip_distance.hist(bins=np.linspace(0., 10., 100))
```



A histogram with pandas, matplotlib, and seaborn

Here are a few references:

- Plotting with pandas at http://pandas.pydata.org/pandas-docs/stable/visualization.html
- Seaborn documentation at http://stanford.edu/~mwaskom/software/seaborn/
- Visualizing distributions with seaborn, at http://stanford.edu/~mwaskom/software/seaborn/tutorial/distributions.html

Manipulating data

Visualizing raw data and computing basic statistics is particularly easy with pandas. All we have to do is choose a couple of columns in a DataFrame and use built-in statistical or visualization functions.

However, more sophisticated data manipulations methods quickly become necessary as we explore a dataset. In this section, we will first see how to make selections of a DataFrame. Then, we will see how to efficiently make transformations and computations on columns.

We first import the NYC taxi dataset, as in the previous section.

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        %matplotlib inline
        data = pd.read_csv('data/nyc_data.csv',
                           parse_dates=['pickup_datetime',
                                        'dropoff_datetime'])
        fare = pd.read_csv('data/nyc_fare.csv',
                           parse_dates=['pickup_datetime'])
```

The `data` and `fare` DataFrames are now loaded in the notebook.

Selecting data

Our dataset contains almost one million rows. Only limited analyses can be done by using the whole dataset. More interesting discoveries can be made by looking at carefully-chosen subsets of the data. For example, what can we say about the taxi rides done on a particular day, a particular month, or a particular day of week? What about those starting or ending at a particular location? A significant part of real-world data analysis involves such fine-grained selections.

pandas offers many facilities for selecting a subset of columns or rows.

Selecting columns

First, let's select a few columns:

```
In [2]: data[['trip_distance', 'trip_time_in_secs']].head(3)
Out[2]:    trip_distance  trip_time_in_secs
        0           0.61                300
        1           3.28                960
        2           1.50                386
```

In Python, the square brackets `[]` are used for selecting elements in a list. The same notation is used by pandas to select columns. We need two pairs of brackets because pandas expects a list of columns to select, here `['trip_distance', 'trip_time_in_secs']`. The end-result is a new DataFrame containing just two columns instead of 14.

This is about all you need to know about selecting columns. There is much more to say about selecting rows.

Selecting rows

Rows of a DataFrame are *indexed*: every row comes with a unique *label* (or *index*). Often, this label is

just an integer between `0` and `n_rows-1`. In some situations, this label can be something else, like a string. If we had a DataFrame giving information about each taxi, the label could be the taxi's medallion (a unique identifier for NYC's taxicabs), or an anonymized version of it.

The `loc` attribute of a DataFrame is used to select row(s) from their labels. Here, we select the first row:

```
In [3]: data.loc[0]
Out[3]: medallion             76942C3205E17D7E7FE5A9F709D16434
        hack_license          25BA06A87905667AA1FE5990E33F0E2E
        vendor_id                                          VTS
        rate_code                                            1
        store_and_fwd_flag                                 NaN
        pickup_datetime                    2013-01-01 00:00:00
        dropoff_datetime                   2013-01-01 00:05:00
        passenger_count                                      3
        trip_time_in_secs                                  300
        trip_distance                                     0.61
        pickup_longitude                                -73.95592
        pickup_latitude                                  40.78189
        dropoff_longitude                               -73.96318
        dropoff_latitude                                 40.77783
        Name: 0, dtype: object
```

Multiple rows can be selected by providing a list of labels:

```
In [4]: data.loc[[0, 100000]]
```

	medallion	hack_license	vendor_id	rate_code	store_and_fwd_flag	pickup_d
0	76942C3205E17D7E7FE5A9F709D16434	25BA06A87905667AA1FE5990E33F0E2E	VTS	1	NaN	2013-01-(00:00:00
100000	7461F7106D33D3A5775F4245724606FD	BACEA353BB4106A005BB7836BDCAC0C3	VTS	1	NaN	2013-02-1 18:10:00

Selecting multiple rows

We can also select regularly spaced rows using *slices*. For example, here is how to select one row out of 10 between rows 1000 and 2000:

```
In [5]: data.loc[1000:2000:10,
                 ['trip_distance', 'trip_time_in_secs']]
Out[5]:       trip_distance  trip_time_in_secs
        1000           1.00                441
        1010           3.80                691
        ....
        1990           0.13                 60
        2000           9.60                963
```

Note how we combined column and row selection here. Two expressions can be passed to `loc`: the row selection first, and the column selection second (the two expressions are separated by a comma).

`loc` expects actual labels and, unlike normal Python slices, the start and end points are both inclusive! Also, we could have used `iloc` instead of `loc` to specify index positions rather than labels.

Filtering with boolean indexing

Instead of selecting rows by labels, we can also select rows satisfying specific properties. This is a more common use-case in data analysis.

For example, let's select the longest rides:

```
In [6]: data.loc[data.trip_distance>50]
```

dropoff_datetime	passenger_count	trip_time_in_secs	trip_distance	pickup_longitude	pickup_latitude	dropoff_longitude
2013-01-01 21:56:37	1	934	52.20	-73.979576	40.743626	-73.941902
2013-01-04 07:17:14	1	1973	96.30	-73.959785	40.762497	-73.962440
2013-01-05 02:23:01	1	1913	52.90	-74.006119	40.735157	-73.958694
2013-01-12 03:24:47	1	1312	66.20	-73.966873	40.683315	-73.916885

Long taxi rides

Here, `data.trip_distance>50` is a Series object containing boolean values for all rows, depending on whether the trip distance is higher or lower than 50. The `loc` attribute also works with booleans instead of explicit labels: it will return all rows represented by a `True` boolean value.

We might want to choose the distance threshold depending on certain conditions. For example, we might want to keep the 1% longest trips. Here, let's show how the IPython widgets can help us do that (this isn't the only method, of course).

We create a slider displaying the number of rows with a distance larger than the threshold:

```
In [7]: from ipywidgets import interact
In [8]: @interact
        def show_nrows(distance_threshold=(0, 200)):
            return len(data.loc[data.trip_distance >
                                distance_threshold])
```

A slider to select long rides

More selection, indexing, and filtering facilities are described in pandas' documentation. Here are a few references:

- http://pandas.pydata.org/pandas-docs/stable/dsintro.html
- http://pandas.pydata.org/pandas-docs/stable/indexing.html

Computing with numbers

The `trip_time_in_secs` column contains the trip durations in seconds. How can we convert these values to minutes? More generally, how can we make computations on DataFrames?

A first approach would be to use a `for` loop, iterating over all rows and making numerical computations successively inside that loop. This is what people with a background in the C programming language tend to do when they start to learn Python. However, this isn't the best way to do things in Python.

Whereas Python loops are possible in this situation, they would be extremely slow. For this reason, they should be avoided as much as possible. We will discuss this issue in the next chapter. In the meantime, there are much better, faster, and actually *simpler* alternatives.

pandas allows you to perform vector operations on DataFrame and Series objects. These operations are quite natural, because they follow standard mathematical notations. For example, let's add a new column containing the trip durations in minutes:

```
In [9]: data['trip_time_in_mins'] = data.trip_time_in_secs / 60.0
In [10]: data[['trip_time_in_secs', 'trip_time_in_mins']].head(3)
Out[10]:    trip_time_in_secs  trip_time_in_mins
         0                300           5.000000
         1                960          16.000000
         2                386           6.433333
```

Let's explain this in more detail. The `data.trip_time_in_secs` notation represents a Series object. The `/` symbol represents floating-point division in Python 3. It normally works with numbers only. However, pandas extends this operator to work with Series and DataFrames as well, in which case it automatically operates on all elements. Here, all elements of `data.trip_time_in_secs` are divided by 60.

The same notation would also work if we had another Series object of the same size in the second term. In that case, the division would occur on an element-wise basis (the first item in the Series on the left divided by the first in the right, the second by the second, and so on).

A Series object is a vector with indices (or labels). The indices determine which values are used when operating Series objects together. Here is an example:

```
In [11]: a = data.trip_distance[:5]
         a
Out[11]: 0    0.61
         1    3.28
         2    1.50
         3    0.00
         4    1.31
         Name: trip_distance, dtype: float64
In [12]: b = data.trip_distance[2:6]
         b
Out[12]: 2    1.50
         3    0.00
         4    1.31
         5    5.81
         Name: trip_distance, dtype: float64
```

These two Series objects have different but overlapping sets of indices. Although they don't have the same size, we can add them together:

```
In [13]: a + b
Out[13]: 0     NaN
         1     NaN
         2    3.00
         3    0.00
         4    2.62
         5     NaN
         Name: trip_distance, dtype: float64
```

The result is a new Series object containing the *aligned* sum of `a` and `b`. The set of indices of `a + b` is the *union* of the indices of `a` and `b`. When one value is missing, we get an operation with an undefined value, which is `NaN` (Not a Number). When the indices overlap, the sum is correctly computed. This feature - **alignment** - makes it quite convenient to operate on labeled data. You'll find more information at http://pandas.pydata.org/pandas-docs/stable/basics.html.

Other mathematical operations (`+`, `*`, etc.) work similarly. Further, NumPy implements many mathematical functions like `np.log()` and `np.sin()`; they not only work on scalar numbers but also on Series and DataFrames. This is called **vectorization**, because this concept relates to mathematical operations performed on *vectors*. We will discuss this concept in greater details in Chapter 3, *Numerical Computing with NumPy*.

Working with text

Efficient vectorized operations can also be done on text. Let's have a look at `data.medallion`:

```
In [14]: data.medallion.head(3)
Out[14]: 0    76942C3205E17D7E7FE5A9F709D16434
         1    517C6B330DBB3F055D007B07512628B3
         2    ED15611F168E41B33619C83D900FE266
         Name: medallion, dtype: object
```

This column contains anonymized versions of the taxis' medallions. The `str` attribute gives us access to many vectorized string processing functions. Here, for example, we extract the first four characters of every medallion:

```
In [15]: data.medallion.str.slice(0, 4).head(3)
Out[15]: 0    7694
         1    517C
         2    ED15
         Name: medallion, dtype: object
```

There are many other functions, including ones that apply regular expressions on all rows. Together, these functions are essential when you're working with text data, particularly when you have datasets so large that `for` loops would be too slow. You will find the full list of string methods at http://pandas.pydata.org/pandas-docs/stable/text.html.

Working with dates and times

pandas provides many methods to operate on dates and times. Common operations include:

- getting the day, day of week, hour, or any other quantity from dates
- selecting ranges of dates
- computing time ranges
- dealing with different time zones

These operations only work on Series with a `datetime64` data type, or with `DatetimeIndex` objects (used to index values with dates or times). In practice, there are many ways to get such objects from raw data like CSV files. Here, we used the `parse_dates` keyword arguments in the `pd.read_csv()` function. Among the other methods, let's mention the `pd.to_datetime()` function. You will find more details in the references below.

The `dt` attribute of datetime objects gives us access to datetime components. For example, here is how to get the day of the week of the taxi trips (Monday=0, Sunday=6):

```
In [16]: data.pickup_datetime.dt.dayofweek[::200000]
Out[16]: 0         1
         200000    6
         400000    5
         600000    0
         800000    1
         dtype: int64
```

Here is a more complex example. Let's select all night trips that finished the next day:

```
In [17]: day_p = data.pickup_datetime.dt.day
         day_d = data.dropoff_datetime.dt.day
         selection = (day_p != day_d)
         print(len(data.loc[selection]))
         data.loc[selection].head(3)
Out[17]: 7716
```

	vendor_id	rate_code	store_and_fwd_flag	pickup_datetime	dropoff_datetime	passenger_count	trip_time_in_secs	trip_distance
ED	VTS	1	NaN	2013-01-01 23:45:00	2013-01-02 00:03:00	1	1080	12.61
95	CMT	1	N	2013-01-01 23:46:22	2013-01-02 00:28:01	1	2498	16.10
49	CMT	1	N	2013-01-01 23:46:53	2013-01-02 00:03:33	1	1000	5.40

Night trips

Like in the *Computing with numbers* subsection, the `(day_p != day_d)` expression is a Series of booleans for selecting the rows that have different pickup and dropoff days. Python's inequality symbol `!=` is understood by pandas as a vectorized operator working on a row-by-row basis.

Here are a few references about date and time operations:

- http://pandas.pydata.org/pandas-docs/stable/timeseries.html
- http://pandas.pydata.org/pandas-docs/stable/timedeltas.html

Handling missing data

We finish this part with a short discussion about missing data. Real-world datasets are rarely perfect, and having missing values in a dataset is the rule rather than the exception. Fortunately, pandas perfectly handles missing data.

In practice, you can consider letting pandas deal seamlessly with missing data while you manipulate and operate on data. However, there are times when you want control over these missing values. For

example, you may want to discard missing data from some analysis. Or, you may want to replace missing data with a default value.

In pandas, missing data is represented by `NaN` (Not a Number) or `None`. pandas provides several Series and DataFrame methods to deal with missing data, notably:

- `isnull()` indicates whether values are null or not
- `notnull()` indicates the opposite
- `dropna()` removes missing data
- `fillna(some_default_value)` replaces missing data with a default value

You will find more details at http://pandas.pydata.org/pandas-docs/stable/missing_data.html.

Complex operations

We've seen how to load, select, filter, and operate on data with pandas. In this section, we will show more complex manipulations that are typically done on full-blown databases based on SQL.

Tip

SQL

Structured Query Language is a domain-specific language widely used to manage data in **relational database management systems** (**RDBMS**). pandas is somewhat inspired by SQL, which is familiar to many data analysts. Additionally, pandas can connect to SQL databases. You will find more information about the links between pandas and SQL at http://pandas.pydata.org/pandas-docs/stable/comparison_with_sql.html.

Let's first import our NYC taxi dataset as in the previous sections.

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn
        %matplotlib inline
        data = pd.read_csv('data/nyc_data.csv',
                           parse_dates=['pickup_datetime',
                                        'dropoff_datetime'])
        fare = pd.read_csv('data/nyc_fare.csv',
                           parse_dates=['pickup_datetime'])
```

Group-by

A *group-by* operation typically consists of one or several of the following steps:

- splitting the data into groups that share common attributes
- applying a function to every group
- recombining the results

Many operations that seem particularly complex are actually group-by operations. pandas provides user-friendly facilities to perform these manipulations. We will illustrate them here.

Let's have a look at the weekly statistics in our dataset. We first need to split the data into weekly groups. pandas provides the `groupby()` method for this purpose, as shown here:

```
In [2]: weekly = data.groupby(data.pickup_datetime.dt.weekofyear)
In [3]: len(weekly)
Out[3]: 52
```

Here, `data.pickup_datetime.dt.weekofyear` is a Series instance with the week number of every ride. The `groupby()` method returns an object with one group per value of `weekofyear`. Since there are 52 different weeks in the year, `weekly` contains 52 different groups.

The `size()` method returns the number of rows in each group, as shown here:

```
In [4]: y = weekly.size()
        y.head(3)
```

```
Out[4]: 1    17042
        2    15941
        3    17017
        dtype: int64
```

We'll now plot the number of rides per week. To create a meaningful plot, we first need to specify the appropriate x-axis with the dates of all 52 weeks:

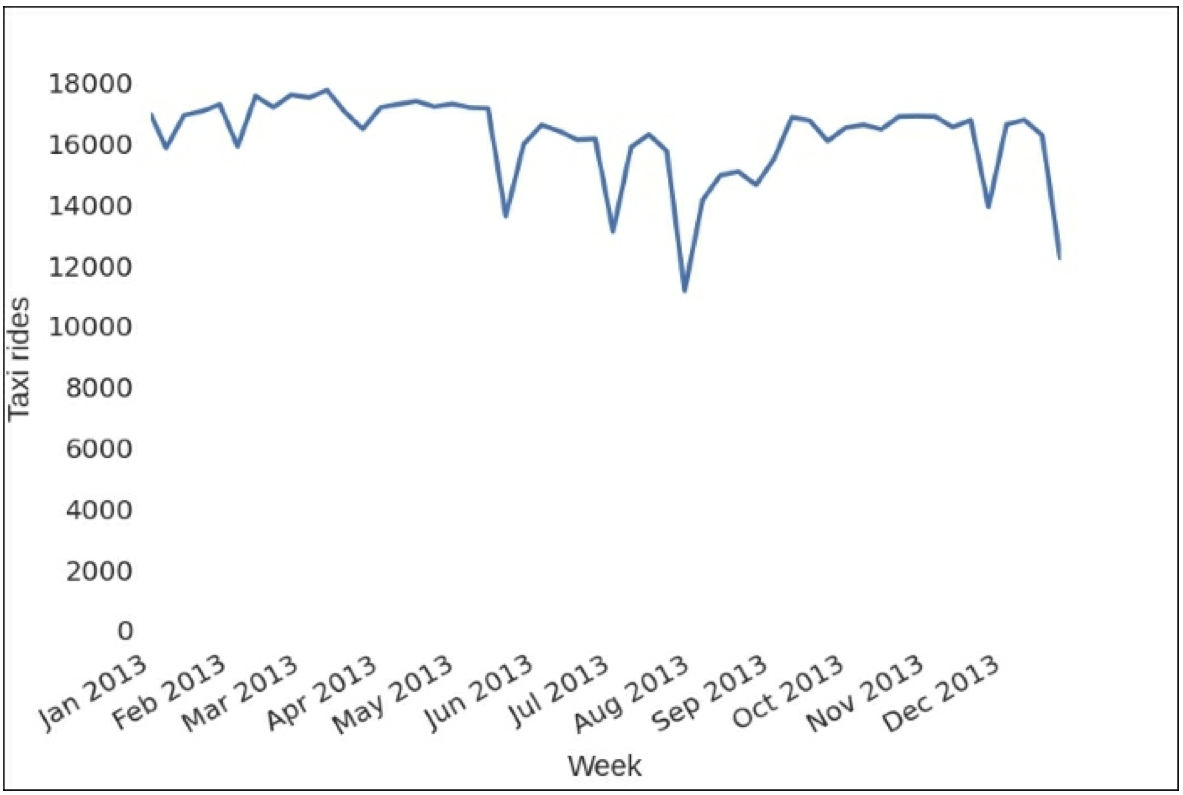
```
In [5]: x = weekly.pickup_datetime.first()
        x.head(3)
Out[5]: 1   2013-01-01 00:00:00
        2   2013-01-07 00:03:00
        3   2013-01-14 00:00:51
        Name: pickup_datetime, dtype: datetime64[ns]
```

This Series contains the date of the first item in every row.

Finally, we create a new Series with the values in our `y` object, and indexed by the dates `x` (we need to use `.values` in order to discard `y`'s indices, since we use `x`'s indices instead). The `plot()` method of this new Series creates the plot we want:

```
In [6]: pd.Series(y.values, index=x).plot()
        plt.ylim(0)  # Set the lower y value to 0.
        plt.xlabel('Week')  # Label of the x axis.
        plt.ylabel('Taxi rides')  # Label of the y axis.
```

Here is the result (let's not forget that our dataset only contains a small fraction of all rides):



Number of taxi rides per week

We'll see more examples in the next subsection.

More information about the powerful group-by operation in pandas can be found at http://pandas.pydata.org/pandas-docs/stable/groupby.html.

Joins

Joins are common operations in relational databases. The idea is to combine several tables together, based on common values shared between the tables.

In the current example, we have two DataFrames, `data` and `fare`, both with the same number of rows (one row per trip). First, from the `fare` DataFrame, we will get the average tip obtained by each taxi. Then, we'll inject this information into the `data` DataFrame.

For the first step, we use `groupby()` again:

```
In [7]: tip = fare[['medallion', 'tip_amount']] \
              .loc[fare.tip_amount>0].groupby('medallion').mean()
        print(len(tip))
        tip.head(3)
Out[7]: 13407
                                          tip_amount
        medallion
        00005007A9F30E289E760362F69E4EAD    1.815854
        000318C2E3E6381580E5C99910A60668    2.857222
        000351EDC735C079246435340A54C7C1    2.099111
```

Here, we considered a reduced DataFrame with just the `medallion` and `tip_amount` columns, removed the trips where the passenger did not tip, grouped by taxi's medallion (a unique identifier for the taxis), and took the mean of this grouped DataFrame. This new DataFrame contains one column `tip_amount` and is indexed by the medallion. It contains 13407 rows: this corresponds to the number of different taxis in our dataset.

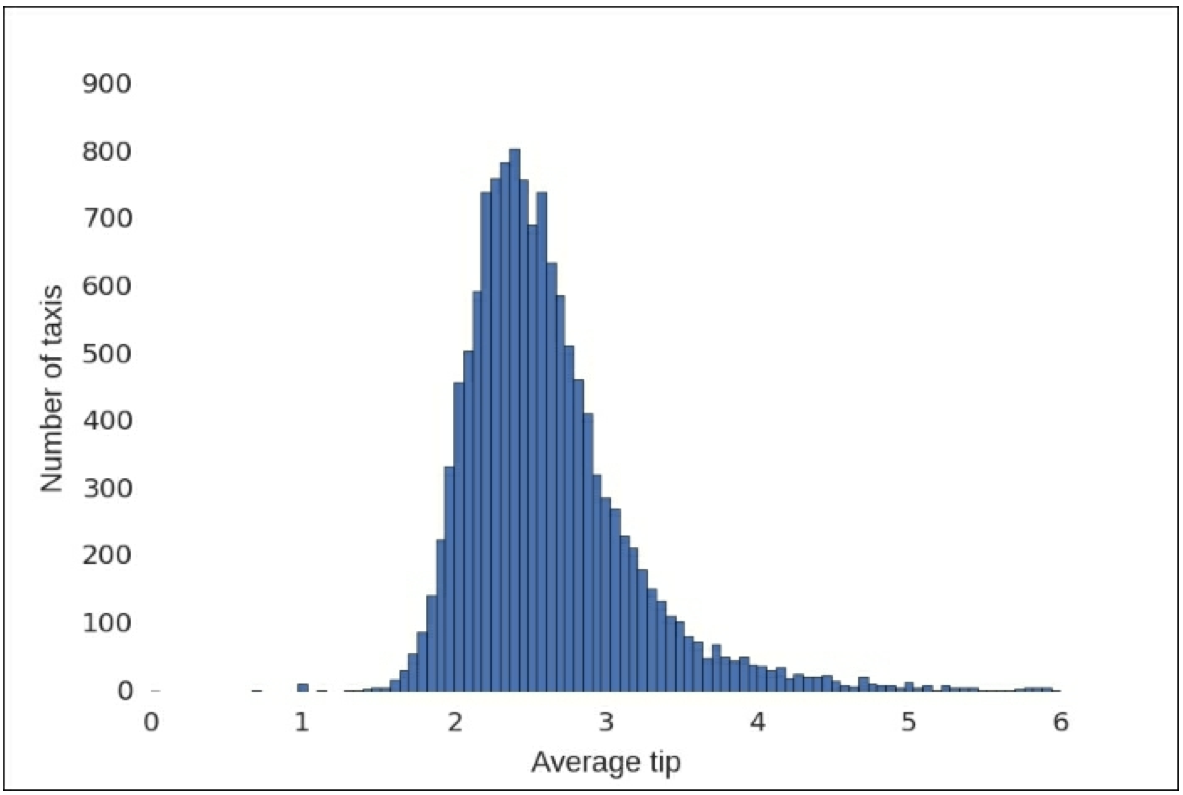
Tip

Chaining syntax

Note how we used the chaining syntax here to perform several operations successively on the `fare` DataFrame (the `obj.fun1().fun2().fun3()` pattern). Each of these operations in pandas returns a new DataFrame. The dot . character applies the next operation to the previous operation's result and forms a concise and readable syntax for chained operations. See also the `pipe()` function at http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.pipe.html.

Let's plot a histogram of these average tips:

```
In [8]: tip.hist(bins=np.linspace(0., 6., 100))
        plt.xlabel('Average tip')
        plt.ylabel('Number of taxis')
```



Average tips earned by taxis

The next step is to reinject this `tip` DataFrame into the `data` DataFrame. The `medallion` column appears in both of our datasets; by identifying this special field (also called the *key*) in both datasets, we can associate every row in `tip` to a row in `data`. This operation is called a **join** in SQL.

We can use the `merge()` function here:

```
In [9]: data_merged = pd.merge(data, tip, how='left',
                               left_on='medallion', right_index=True)
        data_merged.head(3)
```

trip_distance	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	tip_amount
0.61	-73.955925	40.781887	-73.963181	40.777832	3.180417
3.28	-74.005501	40.745735	-73.964943	40.755722	2.863235
1.50	-73.969955	40.799770	-73.954567	40.787392	2.147143

Result of a merge operation

Let's explain how this works:

- We specify the left and right DataFrames to perform the join on.

- There are several types of joins; we choose a *left* join here because we want to keep the keys from the left DataFrame `data`.
- We then specify where to find this key in the left and right DataFrames.
  - On the left, we use the `medallion` column.
  - On the right, we use the index, because the `tip` DataFrame is indexed by the medallion.

The end product is a new DataFrame similar to our original `data` DataFrame, but with an additional column containing the average tip received by the taxi.

Joins and merges form a rich and complex topic. You will find more information at http://pandas.pydata.org/pandas-docs/stable/merging.html.

Finally, here are a few more advanced topics in pandas that are worth exploring:

- Features for time series data at http://pandas.pydata.org/pandas-docs/stable/timeseries.html
- Support for categorical variables at http://pandas.pydata.org/pandas-docs/stable/categorical.html
- Pivot tables (particularly useful when dealing with high-dimensional data) at http://pandas.pydata.org/pandas-docs/stable/reshaping.html

Summary

In this chapter, we covered the basics of data analysis with pandas: loading a dataset, selecting rows and columns, grouping and aggregating quantities, and performing complex operations efficiently.

The next natural step is to conduct statistical analyses: hypothesis testing, modeling, predictions, and so on. Several Python libraries provide such functionality beyond pandas: SciPy, statsmodels, PyMC, and more. The *IPython Cookbook* contains many advanced examples of such analyses.

In the next chapter, we will introduce NumPy, the library underlying the entire SciPy ecosystem.

Chapter 3. Numerical Computing with NumPy

NumPy is the library that underlies the entire SciPy/PyData ecosystem. NumPy provides a multidimensional array data type that is widely used in numerical computing.

In this chapter, we will use NumPy on data analysis and scientific modeling examples, covering the following topics:

- A primer to vector computing
- Creating and loading arrays
- Basic array manipulations
- Computing with NumPy arrays

A primer to vector computing

Vector computing is about efficiently performing mathematical operations on numerical arrays. Many problems in science and engineering actually consist of a sequence of such operations.

This section introduces and demonstrates the multidimensional array data type for numerical computing.

Multidimensional arrays

What is a multidimensional array? Consider a vector containing 1000 real numbers. It has one dimension, since numbers are stored along a single axis. Now, consider a matrix with 1000 rows and 1000 columns. It contains 1,000,000 numbers. Because it has two dimensions, you need to specify both the row and column to refer to a specific number.

More generally, an n-dimensional array, also called **ndarray**, is an n-dimensional matrix (or tensor). Every number is identified by n indices `(i_1, ... i_n)`.

Many types of real-world data can be represented as ndarrays:

- The evolution of a stock exchange price is a 1D array (vector) with one value per day (or per hour, per week, etc.).
- A grayscale image is a `(height, width)` 2D array, with one light intensity per pixel.
- The evolution of a **Partial Differential Equation** (**PDE**) on a 2D grid can be represented as a `(n, m, duration)` 3D array.
- A video is a `(height, width, n_channels, duration)` 4D array, where typically `n_channels=3` for the three RGB (red, green, blue) color components.

It is quite rare to work on arrays with more than 4 dimensions.

Originally, Python didn't provide an adequate structure for representing an ndarray. This is the main goal of a scientific Python library created in the early 2000s called NumPy, which traces its roots back to several years before.

The ndarray

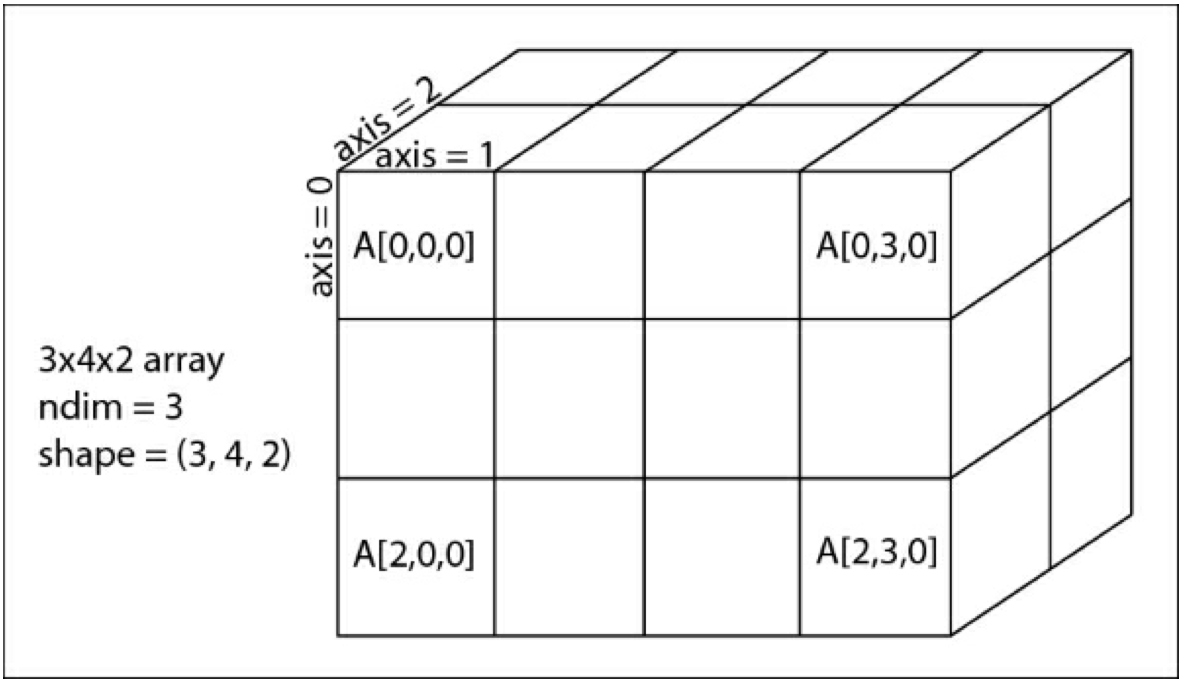
An ndarray is essentially defined by:

- a number of **dimensions**
- a **shape**
- **strides**
- a **data type**, or **dtype**

- the actual data

We already explained the notion of dimension for a numerical array. The **shape** is the length of every axis. For example, the shape of a video would be `(height, width, n_channels, duration)`. There are four elements in this tuple because there are four dimensions in the array. Strides will be explained later in this chapter.

Here is a schematic representation of the structure of a 3D ndarray:



Structure of an ndarray

In an array, all elements must have the same data type. The most common data types are integers, floating-point numbers, booleans, and strings.

You can also define custom data types for **structured arrays** (also called **record arrays**). These are arrays of structs (meant as *C structs*). A typical use-case occurs when you want to load a flat binary file in a complex format. You will find more information at http://docs.scipy.org/doc/numpy/user/basics.rec.html.

Vector operations on ndarrays

NumPy not only provides an ndarray structure for *storing* numerical data, but it also implements fast mathematical operations on ndarrays. The ability to perform highly-efficient operations on ndarrays is one of the major advantages of this structure.

Many operations on arrays follow the same pattern where an elementary mathematical operation is performed on an element-wise basis on two arrays. For example, the sum `C=A+B` of two matrices contains the sums of all corresponding pairs of elements in `A` and `B`: `C_ij = A_ij + B_ij`. Mathematically, this corresponds to operations on vectors and matrices. These are called **vector (or vectorized) operations**. NumPy offers fast implementations of many vector operations.

Another advantage of NumPy is the brevity of the syntax for array operations. Whereas a language

like C or Java would require us to write a loop for a matrix operation as simple as `C=A+B`, NumPy allows us to simply write `C=A+B`. More generally, many complex operations can be concisely written in a few lines of NumPy, whereas they would involve tens of lines in another language.

How fast are vector computations in NumPy?

One of the most important take-home messages of this chapter is that vector operations on ndarrays are much faster than Python loops. Most numerical computations should be done with vector operations in NumPy instead of Python loops.

Let's illustrate this particularly important point by showing two ways of computing the sum of two vectors: in pure Python, and with NumPy.

Let's first create two vectors containing 1,000,000 random numbers each. We use the native `random` module in a list comprehension:

```
In [1]: from random import random
        list_1 = [random() for _ in range(1000000)]
        list_2 = [random() for _ in range(1000000)]
```

We compute the sum of these vectors with another list comprehension. The `zip()` built-in function allows us to loop over the two vectors simultaneously, as shown here:

```
In [2]: out = [x + y for (x, y) in zip(list_1, list_2)]
        out[:3]
Out[2]: [0.843375384328939, 1.507485612134079, 1.4119777108063973]
```

How long does this operation take? Let's use IPython's `%timeit` magic command to find it out:

```
In [3]: %timeit [x + y for (x, y) in zip(list_1, list_2)]
Out[3]: 10 loops, best of 3: 69.7 ms per loop
```

Now, we perform the same operation with NumPy:

```
In [4]: import numpy as np
        arr_1 = np.array(list_1)
        arr_2 = np.array(list_2)
```

The `np.array()` function can convert a Python list into an ndarray (we'll cover this in more detail in the next section). Although `list_1` and `arr_1` contain the same data, they don't have the same data type:

```
In [5]: type(list_1), type(arr_1)
Out[5]: (list, numpy.ndarray)
In [6]: arr_1.shape
Out[6]: (1000000,)
In [7]: arr_1.dtype
Out[7]: dtype('float64')
```

Computing the sum of the two arrays is particularly easy with NumPy; the `+` character directly works with ndarrays of the same shape:

```
In [8]: sum_arr = arr_1 + arr_2
        sum_arr[:3]
Out[8]: array([ 0.84337538,  1.50748561,  1.41197771])
```

How much faster is NumPy over pure Python here?

```
In [9]: %timeit arr_1 + arr_2
Out[9]: 1000 loops, best of 3: 1.57 ms per loop
```

This is about 45 times faster.

Generally speaking, getting one or several orders of magnitude of speed improvements between pure Python and NumPy is not uncommon. We'll explain the technical reasons of this below. In the meantime, just remember that **vectorized operations with NumPy are much faster than Python loops**. Every time you are tempted to write a Python loop, see if you can use NumPy instead.

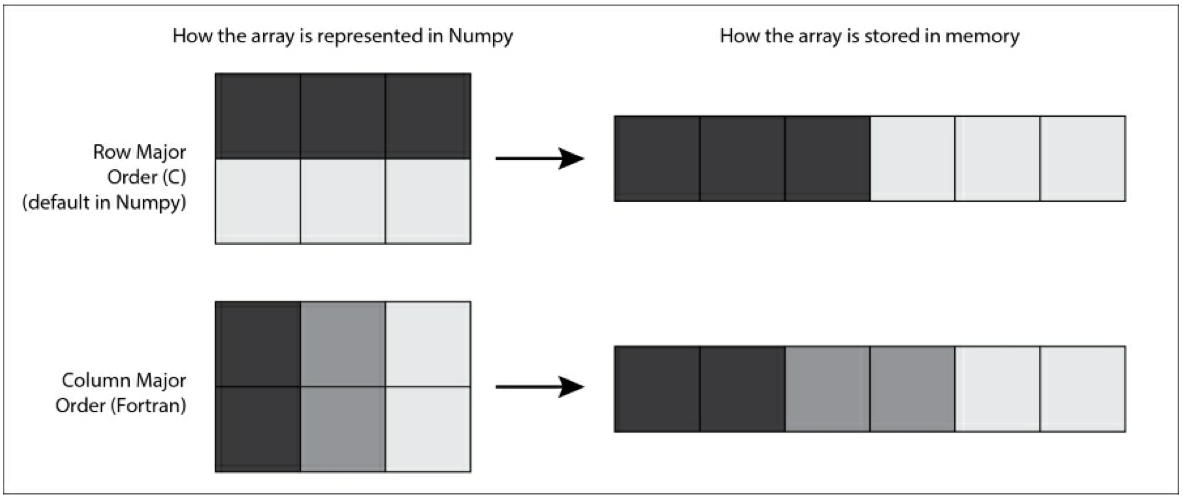
How an ndarray is stored in memory

Let's briefly discuss the internals of NumPy. Although beginners can probably skip this, knowing these details can help you write more efficient code with NumPy.

Internally, an ndarray consists of some metadata about the array's structure, and the actual binary data. The data is stored in a *contiguous* block of memory. For example, the data of a vector containing 10 elements of double-precision floating-point numbers (`float64` dtype, where each number is encoded in 8 bytes) is stored in a contiguous block of 80 bytes.

With this information, you can calculate the memory requirements for an ndarray. For example, a `(10,000, 10,000)` float64 array requires `10,000*10,000*8` bytes, which is about 763 MB of memory. **When working with large arrays, check your available memory to avoid running out of RAM and crashing your computer.**

When there is more than one dimension, there are several ways of storing the elements in the memory block. With a matrix, the elements can be stored in **row-major order** (also known as **C-order**) or **column-major order** (also known as **Fortran-order**). The distinction pertains to which axis among the row or the column moves the fastest as one goes along all elements in the data buffer. The default order in NumPy is the C-order, although this can be configured differently.



C-major and Fortran-order layout

This notion generalizes to multidimensional arrays with the notion of **strides**. Strides describe how the elements of a multidimensional array are organized within the data buffer. NumPy implements

a *strided indexing scheme*, where the position of any element is a linear combination of the element's indices, the coefficients being the strides. In other words, strides describe, in any axis, how many bytes to jump over in the data buffer to go from one item to the next along that axis.

Here are a few references:

- Documentation on strides at http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html#internal-memory-layout-of-an-ndarray
- Advanced NumPy in the SciPy lectures notes at http://www.scipy-lectures.org/advanced/advanced_numpy/
- Getting the best performance out of NumPy, an *IPython Cookbook* recipe, at http://ipython-books.github.io/featured-01/
- Blog post by Eli Bendersky at http://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays/

Why operations on ndarrays are fast

Let's compare the additions on the lists and arrays in the previous example.

Python being a dynamic interpreted language, a `for` loop involves many low-level operations of the CPython interpreter. When there are many iterations, this overhead takes significantly more time than the actual addition.

By contrast, in NumPy, vector operations are implemented in C, which is a more lower-level language than Python. This implementation leads to far fewer CPU instructions than the Python loop. Knowing the address of the memory block and the data type, it is just simple arithmetic to loop over all items.

In a Python list, elements are stored at arbitrary locations in memory, whereas in an array, elements are stored within a contiguous block of memory. CPUs are more efficient at loading consecutive bytes from memory. This is called **sequential locality**.

Further, NumPy can take advantage of the vectorized instructions of modern CPUs, such as Intel's SSE and AVX, AMD's XOP, and so on. For example, multiple consecutive floating-point numbers can be loaded in 128, 256, or 512-bit registers for vectorized arithmetical computations implemented as CPU instructions.

NumPy can also be linked to highly-optimized linear algebra libraries such as BLAS and LAPACK through ATLAS or the Intel **Math Kernel Library** (**MKL**). Finally, a few specific matrix computations, including the matrix product `np.dot()`, may be multithreaded to take advantage of multicore processors.

All of these reasons explain why NumPy is so much faster than Python loops on vector operations.

Creating and loading arrays

In this section, we will see how to create and load NumPy arrays.

Creating arrays

First, there are several NumPy functions for creating common types of arrays. For example, `np.zeros(shape)` creates an array containing only zeros. The `shape` argument is a tuple giving the size of every axis. Hence, `np.zeros((3, 4))` creates an array of size `(3, 4)` (note the double parentheses, because we pass a tuple to the function).

Here are some further examples:

```
In [1]: import numpy as np
        print("ones", np.ones(5))
        print("arange", np.arange(5))
        print("linspace", np.linspace(0., 1., 5))
        print("random", np.random.uniform(size=3))
        print("custom", np.array([2, 3, 5]))
Out[1]: ones [ 1.  1.  1.  1.  1.]
        arange [0 1 2 3 4]
        linspace [ 0.    0.25  0.5   0.75  1.  ]
        random [ 0.68361911  0.33585308  0.70733934]
        custom [2 3 5]
```

The `np.arange()` and `np.linspace()` functions create arrays with regularly spaced numbers. The `np.random` module contains many functions for generating arrays containing independent (pseudo)-random values following various distributions (uniform, exponential, Gaussian, and many others).

The versatile `np.array()` function converts Python objects like lists or tuples into NumPy arrays. It is also used to create small arrays by specifying their values directly, as shown here:

```
In [2]: np.array([[1, 2], [3, 4]])
Out[2]: array([[1, 2],
               [3, 4]])
```

Every array has a fixed data type. You can specify the data type explicitly, or you can let NumPy figure out the data type automatically. For example, `np.ones()` generates an array of floating-point numbers by default, whereas `np.arange()` returns an array of integers. You can specify the data type explicitly as shown here:

```
In [3]: np.ones(5, dtype=np.int64)
Out[3]: array([1, 1, 1, 1, 1])
In [4]: np.arange(5).astype(np.float64)
Out[4]: array([ 0.,  1.,  2.,  3.,  4.])
```

The `astype()` method converts an array to any other type.

Here are a few references:

- Array creation routines at http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html
- NumPy random functions at http://docs.scipy.org/doc/numpy/reference/routines.random.html
- Data type objects at http://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html

- Data types at http://docs.scipy.org/doc/numpy/user/basics.types.html

Loading arrays from files

The `np.load()` and `np.save()` functions allow you to import and export NumPy arrays from/to binary files in a custom format.

Text and binary files can be imported into NumPy arrays. The `np.fromfile()` and `np.fromstring()` functions load arrays from binary/text files or strings. The `np.loadtxt()` and `np.genfromtxt()` functions load arrays from text files, including CSV files. For loading CSV files, text, or some other heterogeneous data, pandas is generally more effective than NumPy. Internally, pandas is based on NumPy. Therefore, data can be easily exchanged between pandas and NumPy structures. Here is an example:

```
In [5]: import pandas as pd
```

Let's load the NYC taxi dataset from Chapter 2, *Interactive Data Analysis with pandas*:

```
In [6]: data = pd.read_csv('../chapter2/data/nyc_data.csv')
```

Going from pandas to NumPy is particularly easy: just use the `.values` attribute, available on all DataFrame and Series objects. More specifically:

- A Series corresponds to a 1D NumPy array.
- A DataFrame corresponds to a 2D NumPy array.
- A Panel corresponds to a 3D NumPy array (we won't cover this pandas structure here).

Here, we obtain a `(N, 2)` NumPy array with the pickup coordinates of all trips:

```
In [7]: pickup = data[['pickup_longitude', 'pickup_latitude']].values
        pickup
Out[7]: array([[-73.955925,  40.781887],
               [-74.005501,  40.745735],
               ...,
               [-73.978477,  40.772945],
               [-73.987206,  40.750568]])
In [8]: pickup.shape
Out[8]: (846945, 2)
```

Here are a few references:

- Links between NumPy and pandas data structures at http://pandas.pydata.org/pandas-docs/stable/dsintro.html#dataframe-interoperability-with-numpy-functions
- Input/output routines at http://docs.scipy.org/doc/numpy/reference/routines.io.html

Basic array manipulations

Let's see some basic array manipulations around multiplication tables.

```
In [1]: import numpy as np
```

We first create an array of integers between 1 and 10, as shown here:

```
In [2]: x = np.arange(1, 11)
In [3]: x
Out[3]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Note that in `np.arange(start, end)`, `start` is included while `end` is excluded.

To create our multiplication table, we first need to transform `x` into a row and column vector. Our vector `x` is a 1D array, whereas row and column vectors are 2D arrays (also known as matrices). There are many ways to transform a 1D array to a 2D array. We will see the two most common methods here.

The first method is to use `reshape()`:

```
In [4]: x_row = x.reshape((1, -1))
        x_row
Out[4]: array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]])
```

The `reshape()` method takes the new shape as parameter. The total number of elements must be unchanged. For example, reshaping a `(2, 3)` array to a `(5,)` array would raise an error. The number `-1` can be used to tell NumPy to figure out automatically the size of that axis.

Here, note the double square brackets, indicating that `x_row` is a 2D array with just one row, while `x` was a 1D array.

In NumPy, the first axis is vertical, while the second axis is horizontal. However, 1D arrays are displayed horizontally, which is slightly confusing.

Another reshaping method is to use a special indexing syntax in NumPy:

```
In [5]: x_col = x[:, np.newaxis]
        x_col
Out[5]: array([[ 1],
               [ 2],
               [ 3],
               [ 4],
               [ 5],
               [ 6],
               [ 7],
               [ 8],
               [ 9],
               [10]])
```

Here, the colon `:` is used to select the entire first axis (vertical), whereas `np.newaxis` is used to create a new second axis (horizontal) with just one item.

We can now create our multiplication table. A first possibility would be to create an empty `(10, 10)` array and fill it with two `for` loops. However, doing it with NumPy leads to faster and much more con-

cise code.

We can use `np.dot()` to compute a matrix product between two vectors:

```
In [6]: np.dot(x_col, x_row)
Out[6]: array([[  1,   2,   3,   4,   5,   6,   7,   8,   9,  10],
               [  2,   4,   6,   8,  10,  12,  14,  16,  18,  20],
               [  3,   6,   9,  12,  15,  18,  21,  24,  27,  30],
               [  4,   8,  12,  16,  20,  24,  28,  32,  36,  40],
               [  5,  10,  15,  20,  25,  30,  35,  40,  45,  50],
               [  6,  12,  18,  24,  30,  36,  42,  48,  54,  60],
               [  7,  14,  21,  28,  35,  42,  49,  56,  63,  70],
               [  8,  16,  24,  32,  40,  48,  56,  64,  72,  80],
               [  9,  18,  27,  36,  45,  54,  63,  72,  81,  90],
               [ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100]])
```

Since `x_col` is a `(10, 1)` array (column vector) and `x_row` is a `(1, 10)` array (row vector), their matrix product is a `(10, 10)` array. Each element `(i, j)` (ith row, jth column) is the product of `x[i]` and `x[j]`, which is what we want for our multiplication table.

Another method is to use the regular NumPy multiplication with the `*` symbol. On arrays, this operation is to be understood as the element-wise multiplication, not the matrix multiplication. Here is an example:

```
In [7]: x_row * x_row
Out[7]: array([[  1,   4,   9,  16,  25,  36,  49,  64,  81, 100]])
```

This returns the squares of all numbers in `x_row`.

In our case, we can also use this operation to compute the multiplication table:

```
In [8]: x_row * x_col
Out[8]: array([[  1,   2,   3,  ...,   9,  10],
               [  2,   4,   6,  ...,  18,  20],
               ...
               [  9,  18,  27,  ...,  81,  90],
               [ 10,  20,  30,  ...,  90, 100]])
```

Why did multiplying a `(1, 10)` array by a `(10, 1)` array resulted in a `(10, 10)` array? The reason is called **broadcasting**. Element-wise array operations like the regular multiplication `*` normally requires arrays to have the same shape. However, NumPy also accepts arrays with compatible but not identical dimensions. The general rule is that *two dimensions are compatible when they are equal, or when one of them is 1*. The dimension equal to one is transparently and silently stretched to match the other dimension, and this operation does not involve any memory copy. Here, broadcasting allows us to compute the multiplication table with an element-wise multiplication operation.

You will find more information at the following pages:

- Array manipulation routines at http://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html
- Broadcasting rules at http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

Computing with NumPy arrays

We now get to the substance of array programming with NumPy. We will perform manipulations and computations on ndarrays.

Let's first import NumPy, pandas, matplotlib, and seaborn:

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

We load the NYC taxi dataset with pandas:

```
In [2]: data = pd.read_csv('../chapter2/data/nyc_data.csv',
                           parse_dates=['pickup_datetime',
                                        'dropoff_datetime'])
```

We get the pickup and dropoff locations of the taxi rides as ndarrays, using the `.values` attribute of pandas DataFrames:

```
In [3]: pickup = data[['pickup_longitude', 'pickup_latitude']].values
        dropoff = data[['dropoff_longitude',
                        'dropoff_latitude']].values
        pickup
Out[3]: array([[-73.955925,  40.781887],
               [-74.005501,  40.745735],
               [-73.969955,  40.79977 ],
               ...,
               [-73.993492,  40.729347],
               [-73.978477,  40.772945],
               [-73.987206,  40.750568]])
```

Selection and indexing

Let's illustrate selection and indexing with NumPy. These operations are similar to those offered by pandas on DataFrame and Series objects.

In NumPy, a given element can be retrieved with `pickup[i, j]`, where `i` is the 0-indexed row number, and `j` is the 0-indexed column number:

```
In [4]: print(pickup[3, 1])
Out[4]: 40.755081
```

A part of the array can be selected with the slicing syntax, which supports a start position, an end position, and an optional step, as shown here:

```
In [5]: pickup[1:7:2, 1:]
Out[5]: array([[ 40.745735],
               [ 40.755081],
               [ 40.768978]])
```

Here, we've selected the elements at `[1, 1]`, `[3, 1]`, and `[5, 1]`. The slicing syntax in Python is `start:end:step` where `start` is included and `end` is excluded. If `start` or `end` are omitted, they default to `0` or the length of the dimension, respectively, whereas `step` defaults to 1. For example, `1:` is equivalent to `1:n:1` where `n` is the size of the axis.

Let's select the longitudes of all pickup locations, in other words, the first column:

```
In [6]: lon = pickup[:, 0]
        lon
Out[6]: array([-73.9559, -74.0055, ..., -73.9784, -73.9872])
```

The result is a 1D ndarray.

We also get the second column of `pickup`:

```
In [7]: lat = pickup[:, 1]
        lat
Out[7]: array([ 40.7818, 40.7457, ..., 40.7729, 40.7505])
```

Boolean operations on arrays

Let's now illustrate filtering operations in NumPy. Again, these are similar to pandas. As an example, we're going to select all trips departing at a given location:

```
In [8]: lon_min, lon_max = (-73.98330, -73.98025)
        lat_min, lat_max = ( 40.76724,  40.76871)
```

In NumPy, symbols like arithmetic, inequality, and boolean operators work on ndarrays on an element-wise basis. Here is how to select all trips where the longitude is between `lon_min` and `lon_max`:

```
In [9]: in_lon = (lon_min <= lon) & (lon <= lon_max)
        in_lon
Out[9]: array([False, False, False, ..., False, False, False],
              dtype=bool)
```

The symbol `&` represents the AND boolean operator, while `|` represents the OR.

Here, the result is a Boolean vector containing as many elements as there are in the `lon` vector.

How many `True` elements are there in this array? NumPy arrays provide a `sum()` method that returns the sum of all elements in the array. When the array contains boolean values, `False` elements are converted to 0 and `True` elements are converted to 1. Therefore, the sum corresponds to the number of `True` elements:

```
In [10]: in_lon.sum()
Out[10]: 69163
```

We can process the latitudes similarly:

```
In [11]: in_lat = (lat_min <= lat) & (lat <= lat_max)
```

Then, we get all trips where both the longitude and latitude belong to our rectangle:

```
In [12]: in_lonlat = in_lon & in_lat
```

```
         in_lonlat.sum()
Out[12]: 3998
```

The `np.nonzero()` function returns the indices corresponding to `True` in a boolean array, as shown here:

```
In [13]: np.nonzero(in_lonlat)[0]
Out[13]: array([   901,   1011,   1066, ..., 845749, 845903, 846080])
```

Finally, we'll need the dropoff coordinates:

```
In [14]: lon1, lat1 = dropoff.T
```

This is a more concise way of writing `lon1 = dropoff[:, 0]; lat1 = dropoff[:, 1]`. The `T` attribute corresponds to the transpose of a matrix, which simply means that a matrix's columns become the corresponding rows of a new matrix, and the new columns are the original matrix's rows. Here, `dropoff.T` is a `(2, N)` array where the first row contains the longitude and the second row contains the latitude. In NumPy, an ndarray is iterable along the first dimension, in other words, along the rows of the matrix. Therefore, the syntax unpacking feature of Python allows us to concisely assign `lon1` to the first row and `lat1` to the second row.

Mathematical operations on arrays

We have the coordinates of all pickup and dropoff locations in NumPy arrays. Let's compute the straight line distance between those two locations, for every taxi trip.

There are several mathematical formulas giving the distance between two points given by their longitudes and latitudes. Here, we will compute a great-circle distance with a spherical Earth approximation.

The following function implements this formula.

```
In [15]: EARTH_R = 6372.8
         def geo_distance(lon0, lat0, lon1, lat1):
             """Return the distance (in km) between two points in
             geographical coordinates."""
             # from: http://en.wikipedia.org/wiki/Great-circle_distance
             # and: http://stackoverflow.com/a/8859667/1595060
             lat0 = np.radians(lat0)
             lon0 = np.radians(lon0)
             lat1 = np.radians(lat1)
             lon1 = np.radians(lon1)
             dlon = lon0 - lon1
             y = np.sqrt(
                 (np.cos(lat1) * np.sin(dlon)) ** 2
                  + (np.cos(lat0) * np.sin(lat1)
                  - np.sin(lat0) * np.cos(lat1) * np.cos(dlon)) ** 2)
             x = np.sin(lat0) * np.sin(lat1) + \
                 np.cos(lat0) * np.cos(lat1) * np.cos(dlon)
             c = np.arctan2(y, x)
             return EARTH_R * c
```

We have made extensive use of trigonometric functions provided by NumPy: `np.radians()` (converting numbers from degrees into radians), `np.cos()`,`np.sin()`,`np.arctan2(x, y)` (returning the

arctangent of `x/y`), and so on. These mathematical functions are defined on real numbers, but NumPy provides *vectorized* versions of them. These vectorized functions not only work on numbers but also on arbitrary numerical ndarrays. As we have explained earlier, these functions are orders of magnitude faster than Python loops. You will find the list of mathematical functions in NumPy at http://docs.scipy.org/doc/numpy/reference/routines.math.html.

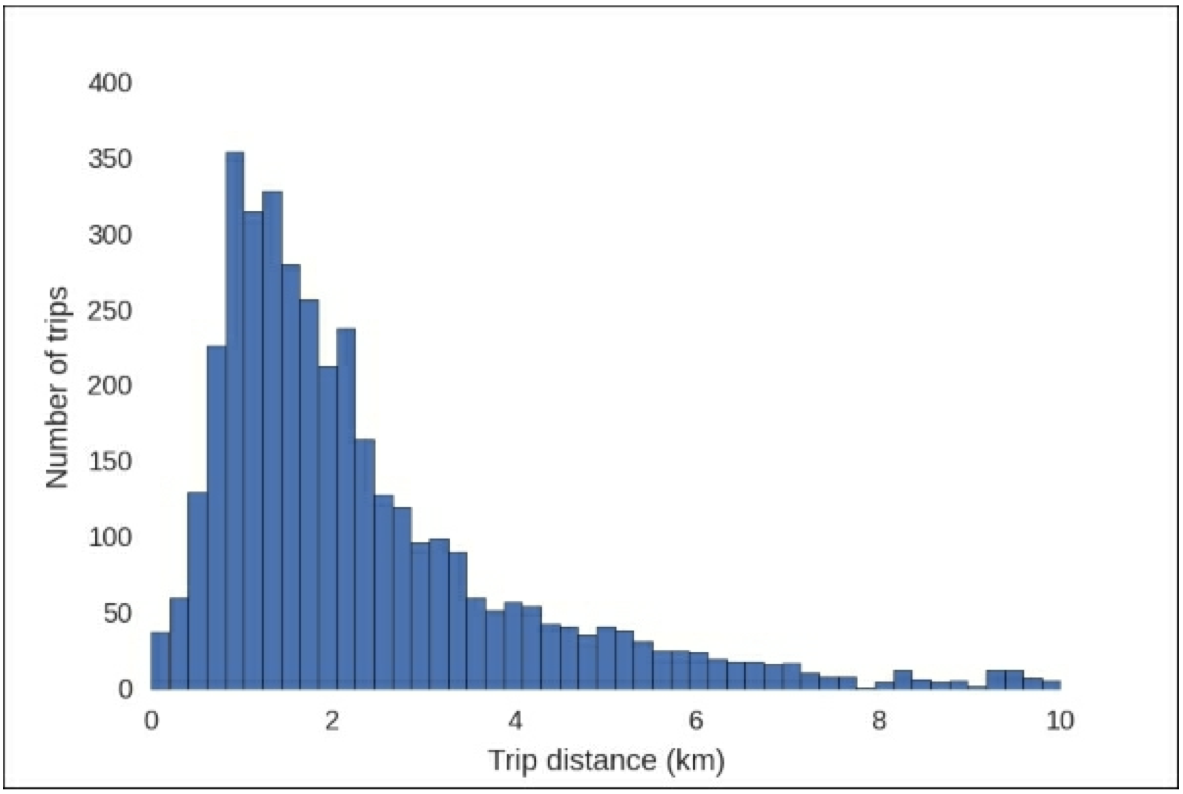
All in all, NumPy makes it quite natural to implement mathematical formulas on arrays of numbers. The syntax is exactly the same as with scalar operations.

Now, let's compute the straight line distances of all taxi trips:

```
In [16]: distances = geo_distance(lon, lat, lon1, lat1)
```

Below is a histogram of these distances for the trips starting at Columbus Circle (the location indicated by the geographical coordinates above). Those trips are indicated by the `in_lonlat` boolean array obtained earlier in this section.

```
In [17]: plt.hist(distances[in_lonlat], np.linspace(0., 10., 50))
         plt.xlabel('Trip distance (km)')
         plt.ylabel('Number of trips')
```



Histogram of trip distances

matplotlib's `plt.hist()` function computes a histogram and plots it. It is a convenient wrapper around NumPy's `np.histogram()` function that simply computes a histogram. You will find more statistical functions in NumPy at http://docs.scipy.org/doc/numpy/reference/routines.statistics.html.

A density map with NumPy

We have reviewed the most common array operations in this section. We will now see a more advanced example combining several techniques. We will compute and display a 2D density map of the most common pickup and dropoff locations, at specific times in the day.

First, let's select the evening taxi trips. This time, we use pandas, which offers particularly rich date and time features. We eventually get a NumPy array with the `.values` attribute:

```
In [18]: evening = (data.pickup_datetime.dt.hour >= 19).values
In [19]: n = np.sum(evening)
In [20]: n
Out[20]: 242818
```

Tip

Pandas and NumPy

Remember that pandas is based on NumPy, and that it is quite common to leverage both libraries in complex data analysis tasks. A natural workflow is to start loading and manipulating data with pandas, and then switch to NumPy when complex mathematical operations are to be performed on arrays. As a rule of thumb, pandas excels at filtering, selecting, grouping, and other data manipulations, whereas NumPy is particularly efficient at vector mathematical operations on numerical arrays.

The `n` variable contains the number of evening trips in our dataset.

Here is how we are going to create our density map: We consider the set of all pickup and dropoff locations for these `n` evening trips. There are `2n` of such points. Every point is associated with a weight of `-1` for pickup locations and `+1` for dropoff locations. The algebraic density of points at a given location, taking into account the weights, reflects whether people tend to leave or to arrive at this location.

To create the `weights` vector for our `2n` points, we first create a vector containing only zeros. Then, we set the first half of the array to `-1` (pickup) and the last half to `+1` (dropoff):

```
In [21]: weights = np.zeros(2 * n)
In [22]: weights[:n] = -1
         weights[n:] = +1
```

Note

Indexing in Python and NumPy starts at 0, and excludes the last element. The first half of `weights` is made of `weights[0]`, `weights[1]`, up to `weights[n-1]`. There are `n` of such elements. The slice `weights[:n]` is equivalent to `weights[0:n]`: it starts at `weights[0]`, and ends at `weights[n]` *excluded*, so the last element is effectively `weights[n-1]`.

We could also have used array manipulation routines provided by NumPy, such as `np.tile()` to concatenate copies of an array along several dimensions, or `np.repeat()` to make copies of every element along several dimensions. You will find the list of manipulation functions at http://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html.

Next, we create a `(2n, 2)` array defined by the vertical concatenation of the pickup and dropoff locations for the evening trips:

```
In [23]: points = np.r_[pickup[evening],
                        dropoff[evening]]
```

```
In [24]: points.shape
Out[24]: (485636, 2)
```

The concise `np.r_[]` syntax allows us to concatenate arrays along the first (vertical) dimension. We could also have used more explicit manipulation functions such as `np.vstack()` or `np.concatenate()`.

Now, we convert these points from geographical coordinates to pixel coordinates, using the same function as in the previous chapter:

```
In [25]: def lat_lon_to_pixels(lat, lon):
             lat_rad = lat * np.pi / 180.0
             lat_rad = np.log(np.tan((lat_rad + np.pi / 2.0) / 2.0))
             x = 100 * (lon + 180.0) / 360.0
             y = 100 * (lat_rad - np.pi) / (2.0 * np.pi)
             return (x, y)
In [26]: lon, lat = points.T
         x, y = lat_lon_to_pixels(lat, lon)
```

We now define the bins for the 2D histogram in our density map. This defines a 2D grid over which we compute the histogram.

```
In [27]: lon_min, lat_min = -74.0214, 40.6978
         lon_max, lat_max = -73.9524, 40.7982
In [28]: x_min, y_min = lat_lon_to_pixels(lat_min, lon_min)
         x_max, y_max = lat_lon_to_pixels(lat_max, lon_max)
In [29]: bin = .00003
         bins_x = np.arange(x_min, x_max, bin)
         bins_y = np.arange(y_min, y_max, bin)
```

These two arrays contain the horizontal and vertical bins.

Finally, we compute the histogram with the `np.histogram2d()` function. We pass as arguments the `y, x` coordinates of the points (reversed because we want the grid's first axis to represent the `y` coordinate), the weights, and the bins. This function computes a weighted sum of the points, in every bin. It returns several objects, the first of which is the density map we are interested in:

```
In [30]: grid, _, _ = np.histogram2d(y, x, weights=weights,
                                      bins=(bins_y, bins_x))
```

You will find the reference documentation of this function at http://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram2d.html#numpy.histogram2d.

Before displaying the density map, we will apply a logistic function to it in order to smooth it:

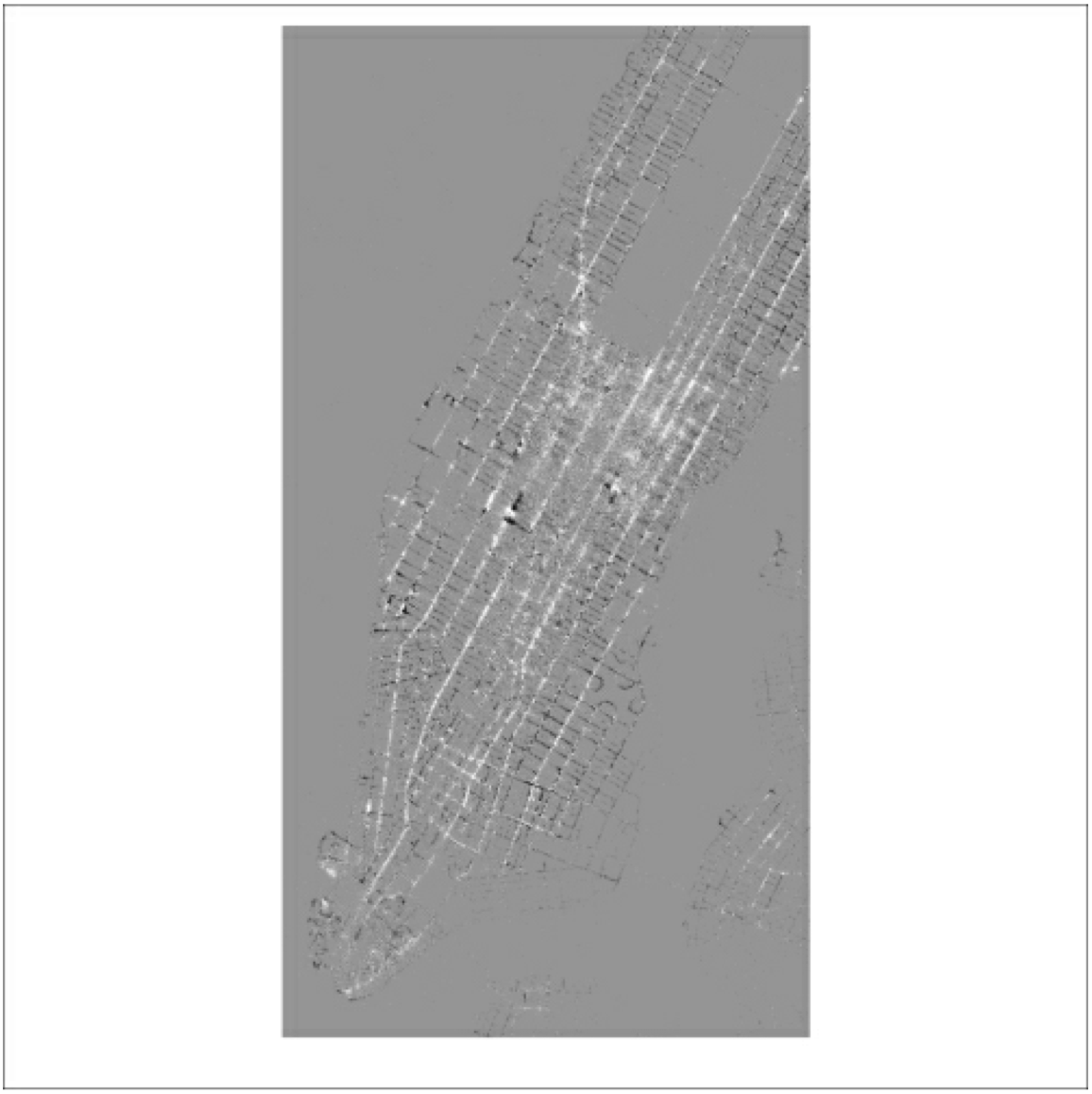
```
In [31]: density = 1. / (1. + np.exp(-.5 * grid))
```

This logistic function is called the **expit function**. It can also be found in the SciPy package at `scipy.special.expit()`. `scipy.special` provides many other special functions such as Bessel functions, Gamma functions, hypergeometric functions, and so on.

Finally, we display the density map with `plt.imshow()`:

```
In [32]: plt.imshow(density,
```

```
                origin='lower',
                interpolation='bicubic'
                )
        plt.axis('off')
```



Sources and sinks in taxi trip data

In this figure, white areas correspond to common dropoff locations whereas dark areas correspond to common pickup locations.

matplotlib's `plt.imshow()` function displays a matrix as an image. It supports several interpolation methods. Here, we used a bicubic interpolation. The `origin` argument is necessary because in our `density` matrix, the top-left corner corresponds to the smallest latitude, so it should correspond to the bottom-left corner in the image.

Other topics

We only scratched the surface of the possibilities offered by NumPy. Further numerical computing topics covered by NumPy and the more specialized SciPy library include:

- Search and sort in arrays
- Set operations
- Linear algebra
- Special mathematical functions
- Fourier transforms and signal processing
- Generation of pseudo-random numbers
- Statistics
- Numerical integration and numerical ODE solvers
- Function interpolation
- Basic image processing
- Numerical optimization

The *IPython Cookbook* covers many of these topics.

Here are a few references:

- NumPy reference at http://docs.scipy.org/doc/numpy/reference/
- SciPy reference at http://docs.scipy.org/doc/scipy/reference/
- *IPython Cookbook* at http://ipython-books.github.io/cookbook/

Summary

In this chapter, we introduced NumPy and the ndarray structure. We explained the main concepts of array computing and the performance benefits it brings over Python loops. We also showed how to use NumPy in conjunction with pandas for advanced data analysis tasks.

In the next chapter, we will explore several options for plotting, visualization, and graphical interfaces.