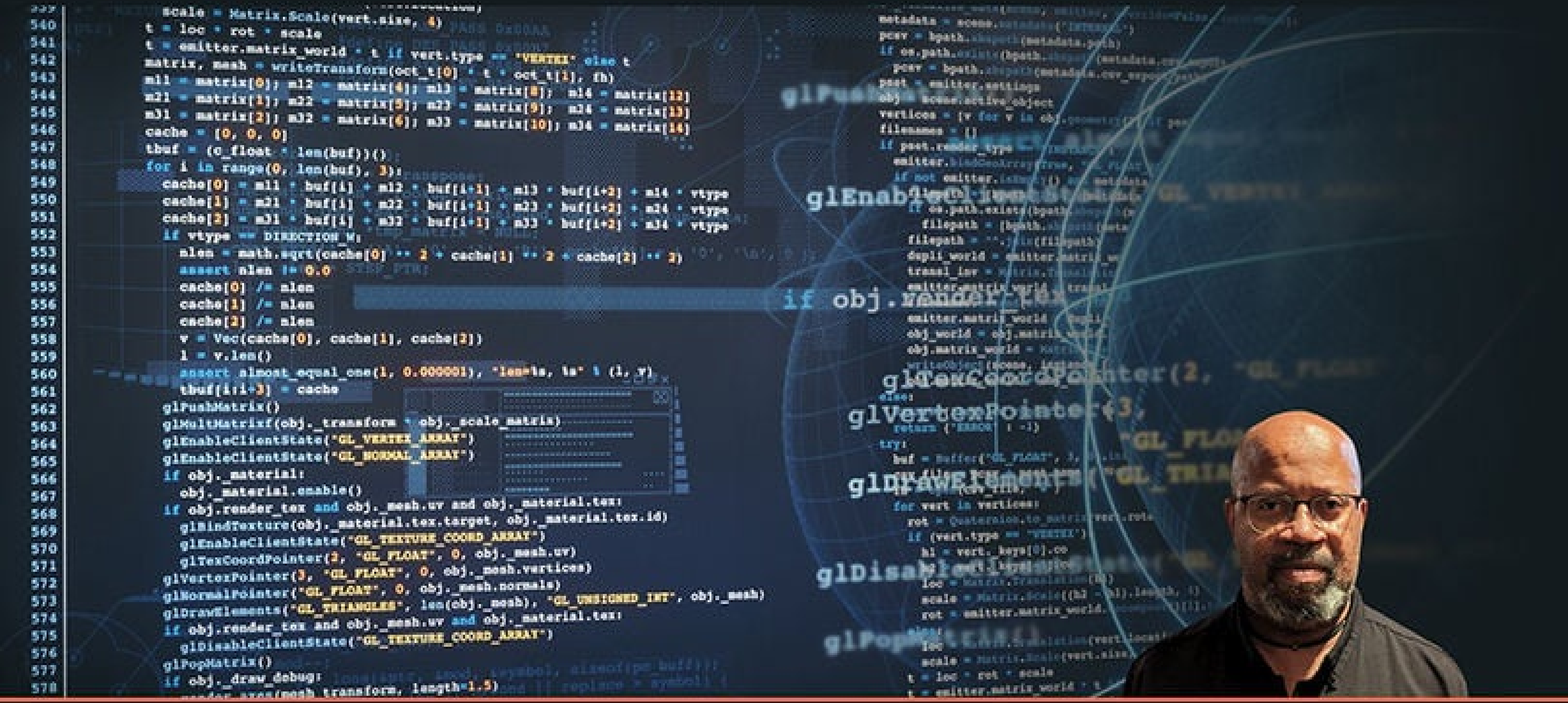
EXPERT INSIGHT

Early Access

Python Data Cleaning Cookbook

Detect and remove dirty data and extract key insights with pandas, machine learning and ChatGPT, Spark, and more

Second Edition



<packt>

Michael Walker

Python Data Cleaning Cookbook

Early Access Publication: Python Data Cleaning Cookbook

Early Access Production Reference: B18596

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-80323-987-3

Table of Contents

Python Data Cleaning Cookbook, Second Edition: Detect and remove dirty data and extract key insights with pandas, machine learning and ChatGPT, Spark, and more

Classes

1 Anticipating Data Cleaning Issues when Importing Tabular Data into Pandas

Scientific distributions of **Python** (Anaconda, WinPython, Canopy, and so on) provide analysts with an impressive range of data manipulation, exploration, and visualization tools. One important tool is pandas. Developed by Wes McKinney in 2008, but really gaining in popularity after 2012, pandas is now an essential library for data analysis in Python. We work with pandas extensively in this book, along with popular packages such as `numpy`, `matplotlib`, and `scipy`.A key pandas object is the **DataFrame**, which represents data as a tabular structure, with rows and columns. In this way, it is similar to the other data stores we discuss in this chapter. However, a pandas DataFrame also has indexing functionality that makes selecting, combining, and transforming data relatively straightforward, as the recipes in this book will demonstrate.Before we can make use of this great functionality, we have to get our data into pandas. Data comes to us in a wide variety of formats: as CSV or Excel files, as tables from SQL databases, from statistical analysis packages such as SPSS, Stata, SAS, or R, from non-tabular sources such as JSON, and from web pages.We examine tools for importing tabular data in this recipe. Specifically, we cover the following topics:

- Importing CSV files
- Importing Excel files
- Importing data from SQL databases

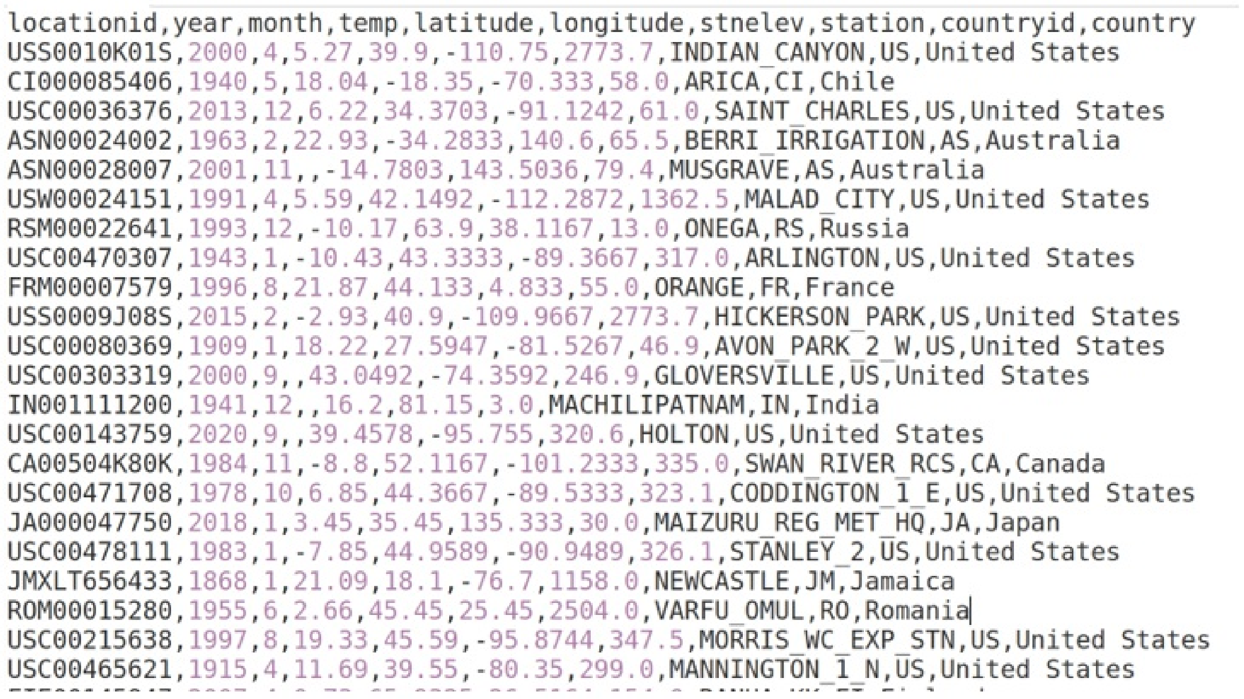
- Importing SPSS, Stata, and SAS data
- Importing R data
- Persisting tabular data

Importing CSV files

The `read_csv` method of the `pandas` library can be used to read a file with **comma separated values** (**CSV**) and load it into memory as a pandas DataFrame. In this recipe, we read a CSV file and address some common issues: creating column names that make sense to us, parsing dates, and dropping rows with critical missing data.Raw data is often stored as CSV files. These files have a carriage return at the end of each line of data to demarcate a row, and a comma between each data value to delineate columns. Something other than a comma can be used as the delimiter, such as a tab. Quotation marks may be placed around values, which can be helpful when the delimiter occurs naturally within certain values, which sometimes happens with commas.All data in a CSV file are characters, regardless of the logical data type. This is why it is easy to view a CSV file, presuming it is not too large, in a text editor. The pandas `read_csv` method will make an educated guess about the data type of each column, but you will need to help it along to ensure that these guesses are on the mark.

Getting ready

Create a folder for this chapter and create a new Python script or **Jupyter Notebook** file in that folder. Create a data subfolder and place the `landtempssample.csv` file in that subfolder. Alternatively, you could retrieve all of the files from the GitHub repository. Here is a screenshot of the beginning of the CSV file:



```
locationid,year,month,temp,latitude,longitude,stnelev,station,countryid,country
USS0010K01S,2000,4,5.27,39.9,-110.75,2773.7,INDIAN_CANYON,US,United States
CI000085406,1940,5,18.04,-18.35,-70.333,58.0,ARICA,CI,Chile
USC00036376,2013,12,6.22,34.3703,-91.1242,61.0,SAINT_CHARLES,US,United States
ASN00024002,1963,2,22.93,-34.2833,140.6,65.5,BERRI_IRRIGATION,AS,Australia
ASN00028007,2001,11,,-14.7803,143.5036,79.4,MUSGRAVE,AS,Australia
USW00024151,1991,4,5.59,42.1492,-112.2872,1362.5,MALAD_CITY,US,United States
RSM00022641,1993,12,-10.17,63.9,38.1167,13.0,ONEGA,RS,Russia
USC00470307,1943,1,-10.43,43.3333,-89.3667,317.0,ARLINGTON,US,United States
FRM00007579,1996,8,21.87,44.133,4.833,55.0,ORANGE,FR,France
USS0009J08S,2015,2,-2.93,40.9,-109.9667,2773.7,HICKERSON_PARK,US,United States
USC00080369,1909,1,18.22,27.5947,-81.5267,46.9,AVON_PARK_2_W,US,United States
USC00303319,2000,9,,43.0492,-74.3592,246.9,GLOVERSVILLE,US,United States
IN001111200,1941,12,,16.2,81.15,3.0,MACHILIPATNAM,IN,India
USC00143759,2020,9,,39.4578,-95.755,320.6,HOLTON,US,United States
CA00504K80K,1984,11,-8.8,52.1167,-101.2333,335.0,SWAN_RIVER_RCS,CA,Canada
USC00471708,1978,10,6.85,44.3667,-89.5333,323.1,CODDINGTON_1_E,US,United States
JA000047750,2018,1,3.45,35.45,135.333,30.0,MAIZURU_REG_MET_HQ,JA,Japan
USC00478111,1983,1,-7.85,44.9589,-90.9489,326.1,STANLEY_2,US,United States
JMXLT656433,1868,1,21.09,18.1,-76.7,1158.0,NEWCASTLE,JM,Jamaica
ROM00015280,1955,6,2.66,45.45,25.45,2504.0,VARFU_OMUL,RO,Romania
USC00215638,1997,8,19.33,45.59,-95.8744,347.5,MORRIS_WC_EXP_STN,US,United States
USC00465621,1915,4,11.69,39.55,-80.35,299.0,MANNINGTON_1_N,US,United States
```

Figure 1.1: Land Temperatures Data

Note

This dataset, taken from the Global Historical Climatology Network integrated database, is made available for public use by the United States National Oceanic and Atmospheric Administration at https://www.ncdc.noaa.gov/data-access/land-based-station-data/land-based-datasets/global-historical-climatology-network-monthly-version-4. This is just a 100,000-row sample of the full dataset, which is also available in the repository.

How to do it...

We will import a CSV file into pandas, taking advantage of some very useful `read_csv` options:

1. Import the `pandas` library and set up the environment to make viewing the output easier:

```
import pandas as pd
pd.options.display.float_format = '{:,.2f}'.format
pd.set_option('display.width', 85)
pd.set_option('display.max_columns', 8)
```

1. Read the data file, set new names for the headings, and parse the date column.

Pass an argument of `1` to the `skiprows` parameter to skip the first row, pass a list of columns to `parse_dates` to create a pandas datetime column from those columns, and set `low_memory` to `False` to reduce the usage of memory during the import process:

```
landtemps = pd.read_csv('data/landtempssample.csv',
...     names=['stationid','year','month','avgtemp','latitude',
...       'longitude','elevation','station','countryid','country
'],
...     skiprows=1,
...     parse_dates=[['month','year']],
...     low_memory=False)
>>> type(landtemps)
<class 'pandas.core.frame.DataFrame'>
```

Note

> We have to use `skiprows` because we are passing a list of column names to `read_csv`. If we use the column names in the CSV file we do not need to specify values for either `names` or `skiprows`.

1. Get a quick glimpse of the data.

- View the first few rows. Show the data type for all columns, as well as the number of rows and columns:

```
>>> landtemps.head(7)
  month_year    stationid  ...  countryid  \

0 2000-04-01  USS0010K01S  ...         US
1 1940-05-01  CI000085406  ...         CI
2 2013-12-01  USC00036376  ...         US
3 1963-02-01  ASN00024002  ...         AS
4 2001-11-01  ASN00028007  ...         AS
5 1991-04-01  USW00024151  ...         US
6 1993-12-01  RSM00022641  ...         RS
         country
```

```
0  United States
1          Chile
2  United States
3      Australia
4      Australia
5  United States
6         Russia
[7 rows x 9 columns]
```

```
>>> landtemps.dtypes
```

```
month_year    datetime64[ns]
stationid        object
avgtemp        float64
latitude        float64
longitude        float64
elevation        float64
station            object
countryid        object
country        object
dtype: object
```

```
>>> landtemps.shape
```

```
(100000, 9)
```

1. Give the date column a better name and view the summary statistics for average monthly temperature:

```
>>> landtemps.rename(columns={'month_year':'measuredate'}, inpla
ce=True)
>>> landtemps.dtypes
```

```
measuredate    datetime64[ns]
stationid        object
avgtemp        float64
latitude        float64
longitude        float64
elevation        float64
station            object
countryid        object
country        object
dtype: object
```

```
>>> landtemps.avgtemp.describe()
```

```
count   85,554.00
mean    10.92
```

```
std    11.52
min    -70.70
25%    3.46
50%    12.22
75%    19.57
max    39.95
Name: avgtemp, dtype: float64
```

1. Look for missing values for each column.

Use `isnull`, which returns `True` for each value that is missing for each column, and `False` when not missing. Chain this with `sum` to count the missings for each column. (When working with Boolean values, `sum` treats `True` as `1` and `False` as `0`. I will discuss method chaining in the *There's more...* section of this recipe):

```
>>> landtemps.isnull().sum()
```

```
measuredate    0
stationid    0
avgtemp    14446
latitude    0
longitude    0
elevation    0
station    0
countryid    0
country    5
dtype: int64
```

1. Remove rows with missing data for `avgtemp`.

Use the `subset` parameter to tell `dropna` to drop rows when `avgtemp` is missing. Set `inplace` to `True`. Leaving `inplace` at its default value of `False` would display the DataFrame, but the changes we have made would not be retained. Use the `shape` attribute of the DataFrame to get the number of rows and columns:

```
>>> landtemps.dropna(subset=['avgtemp'], inplace=True)
>>> landtemps.shape
```

```
(85554, 9)
```

That's it! Importing CSV files into pandas is as simple as that.

How it works...

Almost all of the recipes in this book use the `pandas` library. We refer to it as `pd` to make it easier to reference later. This is customary. We also use `float_format` to display float values in a readable way and `set_option` to make the terminal output wide enough to accommodate the number of variables.Much of the work is done by the first line in *step 2*. We use `read_csv` to load a pandas DataFrame in memory and call it `landtemps`. In addition to passing a filename, we set the `names` parameter to a list of our preferred column headings. We also tell `read_csv` to skip the first row, by setting `skiprows` to 1, since the original column headings are in the first row of the CSV file. If we do not tell it to skip the first row, `read_csv` will treat the header row in the file as actual data. `read_csv` also solves a date conversion issue for us. We use the `parse_dates` parameter to ask it to convert the `month` and `year` columns to a date value.*Step 3* runs through a few standard data checks. We use `head(7)` to print out all columns for the first 7 rows. We use the `dtypes` attribute of the data frame to show the data type of all columns. Each column has the expected data type. In pandas, character data has the object data type, a data type that allows for mixed values. `shape` returns a tuple, whose first element is the number of rows in the data frame (100,000 in this case) and whose second element is the number of columns (9).When we used `read_csv` to parse the `month` and `year` columns, it gave the resulting column the name `month_year`. We use the `rename` method in *step 4* to give that column a better name. We need to specify `inplace=True` to replace the old column name with the new column name in memory. The `describe` method provides summary statistics on the `avgtemp` column.Notice that the count for `avgtemp` indicates that there are 85,554 rows that have valid values for `avgtemp`. This is out of 100,000 rows for the whole DataFrame, as provided by the `shape` attribute. The listing of missing values for each column in *step 5* (`landtemps.isnull().sum()`) confirms this: *100,000 – 85,554 = 14,446*.*Step 6* drops all rows where `avgtemp` is `NaN`. (The `NaN` value, not a number, is the pandas representation of missing values.) `subset` is used to indicate which column to check for missings. The `shape` attribute for `landtemps` now indicates that there are 85,554 rows, which is what we would expect given the previous count from `describe`.

There's more...

If the file you are reading uses a delimiter other than a comma, such as a tab, this can be specified in the `sep` parameter of `read_csv`. When creating the pandas DataFrame, an index was also created. The numbers to the far left of the output when `head` and `sample` were run are index values. Any number of rows can be specified for `head` or `sample`. The default value is `5`.Setting `low_memory` to `False` causes `read_csv` to parse data in chunks. This is easier on systems with lower memory when working with larger files. However, the full DataFrame will still be loaded into memory once `read_csv` completes successfully.The `landtemps.isnull().sum()` statement is an example of chaining methods. First, `isnull` returns a DataFrame of `True` and `False` values, resulting from testing whether each column value is `null. sum` takes that DataFrame and sums the `True` values for each column, interpreting the `True` values as `1` and the `False` values as `0`. We would have obtained the same result if we had used the following two steps:

```
>>> checknull = landtemps.isnull()
>>> checknull.sum()
```

There is no hard and fast rule for when to chain methods and when not to do so. I find it helpful to chain when I really think of something I am doing as being a single step, but only two or more steps, mechanically speaking. Chaining also has the side benefit of not creating extra objects that I might not need.The dataset used in this recipe is just a sample from the full land temperatures database with almost 17 million records. You can run the larger file if your machine can handle it, with the following code:

```
>>> landtemps = pd.read_csv('data/landtemps.zip',
...   compression='zip', names=['stationid','year',
...     'month','avgtemp','latitude','longitude',
...     'elevation','station','countryid','country'],
...     skiprows=1,
...     parse_dates=[['month','year']],
...     low_memory=False)
```

`read_csv` can read a compressed ZIP file. We get it to do this by passing the name of the ZIP file and the type of compression.

See also

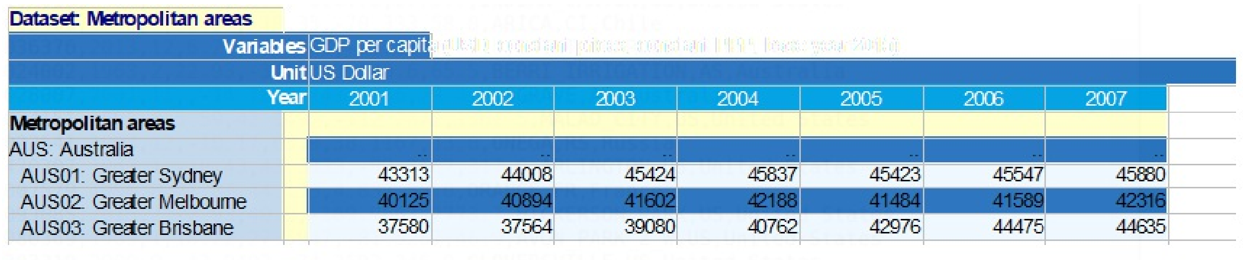
Subsequent recipes in this chapter, and in other chapters, set indexes to improve navigation over rows and merging.A significant amount of reshaping of the Global Historical Climatology Network raw data was done before using it in this recipe. We demonstrate this in *Chapter 8*, *Tidying and Reshaping Data*.

Importing Excel files

The `read_excel` method of the `pandas` library can be used to import data from an Excel file and load it into memory as a pandas DataFrame. In this recipe, we import an Excel file and handle some common issues when working with Excel files: extraneous header and footer information, selecting specific columns, removing rows with no data, and connecting to particular sheets.Despite the tabular structure of Excel, which invites the organization of data into rows and columns, spreadsheets are not datasets and do not require people to store data in that way. Even when some data conforms with those expectations, there is often additional information in rows or columns before or after the data to be imported. Data types are not always as clear as they are to the person who created the spreadsheet. This will be all too familiar to anyone who has ever battled with importing leading zeros. Moreover, Excel does not insist that all data in a column be of the same type, or that column headings be appropriate for use with a programming language such as Python.Fortunately, `read_excel` has a number of options for handling messiness in Excel data. These options make it relatively easy to skip rows and select particular columns, and to pull data from a particular sheet or sheets.

Getting ready

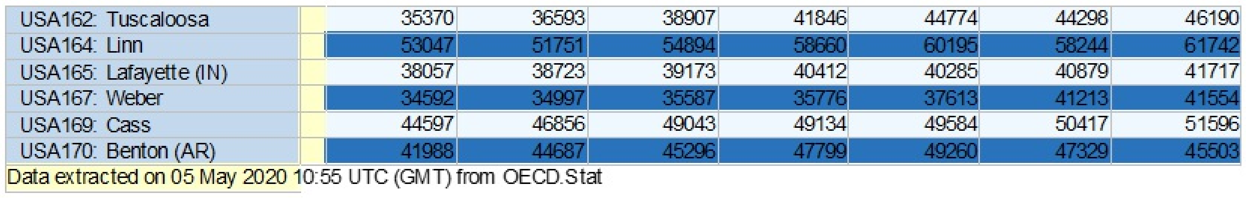
You can download the `GDPpercapita.xlsx` file, as well as the code for this recipe, from the GitHub repository for this book. The code assumes that the Excel file is in a data subfolder. Here is a view of the beginning of the file:



Dataset: Metropolitan areas								
Variables	GDP per capita (USD, constant prices, constant PPP, base year 2015)							
Unit	US Dollar							
Year		2001	2002	2003	2004	2005	2006	2007
Metropolitan areas								
AUS: Australia		..	..	..	..	..	..	..
AUS01: Greater Sydney		43313	44008	45424	45837	45423	45547	45880
AUS02: Greater Melbourne		40125	40894	41602	42188	41484	41589	42316
AUS03: Greater Brisbane		37580	37564	39080	40762	42976	44475	44635

Figure 1.2: View of the dataset

And here is a view of the end of the file:



USA162: Tuscaloosa		35370	36593	38907	41846	44774	44298	46190
USA164: Linn		53047	51751	54894	58660	60195	58244	61742
USA165: Lafayette (IN)		38057	38723	39173	40412	40285	40879	41717
USA167: Weber		34592	34997	35587	35776	37613	41213	41554
USA169: Cass		44597	46856	49043	49134	49584	50417	51596
USA170: Benton (AR)		41988	44687	45296	47799	49260	47329	45503
Data extracted on 05 May 2020 10:55 UTC (GMT) from OECD.Stat								

Figure 1.3: View of the dataset

Note

This dataset, from the Organisation for Economic Co-operation and Development, is available for public use at https://stats.oecd.org/.

How to do it…

We import an Excel file into pandas and do some initial data cleaning:

1. Import the `pandas` library:

```
>>> import pandas as pd
```

1. Read the Excel per capita GDP data.

Select the sheet with the data we need, but skip the columns and rows that we do not want. Use the `sheet_name` parameter to specify the sheet. Set `skiprows` to `4` and `skipfooter` to `1` to skip the first four rows (the first row is hidden) and the last row. We provide values for `usecols` to get data from column `A` and columns `C` through `T` (column `B` is blank). Use `head` to view the first few rows:

```
>>> percapitaGDP = pd.read_excel("data/GDPpercapita.xlsx",
...    sheet_name="OECD.Stat export",
...    skiprows=4,
...    skipfooter=1,
...    usecols="A,C:T")
>>> percapitaGDP.head()

    Year    2001      2017    2018
0    Metropolitan areas    NaN      NaN    NaN
1              AUS: Australia
2       AUS01: Greater Sydney    43313      50578    49860
3    AUS02: Greater Melbourne    40125      43025    42674
4     AUS03: Greater Brisbane    37580      46876    46640
[5 rows x 19 columns]
```

1. Use the `info` method of the data frame to view data types and the `non-null` count:

```
>>> percapitaGDP.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 702 entries, 0 to 701
Data columns (total 19 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0    Year    702    non-null    object
 1    2001    701    non-null    object
 2    2002    701    non-null    object
 3    2003    701    non-null    object
 4    2004    701    non-null    object
 5    2005    701    non-null    object
 6    2006    701    non-null    object
 7    2007    701    non-null    object
 8    2008    701    non-null    object
 9    2009    701    non-null    object
 10    2010    701    non-null    object
 11    2011    701    non-null    object
 12    2012    701    non-null    object
 13    2013    701    non-null    object
 14    2014    701    non-null    object
 15    2015    701    non-null    object
 16    2016    701    non-null    object
 17    2017    701    non-null    object
 18    2018    701    non-null    object
dtypes: object(19)
```

```
memory usage: 104.3+ KB
```

1. Rename the `Year` column to `metro` and remove the leading spaces.

Give an appropriate name to the metropolitan area column. There are extra spaces before the metro values in some cases, and extra spaces after the metro values in others. We can test for leading spaces with startswith(‘ ‘) and then use any to establish whether there are one or more occasions when the first character is blank. We can use endswith(‘ ‘) to examine trailing spaces. We use strip to remove both leading and trailing spaces. When we test for trailing spaces again we see that there are none:

```
>>> percapitaGDP.rename(columns={'Year':'metro'}, inplace=True)
>>> percapitaGDP.metro.str.startswith(' ').any()

True

>>> percapitaGDP.metro.str.endswith(' ').any()

True

>>> percapitaGDP.metro = percapitaGDP.metro.str.strip()
percapitaGDP.metro.str.endswith(' ').any()

False
```

1. Convert the data columns to numeric.

Iterate over all of the GDP year columns (2001-2018) and convert the data type from `object` to `float`. Coerce the conversion even when there is character data – the `..` in this example. We want character values in those columns to become missing, which is what happens. Rename the year columns to better reflect the data in those columns:

```
>>> for col in percapitaGDP.columns[1:]:
...   percapitaGDP[col] = pd.to_numeric(percapitaGDP[col],
...     errors='coerce')
...   percapitaGDP.rename(columns={col:'pcGDP'+col},
...     inplace=True)
...

    metro    pcGDP2001  ...  \
0    Metropolitan areas    NaN  ...
1    AUS: Australia    NaN  ...
```

```
2    AUS01:Greater Sydney    43313  ...
3    AUS02: Greater Melbourne    40125  ...
4    AUS03: Greater Brisbane    37580  ...
    pcGDP2017    pcGDP2018
0    NaN    NaN
1    NaN    NaN
2    50578    49860
3    43025    42674
4    46876    46640
[5 rows x 19 columns]
>>> percapitaGDP.dtypes
metro    object
pcGDP2001    float64
pcGDP2002    float64
abbreviated to save space
pcGDP2017    float64
pcGDP2018    float64
dtype: object
```

1. Use the `describe` method to generate summary statistics for all numeric data in the data frame:

```
>>> percapitaGDP.describe()

    pcGDP2001    pcGDP2002      pcGDP2017    pcGDP2018
count    424    440      445    441
mean    41264    41015      47489    48033
std    11878    12537      15464    15720
min    10988    11435      2745    2832
25%    33139    32636      37316    37908
50%    39544    39684      45385    46057
75%    47972    48611      56023    56638
max    91488    93566      122242    127468
[8 rows x 18 columns]
```

1. Remove rows where all of the per capita GDP values are missing.

Use the `subset` parameter of `dropna` to inspect all columns, starting with the second column (it is zero-based) through the last column. Use `how` to specify that we want to drop rows only if all of the columns specified in `subset` are missing. Use `shape` to show the number of rows and columns in the resulting DataFrame:

```
>>> percapitaGDP.dropna(subset=percapitaGDP.columns[1:], how="al
l", inplace=True)
```

```
>>> percapitaGDP.describe()

    pcGDP2001    pcGDP2002      pcGDP2017    pcGDP2018
count    424    440      445    441
mean    41264    41015      47489    48033
std    11878    12537      15464    15720
min    10988    11435      2745    2832
25%    33139    32636      37316    37908
50%    39544    39684      45385    46057
75%    47972    48611      56023    56638
max    91488    93566      122242    127468
[8 rows x 18 columns]

>>> percapitaGDP.head()
    metro    pcGDP2001    pcGDP2002  \

2    AUS01: Greater Sydney    43313    44008
3    AUS02: Greater Melbourne    40125    40894
4    AUS03:Greater Brisbane    37580    37564
5    AUS04: Greater Perth    45713    47371
6   AUS05: Greater Adelaide    36505    37194
   ...    pcGDP2016    pcGDP2017    pcGDP2018
2  ...    50519    50578    49860
3  ...    42671    43025    42674
4  ...    45723    46876    46640
5  ...    66032    66424    70390
6  ...    39737    40115    39924
[5 rows x 19 columns]

>>> percapitaGDP.shape

(480, 19)
```

1. Set the index for the DataFrame using the metropolitan area column.

Confirm that there are 480 valid values for `metro` and that there are 480 unique values, before setting the index:

```
>>> percapitaGDP.metro.count()
480
>>> percapitaGDP.metro.nunique()
480
>>> percapitaGDP.set_index('metro', inplace=True)
>>> percapitaGDP.head()

    pcGDP2001  ...    pcGDP2018
metro                                ...
```

```
AUS01: Greater Sydney    43313  ...    49860
AUS02: Greater Melbourne    40125  ...    42674
AUS03: Greater Brisbane    37580  ...    46640
AUS04: Greater Perth    45713  ...    70390
AUS05: Greater Adelaide    36505  ...    39924
[5 rows x 18 columns]
```

```
>>> percapitaGDP.loc['AUS02: Greater Melbourne']
```

```
pcGDP2001    40125
pcGDP2002    40894
...
pcGDP2017    43025
pcGDP2018    42674
Name: AUS02: Greater Melbourne, dtype: float64
```

We have now imported the Excel data into a pandas data frame and cleaned up some of the messiness in the spreadsheet.

How it works...

We mostly manage to get the data we want in *step 2* by skipping rows and columns we do not want, but there are still a number of issues: `read_excel` interprets all of the GDP data as character data, many rows are loaded with no useful data, and the column names do not represent the data well. In addition, the metropolitan area column might be useful as an index, but there are leading and trailing blanks and there may be missing or duplicated values. `read_excel` interprets `Year` as the column name for the metropolitan area data because it looks for a header above the data for that Excel column and finds `Year` there. We rename that column `metro` in *step 4*. We also use `strip` to fix the problem with leading and trailing blanks. If there had only been leading blanks, we could have used `lstrip`, or `rstrip` if there had only been trailing blanks. It is a good idea to assume that there might be leading or trailing blanks in any character data and clean that data shortly after the initial import.The spreadsheet authors used `..` to represent missing data. Since this is actually valid character data, those columns get the object data type (that is how pandas treats columns with character or mixed data). We coerce a conversion to numeric in *step 5*. This also results in the original values of `..` being replaced with `NaN` (not a number), pandas missing values for numbers. This is what we want.We can fix all of the per capita GDP columns with just a few lines because pandas makes it easy to iterate over the columns of a DataFrame. By specifying `[1:]`, we iterate from the second column to the last column. We can then change those columns to numeric and rename them to something more appropriate.There are several reasons why it is a good idea to clean up the column headings for the annual GDP columns: it helps us to remember what the data actually is; if we merge it with other data by metropolitan area, we will not have to worry about conflicting variable names; and we can use attribute access to work with pandas series based on those columns, which I will discuss in more detail in the *There's more…* section of this recipe. `describe` in *step 6* shows us that fewer than 500 rows have valid data for per capita GDP. When we drop all rows that have missing values for all per capita GDP columns in *step 7*, we end up with 480 rows in the DataFrame.

There's more…

Once we have a pandas DataFrame, we have the ability to treat columns as more than just columns. We can use attribute access (such as `percapitaGPA.metro`) or bracket notation (`percapitaGPA['metro']`) to get the functionality of a pandas data series. Either method makes it possible to use data series string inspecting methods such as `str.startswith`, and counting methods such as `nunique`. Note that the original column names of `20##` did not allow for attribute access because they started with a number, so `percapitaGDP.pcGDP2001.count()` works, but `percapitaGDP.2001.count()` returns a syntax error because `2001` is not a valid Python identifier (since it starts with a number).pandas is rich with features for string manipulation and for data series operations. We will try many of them out in subsequent recipes. This recipe showed those I find most useful when importing Excel data.

See also

There are good reasons to consider reshaping this data. Instead of 18 columns of GDP per capita data for each metropolitan area, we should have 18 rows of data for each metropolitan area, with columns for year and GDP per capita. Recipes for reshaping data can be found in *Chapter 9, Tidying and Reshaping Data*.

Importing data from SQL databases

In this recipe, we will use `pymssql` and `mysql apis` to read data from **Microsoft SQL Server** and **MySQL** (now owned by **Oracle**) databases, respectively. Data from sources such as these tends to be well structured since it is designed to facilitate simultaneous transactions by members of organizations, and those who interact with them. Each transaction is also likely related to some other organizational transaction.This means that although data tables from enterprise systems are more reliably structured than data from CSV files and Excel files, their logic is less likely to be self-contained. You need to know how the data from one table relates to data from another table to understand its full meaning. These relationships need to be preserved, including the integrity of primary and foreign keys, when pulling data. Moreover, well-structured data tables are not necessarily uncomplicated data tables. There are often sophisticated coding schemes that determine data values, and these coding schemes can change over time. For example, codes for staff ethnicity at a retail store chain might be different in 1998 than they are in 2020. Similarly, frequently there are codes for missing values, such as 99999, that pandas will understand as valid values.Since much of this logic is business logic, and implemented in stored procedures or other applications, it is lost when pulled out of this larger system. Some of what is lost will eventually have to be reconstructed when preparing data for analysis. This almost always involves combining data from multiple tables, so it is important to preserve the ability to do that. But it also may involve adding some of the coding logic back after loading the SQL table into a pandas DataFrame. We explore how to do that in this recipe.

Getting ready

This recipe assumes you have `pymssql` and `mysql apis` installed. If you do not, it is relatively straightforward to install them with `pip`. From the terminal, or `powershell` (in Windows), enter `pip install pymssql` or `pip install mysql-connector-python`.

Note

The dataset used in this recipe is available for public use at https://archive.ics.uci.edu/ml/machine-learning-databases/00320/.

How to do it…

We import SQL Server and MySQL data tables into a pandas data frame as follows:

1. Import `pandas`, `numpy`, `pymssql`, and `mysql`.

This step assumes that you have installed `pymssql` and `mysql apis`:

```
>>> import pandas as pd
>>> import numpy as np
>>> import pymssql
>>> import mysql.connector
```

1. Use `pymssql api` and `read_sql` to retrieve and load data from a SQL Server instance.

Select the columns we want from the SQL Server data and use SQL aliases to improve column names (for example, `fedu AS fathereducation`). Create a connection to the SQL Server data by passing database credentials to the `pymssql` connect function. Create a pandas data frame by passing the `select` statement and `connection` object to `read_sql`. Close the connection to return it to the pool on the server:

```
>>> sqlselect = "SELECT studentid, school, sex, age, famsize,\
...   medu AS mothereducation, fedu AS fathereducation,\
...   traveltime, studytime, failures, famrel, freetime,\
...   goout, g1 AS gradeperiod1, g2 AS gradeperiod2,\
...   g3 AS gradeperiod3 From studentmath"
>>>
>>> server = "pdcc.c9sqqzd5fulv.us-west-2.rds.amazonaws.com"
>>> user = "pdccuser"
>>> password = "pdccpass"
>>> database = "pdcctest"
>>>
>>> conn = pymssql.connect(server=server,
...   user=user, password=password, database=database)
>>>
>>> studentmath = pd.read_sql(sqlselect,conn)
>>> conn.close()
```

1. Check the data types and the first few rows:

```
>>> studentmath.dtypes

studentid    object
school    object
sex    object
age    int64
famsize    object
mothereducation    int64
fathereducation    int64
traveltime    int64
studytime    int64
failures    int64
famrel    int64
freetime    int64
goout    int64
gradeperiod1    int64
gradeperiod2    int64
gradeperiod3    int64
dtype: object

>>> studentmath.head()
    studentid    school  ...    gradeperiod2    gradeperiod3

0    001    GP  ...    6    6
1    002    GP  ...    5    6
2    003    GP  ...    8    10
3    004    GP  ...    14    15
4    005    GP  ...    10    10
[5 rows x 16 columns]
```

1. (Alternative) Use the `mysql` connector and `read_sql` to get data from MySQL.

Create a connection to the `mysql` data and pass that connection to `read_sql` to retrieve the data and load it into a pandas data frame. (The same data file on student math scores was uploaded to SQL Server and MySQL, so we can use the same SQL select statement we used in the previous step.):

```
>>> host = "pdccmysql.c9sqqzd5fulv.us-west-2.rds.amazonaws.com"
>>> user = "pdccuser"
>>> password = "pdccpass"
>>> database = "pdccschema"
>>> connmysql = mysql.connector.connect(host=host, \
...   database=database,user=user,password=password)
>>> studentmath = pd.read_sql(sqlselect,connmysql)
>>> connmysql.close()
```

1. Rearrange the columns, set an index, and check for missing values.

Move the grade data to the left of the DataFrame, just after `studentid`. Also move the `freetime` column to the right after `traveltime` and `studytime`. Confirm that each row has an ID and that the IDs are unique, and set `studentid` as the index:

```
>>> newcolorder = ['studentid', 'gradeperiod1',
...   'gradeperiod2','gradeperiod3', 'school',
...   'sex', 'age', 'famsize','mothereducation',
...   'fathereducation', 'traveltime',
...   'studytime', 'freetime', 'failures',
...   'famrel','goout']
>>> studentmath = studentmath[newcolorder]
>>> studentmath.studentid.count()

395

>>> studentmath.studentid.nunique()

395

>>> studentmath.set_index('studentid', inplace=True)
```

1. Use the data frame's `count` function to check for missing values:

```
>>> studentmath.count()

gradeperiod1    395
gradeperiod2    395
gradeperiod3    395
school    395
sex    395
age    395
famsize    395
mothereducation    395
fathereducation    395
traveltime    395
studytime    395
freetime    395
failures    395
famrel    395
```

```
goout    395
dtype: int64
```

1. Replace coded data values with more informative values.

Create a dictionary with the replacement values for the columns, and then use `replace` to set those values:

```
>>> setvalues= \
...   {"famrel":{1:"1:very bad",2:"2:bad",
...     3:"3:neutral",4:"4:good",5:"5:excellent"},
...   "freetime":{1:"1:very low",2:"2:low",
...     3:"3:neutral",4:"4:high",5:"5:very high"},
...   "goout":{1:"1:very low",2:"2:low",3:"3:neutral",
...     4:"4:high",5:"5:very high"},
...   "mothereducation":{0:np.nan,1:"1:k-4",2:"2:5-9",
...     3:"3:secondary ed",4:"4:higher ed"},
...   "fathereducation":{0:np.nan,1:"1:k-4",2:"2:5-9",
...     3:"3:secondary ed",4:"4:higher ed"}}

>>> studentmath.replace(setvalues, inplace=True)
```

1. Change the type for columns with the changed data to `category`.

Check any changes in memory usage:

```
>>> setvalueskeys = [k for k in setvalues]
>>> studentmath[setvalueskeys].memory_usage(index=False)

famrel    3160
freetime    3160
goout    3160
mothereducation    3160
fathereducation    3160
dtype: int64

>>> for col in studentmath[setvalueskeys].columns:
...   studentmath[col] = studentmath[col]. \
...     astype('category')
...
>>> studentmath[setvalueskeys].memory_usage(index=False)

famrel    607
freetime    607
goout    607
mothereducation    599
```

```
fathereducation    599
dtype: int64
```

1. Calculate percentages for values in the `famrel` column.

Run `value_counts` and set `normalize` to `True` to generate percentages:

```
>>> studentmath['famrel'].value_counts(sort=False, normalize=Tru
e)

1:very bad    0.02
2:bad    0.05
3:neutral    0.17
4:good    0.49
5:excellent    0.27
Name: famrel, dtype: float64
```

1. Use `apply` to calculate percentages for multiple columns:

```
>>> studentmath[['freetime','goout']].\
...   apply(pd.Series.value_counts, sort=False,
...   normalize=True)
             freetime  goout

1:very low    0.05    0.06
2:low    0.16    0.26
3:neutral    0.40    0.33
4:high    0.29    0.22
5:very high    0.10    0.13

>>> studentmath[['mothereducation','fathereducation']].\
...   apply(pd.Series.value_counts, sort=False,
...   normalize=True)

    mothereducation    fathereducation
1:k-4    0.15    0.21
2:5-9    0.26    0.29
3:secondary ed    0.25    0.25
4:higher ed    0.33    0.24
```

The preceding steps retrieved a data table from a SQL database, loaded that data into pandas, and did some initial data checking and cleaning.

How it works...

Since data from enterprise systems is typically better structured than CSV or Excel files, we do not need to do things such as skip rows or deal with different logical data types in a column. But some massaging is still usually required before we can begin exploratory analysis. There are often more columns than we need, and some column names are not intuitive or not ordered in the best way for analysis. The meaningfulness of many data values is not stored in the data table, to avoid entry errors and save on storage space. For example, `3` is stored for `mother's education` rather than `secondary education`. It is a good idea to reconstruct that coding as early in the cleaning process as possible.To pull data from a SQL database server, we need a connection object to authenticate us on the server, and a SQL select string. These can be passed to `read_sql` to retrieve the data and load it into a pandas DataFrame. I usually use the SQL `SELECT` statement to do a bit of cleanup of column names at this point. I sometimes also reorder columns, but I do that later in this recipe.We set the index in *step 5*, first confirming that every row has a value for `studentid` and that it is unique. This is often more important when working with enterprise data because we will almost always need to merge the retrieved data with other data files on the system. Although an index is not required for this merging, the discipline of setting one prepares us for the tricky business of merging data down the road. It will also likely improve the speed of the merge.We use the DataFrame's `count` function to check for missing values and there are no missing values – non-missing values is 395 (the number of rows) for every column. This is almost too good to be true. There may be values that are logically missing; that is, valid numbers that nonetheless connote missing values, such as -1, 0, 9, or 99. We address this possibility in the next step.*Step 7* demonstrates a useful technique for replacing data values for multiple columns. We create a dictionary to map original values to new values for each column, and then run it using `replace`. To reduce the amount of storage space taken up by the new verbose values, we convert the data type of those columns to `category`. We do this by generating a list of the keys of our `setvalues` dictionary – `setvalueskeys = [k for k in setvalues]` generates [`famrel`, `freetime`, `goout`, `mothereducation`, and `fathereducation`]. We then iterate over those five columns and use the `astype` method to change the data type to `category`. Notice that the memory usage for those columns is reduced substantially.Finally, we check the assignment of new values by using `value_counts` to view relative frequencies. We use `apply` because we want to run `value_counts` on multiple columns. To avoid `value_counts` sorting by frequency, we set sort to `False`.The DataFrame `replace` method is also a handy tool for dealing with logical missing values that will not be recognized as missing when retrieved by `read_sql`. `0` values for `mothereducation` and `fathereducation` seem to fall into that category. We fix this problem in the `setvalues` dictionary by indicating that `0` values for `mothereducation` and `fathereducation` should be replaced with `NaN`. It is important to address these kinds of missing values shortly after the initial import because they are not always obvious and can significantly impact all subsequent work.Users of packages such as *SPPS, SAS,* and *R* will notice the difference between this approach and value labels in SPSS and R, and `proc format` in SAS. In pandas, we need to change the actual data to get more informative values. However, we reduce how much data is actually stored by giving the column a `category` data type. This is similar to factors in R.

There's more...

I moved the grade data to near the beginning of the DataFrame. I find it helpful to have potential target or dependent variables in the leftmost columns, to keep them at the forefront of my thinking. It is also helpful to keep similar columns together. In this example, personal demographic variables (sex, age) are next to one another, as are family variables (`mothereducation`, `fathereducation`), and how students spend their time (`traveltime`, `studytime`, and `freetime`).You could have used `map` instead of `replace` in *step 7*. Prior to version 19.2 of pandas, `map` was significantly more efficient. Since then, the difference in efficiency has been much smaller. If you are working with a very large dataset, the difference may still be enough to consider using map.

See also

The recipes in *Chapter 8, Addressing Data Issues when Combining Data Frames*, go into detail on merging data. We will take a closer look at bivariate and multivariate relationships between variables in *Chapter 4, Identifying Missing Values and Outliers in Subsets of Data*. We demonstrate how to use some of these same approaches in packages such as SPSS, SAS, and R in subsequent recipes in this chapter.

Importing SPSS, Stata, and SAS data

We will use `pyreadstat` to read data from three popular statistical packages into pandas. The key advantage of `pyreadstat` is that it allows data analysts to import data from these packages without losing metadata, such as variable and value labels.The SPSS, Stata, and SAS data files we receive often come to us with the data issues of CSV and Excel files and SQL databases having been resolved. We do not typically have the invalid column names, changes in data types, and unclear missing values that we can get with CSV or Excel files, nor do we usually get the detachment of data from business logic, such as the meaning of data codes, that we often get with SQL data. When someone or some organization shares a data file from one of these packages with us, they have often added variable labels and value labels for categorical data. For example, a hypothetical data column called `presentsat` has the variable label `overall satisfaction with presentation` and value labels 1-5, with `1` being not at all satisfied and `5` being highly satisfied.The challenge is retaining that metadata when importing data from those systems into pandas. There is no precise equivalent to variable and value labels in pandas, and built-in tools for importing SAS, Stata, and SAS data lose the metadata. In this recipe, we will use `pyreadstat` to load variable and value label information and use a couple of techniques for representing that information in pandas.

Getting ready

This recipe assumes you have installed the `pyreadstat` package. If it is not installed, you can install it with `pip`. From the terminal, or `powershell` (in Windows), enter `pip install pyreadstat`. You will need the SPSS, Stata, and SAS data files for this recipe to run the code.We will work with data from the **United States National Longitudinal Survey of Youth** (**NLS**).

Note

> The National Longitudinal Survey of Youth is conducted by the United States Bureau of Labor Statistics. This survey started with a

cohort of individuals in 1997. Each survey respondent was high school age when they first completed the survey, having been born between 1980 and 1985. There were annual follow-up surveys each year through 2017. For this recipe, I pulled 42 variables on grades, employment, income, and attitudes toward government, from the hundreds of data items on the survey. Separate files for SPSS, Stata, and SAS can be downloaded from the repository. NLS data can be downloaded from https://www.nlsinfo.org/investigator/pages/search.

How to do it...

We will import data from SPSS, Stata, and SAS, retaining metadata such as value labels:

1. Import `pandas`, `numpy`, and `pyreadstat`.

This step assumes that you have installed `pyreadstat`:

```
>>> import pandas as pd
>>> import numpy as np
>>> import pyreadstat
```

1. Retrieve the SPSS data.

Pass a path and filename to the `read_sav` method of `pyreadstat`. Display the first few rows and a frequency distribution. Notice that the column names and value labels are non-descriptive, and that `read_sav` returns both a pandas DataFrame and a meta object:

```
>>> nls97spss, metaspss = pyreadstat.read_sav('data/nls97.sav')
>>> nls97spss.dtypes

R0000100    float64
R0536300    float64
R0536401    float64
...
U2962900    float64
U2963000    float64
Z9063900    float64
dtype: object
```

```
>>> nls97spss.head()

   R0000100  R0536300  ...  U2963000  Z9063900
0    1    2  ...    nan    52
1    2    1  ...    6    0
2    3    2  ...    6    0
3    4    2  ...    6    4
4    5    1  ...    5    12
[5 rows x 42 columns]

>>> nls97spss['R0536300'].value_counts(normalize=True)

1.00    0.51
2.00    0.49
Name: R0536300, dtype: float64
```

1. Grab the metadata to improve column labels and value labels.

The `metaspss` object created when we called `read_sav` has the column labels and the value labels from the SPSS file. Use the `variable_value_labels` dictionary to map values to value labels for one column (`R0536300`). (This does not change the data. It only improves our display when we run `value_counts` .) Use the `set_value_labels` method to actually apply the value labels to the DataFrame:

```
>>> metaspss.variable_value_labels['R0536300']
{0.0: 'No Information', 1.0: 'Male', 2.0: 'Female'}
>>> nls97spss['R0536300'].\
...   map(metaspss.variable_value_labels['R0536300']).\
...   value_counts(normalize=True)

Male    0.51
Female    0.49
Name: R0536300, dtype: float64

>>> nls97spss = pyreadstat.set_value_labels(nls97spss, metaspss,
 formats_as_category=True)
```

1. Use column labels in the metadata to rename the columns.

- To use the column labels from `metaspss` in our DataFrame, we can simply assign the column labels in `metaspss` to our DataFrame’s column names. Clean up the column names a bit by changing them to lowercase, changing spaces to underscores, and removing all remaining non-alphanumeric characters:

```
>>> nls97spss.columns = metaspss.column_labels
>>> nls97spss['KEY!SEX (SYMBOL) 1997'].value_counts(normalize=Tr
ue)

Male    0.51
Female    0.49
Name: KEY!SEX (SYMBOL) 1997, dtype: float64

>>> nls97spss.dtypes

PUBID - YTH ID CODE 1997    float64
KEY!SEX (SYMBOL) 1997    category
KEY!BDATE M/Y (SYMBOL) 1997    float64
KEY!BDATE M/Y (SYMBOL) 1997    float64
CV_SAMPLE_TYPE 1997    category
KEY!RACE_ETHNICITY (SYMBOL) 1997    category
HRS/WK R WATCHES TELEVISION 2017    category
HRS/NIGHT R SLEEPS 2017    float64
CVC_WKSWK_YR_ALL L99    float64
dtype: object

>>> nls97spss.columns = nls97spss.columns.\
...     str.lower().\
...     str.replace(' ','_').\
...     str.replace('[^a-z0-9_]', '')
>>> nls97spss.set_index('pubid__yth_id_code_1997', inplace=True)
```

1. Simplify the process by applying the value labels from the beginning.

The data values can actually be applied in the initial call to `read_sav` by setting `apply_value_formats` to `True`. This eliminates the need to call the `set_value_labels` function later:

```
>>> nls97spss, metaspss = pyreadstat.read_sav('data/nls97.sav',
apply_value_formats=True, formats_as_category=True)
>>> nls97spss.columns = metaspss.column_labels
>>> nls97spss.columns = nls97spss.columns.\
...   str.lower().\
...   str.replace(' ','_').\
...   str.replace('[^a-z0-9_]', '')
```

1. Show the columns and a few rows:

```
>>> nls97spss.dtypes
```

```
pubid__yth_id_code_1997    float64
keysex_symbol_1997    category
keybdate_my_symbol_1997    float64
keybdate_my_symbol_1997    float64
hrsnight_r_sleeps_2017    float64
cvc_wkswk_yr_all_l99    float64
dtype: object

>>> nls97spss.head()

   pubid__yth_id_code_1997 keysex_symbol_1997  ...  \
0    1    Female  ...
1    2    Male  ...
2    3    Female  ...
3    4    Female  ...
4    5    Male  ...

hrsnight_r_sleeps_2017  cvc_wkswk_yr_all_l99

0    nan    52
1    6    0
2    6    0
3    6    4
4    5    12
[5 rows x 42 columns]
```

1. Run frequencies on one of the columns and set the index:

```
>>> nls97spss.govt_responsibility__provide_jobs_2006.\
...   value_counts(sort=False)

Definitely should be    454
Definitely should not be    300
Probably should be    617
Probably should not be    462
Name: govt_responsibility__provide_jobs_2006, dtype: int64

>>> nls97spss.set_index('pubid__yth_id_code_1997', inplace=True)
```

1. Import the Stata data, apply value labels, and improve the column headings.

Use the same methods for the Stata data that we use for the SPSS data:

```
>>> nls97stata, metastata = pyreadstat.read_dta('data/nls97.dta'
, apply_value_formats=True, formats_as_category=True)
>>> nls97stata.columns = metastata.column_labels
>>> nls97stata.columns = nls97stata.columns.\
...     str.lower().\
...     str.replace(' ','_').\
...     str.replace('[^a-z0-9_]', '')
>>> nls97stata.dtypes

pubid__yth_id_code_1997    float64
keysex_symbol_1997    category
keybdate_my_symbol_1997    float64
keybdate_my_symbol_1997    float64
hrsnight_r_sleeps_2017    float64
cvc_wkswk_yr_all_l99    float64
dtype: object
```

1. View a few rows of the data and run a `frequency`:

```
>>> nls97stata.head()
    pubid__yth_id_code_1997 keysex_symbol_1997  ...  \

0    1    Female  ...
1    2    Male  ...
2    3    Female  ...
3    4    Female  ...
4    5    Male  ...

    hrsnight_r_sleeps_2017  cvc_wkswk_yr_all_l99

0    -5    52
1    6    0
2    6    0
3    6    4
4    5    12

[5 rows x 42 columns]
>>> nls97stata.govt_responsibility__provide_jobs_2006.\
...   value_counts(sort=False)

-5.0    1425
-4.0    5665
-2.0    56
-1.0    5
Definitely should be    454
Definitely should not be    300
Probably should be    617
Probably should not be    462
Name: govt_responsibility__provide_jobs_2006, dtype: int64
```

1. Fix the logical missing values that show up with the Stata data and set an index:

```
>>> nls97stata.min()

pubid__yth_id_code_1997    1
keysex_symbol_1997    Female
keybdate_my_symbol_1997    1
keybdate_my_symbol_1997    1,980
cv_bio_child_hh_2017    -5
cv_bio_child_nr_2017    -5
hrsnight_r_sleeps_2017    -5
cvc_wkswk_yr_all_l99    -4
dtype: object

>>> nls97stata.replace(list(range(-9,0)), np.nan, inplace=True)
>>> nls97stata.min()

pubid__yth_id_code_1997    1
keysex_symbol_1997    Female
keybdate_my_symbol_1997    1
keybdate_my_symbol_1997    1,980
cv_bio_child_hh_2017    0
cv_bio_child_nr_2017    0
hrsnight_r_sleeps_2017    0
cvc_wkswk_yr_all_l99    0
dtype: object

>>> nls97stata.set_index('pubid__yth_id_code_1997', inplace=True)
```

1. Retrieve the SAS data, using the SAS catalog file for value labels:

The data values for SAS are stored in a catalog file. Setting the catalog file path and filename retrieves the value labels and applies them:

```
>>> nls97sas, metasas = pyreadstat.read_sas7bdat('data/nls97.sas
7bdat', catalog_file='data/nlsformats3.sas7bcat', formats_as_cat
egory=True)
>>> nls97sas.columns = metasas.column_labels
>>>
>>> nls97sas.columns = nls97sas.columns.\
...     str.lower().\
...     str.replace(' ','_').\
...     str.replace('[^a-z0-9_]', '')
>>>
>>> nls97sas.head()
```

```
   pubid__yth_id_code_1997 keysex_symbol_1997  ...  \

0    1    Female  ...
1    2    Male  ...
2    3    Female  ...
3    4    Female  ...
4    5    Male  ...

   hrsnight_r_sleeps_2017  cvc_wkswk_yr_all_l99

0    nan    52
1    6    0
2    6    0
3    6    4
4    5    12
[5 rows x 42 columns]

>>> nls97sas.keysex_symbol_1997.value_counts()

Male    4599
Female    4385
Name: keysex_symbol_1997, dtype: int64

>>> nls97sas.set_index('pubid__yth_id_code_1997', inplace=True)
```

This demonstrates how to import SPSS, SAS, and Stata data without losing important metadata.

How it works...

The `read_sav`, `read_dta`, and `read_sas7bdat` methods of `Pyreadstat`, for SPSS, Stata, and SAS data files, respectively, work in a similar manner. Value labels can be applied when reading in the data by setting `apply_value_formats` to `True` for SPSS and Stata files (*steps 5 and 8*), or by providing a catalog file path and filename for SAS (*step 11*). We can set `formats_as_category` to `True` to change the data type to `category` for those columns where the data values will change. The meta object has the column names and the column labels from the statistical package, so metadata column labels can be assigned to pandas data frame column names at any point (`nls97spss.columns = metaspss.column_labels`). We can even revert to the original column headings after assigning meta column labels to them by setting pandas column names to the metadata column names (`nls97spss.columns = metaspss.column_names`).In *step 3*, we read the SPSS data without applying value labels. We looked at the dictionary for one variable (`metaspss.variable_value_labels['R0536300']`), but we could have viewed it for all variables (`metaspss.variable_value_labels`). When we are satisfied that the labels make sense, we can set them by calling the `set_value_labels` function. This is a good approach when you do not know the data well and want to inspect the labels before applying them.The column labels from the meta object are often a better choice than the original column headings. Column headings can be quite cryptic, particularly when the SPSS, Stata, or SAS file is based on a large survey, as in this example. But the labels are not usually ideal for column headings either. They sometimes have spaces, capitalization that is not helpful, and non-alphanumeric characters. We chain some string operations to switch to lowercase, replace spaces with underscores, and remove non-alphanumeric characters.Handling missing values is not always straightforward with these data files, since there are often many reasons why data is missing. If the file is from a survey, the missing value may be because of a survey skip pattern, or a respondent failed to respond, or the response was invalid, and so on. The National Longitudinal Survey has 9 possible values for missing, from -1 to -9. The SPSS import automatically set those values to `NaN` , while the Stata import retained the original values. (We could have gotten the SPSS import to retain those values by setting `user_missing` to `True` .) For the Stata data, we need to tell it to replace all values from -1 to -9 with `NaN` . We do this by using the DataFrame's `replace` function and passing it a list of integers from -9 to -1 (`list(range(-9,0))`).

There's more...

You may have noticed similarities between this recipe and the previous one in terms of how value labels are set. The `set_value_labels` function is like the DataFrame `replace` operation we used to set value labels in that recipe. We passed a dictionary to `replace` that mapped columns to value labels. The `set_value_labels` function in this recipe essentially does the same thing, using the `variable_value_labels` property of the meta object as the dictionary.Data from statistical packages is often not as well structured as SQL databases tend to be in one significant way. Since they are designed to facilitate analysis, they often violate database normalization rules. There is often an implied relational structure that might have to be *unflattened* at some point. For example, the data combines individual and event level data – person and hospital visits, brown bear and date emerged from hibernation. Often, this data will need to be reshaped for some aspects of the analysis.

See also

The `pyreadstat` package is nicely documented at https://github.com/Roche/pyreadstat. The package has many useful options for selecting columns and handling missing data that space did not permit me to demonstrate in this recipe. In *Chapter 8, Tidying and Reshaping Data* we will examine how to normalize data that may have been flattened for analytical purposes.

Importing R data

We will use `pyreadr` to read an R data file into pandas. Since `pyreadr` cannot capture the metadata, we will write code to reconstruct value labels (analogous to R factors) and column headings. This is similar to what we did in the *Importing data from SQL databases* recipe.The R statistical package is, in many ways, similar to the combination of Python and pandas, at least in its scope. Both have strong tools across a range of data preparation and data analysis tasks. Some data scientists work with both R and Python, perhaps doing data manipulation in Python and statistical analysis in R, or vice-versa, depending on their preferred packages. But there is currently a scarcity of tools for reading data saved in R, as `rds` or `rdata` files, into Python. The analyst often saves the data as a CSV file first, and then loads the CSV file into Python. We will use `pyreadr` , from the same author as `pyreadstat` , because it does not require an installation of R.When we receive an R file, or work with one we have created ourselves, we can count on it being fairly well structured, at least compared to CSV or Excel files. Each column will have only one data type, column headings will have appropriate names for Python variables, and all rows will have the same structure. However, we may need to restore some of the coding logic, as we did when working with SQL data.

Getting ready

This recipe assumes you have installed the `pyreadr` package. If it is not installed, you can install it with `pip`. From the terminal, or `powershell` (in Windows), enter `pip install pyreadr`.We will again work with the National Longitudinal Survey in this recipe. You will need to download the `rds` file used in this recipe from the GitHub repository in order to run the code.

How to do it...

We will import data from R without losing important metadata:

1. Load `pandas`, `numpy`, `pprint`, and the `pyreadr` package:

```
>>> import pandas as pd
>>> import numpy as np
>>> import pyreadr
>>> import pprint
```

1. Get the R data.

Pass the path and filename to the `read_r` method to retrieve the R data and load it into memory as a pandas DataFrame. `read_r` can return one or more objects. When reading an `rds` file (as opposed to an `rdata` file), it will return one object, having the key `None`. We indicate `None` to get the pandas DataFrame:

```
>>> nls97r = pyreadr.read_r('data/nls97.rds')[None]
>>> nls97r.dtypes

R0000100    int32
R0536300    int32
...
U2962800    int32
U2962900    int32
U2963000    int32
Z9063900    int32

dtype: object
>>> nls97r.head(10)
   R0000100  R0536300  ...  U2963000  Z9063900

0    1    2  ...    -5    52
```

```
1    2    1  ...    6    0
2    3    2  ...    6    0
3    4    2  ...    6    4
4    5    1  ...    5    12
5    6    2  ...    6    6
6    7    1  ...    -5    0
7    8    2  ...    -5    39
8    9    1  ...    4    0
9    10    1  ...    6    0
[10 rows x 42 columns]
```

1. Set up dictionaries for value labels and column headings.

Load a dictionary that maps columns to the value labels and create a list of preferred column names as follows:

```
>>> with open('data/nlscodes.txt', 'r') as reader:
...     setvalues = eval(reader.read())
...
>>> pprint.pprint(setvalues)

{'R0536300': {0.0: 'No Information', 1.0: 'Male', 2.0: 'Female'}
,
 'R1235800': {0.0: 'Oversample', 1.0: 'Cross-sectional'},
 'S8646900': {1.0: '1. Definitely',
              2.0: '2. Probably ',
              3.0: '3. Probably not',
              4.0: '4. Definitely not'}}

>>> newcols = ['personid','gender','birthmonth',
...   'birthyear','sampletype','category',
...   'satverbal','satmath','gpaoverall',
...   'gpaeng','gpamath','gpascience','govjobs',
...   'govprices','govhealth','goveld','govind',
...   'govunemp','govinc','govcollege',
...   'govhousing','govenvironment','bacredits',
...   'coltype1','coltype2','coltype3','coltype4',
...   'coltype5','coltype6','highestgrade',
...   'maritalstatus','childnumhome','childnumaway',
...   'degreecol1','degreecol2','degreecol3',
...   'degreecol4','wageincome','weeklyhrscomputer',
...   'weeklyhrstv','nightlyhrssleep',
...   'weeksworkedlastyear']
```

1. Set value labels and missing values, and change selected columns to `category` data type.

Use the `setvalues` dictionary to replace existing values with value labels. Replace all values from -9 to -1 with `NaN`:

```
>>> nls97r.replace(setvalues, inplace=True)
>>> nls97r.head()

   R0000100   R0536300  ...  U2963000  Z9063900
0    1    Female  ...    -5    52
1    2    Male  ...    6    0
2    3    Female  ...    6    0
3    4    Female  ...    6    4
4    5    Male  ...    5    12
[5 rows x 42 columns]

>>> nls97r.replace(list(range(-9,0)), np.nan, inplace=True)
>>> for col in nls97r[[k for k in setvalues]].columns:
...     nls97r[col] = nls97r[col].astype('category')
...
>>> nls97r.dtypes

R0000100    int64
R0536300    category
R0536401    int64
R0536402    int64
R1235800    category
              ...
U2857300    category
U2962800    category
U2962900    category
U2963000    float64
Z9063900    float64
Length: 42, dtype: object
```

1. Set meaningful column headings:

```
>>> nls97r.columns = newcols
>>> nls97r.dtypes

personid    int64
gender    category
birthmonth    int64
birthyear    int64
sampletype    category
                          ...
wageincome    category
weeklyhrscomputer    category
weeklyhrstv    category
```

```
nightlyhrssleep    float64
weeksworkedlastyear    float64
Length: 42, dtype: object
```

This shows how R data files can be imported into pandas and value labels assigned.

How it works...

Reading R data into pandas with `pyreadr` is fairly straightforward. Passing a filename to the `read_r` function is all that is required. Since `read_r` can return multiple objects with one call, we need to specify which object. When reading an `rds` file (as opposed to an `rdata` file), only one object is returned. It has the key `None` .In *step 3*, we load a dictionary that maps our variables to value labels, and a list for our preferred column headings. In *step 4* we apply the value labels. We also change the data type to `category` for the columns where we applied the values. We do this by generating a list of the keys of our `setvalues` dictionary with `[k for k in setvalues]` and then iterating over those columns.We change the column headings in *step 5* to ones that are more intuitive. Note that the order matters here. We need to set the value labels before changing the column names, since the `setvalues` dictionary is based on the original column headings.The main advantage of using `pyreadr` to read R files directly into pandas is that we do not have to convert the R data into a CSV file first. Once we have written our Python code to read the file, we can just rerun it whenever the R data changes. This is particularly helpful when we do not have R on the machine where we are working.

There's more...

`Pyreadr` is able to return multiple data frames. This is useful when we save several data objects in R as an `rdata` file. We can return all of them with one call. `Pprint` is a handy tool for improving the display of Python dictionaries.

See also

Clear instructions and examples for `pyreadr` are available at https://github.com/ofajardo/pyreadr.Feather files, a relatively new format, can be read by both R and Python. I discuss those files in the next recipe.We could have used `rpy2` instead of `pyreadr` to import R data. `rpy2` requires that R also be installed, but it is more powerful than `pyreadr`. It will read R factors and automatically set them to pandas DataFrame values. See the following code:

```
>>> import rpy2.robjects as robjects
>>> from rpy2.robjects import pandas2ri
>>> pandas2ri.activate()
>>> readRDS = robjects.r['readRDS']
>>> nls97withvalues = readRDS('data/nls97withvalues.rds')
>>> nls97withvalues
```

```
      R0000100      R0536300  ...    U2963000  Z9063900
1    1    Female  ...    -2147483648    52
2    2    Male  ...    6    0
3    3    Female  ...    6    0
4    4    Female  ...    6    4
5    5    Male  ...    5    12
...        ...      ...  ...          ...        ...
8980    9018    Female  ...    4    49
8981    9019    Male  ...    6    0
8982    9020    Male  ...    -2147483648    15
8983    9021    Male  ...    7    50
8984    9022    Female  ...    7    20
[8984 rows x 42 columns]
```

This generates unusual *-2147483648* values. This is what happened when `readRDS` interpreted missing data in numeric columns. A global replace of that number with `NaN`, after confirming that that is not a valid value, would be a good next step.

Persisting tabular data

We persist data, copy it from memory to local or remote storage, for several reasons: to be able to access the data without having to repeat the steps we used to generate it; to share the data with others; or to make it available for use with different software. In this recipe, we save data that we have loaded into a pandas data frame as different file types (CSV, Excel, Pickle, and Feather).Another important, but sometimes overlooked, reason to persist data is to preserve some segment of our data that needs to be examined more closely; perhaps it needs to be scrutinized by others before our analysis can be completed. For analysts who work with operational data in medium- to large-sized organizations, this process is part of the daily data cleaning workflow.In addition to these reasons for persisting data, our decisions about when and how to serialize data are shaped by several other factors: where we are in terms of our data analysis projects, the hardware and software resources of the machine(s) saving and reloading the data, and the size of our dataset. Analysts end up having to be much more intentional when saving data than they are when pressing *CTRL+S* in their word processing application.Once we persist data, it is stored separately from the logic that we used to create it. I find this to be one of the most important threats to the integrity of our analysis. Often, we end up loading data that we saved some time in the past (a week ago? a month ago? a year ago?) and forget how a variable was defined and how it relates to other variables. If we are in the middle of a data cleaning task, it is best not to persist our data, so long as our workstation and network can easily handle the burden of regenerating the data. It is a good idea to persist data only once we have reached milestones in our work.Beyond the question of *when* to persist data, there is the question of *how*. If we are persisting it for our own reuse with the same software, it is best to save it in a binary format native to that software. That is pretty straightforward for tools such as SPSS, SAS, Stata, and R, but not so much for pandas. But that is good news in a way. We have lots of choices, from CSV and Excel to pickle and feather. We save to all these file types in this recipe.

Getting ready

You will need to install feather if you do not have it on your system. You can do that by entering `pip install pyarrow` in a terminal window or `powershell` (in Windows). If you do not already have a subfolder named *Views* in your `chapter 1` folder, you will need to create it in order to run the code for this recipe.

Note

This dataset, taken from the Global Historical Climatology

Network integrated database, is made available for public use by the United States National Oceanic and Atmospheric Administration at https://www.ncdc.noaa.gov/data-access/land-based-station-data/land-based-datasets/global-historical-climatology-network-monthly-version-4. This is just a 100,000-row sample of the full dataset, which is also available in the repository.

How to do it...

We will load a CSV file into pandas and then save it as a pickle and as a feather file. We will also save subsets of the data to CSV and Excel formats:

1. Import pandas and `pyarrow` and adjust the display.

`Pyarrow` needs to be imported in order to save pandas to feather:

```
>>> import pandas as pd
>>> import pyarrow
```

1. Load the land temperatures CSV file into pandas, drop rows with missing data, and set an index:

```
>>> landtemps = \
...   pd.read_csv('data/landtempssample.csv',
...     names=['stationid','year','month','avgtemp',
...       'latitude','longitude','elevation',
...       'station','countryid','country'],
...     skiprows=1,
...     parse_dates=[['month','year']],
...     low_memory=False)
>>> landtemps.rename(columns={'month_year':'measuredate'}, inpla
ce=True)
>>> landtemps.dropna(subset=['avgtemp'], inplace=True)
>>> landtemps.dtypes

measuredate    datetime64[ns]
stationid    object
avgtemp    float64
latitude    float64
longitude    float64
elevation    float64
station    object
countryid    object
```

```
country    object
dtype: object
```

```
>>> landtemps.set_index(['measuredate','stationid'], inplace=Tru
e)
```

1. Write extreme values for temperature to CSV and Excel files.

Use the `quantile` method to select outlier rows, those at the 1 in 1,000 level at each end of the distribution:

```
>>> extremevals = landtemps[(landtemps.avgtemp < landtemps.avgte
mp.quantile(.001)) | (landtemps.avgtemp > landtemps.avgtemp.quan
tile(.999))]
>>> extremevals.shape
(171, 7)
>>> extremevals.sample(7)

                         avgtemp  ...  country
measuredate stationid             ...
2013-08-01    QAM00041170    35.30  ...    Qatar
2005-01-01    RSM00024966    -40.09  ...    Russia
1973-03-01    CA002401200    -40.26  ...    Canada
2007-06-01    KU000405820    37.35  ...    Kuwait
1987-07-01    SUM00062700    35.50  ...    Sudan
1998-02-01    RSM00025325    -35.71  ...    Russia
1968-12-01    RSM00024329    -43.20  ...    Russia
[7 rows x 7 columns]
```

```
>>> extremevals.to_excel('views/tempext.xlsx')
>>> extremevals.to_csv('views/tempext.csv')
```

1. Save to pickle and feather files.

The index needs to be reset in order to save a feather file:

```
>>> landtemps.to_pickle('data/landtemps.pkl')
>>> landtemps.reset_index(inplace=True)
>>> landtemps.to_feather("data/landtemps.ftr")
```

1. Load the pickle and feather files we just saved.

Notice that our index was preserved when saving and loading the pickle file:

```
>>> landtemps = pd.read_pickle('data/landtemps.pkl')
>>> landtemps.head(2).T

measuredate    2000-04-01    1940-05-01
stationid    USS0010K01S    CI000085406
avgtemp    5.27    18.04
latitude    39.90    -18.35
longitude    -110.75    -70.33
elevation    2,773.70    58.00
station    INDIAN_CANYON    ARICA
countryid    US    CI
country    United States    Chile

>>> landtemps = pd.read_feather("data/landtemps.ftr")
>>> landtemps.head(2).T

                               0                    1
measuredate    2000-04-01 00:00:00    1940-05-01 00:00:00
stationid    USS0010K01S    CI000085406
avgtemp    5.27    18.04
latitude    39.90    -18.35
longitude    -110.75    -70.33
elevation    2,773.70    58.00
station    INDIAN_CANYON    ARICA
countryid    US    CI
country    United States    Chile
```

The previous steps demonstrate how to serialize pandas data frames using two different formats, pickle and feather.

How it works...

Persisting pandas data is quite straightforward. DataFrames have `to_csv`, `to_excel`, `to_pickle`, and `to_feather` methods. Pickling preserves our index.

There's more...

The advantage of storing data in CSV files is that saving it uses up very little additional memory. The disadvantage is that writing CSV files is quite slow and we lose important metadata, such as data types. (`read_csv` can often figure out the data type when we reload the file, but not always.) Pickle files keep that data, but can burden a system that is low on resources when serializing. Feather is easier on resources, and can be easily loaded in R as well as Python, but we have to sacrifice our index in order to serialize. Also, the authors of feather make no promises regarding long-term support.You may have noticed that I do not make a recommendation about what to use for data serialization – other than to limit your persistence of full datasets to project milestones. This is definitely one of those “right tools for the right job” kind of situations. I use CSV or Excel files when I want to share a segment of a file with colleagues for discussion. I use feather for ongoing Python projects, particularly when I am using a machine with sub-par RAM and an outdated chip, and I am also using R. When I am wrapping up a project, I pickle the DataFrames.

2 Anticipating Data Cleaning Issues when Working with HTML, JSON, and Spark Data

This chapter continues our work on importing data from a variety of sources, and the initial checks we should do on the data after importing it. Gradually, over the last 25 years, data analysts have found that they increasingly need to work with data in non-tabular, semi-structured forms. Sometimes they even create and persist data in those forms. We work with a common alternative to traditional tabular datasets in this chapter, JSON, but the general concepts can be extended to XML and NoSQL data stores such as MongoDB. We also go over common issues that occur when scraping data from websites.Data analysts have also been finding that increases in the volume of data to be analyzed have been even greater than improvement in machine processing power, at least those computing resources that are available locally. Working with big data sometimes requires us to rely on technology like Apache Spark, which can take advantage of distributed resources.In this chapter, we will work through the following recipes:

- Importing simple JSON data
- Importing more complicated JSON data from an API
- Importing data from web pages
- Working with data in Spark
- Persisting JSON data

Importing simple JSON data

JavaScript Object Notation (**JSON**) has turned out to be an incredibly useful standard for transferring data from one machine, process, or node to another. Often a client sends a data request to a server, upon which that server queries the data in the local storage and then converts it from something like a **SQL Server table** or tables into JSON, which the client can consume. This is sometimes complicated further by the first server (say, a web server) forwarding the request to a database server. JSON facilitates this, as does XML, by doing the following:

- Being readable by humans
- Being consumable by most client devices
- Not being limited in structure

JSON is quite flexible, which means that it can accommodate just about anything, no matter how unwise. The structure can even change within a JSON file, so different keys might be present at different points. For example, the file might begin with some explanatory keys that have a very different structure than the remaining *data* keys. Or some keys might be present in some cases, but not others. We go over some approaches for dealing with that messiness (uh, flexibility).

Getting ready…

We are going to work with data on news stories about political candidates in this recipe. This data is made available for public use at `dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/` . I have combined the JSON files there into one file and randomly selected 60,000 news stories from the combined data. This sample (`allcandidatenewssample.json`) is available in the GitHub repository of this book.We will do a little work with list and dictionary comprehensions in this recipe. *DataCamp* has good guides to list comprehensions (https://www.datacamp.com/community/tutorials/python-list-comprehension) and dictionary comprehensions (https://www.datacamp.com/community/tutorials/python-dictionary-comprehension) if you are feeling a little rusty.

How to do it...

We will import a JSON file into pandas after doing some data checking and cleaning:

1. Import the `json` and `pprint` libraries.

`pprint` improves the display of the lists and dictionaries that are returned when we load JSON data:

```
import pandas as pd
import numpy as np
import json
import pprint
from collections import Counter
```

1. Load the JSON data and look for potential issues.

Use the `json load` method to return data on news stories about political candidates. `load` returns a list of dictionaries. Use `len` to get the size of the list, which is the total number of news stories in this case. (Each list item is a dictionary with keys for the title, source, and so on, and their respective values.) Use `pprint` to display the first two dictionaries. Get the value from the source key for the first list item:

```
with open('data/allcandidatenewssample.json') as f:
...   candidatenews = json.load(f)
...
len(candidatenews)

60000

pprint.pprint(candidatenews[0:2])

[{'date': '2019-12-25 10:00:00',
  'domain': 'www.nbcnews.com',
  'panel_position': 1,
  'query': 'Michael Bloomberg',
  'source': 'NBC News',
  'story_position': 6,
  'time': '18 hours ago',
  'title': 'Bloomberg cuts ties with company using prison inmate
s to make campaign calls',
  'url': 'https://www.nbcnews.com/politics/2020-election/bloombe
rg-cuts-ties-company-using-prison-inmates-make-campaign-calls-n1
106971'},
 {'date': '2019-11-09 08:00:00',
  'domain': 'www.townandcountrymag.com',
  'panel_position': 1,
  'query': 'Amy Klobuchar',
  'source': 'Town & Country Magazine',
  'story_position': 3,
  'time': '18 hours ago',
  'title': "Democratic Candidates React to Michael Bloomberg's P
otential Run",
  'url': 'https://www.townandcountrymag.com/society/politics/a29
739854/michael-bloomberg-democratic-candidates-campaign-reaction
s/'}]
```

```
pprint.pprint(candidatenews[0]['source'])
```

```
'NBC News'
```

1. Check for differences in the structure of the dictionaries.

Use `Counter` to check for any dictionaries in the list with fewer than, or more than, the 9 keys that is normal. Look at a few of the dictionaries with almost no data (those with just two keys) before removing them. Confirm that the remaining list of dictionaries has the expected length – *60000-2382=57618*:

```
Counter([len(item) for item in candidatenews])
```

```
Counter({9: 57202, 2: 2382, 10: 416})
```

```
pprint.pprint(next(item for item in candidatenews if len(item)<9
))
```

```
{'date': '2019-09-11 18:00:00', 'reason': 'Not collected'}
```

```
pprint.pprint(next(item for item in candidatenews if len(item)>9
))
```

```
{'category': 'Satire',
 'date': '2019-08-21 04:00:00',
 'domain': 'politics.theonion.com',
 'panel_position': 1,
 'query': 'John Hickenlooper',
 'source': 'Politics | The Onion',
 'story_position': 8,
 'time': '4 days ago',
 'title': '‘And Then There Were 23,’ Says Wayne Messam Crossing
Out '
          'Hickenlooper Photo \n'
          'In Elaborate Grid Of Rivals',
 'url': 'https://politics.theonion.com/and-then-there-were-23-sa
ys-wayne-messam-crossing-ou-1837311060'}
```

```
pprint.pprint([item for item in candidatenews if len(item)==2][0
:10])
```

```
[{'date': '2019-09-11 18:00:00', 'reason': 'Not collected'},
 {'date': '2019-07-24 00:00:00', 'reason': 'No Top stories'},
...
 {'date': '2019-01-03 00:00:00', 'reason': 'No Top stories'}]
```

```
candidatenews = [item for item in candidatenews if len(item)>2]
len(candidatenews)
```

```
57618
```

1. Generate counts from the JSON data.

Get the dictionaries just for *Politico* (a website that covers political news) and display a couple of dictionaries:

```
politico = [item for item in candidatenews if item["source"] ==
"Politico"]
len(politico)
```

```
2732
```

```
pprint.pprint(politico[0:2])
```

```
[{'date': '2019-05-18 18:00:00',
  'domain': 'www.politico.com',
  'panel_position': 1,
  'query': 'Marianne Williamson',
  'source': 'Politico',
  'story_position': 7,
  'time': '1 week ago',
  'title': 'Marianne Williamson reaches donor threshold for Dem
debates',
  'url': 'https://www.politico.com/story/2019/05/09/marianne-wil
liamson-2020-election-1315133'},
 {'date': '2018-12-27 06:00:00',
  'domain': 'www.politico.com',
  'panel_position': 1,
  'query': 'Julian Castro',
  'source': 'Politico',
  'story_position': 1,
  'time': '1 hour ago',
  'title': "O'Rourke and Castro on collision course in Texas",
  'url': 'https://www.politico.com/story/2018/12/27/orourke-juli
an-castro-collision-texas-election-1073720'}]
```

1. Get the `source` data and confirm that it has the anticipated length.

Show the first few items in the new `sources` list. Generate a count of news stories by source and display the 10 most popular sources. Notice that stories from *The Hill* can have `TheHill` (without a space) or `The Hill` as the value for `source`:

```
sources = [item.get('source') for item in candidatenews]
type(sources)
```

```
<class 'list'>
```

```
len(sources)
```

```
57618
```

```
sources[0:5]
```

```
['NBC News', 'Town & Country Magazine', 'TheHill', 'CNBC.com', '
Fox News']
```

```
pprint.pprint(Counter(sources).most_common(10))
```

```
[('Fox News', 3530),
 ('CNN.com', 2750),
 ('Politico', 2732),
 ('TheHill', 2383),
 ('The New York Times', 1804),
 ('Washington Post', 1770),
 ('Washington Examiner', 1655),
 ('The Hill', 1342),
 ('New York Post', 1275),
 ('Vox', 941)]
```

1. Fix any errors in the values in the dictionary.

Fix the `source` values for `The Hill`. Notice that `The Hill` is now the most frequent source for news stories:

```
for newsdict in candidatenews:
...     newsdict.update((k, "The Hill") for k, v in newsdict.ite
ms()
...      if k == "source" and v == "TheHill")
...
sources = [item.get('source') for item in candidatenews]
pprint.pprint(Counter(sources).most_common(10))
```

```
[('The Hill', 3725),
 ('Fox News', 3530),
 ('CNN.com', 2750),
 ('Politico', 2732),
 ('The New York Times', 1804),
 ('Washington Post', 1770),
 ('Washington Examiner', 1655),
 ('New York Post', 1275),
 ('Vox', 941),
 ('Breitbart', 799)]
```

1. Create a pandas DataFrame.

Pass the JSON data to the pandas `DataFrame` method. Convert the `date` column to a `datetime` data type:

```
candidatenewsdf = pd.DataFrame(candidatenews)
candidatenewsdf.dtypes
```

```
title              object
url                object
source             object
time               object
date               object
query              object
story_position      int64
panel_position     object
domain             object
category           object
dtype: object
```

1. Confirm that we are getting the expected values for `source`.

Also, rename the `date` column:

```
candidatenewsdf.rename(columns={'date':'storydate'}, inplace=Tru
e)
candidatenewsdf.storydate = candidatenewsdf.storydate.astype('da
tetime64[ns]')
candidatenewsdf.shape
```

```
(57618, 10)
```

```
candidatenewsdf.source.value_counts(sort=True).head(10)
```

```
The Hill               3725
Fox News               3530
CNN.com                2750
Politico               2732
The New York Times     1804
Washington Post        1770
Washington Examiner    1655
New York Post          1275
Vox                     941
Breitbart               799
Name: source, dtype: int64
```

We now have a pandas DataFrame with only the news stories where there is meaningful data, and with the values for `source` fixed.

How it works...

The `json.load` method returns a list of dictionaries. This makes it possible to use a number of familiar tools when working with this data: list methods, slicing, list comprehensions, dictionary updates, and so on. There are times, maybe when you just have to populate a list or count the number of individuals in a given category, when there is no need to use pandas.In *steps 2* to *6*, we use list methods to do many of the same checks we have done with pandas in previous recipes. In *step 3* we use `Counter` with a list comprehension (`Counter([len(item) for item in candidatenews])`) to get the number of keys in each dictionary. This tells us that there are 2,382 dictionaries with just 2 keys and 416 with 10. We use `next` to look for an example of dictionaries with fewer than 9 keys or more than 9 keys to get a sense of the structure of those items. We use slicing to show 10 dictionaries with 2 keys to see if there is any data in those dictionaries. We then select only those dictionaries with more than 2 keys.In *step 4* we create a subset of the list of dictionaries, one that just has `source` equal to `Politico`, and take a look at a couple of items. We then create a list with just the source data and use `Counter` to list the 10 most common sources in *step 5*.*Step 6* demonstrates how to replace key values conditionally in a list of dictionaries. In this case, we update the key value to `The Hill` whenever the `key (k)` is `source` and `value (v)` is `TheHill`. The `for k, v in newsdict.items()` section is the unsung hero of this line. It loops through all key/value pairs for all dictionaries in `candidatenews` .It is easy to create a pandas DataFrame by passing the list of dictionaries to the pandas `DataFrame` method. We do this in *step 7*. The main complication is that we need to convert the date column from a string to a date, since dates are just strings in JSON.

There's more...

In *steps 5* and *6* we use `item.get('source')` instead of `item['source']`. This is handy when there might be missing keys in a dictionary. `get` returns `None` when the key is missing, but we can use an optional second argument to specify a value to return.I renamed the `date` column to `storydate` in *step 8*. This is not necessary, but is a good idea. Not only does `date` not tell you anything about what the dates actually represent; it is also so generic a column name that it is bound to cause problems at some point.The news stories data fits nicely into a tabular structure. It makes sense to represent each list item as one row, and the key/value pairs as columns and column values for that row. There are no significant complications, such as key values that are themselves lists of dictionaries. Imagine an authors key for each story with a list item for each author as the key value, and that list item is a dictionary of information about the author. This is not at all unusual when working with JSON data in Python. The next recipe shows how to work with data structured in this way.

Importing more complicated JSON data from an API

In the previous recipe, we discussed one significant advantage (and challenge) of working with JSON data – its flexibility. A JSON file can have just about any structure its authors can imagine. This often means that this data does not have the tabular structure of the data sources we have discussed so far, and that pandas DataFrames have. Often, analysts and application developers use JSON precisely because it does not insist on a tabular structure. I know I do! Retrieving data from multiple tables often requires us to do a one-to-many merge. Saving that data to one table or file means duplicating data on the “one” side of the one-to-many relationship. For example, student demographic data is merged with data on the courses studied, and the demographic data is repeated for each course. With JSON, duplication is not required to capture these items of data in one file. We can have data on the courses studied nested within the data for each student.But doing analysis with JSON structured in this way will eventually require us to either: 1) manipulate the data in a very different way than we are used to doing; or 2) convert the JSON to a tabular form. We examine the first approach in the *Classes that handle non-tabular data structures* recipe in *Chapter 10*, *User Defined Functions and Classes to Automate Data Cleaning*. This recipe takes the second approach. It uses a very handy tool for converting selected nodes of JSON to a tabular structure – `json_normalize`.We first use an API to get JSON data because that is how JSON is frequently consumed. One advantage of retrieving the data with an API, rather than working from a file we have saved locally, is that it is easier to rerun our code when the source data is refreshed.

Getting ready...

This recipe assumes you have the `requests` and `pprint` libraries already installed. If they are not installed, you can install them with pip. From the terminal (or PowerShell in Windows), enter `pip install requests` and `pip install pprint`.The following is the structure of the JSON file that is created when using the collections API of the Cleveland Museum of Art. There is a helpful *info* section at the beginning, but we are interested in the *data* section. This data does not fit nicely into a tabular data structure. There may be several `citations` objects and several `creators` objects for each collection object. I have abbreviated the JSON file to save space:

```
{"info": { "total": 778, "parameters": {"african_american_artist
s": "" }},
"data": [
{
"id": 165157,
"accession_number": "2007.158",
"title": "Fulton and Nostrand",
"creation_date": "1958",
“citations": [
  {
   "citation": "Annual Exhibition: Sculpture, Paintings...",
   "page_number": "Unpaginated, [8],[12]",
   "url": null
   },
  {
   "citation": "\"Moscow to See Modern U.S. Art,\"<em> New York.
..",
   "page_number": "P. 60",
   "url": null
  }]
"creators": [
      {
     "description": "Jacob Lawrence (American, 1917-2000)",
     "extent": null,
     "qualifier": null,
     "role": "artist",
     "birth_year": "1917",
     "death_year": "2000"
     }
  ]
 }
```

Note

The API used in this recipe is provided by the Cleveland Museum of Art. It is available for public use at https://openaccess-api.clevelandart.org/.

How to do it…

Create a DataFrame from the museum’s collections data with one row for each `citation`, and the `title` and `creation_date` duplicated:

1. Import the `json`, `requests`, and `pprint` libraries.

We need the `requests` library to use an API to retrieve JSON data. `pprint` improves the display of lists and dictionaries:

```
import pandas as pd
import numpy as np
import json
import pprint
import requests
```

1. Use an API to load the JSON data.

Make a `get` request to the collections API of the Cleveland Museum of Art. Use the query string to indicate that you just want collections from African-American artists. Display the first collection item. I have truncated the output for the first item to save space:

```
response = requests.get("https://openaccess-api.clevelandart.org
/api/artworks/?african_american_artists")
camcollections = json.loads(response.text)
print(len(camcollections['data']))
```

```
778
```

```
pprint.pprint(camcollections['data'][0])
```

```
{'accession_number': '2007.158',
 'catalogue_raisonne': None,
 'citations': [
   {'citation': 'Annual Exhibition: Sculpture...',
    'page_number': 'Unpaginated, [8],[12]',
    'url': None},
  {'citation': '"Moscow to See Modern U.S....',
    'page_number': 'P. 60',
    'url': None}]
 'collection': 'American - Painting',
 'creation_date': '1958',
 'creators': [
  {'biography': 'Jacob Lawrence (born 1917)...',
   'birth_year': '1917',
   'description': 'Jacob Lawrence (American...)',
   'role': 'artist'}],
 'type': 'Painting'}
```

1. Flatten the JSON data.

Create a DataFrame from the JSON data using the `json_normalize` method. Indicate that the number of citations will determine the number of rows, and that `accession_number`, `title`, `creation_date`, `collection`, `creators`, and `type` will be repeated. Observe that the data has been flattened by displaying the first two observations, transposing them with the `.T` option to make it easier to view:

```
camcollectionsdf = \
...   pd.json_normalize(camcollections['data'],
...    'citations',
...    ['accession_number','title','creation_date',
...    'collection','creators','type'])
camcollectionsdf.head(2).T

                          0                        1
citation  Annual Exhibiti...  "Moscow to See Modern...
page_number     Unpaginated,                    P. 60
url                     None                     None
accession_number   2007.158                  2007.158
title      Fulton and No...          Fulton and No...
creation_date           1958                     1958
collection American - Pa...          American - Pa...
creators [{'description':'J...   [{'description':'J...
type               Painting                  Painting
```

Pull the `birth_year` value from `creators`:

```
creator = camcollectionsdf[:1].creators[0]
type(creator[0])

<class 'dict'>

pprint.pprint(creator)

[{'biography': 'Jacob Lawrence (born 1917) has been a prominent
art...',
  'birth_year': '1917',
  'death_year': '2000',
  'description': 'Jacob Lawrence (American, 1917-2000)',
  'extent': None,
  'name_in_original_language': None,
  'qualifier': None,
  'role': 'artist'}]
camcollectionsdf['birthyear'] = camcollectionsdf.\
...   creators.apply(lambda x: x[0]['birth_year'])
camcollectionsdf.birthyear.value_counts().\
...   sort_index().head()
1821    18
1886     2
1888     1
```

```
1892    13
1899    17
Name: birthyear, dtype: int64
```

This gives us a pandas DataFrame with one row for each `citation` for each collection item, with the collection information (`title` , `creation_date` , and so on) duplicated.

How it works...

We work with a much more *interesting* JSON file in this recipe than in the previous one. Each object in the JSON file is an item in the collection of the Cleveland Museum of Art. Nested within each collection item are one or more citations. The only way to capture this information in a tabular DataFrame is to flatten it. There are also one or more dictionaries for creators of the collection item (the artist or artists). That dictionary (or dictionaries) contains the `birth_year` value that we want.We want one row for every citation for all collection items. To understand this, imagine that we are working with relational data and have a collections table and a citations table, and that we are doing a one-to-many merge from collections to citations. We do something similar with `json_normalize` by using *citations* as the second parameter. That tells `json_normalize` to create one row for each citation and use the key values in each citation dictionary – for `citation` , `page_number` , and `url` – as data values.The third parameter in the call to `json_normalize` has the list of column names for the data that will be repeated with each citation. Notice that `access_number` , `title` , `creation_date` , `collection` , `creators` , and `type` are repeated in observations one and two. `Citation` and `page_number` change. (`url` is the same value for the first and second citations. Otherwise, it would also change.)This still leaves us with the problem of the creators dictionaries (there can be more than one creator). When we ran `json_normalize` it grabbed the value for each key we indicated (in the third parameter) and stored it in the data for that column and row, whether that value was simple text or was a list of dictionaries, as is the case for creators. We take a look at the first (and in this case, only) `creators` item for the first collections row in *step 10*, naming it `creator` . (Note that the creators list is duplicated across all `citations` for a collection item, just as the values for `title` , `creation_date` , and so on are.)We want the birth year for the first creator for each collection item, which can be found at `creator[0][‘birth_year’]`. To create a `birthyear` series using this, we use `apply` and a `lambda` function:

```
camcollectionsdf['birthyear'] = camcollectionsdf.\
...   creators.apply(lambda x: x[0]['birth_year'])
```

We take a closer look at lambda functions in *Chapter 6, Cleaning and Exploring Data with Series Operations*. Here, it is helpful to think of the `x` as representing the `creators` series, so `x[0]` gives us the list item we want, `creators[0]`. We grab the value from the `birth_year` key.

There’s more...

You may have noticed that we left out some of the JSON returned by the API in our call to `json_normalize`. The first parameter that we passed to `json_normalize` was `camcollections[‘data’]`. Effectively, we ignore the info object at the beginning of the JSON data. The information we want does not start until the data object. This is not very different conceptually from the `skiprows` parameter in the second recipe of the previous chapter. There is sometimes metadata like this at the beginning of JSON files.

See also

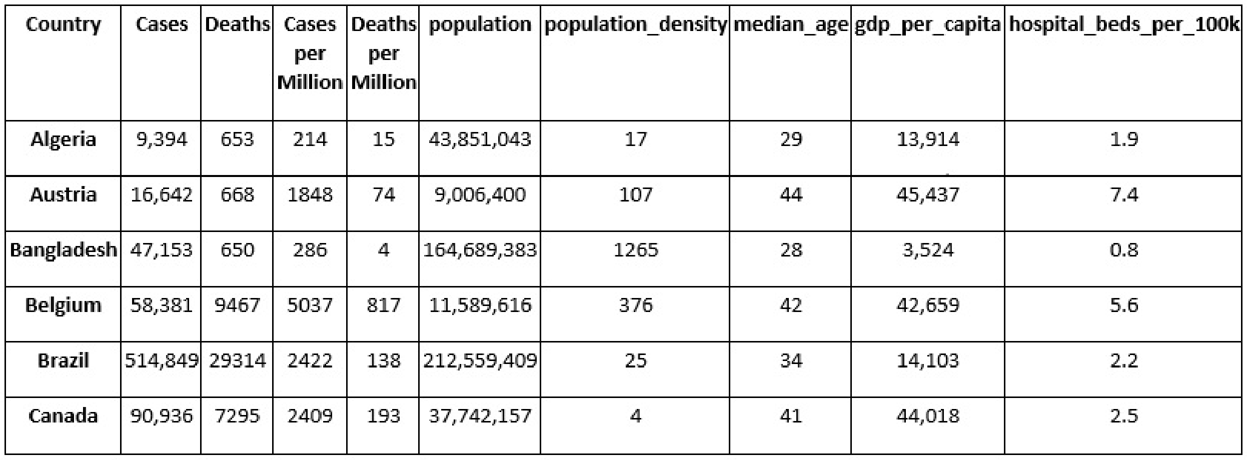
The preceding recipe demonstrates some useful techniques for doing data integrity checks without pandas, including list operations and comprehensions. Those are all relevant for the data in this recipe as well.

Importing data from web pages

We use **Beautiful Soup** in this recipe to scrape data from a web page and load that data into pandas. **Web scraping** is very useful when there is data at a website that is updated regularly, but there is no API. We can rerun our code to generate new data whenever the page is updated.Unfortunately, the web scrapers we build can be broken when the structure of the targeted page changes. That is less likely to happen with APIs because they are designed for data exchange, and carefully curated with that end in mind. The priority for most web designers is the quality of the display of information, not the reliability and ease of data exchange. This causes data cleaning challenges unique to web scraping, including HTML elements that house the data being in surprising and changing locations, formatting tags that obfuscate the underlying data, and explanatory text that aid data interpretation being difficult to retrieve. In addition to these challenges, scraping presents data cleaning issues that are familiar, such as changing data types in columns, less than ideal headings, and missing values. We deal with data issues that occur most frequently in this recipe.

Getting ready…

You will need Beautiful Soup installed to run the code in this recipe. You can install it with pip by entering `pip install beautifulsoup4` in a terminal window or Windows PowerShell.We will scrape data from a web page, find the following table in that page, and load it into a pandas DataFrame:



Country	Cases	Deaths	Cases per Million	Deaths per Million	population	population_density	median_age	gdp_per_capita	hospital_beds_per_100k
Algeria	9,394	653	214	15	43,851,043	17	29	13,914	1.9
Austria	16,642	668	1848	74	9,006,400	107	44	45,437	7.4
Bangladesh	47,153	650	286	4	164,689,383	1265	28	3,524	0.8
Belgium	58,381	9467	5037	817	11,589,616	376	42	42,659	5.6
Brazil	514,849	29314	2422	138	212,559,409	25	34	14,103	2.2
Canada	90,936	7295	2409	193	37,742,157	4	41	44,018	2.5

Figure 2.1 – COVID-19 data from six countries

Note

I created this web page, http://www.alrb.org/datacleaning/covidcaseoutliers.html, based on COVID-19 data for public use from *Our World in Data*, available at https://ourworldindata.org/coronavirus-source-data.

How to do it…

We scrape the COVID data from the website and do some routine data checks:

1. Import the `pprint`, `requests`, and `Beautiful Soup` libraries:

```
import pandas as pd
import numpy as np
import json
import pprint
import requests
from bs4 import BeautifulSoup
```

1. Parse the web page and get the header row of the table.

Use Beautiful Soup's `find` method to get the table we want and then use `find_all` to retrieve the elements nested within the `th` elements for that table. Create a list of column labels based on the text of the `th` rows:

```
webpage = requests.get("http://www.alrb.org/datacleaning/covidca
seoutliers.html")
bs = BeautifulSoup(webpage.text, 'html.parser')
theadrows = bs.find('table', {'id':'tblDeaths'}).thead.find_all(
'th')
type(theadrows)

<class 'bs4.element.ResultSet'>

labelcols = [j.get_text() for j in theadrows]
labelcols[0] = "rowheadings"
labelcols

['rowheadings', 'Cases', 'Deaths', 'Cases per Million', 'Deaths
per Million', 'population', 'population_density', 'median_age',
'gdp_per_capita', 'hospital_beds_per_100k']
```

1. Get the data from the table cells.

Find all of the table rows for the table we want. For each table row, find the `th` element and retrieve the text. We will use that text for our row labels. Also, for each row, find all the `td` elements (the table cells with the data) and save text from all of them in a list. This gives us `datarows`, which has all the numeric data in the table. (You can confirm that it matches the table from the web page.) We then insert the `labelrows` list (which has the row headings) at the beginning of each list in `datarows` :

```
rows = bs.find('table', {'id':'tblDeaths'}).tbody.find_all('tr')
datarows = []
labelrows = []
for row in rows:
...   rowlabels = row.find('th').get_text()
...   cells = row.find_all('td', {'class':'data'})
...   if (len(rowlabels)>3):
...     labelrows.append(rowlabels)
...   if (len(cells)>0):
...     cellvalues = [j.get_text() for j in cells]
...     datarows.append(cellvalues)
...
pprint.pprint(datarows[0:2])

[['9,394', '653', '214', '15', '43,851,043', '17', '29', '13,914
', '1.9'],
 ['16,642', '668', '1848', '74', '9,006,400', '107', '44', '45,4
37', '7.4']]

pprint.pprint(labelrows[0:2])

['Algeria', 'Austria']

for i in range(len(datarows)):
...   datarows[i].insert(0, labelrows[i])
...
pprint.pprint(datarows[0:1])

[['Algeria','9,394','653','214','15','43,851,043','17','29','13,
914','1.9']]
```

1. Load the data into pandas.

Pass the `datarows` list to the `DataFrame` method of pandas. Notice that all data is read into pandas with the object data type, and that some data has values that cannot be converted into numeric values in their current form (due to the commas):

```
totaldeaths = pd.DataFrame(datarows, columns=labelcols)
totaldeaths.iloc[:,1:5].head()

     Cases     Deaths     Cases per Million  \
0    9,394    653    214
1   16,642    668    1848
2   47,153    650    286
3   58,381    9467    5037
4  514,849    29314    2422
  Deaths per Million
0                 15
1                 74
2                  4
3                817
4                138
```

```
totaldeaths.dtypes
```

```
rowheadings    object
Cases    object
Deaths    object
Cases per Million    object
Deaths per Million    object
population    object
population_density    object
median_age    object
gdp_per_capita    object
hospital_beds_per_100k    object
dtype: object
```

1. Fix the column names and convert the data to numeric values.

Remove spaces from column names. Remove all non-numeric data from the first columns with data, including the commas (`str.replace("[^0-9] ","")`). Convert to numeric values, except for the `rowheadings` column:

```
totaldeaths.columns = totaldeaths.columns.str.replace(" ", "_").
str.lower()
for col in totaldeaths.columns[1:-1]:
...   totaldeaths[col] = totaldeaths[col].\
...     str.replace("[^0-9]","").astype('int64')
...
totaldeaths['hospital_beds_per_100k'] = totaldeaths['hospital_be
ds_per_100k'].astype('float')
totaldeaths.head()
  rowheadings   cases  ...  gdp_per_capita  \

0    Algeria    9394  ...    13914
1    Austria    16642  ...    45437
```

```
2      Bangladesh    47153  ...    3524
3    Belgium    58381  ...    42659
4    Brazil    514849  ...    14103
```

```
totaldeaths.dtypes
```

```
rowheadings    object
cases    int64
deaths    int64
cases_per_million    int64
deaths_per_million    int64
population    int64
population_density    int64
median_age    int64
gdp_per_capita    int64
hospital_beds_per_100k    float64
dtype: object
```

We have now created a pandas DataFrame from an `html` table.

How it works...

Beautiful Soup is a very useful tool for finding specific HTML elements in a web page and retrieving text from them. You can get one HTML element with `find` and get one or more with `find_all`. The first argument for both `find` and `find_all` is the HTML element to get. The second argument takes a Python dictionary of attributes. You can retrieve text from all of the HTML elements you find with `get_text`.Some amount of looping is usually necessary to process the elements and text, as with *step 2* and *step 3*. These two statements in *step 2* are fairly typical:

```
theadrows = bs.find('table', {'id':'tblDeaths'}).thead.find_all(
'th')
labelcols = [j.get_text() for j in theadrows]
```

The first statement finds all the `th` elements we want and creates a Beautiful Soup result set called `theadrows` from the elements it found. The second statement iterates over the `theadrows` Beautiful Soup result set using the `get_text` method to get the text from each element, and stores it in the `labelcols` list.*Step 3* is a little more involved, but makes use of the same Beautiful Soup methods. We find all of the table rows (`tr`) in the target table (`rows = bs.find('table', {'id':'tblDeaths'}).tbody.find_all('tr')`

). We then iterate over each of those rows, finding the `th` element and getting the text in that element (`rowlabels = row.find('th').get_text()`). We also find all of the table cells (`td`) for each row (`cells = row.find_all('td', {'class':'data'}`) and get the text from all table cells (`cellvalues = [j.get_text() for j in cells]`). Note that this code is dependent on the class of the `td` elements being `data`. Finally, we insert the row labels we get from the `th` elements at the beginning of each list in `datarows`:

```
for i in range(len(datarows)):
    datarows[i].insert(0, labelrows[i])
```

In *step 4*, we use the `DataFrame` method to load the list we created in *steps 2* and *3* into pandas. We then do some cleaning similar to what we have done in previous recipes in this chapter. We use `string replace` to remove spaces from column names and to remove all non-numeric data, including commas, from what are otherwise valid numeric values. We convert all columns, except for the `rowheadings` column, to numeric.

There's more...

Our scraping code is dependent on several aspects of the web page's structure not changing: the ID of the main table, the presence of `th` tags with column and row labels, and the `td` elements continuing to have their class equal to data. The good news is that if the structure of the web page does change, this will likely only affect the `find` and `find_all` calls. The rest of the code would not need to change.

Working with Spark data

When working with large datasets we sometimes need to rely on distributed resources to clean and manipulate our data. With Apache Spark, analysts can take advantage of the combined processing power of many machines. We will use PySpark, a Python API for working with Spark, in this recipe. We will also go over how to use PySpark tools to take a first look at our data, select parts of a our data, and generate some simple summary statistics.

Getting ready...

To run the code in this section you need to get Spark running on your computer. If you have installed Anaconda you can follow these steps to work with Spark:

- Install Java with `conda install openjdk`
- Install PySpark with `conda install pyspark` or `conda install -c conda forge pyspark`
- Install findspark with `conda install -c conda-forge findspark`

We will work with the land temperatures data from *chapter 1* and the candidate news data from this chapter. All data and the code we will be running in this recipe are available in the GitHub repository for this book.

Note

> This dataset, taken from the Global Historical Climatology Network integrated database, is made available for public use by the United States National Oceanic and Atmospheric Administration at https://www.ncdc.noaa.gov/data-access/land-based-station-data/land-based-datasets/global-historical-climatology-network-monthly-version-4.

1. Let's start a Spark session and load the land temperatures data. We can use the read method of the session object to create a Spark DataFrame. We indicate that the first row of the CSV file we are importing has a header.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
   .getOrCreate()
landtemps = spark.read.option("header",True) \
     .csv("data/landtemps.csv")
type(landtemps)
```

```
pyspark.sql.dataframe.DataFrame
```

Notice that the read method returns a Spark DataFrame, not a pandas DataFrame. We will need to use different methods to view our data than those we have used so far.We load the full dataset, not just a 100,000 row sample as we did in the first chapter. If your system is low on resources, you can import the *landtempssample.csv* file instead.

1. We should take a look at the number of rows and the column names and data types that were imported. The `temp` column was read as string. It should be a float. We will fix that in a later step.

```
landtemps.count()

16904868

landtemps.printSchema()

root
 |-- locationid: string (nullable = true)
 |-- year: string (nullable = true)
 |-- month: string (nullable = true)
 |-- temp: string (nullable = true)
 |-- latitude: string (nullable = true)
 |-- longitude: string (nullable = true)
 |-- stnelev: string (nullable = true)
 |-- station: string (nullable = true)
 |-- countryid: string (nullable = true)
 |-- country: string (nullable = true)
```

1. Let’s look at the data for a few rows. We can choose a subset of the columns by using the `select` method.

```
landtemps.select("station",'country','month','year','temp') \
    .show(5, False)

+-------+-------------------+-----+----+-----+
|station|country                    |month|year|temp |
+-------+-------------------+-----+----+-----+
|SAVE   |Antigua and Barbuda|1    |1961|-0.85|
|SAVE   |Antigua and Barbuda|1    |1962|1.17 |
|SAVE   |Antigua and Barbuda|1    |1963|-7.09|
|SAVE   |Antigua and Barbuda|1    |1964|0.66 |
|SAVE   |Antigua and Barbuda|1    |1965|0.48 |
+-------+-------------------+-----+----+-----+
only showing top 5 rows
```

1. We should fix the the data type of the `temp` column. We can use the `withColumn` function to do a range of column operations in Spark. Here

we use it to `cast` the `temp` column to `float` .

```
landtemps = landtemps \
  .withColumn("temp",landtemps.temp.cast('float'))
landtemps.select("temp").dtypes
```

```
[('temp', 'float')]
```

1. Now we can run summary statistics on the `temp` variable. We can use the `describe` method for that.

```
landtemps.describe('temp').show()
```

```
+-------+------------------+
|summary|              temp|
+-------+------------------+
|  count|          14461547|
|   mean|10.880725773138536|
| stddev|11.509636369381685|
|    min|             -75.0|
|    max|             42.29|
+-------+------------------+
```

1. The Spark session's read method can import a variety of different data files, not just CSV files. Let's try that with the *allcandidatenews* JSON file that we worked with earlier in this chapter.

```
allcandidatenews = spark.read \
     .json("data/allcandidatenewssample.json")
allcandidatenews \
  .select("source","title","story_position") \
  .show(5)
```

```
+--------------------+--------------------+--------------+
|              source    |               title    |story_positio
n|
+--------------------+--------------------+--------------+
|            NBC News    |Bloomberg cuts ti...    |
6|
|Town & Country Ma...    |Democratic Candid...|            3|
|                null    |                null    |          nul
l|
|             TheHill    |Sanders responds ...    |
7|
|            CNBC.com    |From Andrew Yang'...|            2|
+--------------------+--------------------+--------------+
only showing top 5 rows
```

1. We can use the `count` and `printSchema` methods again to look at our data.

```
allcandidatenews.count()
```

```
60000
```

```
allcandidatenews.printSchema()
```

```
root
 |-- category: string (nullable = true)
 |-- date: string (nullable = true)
 |-- domain: string (nullable = true)
 |-- panel_position: string (nullable = true)
 |-- query: string (nullable = true)
 |-- reason: string (nullable = true)
 |-- source: string (nullable = true)
 |-- story_position: long (nullable = true)
 |-- time: string (nullable = true)
 |-- title: string (nullable = true)
 |-- url: string (nullable = true)
```

1. We can also generate some summary statistics on the `story_position` variable.

```
allcandidatenews \
   .describe('story_position') \
   .show()
```

```
+-------+-----------------+
|summary|   story_position|
+-------+-----------------+
|  count|            57618|
|   mean|5.249626852719636|
| stddev|2.889001922195635|
|    min|                1|
|    max|               10|
+-------+-----------------+
```

These steps demonstrate how to import data files into a Spark DataFrame, and then view the structure of the data and generate summary statistics.

How it works...

The PySpark API significantly reduces the amount of work Python programmers have to do to use Apache Spark to handle large data files. We get methods to work with that are not very different from the methods we use with pandas DataFrames. We can see the number of rows and columns, examine and change data types, and get summary statistics.

There's more...

At some point in our analysis we might want to convert the Spark DataFrame into a pandas DataFrame. This is a fairly expensive process, and we will lose the benefits of working with Spark, so we typically will not do that unless we are at the point of our analysis when we require the pandas library, or a library that depends on pandas. But when we need to move to pandas, it is very easy to do -- though if you are working with a lot of data and your machine's processer and ram are not exactly top of the line, you might want to start the conversion and then go have some tea or coffee.The following code converts the `allcandidatenews` Spark DataFrame that we created to a pandas DataFrame and displays the resulting DataFrame structure.

```
allcandidatenewsdf = allcandidatenews.toPandas()
allcandidatenewsdf.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 60000 entries, 0 to 59999
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   category        416 non-null    object
 1   date            60000 non-null  object
 2   domain          57618 non-null  object
 3   panel_position  57618 non-null  object
 4   query           57618 non-null  object
 5   reason          2382 non-null   object
 6   source          57618 non-null  object
 7   story_position  57618 non-null  float64
 8   time            57618 non-null  object
 9   title           57618 non-null  object
 10  url             57618 non-null  object
dtypes: float64(1), object(10)
memory usage: 5.0+ MB
```

We have been largely working with non-traditional data stores in this chapter:

JSON files, data from HTML pages, and Spark files. Often we reach a point in our data cleaning work that it makes sense to preserve the results of that cleaning by persisting data. At the end of *chapter 1* we examined how to persist tabular data. That works fine in cases where our data can be captured well with columns and rows. When it cannot, say when we are working with a JSON file that has complicated subdocuments, we might want to preserve that structure when persisting data. In th next recipe we go over persisting JSON data.

Persisting JSON data

There are several reasons why we might want to serialize a JSON file:

- We may have retrieved the data with an API, but need to keep a snapshot of the data.
- The data in the JSON file is relatively static and informs our data cleaning and analysis over multiple phases of a project.
- We might decide that the flexibility of a schema-less format such as JSON helps us solve many data cleaning and analysis problems.

It is worth highlighting this last reason to use JSON – that it can solve many data problems. Although tabular data structures clearly have many benefits, particularly for operational data, they are often not the best way to store data for analysis purposes. In preparing data for analysis, a substantial amount of time is spent either merging data from different tables or dealing with data redundancy when working with flat files. Not only are these processes time consuming, but every merge or reshaping leaves the door open to a data error of broad scope. This can also mean that we end up paying too much attention to the mechanics of manipulating data and too little to the conceptual issues at the core of our work.We return to the Cleveland Museum of Art collections data in this recipe. There are at least three possible units of analysis for this data file – the collection item level, the creator level, and the citation level. JSON allows us to nest citations and creators within collections. (You can examine the structure of the JSON file in the *Getting ready…* section of this recipe.) This data cannot be persisted in a tabular structure without flattening the file, which we did in an earlier recipe in this chapter. In this recipe, we will use two different methods to persist JSON data, each with its own advantages and disadvantages.

Getting ready...

We will be working with data on the Cleveland Museum of Art's collection of works by African-American artists. The following is the structure of the JSON data returned by the API. It has been abbreviated to save space:

```
{"info": { "total": 778, "parameters": {"african_american_artist
s": "" }},
"data": [
{
"id": 165157,
"accession_number": "2007.158",
"title": "Fulton and Nostrand",
"creation_date": "1958",
“citations": [
  {
   "citation": "Annual Exhibition: Sculpture, Paintings...",
   "page_number": "Unpaginated, [8],[12]",
   "url": null
   },
  {
   "citation": "\"Moscow to See Modern U.S. Art,\"<em> New York.
..",
   "page_number": "P. 60",
   "url": null
  }]
"creators": [
      {
     "description": "Jacob Lawrence (American, 1917-2000)",
     "extent": null,
     "qualifier": null,
     "role": "artist",
     "birth_year": "1917",
     "death_year": "2000"
     }
  ]
 }
```

How to do it...

We will serialize the JSON data using two different methods:

1. Load the `pandas`, `json`, `pprint`, `requests`, and `msgpack` libraries:

```
import pandas as pd
import json
import pprint
import requests
import msgpack
```

1. Load the JSON data from an API. I have abbreviated the JSON output:

```
response = requests.get("https://openaccess-api.clevelandart.org
/api/artworks/?african_american_artists")
camcollections = json.loads(response.text)
print(len(camcollections['data']))

778

pprint.pprint(camcollections['data'][0])
{'accession_number': '2007.158',
 'catalogue_raisonne': None,
 'citations': [
   {'citation': 'Annual Exhibition: Sculpture...',
    'page_number': 'Unpaginated, [8],[12]',
    'url': None},
  {'citation': '"Moscow to See Modern U.S....',
    'page_number': 'P. 60',
    'url': None}]
 'collection': 'American - Painting',
 'creation_date': '1958',
 'creators': [
  {'biography': 'Jacob Lawrence (born 1917)...',
   'birth_year': '1917',
   'description': 'Jacob Lawrence (American...)',
   'role': 'artist'}],
 'type': 'Painting'}
```

1. Save and reload the JSON file using Python's `json` library.

Persist the JSON data in human-readable form. Reload it from the saved file and confirm that it worked by retrieving the `creators` data from the first collections item:

```
with open("data/camcollections.json","w") as f:
...   json.dump(camcollections, f)
...
```

```
with open("data/camcollections.json","r") as f:
...   camcollections = json.load(f)
...
pprint.pprint(camcollections['data'][0]['creators'])

[{'biography': 'Jacob Lawrence (born 1917) has been a prominent
artist since...'
  'birth_year': '1917',
  'description': 'Jacob Lawrence (American, 1917-2000)',
  'role': 'artist'}]
```

1. Save and reload the JSON file using `msgpack`:

```
with open("data/camcollections.msgpack", "wb") as outfile:
...     packed = msgpack.packb(camcollections)
...     outfile.write(packed)
...

1586507

with open("data/camcollections.msgpack", "rb") as data_file:
...     msgbytes = data_file.read()
...
camcollections = msgpack.unpackb(msgbytes)
pprint.pprint(camcollections['data'][0]['creators'])

[{'biography': 'Jacob Lawrence (born 1917) has been a prominent.
..',
  'birth_year': '1917',
  'death_year': '2000',
  'description': 'Jacob Lawrence (American, 1917-2000)',
  'role': 'artist'}]
```

How it works...

We use the Cleveland Museum of Art's collections API to retrieve collections items. The `african_american_artists` flag in the query string indicates that we just want collections for those creators. `json.loads` returns a dictionary called `info` and a list of dictionaries called `data`. We check the length of the `data` list. This tells us that there are 778 items in collections. We then display the first item of collections to get a better look at the structure of the data. (I have abbreviated the JSON output.)We save and then reload the data using Python's JSON library in *step 3*. The advantage of persisting the data in this way is that it keeps the data in human-readable form. Unfortunately, it has two disadvantages: saving takes longer than alternative serialization methods, and it uses more storage space.In *step 4*, we use `msgpack` to persist our data. This is faster than Python's `json` library, and the saved file uses less space. Of course, the disadvantage is that the resulting JSON is binary rather than text-based.

There's more...

I use both methods for persisting JSON data in my work. When I am working with small amounts of data, and that data is relatively static, I prefer human-readable JSON. A great use case for this is the recipes in the previous chapter where we needed to create value labels.I use `msgpack` when I am working with large amounts of data, where that data changes regularly. `msgpack` files are also great when you want to take regular snapshots of key tables in enterprise databases.The Cleveland Museum of Art's collections data is similar in at least one important way to the data we work with every day. The unit of analysis frequently changes. Here we are looking at collections, citations, and creators. In our work, we might have to simultaneously look at students and courses, or households and deposits. An enterprise database system for the museum data would likely have separate collections, citations, and creators tables that we would eventually need to merge. The resulting merged file would have data redundancy issues that we would need to account for whenever we changed the unit of analysis.When we alter our data cleaning process to work directly from JSON or parts of it, we end up eliminating a major source of errors. We do more data cleaning with JSON in the *Classes that handle non-tabular data structures* recipe in *Chapter 10, User Defined Functions and Classes to Automate Data Cleaning*.

3 Taking the Measure of Your Data

Within a week of receiving a new dataset, at least one person is likely to ask us a familiar question – “so, how does it look?” This is not always asked relaxedly, and others are not usually excited to hear about all of the red flags we have already found. There might be a sense of urgency to declare the data ready for analysis. Of course, if we sign off on it too soon, this can create much larger problems; the presentation of invalid results, the misinterpretation of variable relationships, and having to redo major chunks of our analysis. The key is sorting out what we need to know about the data before we explore anything else in the data. The recipes in this chapter offer techniques for determining if the data is in good enough shape to begin the analysis, so that even if we cannot say, “it looks fine,” we can at least say, “I’m pretty sure I have identified the main issues, and here they are.”Often our domain knowledge is quite limited, or at least not nearly as good as those who created the data. We have to quickly get a sense of what we are looking at even when we have little substantive understanding of the individuals or events reflected in the data. Many times (for some of us, most of the time) there is not anything like a data dictionary or codebook accompanying the receipt of the data.Quick. Ask yourself what the first few things you try to find out in this situation are; that is, when you first get data about which you know little. It is probably something like this:

- How are the rows of the dataset uniquely identified? (What is the unit of analysis?)
- How many rows and columns are in the dataset?
- What are the key categorical variables and the frequencies of each value?
- How are important continuous variables distributed?
- How might variables be related to each other – for example, how might the distribution of continuous variables vary according to categories in the data?
- What variable values are out of expected ranges, and how are missing values distributed?

We go over essential tools and strategies for answering the first four questions in this chapter. We look into the last two questions in the following chapter.I should point out that this first take on our data is important even when the structure of the data is familiar; when, for example, we receive data for a new month or year with the same column names and data types as in previous periods. It is hard to guard against the sense that we can just rerun our old programs; to be as vigilant as we were the first few times we prepared the data for analysis. Most of us have probably been in situations where we receive new data with a familiar structure, but the answers to the preceding questions are meaningfully different: new valid values for key categorical variables; rare values that have always been permissible but that have not been seen for several periods; and unexpected changes in the status of `clients/students/customers`. It is important to build routines for understanding our data that we follow regardless of our familiarity with it.Specifically, we will cover the following topics in this chapter:

- Getting a first look at your data
- Selecting and organizing columns
- Selecting rows
- Generating frequencies for categorical variables
- Generating statistics for continuous variables
- Using generative AI to generate descriptive statistics

Getting a first look at your data

We will work with two datasets in this chapter: The National Longitudinal Survey of Youth for 1997, a survey conducted by the United States government that surveyed the same group of individuals from 1997 through 2017; and the counts of COVID cases and deaths by country from *Our World in Data*.

Getting ready...

We will mainly be using the pandas library for this recipe. We will use pandas tools to take a closer look at the **National Longitudinal Survey** (**NLS**) and coronavirus case data.

Note

> The NLS of Youth was conducted by the United States Bureau of Labor Statistics. This survey started with a cohort of individuals in 1997 who were born between 1980 and 1985, with annual follow-ups each year through 2017. For this recipe, I pulled 89 variables on grades, employment, income, and attitudes toward government from the hundreds of data items on the survey. Separate files for SPSS, Stata, and SAS can be downloaded from the repository. NLS data can be downloaded from https://www.nlsinfo.org/investigator/pages/search.
>
> *Our World in Data* provides COVID-19 public use data at https://ourworldindata.org/coronavirus-source-data.

How to do it…

We will get an initial look at the NLS and COVID data, including the number of rows and columns, and the data types:

1. Import libraries and load the DataFrames:

```
import pandas as pd
import numpy as np
nls97 = pd.read_csv("data/nls97.csv")
covidtotals = pd.read_csv("data/covidtotals.csv",
...   parse_dates=['lastdate'])
```

1. Set and show the index and the size of the `nls97` data.

Also check to see whether the index values are unique:

```
nls97.set_index("personid", inplace=True)
nls97.index

Int64Index([100061, 100139, 100284, 100292, 100583, 100833,
            999543, 999698, 999963],
           dtype='int64', name='personid', length=8984)

nls97.shape

(8984, 88)

nls97.index.nunique()

8984
```

1. Show the data types and `non-null` value counts:

```
nls97.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 8984 entries, 100061 to 999963
Data columns (total 88 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   gender    8984 non-null    object
 1   birthmonth    8984 non-null    int64
 2   birthyear    8984 non-null    int64
 3   highestgradecompleted    6663 non-null    float64
 4   maritalstatus    6672 non-null    object
 5   childathome    4791 non-null    float64
 6   childnotathome    4791 non-null    float64
 7   wageincome    5091 non-null    float64
 8   weeklyhrscomputer    6710 non-null    object
 9   weeklyhrstv    6711 non-null    object
 10  nightlyhrssleep    6706 non-null    float64
 11  satverbal    1406 non-null    float64
 12  satmath    1407 non-null    float64
...

 83  colenroct15    7469 non-null    object
 84  colenrfeb16    7036 non-null    object
 85  colenroct16    6733 non-null    object
 86  colenrfeb17    6733 non-null    object
 87  colenroct17    6734 non-null    object
dtypes: float64(29), int64(2), object(57)
memory usage: 6.1+ MB
```

1. Show the first row of the `nls97` data.

Use transpose to show a little more of the output:

```
nls97.head(2).T
```

```
personid    100061    100139
gender    Female    Male
birthmonth    5    9
birthyear    1980    1983
highestgradecompleted    13    12
maritalstatus    Married    Married
...                                              ...
colenroct15    1. Not enrolled    1. Not enrolled
colenrfeb16    1. Not enrolled    1. Not enrolled
colenroct16    1. Not enrolled    1. Not enrolled
colenrfeb17    1. Not enrolled    1. Not enrolled
colenroct17    1. Not enrolled    1. Not enrolled
```

1. Set and show the index and size for the COVID data.

Also check to see whether index values are unqiue:

```
covidtotals.set_index("iso_code", inplace=True)
covidtotals.index
```

```
Index(['AFG', 'ALB', 'DZA', 'AND', 'AGO', 'AIA', 'ATG', 'ARG', '
ARM',
       'ABW',
       ...
       'VIR', 'URY', 'UZB', 'VAT', 'VEN', 'VNM', 'ESH', 'YEM', '
ZMB',
       'ZWE'],
      dtype='object', name='iso_code', length=209)
```

```
covidtotals.shape
```

```
(209, 12)
```

```
covidtotals.index.nunique()
```

```
209
```

1. Show the data types and `non-null` value counts:

```
covidtotals.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 209 entries, AFG to ZWE
Data columns (total 12 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   lastdate    209    non-null    datetime64[ns]
 1   location    209    non-null    object
 2   total_cases    209    non-null    float64
 3   total_deaths    209    non-null    float64
 4   total_cases_pm    209    non-null    float64
 5   total_deaths_pm    209    non-null    float64
 6   population    209    non-null    float64
 7   pop_density    198    non-null    float64
 8   median_age    185    non-null    float64
 9   gdp_per_capita    182    non-null    float64
 10  hosp_beds    164    non-null    float64
 11  region    209    non-null    object
dtypes: datetime64[ns](1), float64(9), object(2)
memory usage: 29.3+ KB
```

1. Show a sample of a few rows of the COVID case data:

```
covidtotals.sample(2, random_state=1).T
```

```
iso_code    COG    THA
lastdate    2020-07-12 00:00:00    2020-07-12 00:00:00
location    Congo    Thailand
total_cases    2,028    3,217
total_deaths    47    58
total_cases_pm    368    46
total_deaths_pm    9    1
population    5,518,092    69,799,978
pop_density    15    135
median_age    19    40
gdp_per_capita    4,881    16,278
hosp_beds    NaN    2
region    Central Africa    East Asia
```

This has given us a good foundation for understanding our DataFrames, including their size and column data types.

How it works...

We set and display the index of the `nls97` DataFrame, which is called `personid`, in *step 2*. It is a more meaningful index than the default pandas `RangeIndex`, which is essentially the row numbers with zero base. Often there is a unique identifier when working with individuals as the unit of analysis. This is a good candidate for an index. It makes selecting a row by that identifier easier. Rather than using the statement `nls97.loc[personid==1000061]` to get the row for that person, we can use `nls97.loc[1000061]`. We try this out in the next recipe.Pandas makes it easy to view the number of rows and columns, the data type and number of non-missing values for each column, and the values for the columns for a few rows of your data. This can be accomplished by using the `shape` attribute and calling the `info` and `head`, or `sample`, methods. Using the `head(2)` method shows the first two rows, but sometimes it is helpful to grab a row from anywhere in the DataFrame, in which case we would use `sample`. (We set the seed when we call `sample` (`random_state=1`) to get the same results whenever we run the code.) We can chain our call to `head` or `sample` with a `T` to transpose it. This reverses the display of rows and columns. That is helpful when there are more columns than can be shown horizontally and you want to be able to see all of them. By transposing the rows and columns we are able to see all of the columns.The `shape` attribute of the `nls97` DataFrame tells us that there are 8,984 rows and 88 non-index columns. Since `personid` is the index, it is not included in the column count. The `info` method shows us that many of the columns have object data types and that some have a large number of missing values. `satverbal` and `satmath` have only about 1,400 valid values.The `shape` attribute of the `covidtotals` DataFrame tells us that there are 210 rows and 11 columns, which does not include the country `iso_code` column used for the index (`iso_code` is a unique three-digit identifier for each country). The key variables for most analyses we would do are `total_cases`, `total_deaths`, `total_cases_pm, and total_deaths_pm`. `total_cases` and `total_deaths` are present for each country, but `total_cases_pm` and `total_deaths_pm` are missing for one country.

There's more...

I find that thinking through the index when working with a data file can remind me of the unit of analysis. That is not actually obvious with the NLS data, as it is actually panel data disguised as person-level data. Panel, or longitudinal, datasets have data for the same individuals over some regular duration. In this case, data was collected for each person over a 21-year span, from 1997 till 2017. The administrators of the survey have flattened it for analysis purposes by creating columns for certain responses over the years, such as college enrollment (`colenroct15` through `colenroct17`). This is a fairly standard practice, but it is likely that we will need to do some reshaping for some analyses.One thing I pay careful attention to when receiving any panel data is drop-off in responses to key variables over time. Notice the drop off in valid values from `colenroct15` to `colenroct17` . By October of 2017, only 75% of respondents provided a valid response (6,734/8,984). That is definitely worth keeping in mind during subsequent analysis, since the 6,734 remaining respondents may be different in important ways from the overall sample of 8,984.

See also

A recipe in *Chapter 1, Anticipating Data Cleaning Issues when Importing Tabular Data into Pandas*, shows how to persist pandas DataFrames as feather or pickle files. In later recipes in this chapter, we will look at descriptives and frequencies for these two DataFrames.We reshape the NLS data in *Chapter 9, Tidying and Reshaping Data*, recovering some of its actual structure as panel data. This is necessary for statistical methods such as survival analysis, and is closer to tidy data ideals.

Selecting and organizing columns

We explore several ways to select one or more columns from your DataFrame in this recipe. We can select columns by passing a list of column names to the `[]` bracket operator, or by using the pandas-specific data accessors `loc` and `iloc` .When cleaning data or doing exploratory or statistical analyses, it is helpful to focus on the variables that are relevant to the issue or analysis at hand. This makes it important to group columns according to their substantive or statistical relationships with each other, or to limit the columns we are investigating at any one time. How many times have we said to ourselves something like, *"Why does variable A have a value of x when variable B has a value of y?"* We can only do that when the amount of data we are viewing at a given moment does not exceed our perceptive abilities at that moment.

Getting Ready…

We will continue working with the **National Longitudinal Survey** (**NLS**) data in this recipe.

How to do it…

We will explore several ways to select columns:

1. Import the `pandas` library and load the NLS data into pandas:

Also convert all columns with object data type in the NLS data to category data type. Do this by selecting object data type columns with `select_dtypes` and using `apply` plus a `lambda` function to change the data type to category.

```
import pandas as pd
import numpy as np
nls97 = pd.read_csv("data/nls97.csv")
nls97.set_index("personid", inplace=True)
nls97.loc[:, nls97.dtypes == 'object'] = \
...   nls97.select_dtypes(['object']). \
...   apply(lambda x: x.astype('category'))
```

1. Select a column using the pandas `[]` bracket operator, and the `loc` and `iloc` accessors.

We pass a string matching a column name to the bracket operator to return a pandas series. If we pass a list of one element with that column name (`nls97[['gender']]`), a DataFrame is returned. We can also use the `loc` and `iloc` accessors to select columns:

```
analysisdemo = nls97['gender']
type(analysisdemo)
```

```
<class 'pandas.core.series.Series'>

analysisdemo = nls97[['gender']]
type(analysisdemo)

<class 'pandas.core.frame.DataFrame'>

analysisdemo = nls97.loc[:,['gender']]
type(analysisdemo)

<class 'pandas.core.frame.DataFrame'>

analysisdemo = nls97.iloc[:,[0]]
type(analysisdemo)

<class 'pandas.core.frame.DataFrame'>
```

1. Select multiple columns from a pandas DataFrame.

Use the bracket operator and `loc` to select a few columns:

```
analysisdemo = nls97[['gender','maritalstatus',
...  'highestgradecompleted']]
analysisdemo.shape

(8984, 3)

analysisdemo.head()

              gender  maritalstatus  highestgradecompleted
personid
100061    Female    Married    13
100139    Male    Married    12
100284    Male    Never-married    7
100292    Male    NaN    nan
100583    Male    Married    13

analysisdemo = nls97.loc[:,['gender','maritalstatus',
...  'highestgradecompleted']]
analysisdemo.shape

(8984, 3)

analysisdemo.head()

                   gender      maritalstatus highestgradecomplet
ed
personid
```

```
100061    Female    Married    13
100139    Male    Married    12
100284    Male    Never-married    7
100292    Male    NaN    nan
100583    Male    Married    13
```

1. Select multiple columns based on a list of columns.

If you are selecting more than a few columns, it is helpful to create the list of column names separately. Here, we create a `keyvars` list of key variables for analysis:

```
keyvars = ['gender','maritalstatus',
...  'highestgradecompleted','wageincome',
...  'gpaoverall','weeksworked17','colenroct17']
analysiskeys = nls97[keyvars]
analysiskeys.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8984 entries, 100061 to 999963
Data columns (total 7 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   gender    8984 non-null    category
 1   maritalstatus    6672 non-null    category
 2   highestgradecompleted    6663 non-null    float64
 3   wageincome    5091 non-null    float64
 4   gpaoverall    6004 non-null    float64
 5   weeksworked17    6670 non-null    float64
 6   colenroct17    6734 non-null    category
dtypes: category(3), float64(4)
memory usage: 377.7 KB
```

1. Select one or more columns by filtering on column name.

Select all of the `weeksworked##` columns using the `filter` operator:

```
analysiswork = nls97.filter(like="weeksworked")
analysiswork.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8984 entries, 100061 to 999963
Data columns (total 18 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
```

```
 0   weeksworked00  8603 non-null   float64
 1   weeksworked01  8564 non-null   float64
 2   weeksworked02  8556 non-null   float64
 3   weeksworked03  8490 non-null   float64
 4   weeksworked04  8458 non-null   float64
 5   weeksworked05  8403 non-null   float64
 6   weeksworked06  8340 non-null   float64
 7   weeksworked07  8272 non-null   float64
 8   weeksworked08  8186 non-null   float64
 9   weeksworked09  8146 non-null   float64
 10  weeksworked10  8054 non-null   float64
 11  weeksworked11  7968 non-null   float64
 12  weeksworked12  7747 non-null   float64
 13  weeksworked13  7680 non-null   float64
 14  weeksworked14  7612 non-null   float64
 15  weeksworked15  7389 non-null   float64
 16  weeksworked16  7068 non-null   float64
 17  weeksworked17  6670 non-null   float64
dtypes: float64(18)
memory usage: 1.3 MB
```

1. Select all columns with the category data type.

Use the `select_dtypes` method to select columns by data type:

```
analysiscats = nls97.select_dtypes(include=["category"])
analysiscats.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8984 entries, 100061 to 999963
Data columns (total 57 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   gender    8984 non-null    category
 1   maritalstatus    6672 non-null    category
 2   weeklyhrscomputer    6710 non-null    category
 3   weeklyhrstv    6711 non-null    category
 4   highestdegree    8953 non-null    category
...
 49  colenrfeb14    7624 non-null    category
 50  colenroct14    7469 non-null    category
 51  colenrfeb15    7469 non-null    category
 52  colenroct15    7469 non-null    category
 53  colenrfeb16    7036 non-null    category
 54  colenroct16    6733 non-null    category
 55  colenrfeb17    6733 non-null    category
 56  colenroct17    6734 non-null    category
```

```
dtypes: category(57)
memory usage: 580.0 KB
```

1. Select all columns with numeric data types:

```
analysisnums = nls97.select_dtypes(include=["number"])
analysisnums.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8984 entries, 100061 to 999963
Data columns (total 31 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   birthmonth    8984 non-null    int64
 1   birthyear    8984 non-nul    int64
 2   highestgradecompleted    6663 non-null    float64
...
 23  weeksworked10    8054 non-null    float64
 24  weeksworked11    7968 non-null    float64
 25  weeksworked12    7747 non-null    float64
 26  weeksworked13    7680 non-null    float64
 27  weeksworked14    7612 non-null    float64
 28  weeksworked15    7389 non-null    float64
 29  weeksworked16    7068 non-null    float64
 30  weeksworked17    6670 non-null    float64
dtypes: float64(29), int64(2)
memory usage: 2.2 MB
```

1. Organize columns using lists of column names.

Use lists to organize the columns in your DataFrame. You can easily change the order of columns or exclude some columns in this way. Here, we move the columns in the `demoadult` list to the front:

```
demo = ['gender','birthmonth','birthyear']
highschoolrecord = ['satverbal','satmath','gpaoverall',
...  'gpaenglish','gpamath','gpascience']
govresp = ['govprovidejobs','govpricecontrols',
...   'govhealthcare','govelderliving','govindhelp',
...   'govunemp','govincomediff','govcollegefinance',
...   'govdecenthousing','govprotectenvironment']
demoadult = ['highestgradecompleted','maritalstatus',
...   'childathome','childnotathome','wageincome',
...   'weeklyhrscomputer','weeklyhrstv','nightlyhrssleep',
...   'highestdegree']
weeksworked = ['weeksworked00','weeksworked01',
...   'weeksworked02','weeksworked03','weeksworked04',
       ...
      'weeksworked14','weeksworked15','weeksworked16',
...   'weeksworked17']
colenr = ['colenrfeb97','colenroct97','colenrfeb98',
...   'colenroct98','colenrfeb99','colenroct99',
       .
...   'colenrfeb15','colenroct15','colenrfeb16',...   'colenroct
16','colenrfeb17','colenroct17']
```

1. Create the new reorganized DataFrame:

```
nls97 = nls97[demoadult + demo + highschoolrecord + \
...   govresp + weeksworked + colenr]
nls97.dtypes
```

```
highestgradecompleted    float64
maritalstatus    category
childathome    float64
childnotathome    float64
wageincome    float64
                           ...
colenroct15    category
colenrfeb16    category
colenroct16    category
colenrfeb17    category
colenroct17    category
Length: 88, dtype: object
```

The preceding steps showed how to select columns and change the order of columns in a pandas DataFrame.

How it works...

Both the `[]` bracket operator and the `loc` data accessor are very handy for selecting and organizing columns. Each returns a DataFrame when passed a list of names of columns. The columns will be ordered according to the passed list of column names.In *step 1*, we use `nls97.select_dtypes(['object'])` to select columns with object data type and chain that with `apply` and a `lambda` function (`apply(lambda x: x.astype('category'))`) to change those columns to category. We use the `loc` accessor to only update columns with object data type (`nls97.loc[:, nls97.dtypes == 'object']`). We go into much more detail on `apply` and `lambda` functions in *Chapter 6*, *Cleaning and Exploring Data with Series Operations*.We also select columns by data type in *steps 6* and *7*. `select_dtypes` becomes quite useful when passing columns to methods such as `describe` or `value_counts` and you want to limit the analysis to continuous or categorical variables.In *step* 9, we concatenate six different lists when using the bracket operator. This moves the column names in `demoadult` to the front and organizes all of the columns by those six groups. There are now clear *high school record* and *weeks worked* sections in our DataFrame's columns.

There's more...

We can also use `select_dtypes` to exclude data types. Also, if we are just interested in the `info` results, we can chain the `select_dtypes` call with the `info` method:

```
nls97.select_dtypes(exclude=["category"]).info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8984 entries, 100061 to 999963
Data columns (total 31 columns):
 #   Column    Non-Null Count    Dtype
---  ------                 --------------  -----
 0   highestgradecompleted    6663 non-null    float64
 1   childathome    4791 non-null    float64
 2   childnotathome    4791 non-null    float64
 3   wageincome    5091 non-null    float64
 4   nightlyhrssleep    6706 non-null    float64
 5   birthmonth    8984 non-null    int64
 6   birthyear    8984 non-null    int64
...
 25  weeksworked12    7747 non-null    float64
 26  weeksworked13    7680 non-null    float64
 27  weeksworked14    7612 non-null    float64
 28  weeksworked15    7389 non-null    float64
 29  weeksworked16    7068 non-null    float64
 30  weeksworked17    6670 non-null    float64
dtypes: float64(29), int64(2)
memory usage: 2.2 MB
```

The filter operator can also take a regular expression. For example, you can return the columns that have `income` in their names:

```
nls97.filter(regex='income')

    wageincome    govincomediff
personid
100061    12,500    NaN
100139    120,000    NaN
100284    58,000    NaN
100292    nan    NaN
100583    30,000    NaN
...              ...             ...
999291    35,000    NaN
999406    116,000    NaN
999543    nan    NaN
999698    nan    NaN
999963    50,000    NaN
```

See also

Many of these techniques can be used to create pandas series as well as DataFrames. We demonstrate this in *Chapter 6*, *Cleaning and Exploring Data With Series Operations*.

Selecting rows

When we are taking the measure of our data and otherwise answering the question, *"How does it look?"*, we are constantly zooming in and out. We are looking at aggregated numbers and particular rows. But there are also important data issues that are only obvious at an intermediate zoom level, issues that we only notice when looking at some subset of rows. This recipe demonstrates how to use the pandas tools for detecting data issues in subsets of our data.

Getting ready...

We will continue working with the NLS data in this recipe.

How to do it...

We will go over several techniques for selecting rows in a pandas DataFrame.

1. Import `pandas` and `numpy`, and load the `nls97` data:

```
import pandas as pd
import numpy as np
nls97 = pd.read_csv("data/nls97.csv")
nls97.set_index("personid", inplace=True)
```

1. Use slicing to start at the 1001st row and go to the 1004th row:

`nls97[1000:1004]` selects every row starting from the row indicated by the integer to the left of the colon (`1000`, in this case) to, but not including, the row indicated by the integer to the right of the colon (`1004`). The row at `1000` is actually the 1001st row because of zero-based indexing. Each row appears as a column in the output since we have transposed the resulting DataFrame:

```
nls97[1000:1004].T

personid    195884    195891    195970    195996
gender    Male    Male    Female    Female
birthmonth    12    9    3    9
birthyear    1981    1980    1982    1980
highestgradecompleted  NaN    12    17    NaN
maritalstatus    NaN    Never-married    Never-married     NaN
...              ...              ...              ...     ...
colenroct15      NaN  1. Not enrolled  1. Not enrolled     NaN
colenrfeb16      NaN  1. Not enrolled  1. Not enrolled     NaN
colenroct16      NaN  1. Not enrolled  1. Not enrolled     NaN
colenrfeb17      NaN  1. Not enrolled  1. Not enrolled     NaN
colenroct17      NaN  1. Not enrolled  1. Not enrolled     NaN
```

1. Use slicing to start at the 1001st row and go to the 1004th row, skipping every other row.

The integer after the second colon (`2` in this case) indicates the size of the step. When the step is excluded it is assumed to be 1. Notice that by setting the value of the step to `2`, we are skipping every other row:

```
nls97[1000:1004:2].T

personid    195884    195970
gender    Male    Female
birthmonth    12    3
```

```
birthyear    1981    1982
highestgradecompleted    NaN    17
maritalstatus    NaN    Never-married
...                    ...              ...
colenroct15    NaN    1. Not enrolled
colenrfeb16    NaN    1. Not enrolled
colenroct16    NaN    1. Not enrolled
colenrfeb17    NaN    1. Not enrolled
colenroct17    NaN    1. Not enrolled
```

1. Select the first three rows using `head` and `[]` operator slicing.

Note that `nls97[:3]` returns the same DataFrame as `nls97.head(3)`. By not providing a value to the left of the colon in `[:3]`, we are telling the operator to get rows from the start of the DataFrame:

```
nls97.head(3).T
```

```
personid    100061    100139    100284
gender    Female    Male    Male
birthmonth    5    9    11
birthyear    1980    1983    1984
...                    ...              ...              ...
colenroct15  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenrfeb16  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct16  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenrfeb17  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct17  1. Not enrolled  1. Not enrolled  1. Not enrolled
```

```
nls97[:3].T
```

```
personid    100061    100139    100284
gender    Female    Male    Male
birthmonth    5    9    11
birthyear    1980    1983    1984
...                    ...              ...              ...
colenroct15  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenrfeb16  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct16  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenrfeb17  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct17  1. Not enrolled  1. Not enrolled  1. Not enrolled
```

1. Select the last three rows using `tail` and `[]` operator slicing.

Note that `nls97.tail(3)` returns the same DataFrame as `nls97[-3:]`:

```
nls97.tail(3).T

personid    999543    999698    999963
gender    Female    Female    Female
birthmonth    8    5    9
birthyear    1984    1983    1982
...                    ...               ...               ...
colenroct15  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenrfeb16  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct16  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenrfeb17  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct17  1. Not enrolled  1. Not enrolled  1. Not enrolled
```

```
nls97[-3:].T

personid    999543    999698    999963
gender    Female    Female    Female
birthmonth    8    5    9
birthyear    1984    1983    1982
...                    ...               ...               ...
colenroct15  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenrfeb16  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct16  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenrfeb17  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct17  1. Not enrolled  1. Not enrolled  1. Not enrolled
```

1. Select a few rows using the `loc` data accessor.

Use the `loc` accessor to select by `index` label. We can pass a list of index labels or we can specify a range of labels. (Recall that we have set `personid` as the index.) Note that `nls97.loc[[195884,195891,195970]]` and `nls97.loc[195884:195970]` return the same DataFrame:

```
nls97.loc[[195884,195891,195970]].T

personid    195884    195891    195970
gender    Male    Male    Female
birthmonth    12    9    3
birthyear    1981    1980    1982
highestgradecompleted    NaN    12    17
maritalstatus            NaN    Never-married    Never-married
...                      ...               ...               ...
colenroct15              NaN  1. Not enrolled  1. Not enrolled
colenrfeb16              NaN  1. Not enrolled  1. Not enrolled
colenroct16              NaN  1. Not enrolled  1. Not enrolled
colenrfeb17              NaN  1. Not enrolled  1. Not enrolled
```

```
colenroct17              NaN  1. Not enrolled  1. Not enrolled
```

```
nls97.loc[195884:195970].T
```

```
personid    195884    195891    195970
gender    Male    Male    Female
birthmonth    12    9    3
birthyear    1981    1980    1982
highestgradecompleted    NaN    12    17
maritalstatus            NaN    Never-married    Never-married
...                      ...              ...              ...
colenroct15              NaN  1. Not enrolled  1. Not enrolled
colenrfeb16              NaN  1. Not enrolled  1. Not enrolled
colenroct16              NaN  1. Not enrolled  1. Not enrolled
colenrfeb17              NaN  1. Not enrolled  1. Not enrolled
colenroct17              NaN  1. Not enrolled  1. Not enrolled
```

1. Select a row from the beginning of the DataFrame with the `iloc` data accessor.

`iloc` differs from `loc` in that it takes a list of row position integers, rather than index labels. For that reason, it works similarly to bracket operator slicing. In this step, we first pass a one-item list with the value of `0`. That returns a DataFrame with the first row:

```
nls97.iloc[[0]].T
```

```
personid    100061
gender    Female
birthmonth    5
birthyear    1980
highestgradecompleted    13
maritalstatus    Married
...                             ...
colenroct15    1. Not enrolled
colenrfeb16    1. Not enrolled
colenroct16    1. Not enrolled
colenrfeb17    1. Not enrolled
colenroct17    1. Not enrolled
```

1. Select a few rows from the beginning of the DataFrame with the `iloc` data accessor.

We pass a three-item list, `[0,1,2]`, to return a DataFrame of the first three rows of `nls97`. We would get the same result if we passed `[0:3]` to the accessor:

```
nls97.iloc[[0,1,2]].T

personid    100061    100139    100284
gender    Female    Male    Male
birthmonth    5    9    11
birthyear    1980    1983    1984
...                   ...              ...              ...
colenroct15  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenrfeb16  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct16  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenrfeb17  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct17  1. Not enrolled  1. Not enrolled  1. Not enrolled

nls97.iloc[0:3].T

personid    100061    100139    100284
gender    Female    Male    Male
birthmonth    5    9    11
birthyear    1980    1983    1984
...                   ...              ...              ...
colenroct15  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenrfeb16  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct16  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenrfeb17  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct17  1. Not enrolled  1. Not enrolled  1. Not enrolled
```

1. Select a few rows from the end of the DataFrame with the `iloc` data accessor.

Use `nls97.iloc[[-3,-2,-1]]` , and `nls97.iloc[-3:]` to retrieve the last three rows of the DataFrame. By not providing a value to the right of the colon in `[-3:]` , we are telling the accessor to get all rows from the third-to-last row to the end of the DataFrame:

```
nls97.iloc[[-3,-2,-1]].T

personid    999543    999698    999963
gender    Female    Female    Female
birthmonth    8    5    9
birthyear    1984    1983    1982
...                   ...              ...              ...
colenroct15  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenrfeb16  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct16  1. Not enrolled  1. Not enrolled  1. Not enrolled
```

```
colenrfeb17  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct17  1. Not enrolled  1. Not enrolled  1. Not enrolled
```

```
nls97.iloc[-3:].T
```

```
personid    999543    999698    999963
gender    Female    Female    Female
birthmonth    8    5    9
birthyear    1984    1983    1982
...                     ...             ...             ...
colenroct15  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenrfeb16  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct16  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenrfeb17  1. Not enrolled  1. Not enrolled  1. Not enrolled
colenroct17  1. Not enrolled  1. Not enrolled  1. Not enrolled
```

1. Select multiple rows conditionally using boolean indexing.

Create a DataFrame of just individuals receiving very little sleep. About 5% of survey respondents got 4 or fewer hours' sleep per night, of the 6,706 individuals who responded to that question. Test who is getting 4 or fewer hours of sleep with `nls97.nightlyhrssleep<=4`, which generates a pandas series of `True` and `False` values that we assign to `sleepcheckbool`. Pass that series to the `loc` accessor to create a `lowsleep` DataFrame. `lowsleep` has approximately the number of rows we are expecting. We do not need to do the extra step of assigning the boolean series to a variable. This is done here only for explanatory purposes:

```
nls97.nightlyhrssleep.quantile(0.05)
```

```
4.0
```

```
nls97.nightlyhrssleep.count()
```

```
6706
```

```
sleepcheckbool = nls97.nightlyhrssleep<=4
sleepcheckbool
```

```
personid
100061    False
100139    False
100284    False
100292    False
100583    False
```

```
          ...
999291    False
999406    False
999543    False
999698    False
999963    False
Name: nightlyhrssleep, Length: 8984, dtype: bool
```

```
lowsleep = nls97.loc[sleepcheckbool]
lowsleep.shape
```

```
(364, 88)
```

1. Select rows based on multiple conditions.

It may be that folks who are not getting a lot of sleep also have a fair number of children who live with them. Use `describe` to get a sense of the distribution of the number of children for those who have `lowsleep`. About a quarter have three or more children. Create a new DataFrame with individuals who have `nightlyhrssleep` of 4 or less and the number of children at home of 3 or more. The `&` is the logical *and* operator in pandas and indicates that both conditions have to be true for the row to be selected. (We would have gotten the same result if we worked from the `lowsleep` DataFrame – `lowsleep3pluschildren = lowsleep.loc[lowsleep.childathome>=3]` – but then we would not have been able to demonstrate testing multiple conditions):

```
lowsleep.childathome.describe()
```

```
count    293.00
mean    1.79
std    1.40
min    0.00
25%    1.00
50%    2.00
75%    3.00
max    9.00
```

```
lowsleep3pluschildren = nls97.loc[(nls97.nightlyhrssleep<=4) & (
nls97.childathome>=3)]
lowsleep3pluschildren.shape
```

```
(82, 88)
```

1. Select rows and columns based on multiple conditions.

Pass the condition to the `loc` accessor to select rows. Also, pass a list of column names to select:

```
lowsleep3pluschildren = nls97.loc[(nls97.nightlyhrssleep<=4) & (
nls97.childathome>=3), ['nightlyhrssleep','childathome']]
lowsleep3pluschildren
```

```
          nightlyhrssleep    childathome
personid
119754    4    4
141531    4    5
152706    4    4
156823    1    3
158355    4    4
...                   ...           ...
905774    4    3
907315    4    3
955166    3    3
956100    4    6
991756    4    3
```

The preceding steps demonstrated the key techniques for selecting rows in pandas.

How it works...

We used the `[]` bracket operator in *steps 2* through *5* to do standard Python-like slicing to select rows. That operator allows us to easily select rows based on a list or a range of values indicated with slice notation. This notation takes the form of `[start:end:step]` , where a value of `1` for `step` is assumed if no value is provided. When a negative number is used for `start` , it represents the number of rows from the end of the DataFrame.The `loc` accessor, used in *step 6*, selects rows based on row index labels. Since `personid` is the index for the DataFrame, we can pass a list of one or more `personid` values to the `loc` accessor to get a DataFrame with rows for those index labels. We can also pass a range of index labels to the accessor, which will return a DataFrame with all rows having index labels between the label to the left of the colon and the label to the right (inclusive); so, `nls97.loc[195884:195970]` returns a DataFrame for rows with `personid` between `195884` and `195970`, including those two values.The `iloc` accessor works very much like the bracket operator. We see this in *steps 7* through *9*. We can pass either a list of integers or a range using slicing notation.One of the most valuable pandas capabilities is boolean indexing. It makes it easy to select rows conditionally. We see this in *step 10*. A test returns a boolean series. The `loc` accessor selects all rows for which the test is `True`. We actually didn't need to assign the boolean data series to the variable that we then passed to the `loc` operator. We could have just passed the test to the `loc` accessor with `nls97.loc[nls97.nightlyhrssleep<=4]`.We should take a closer look at how we used the `loc` accessor to select rows in *step 11*. Each condition in `nls97.loc[(nls97.nightlyhrssleep<=4) & (nls97.childathome>=3)]` is placed in parentheses. An error will be generated if the parentheses are excluded. The `&` operator is the equivalent of `and` in standard Python, meaning that *both* conditions have to be `True` for the row to be selected. We would have used `|` for `or` if we had wanted to select the row if *either* condition was `True`.Finally, *step 12* demonstrates how to select both rows and columns in one call to the `loc` accessor. The criteria for rows appear before the comma and the columns to select appear after the comma, as in the following statement:

```
nls97.loc[(nls97.nightlyhrssleep<=4) & (nls97.childathome>=3), [
'nightlyhrssleep','childathome']]
```

This returns the `nightlyhrssleep` and `childathome` columns for all rows where the individual has `nightlyhrssleep` of less than or equal to 4, and `childathome` greater than or equal to 3.

There's more...

We used three different tools to select rows from a pandas DataFrame in this recipe: the `[]` bracket operator, and two pandas-specific accessors, `loc` and `iloc`. This is a little confusing if you are new to pandas, but it becomes clear which tool to use in which situation after just a few months. If you came to pandas with a fair bit of Python and NumPy experience, you likely find the `[]` operator most familiar. However, the pandas documentation recommends against using the `[]` operator for production code. I have settled on a routine of using that operator only for selecting columns from a DataFrame. I use the `loc` accessor when selecting rows by boolean indexing or by index label, and the `iloc` accessor for selecting rows by row number. Since my workflow has me using a fair bit of boolean indexing, I use `loc` much more than the other methods.

See also

The recipe immediately preceding this one has a more detailed discussion on selecting columns.

Generating frequencies for categorical variables

Many years ago, a very seasoned researcher said to me, *"90% of what we're going to find, we'll see in the frequency distributions."* That message has stayed with me. The more one-way and two-way frequency distributions (crosstabs) I do on a DataFrame, the better I understand it. We will do one-way distributions in this recipe, and crosstabs in subsequent recipes.

Getting ready...

We continue our work with the NLS. We will also be doing a fair bit of column selection using filter methods. It is not necessary to review the recipe in this chapter on column selection, but it might be helpful.

How to do it...

We use pandas tools to generate frequencies, particularly the very handy `value_counts`:

1. Load the `pandas` library and the `nls97` file:

Also convert the columns with object data type to category data type.

```
import pandas as pd
nls97 = pd.read_csv("data/nls97.csv")
nls97.set_index("personid", inplace=True)
nls97.loc[:, nls97.dtypes == 'object'] = \
...   nls97.select_dtypes(['object']). \
...   apply(lambda x: x.astype('category'))
```

1. Show the names for columns with the category data type and check for the number of missing values.

Notice that there are no missing values for `gender` and few for `highestdegree`, but many for `maritalstatus` and other columns:

```
catcols = nls97.select_dtypes(include=["category"]).columns
nls97[catcols].isnull().sum()

gender    0
maritalstatus    2312
weeklyhrscomputer    2274
weeklyhrstv    2273
highestdegree    31
                     ...
colenroct15    1515
colenrfeb16    1948
colenroct16    2251
colenrfeb17    2251
colenroct17    2250
Length: 57, dtype: int64
```

1. Show the frequencies for marital status:

```
nls97.maritalstatus.value_counts()

Married    3066
Never-married    2766
Divorced    663
Separated    154
Widowed    23
Name: maritalstatus, dtype: int64
```

1. Turn off sorting by frequency:

```
nls97.maritalstatus.value_counts(sort=False)

Divorced    663
Married    3066
Never-married    2766
Separated    154
Widowed    23
Name: maritalstatus, dtype: int64
```

1. Show percentages instead of counts:

```
nls97.maritalstatus.value_counts(sort=False, normalize=True)

Divorced    0.10
Married    0.46
Never-married    0.41
Separated    0.02
Widowed    0.00
Name: maritalstatus, dtype: float64
```

1. Show the percentages for all government responsibility columns.

Filter the DataFrame for just the government responsibility columns, then use `apply` to run `value_counts` on all columns in that DataFrame:

```
nls97.filter(like="gov").apply(pd.value_counts, normalize=True)
                   govprovidejobs  govpricecontrols  ...  \

1. Definitely    0.25    0.54  ...
2. Probably    0.34    0.33  ...
3. Probably not    0.25    0.09  ...
4. Definitely not    0.16    0.04  ...

        govdecenthousing  govprotectenvironment

1. Definitely    0.44    0.67
2. Probably    0.43    0.29
3. Probably not    0.10    0.03
4. Definitely not    0.02    0.02
```

1. Find the percentages for all government responsibility columns of people who are married.

Do what we did in *step 6*, but first select only rows with marital status equal to `Married`:

```
nls97[nls97.maritalstatus=="Married"].\
... filter(like="gov").\
... apply(pd.value_counts, normalize=True)

                   govprovidejobs  govpricecontrols  ...  \
1. Definitely    0.17    0.46  ...
2. Probably    0.33    0.38  ...
3. Probably not    0.31    0.11  ...
```

```
4. Definitely not    0.18    0.05  ...
                   govdecenthousing  govprotectenvironment
1. Definitely    0.36    0.64
2. Probably    0.49    0.31
3. Probably not    0.12    0.03
4. Definitely not    0.03    0.01
```

1. Find the frequencies and percentages for all category columns in the DataFrame.

First, open a file to write out the frequencies:

```
freqout = open('views/frequencies.txt', 'w')
for col in nls97.select_dtypes(include=["category"]):
...   print(col, "----------------------", "frequencies",
...   nls97[col].value_counts(sort=False),"percentages",
...   nls97[col].value_counts(normalize=True, sort=False),
...   sep="\n\n", end="\n\n\n", file=freqout)
...
freqout.close()
```

This generates a file, the beginning of which looks like this:

```
gender
----------------------
frequencies
Female    4385
Male    4599
Name: gender, dtype: int64
percentages
Female    0.49
Male    0.51
Name: gender, dtype: float64
```

As these steps demonstrate, `value_counts` is quite useful when we need to generate frequencies for one or more columns of a DataFrame.

How it works...

Most of the columns in the `nls97` DataFrame (57 out of 88) have the object data type. If we are working with data that is logically categorical, but does not have a category data type in pandas, there are good reasons to convert it to the category type. Not only does this save memory, it also makes data cleaning a little easier, as we saw in this recipe.The 0star of the show for this recipe is the `value_counts` method. It can generate frequencies for a series, as we do with `nls97.maritalstatus.value_counts`. It can also be run on a whole DataFrame as we0 do with `nls97.filter(like="gov").apply(pd.value_counts, normalize=True)`. We first create a DataFrame with just the government responsibility columns and then pass the resulting DataFrame to `value_counts` with `apply`.You probably noticed that in *step 7*, I split the chaining over several lines to make it easier to read. There is no rule about when it makes sense to do that. I generally try to do that whenever the chaining involves three or more operations.In *step 8*, we iterate over all of the columns with the category data type: `for col in nls97.select_dtypes(include=["category"])`. For each of those columns, we run `value_counts` to get frequencies and `value_counts` again to get percentages. We use a `print` function so that we can generate the carriage returns necessary to make the output readable. All of this is saved to the `frequencies.txt` file in the `views` subfolder. I find it handy to have a bunch of one-way frequencies around just to check before doing any work with categorical variables. *Step 8* accomplishes that.

There's more...

Frequency distributions may be the most important statistical tool for discovering potential data issues with categorical data. The one-way frequencies we generate in this recipe are a good foundation for further insights.However, we often only detect problems once we examine the relationships between categorical variables and other variables, categorical or continuous. Although we stop short of doing two-way frequencies in this recipe, we do start the process of splitting up the data for investigation in *step 7*. In that step, we look at government responsibility responses for married individuals and see that those responses differ from those for the sample overall.This raises several questions about our data that we need to explore. Are there important differences in response rates by marital status, and might this matter for the distribution of the government responsibility variables? We also want to be careful about drawing conclusions before considering potential confounding variables. Are married respondents likely to be older or to have more children, and are those more important factors in their government responsibility answers?I am using the marital status variable as an example of the kind of queries that producing one-way frequencies, like the ones in this recipe, are likely to generate. It is always good to have some bivariate analyses (a correlation matrix, some crosstabs, or a few scatter plots) at the ready should questions like these come up. We will generate those in the next two chapters.

Generating summary statistics for continuous variables

Pandas has a good number of tools we can use to get a sense of the distribution of continuous variables. We will focus on the splendid functionality of `describe` in this recipe and demonstrate the usefulness of histograms for visualizing variable distributions.Before doing any analysis with a continuous variable it is important to have a good understanding of how it is distributed – its central tendency, its spread, and its skewness. This understanding greatly informs our efforts to identify outliers and unexpected values. But it is also crucial information in and of itself. I do not think it overstates the case to say that we understand a particular variable well if we have a good understanding of how it is distributed, and any interpretation without that understanding will be incomplete or flawed in some way.

Getting ready...

We will work with the COVID totals data in this recipe. You will need **Matplotlib** to run this. If it is not installed on your machine already, you can install it at the terminal by entering `pip install matplotlib` .

How to do it...

We take a look at the distribution of a few key continuous variables:

1. Import `pandas` , `numpy` , and `matplotlib` , and load the COVID case totals data:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
covidtotals = pd.read_csv("data/covidtotals720.csv",
...   parse_dates=['lastdate'])
covidtotals.set_index("iso_code", inplace=True)
```

1. Let’s remind ourselves of the structure of the data:

```
covidtotals.shape
```

```
(209, 12)
```

```
covidtotals.sample(1, random_state=1).T
```

```
iso_code                         COG
lastdate    2020-07-12 00:00:00
location    Congo
total_cases    2,028.00
total_deaths    47.00
total_cases_pm    367.52
total_deaths_pm    8.52
population    5,518,092.00
pop_density    15.40
median_age    19.00
gdp_per_capita    4,881.41
hosp_beds    NaN
region    Central Africa
```

```
covidtotals.dtypes
```

```
lastdate    datetime64[ns]
location    object
total_cases    float64
total_deaths    float64
total_cases_pm    float64
total_deaths_pm    float64
population    float64
pop_density    float64
median_age    float64
gdp_per_capita    float64
hosp_beds    float64
region    object
dtype: object
```

1. Get the descriptive statistics on the COVID totals and demographic columns:

```
covidtotals.describe()
```

```
       total_cases  total_deaths  total_cases_pm  ...  \
count    209.0    209.0    209.0  ...
mean    60,757.4    2,703.0    2,297.0  ...
std    272,440.1    11,895.0    4,039.8  ...
min    3.0    0.0    1.2  ...
25%    342.0    9.0    202.8  ...
50%    2,820.0    53.0    868.9  ...
75%    25,611.0    386.0    2,784.9  ...
max    3,247,684.0    134,814.0    35,795.2  ...
       median_age  gdp_per_capita  hosp_beds
count    185.0    182.0    164.0
mean    30.6    19,285.0    3.0
std    9.1    19,687.7    2.5
min    15.1    661.2    0.1
25%    22.2    4,485.3    1.3
50%    29.9    13,031.5    2.4
75%    38.7    27,882.1    3.9
max    48.2    116,935.6    13.8
[8 rows x 9 columns]
```

1. Take a closer look at the distribution of values for the cases and deaths columns.

Use NumPy's `arange` method to pass a list of floats from 0 to 1.0 to the quantile method of the DataFrame:

```
totvars = ['location','total_cases','total_deaths',
...   'total_cases_pm','total_deaths_pm']
covidtotals[totvars].quantile(np.arange(0.0, 1.1, 0.1))
```

```
     total_cases  total_deaths  total_cases_pm  total_deaths_pm
0.0    3.0    0.0    1.2    0.0
0.1    63.6    0.0    63.3    0.0
0.2    231.2    3.6    144.8    1.2
0.3    721.6    14.4    261.5    3.8
0.4    1,324.4    28.4    378.8    7.0
0.5    2,820.0    53.0    868.9    15.2
0.6    6,695.6    116.6    1,398.3    29.4
0.7    14,316.4    279.0    2,307.9    47.7
0.8    40,245.4    885.2    3,492.3    76.3
0.9    98,632.8    4,719.0    5,407.7    201.4
1.0    3,247,684.0    134,814.0    35,795.2    1,237.6
```

1. View the distribution of total cases:

```
plt.hist(covidtotals['total_cases']/1000, bins=12)
plt.title("Total Covid Cases")
plt.xlabel('Cases')
plt.ylabel("Number of Countries")
plt.show()
```

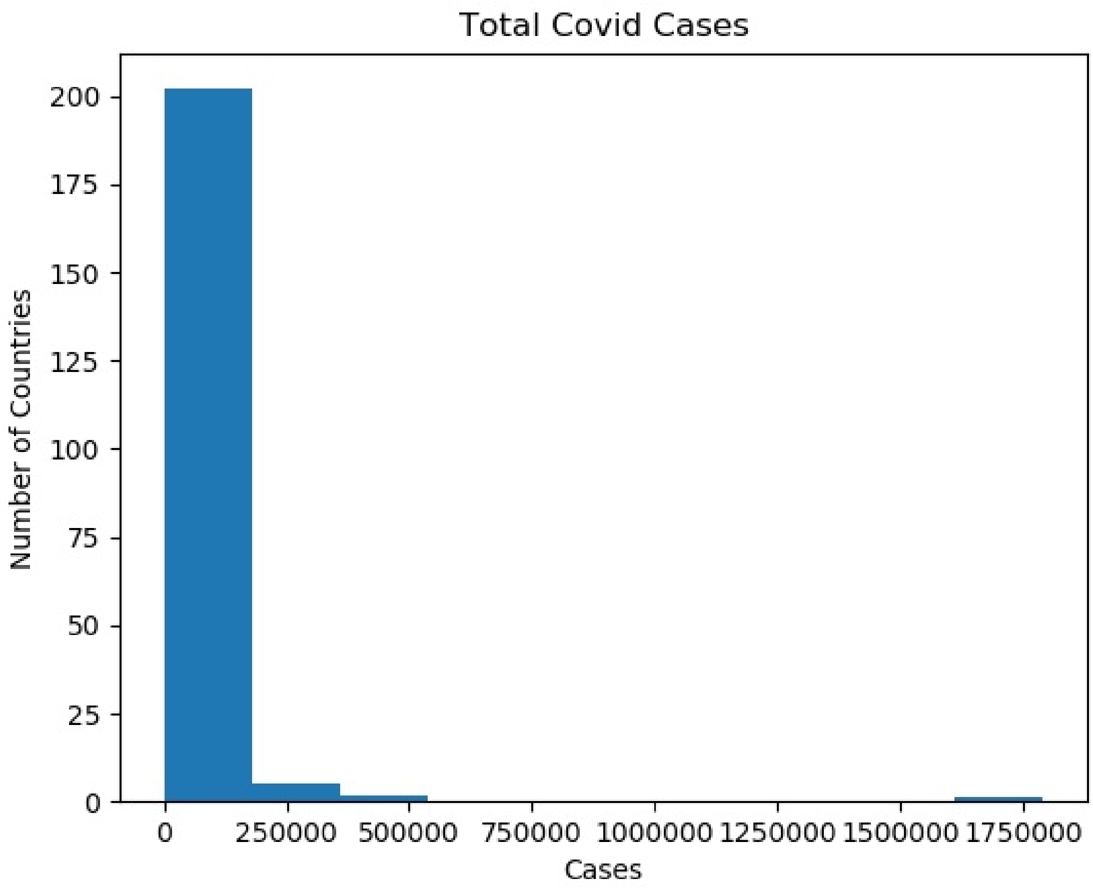


Figure 3.1: Total COVID Cases

The preceding steps demonstrated the use of `describe` and Matplotlib's `hist` method, which are essential tools when working with continuous variables.

How it works...

We use the `describe` method in *step 3* to examine some summary statistics and the distribution of the key variables. It is often a red flag when the mean and median (50%) have dramatically different values. Cases and deaths are heavily skewed to the right (reflected in the mean being much higher than the median). This alerts us to the presence of outliers at the upper end. This is true even with the adjustment for population size, as both `total_cases_pm` and `total_deaths_pm` show this same skew. We do more analysis of outliers in the next chapter.The more detailed percentile data in *step 4* further supports this sense of skewness. For instance, the gap between the 90th-percentile and 100th-percentile values for cases and deaths is substantial. These are good first indicators that we are not dealing with normally distributed data here. Even if this is not due to errors, this matters for the statistical testing we will do down the road. On the list of things we want to note when asked, *"How does the data look?"*, this is one of the first things we want to say.We should also note the large number of zero values for total deaths, over 10%. This will also matter for statistical testing when we get to that point.The histogram of total cases confirms that much of the distribution is between 0 and 150,000, with a few outliers and 1 extreme outlier. Visually, the distribution looks much more log-normal than normal. Log-normal distributions have fatter tails and do not have negative values.

See also

We take a closer look at outliers and unexpected values in the next chapter. We do much more with visualizations in *Chapter 5, Using Visualizations for Exploratory Analysis*.

Using generative AI to view our data

Generative AI tools provide data scientists with a great opportunity to streamline the data cleaning and exploration parts of our workflow. Large language models, in particular, have the potential to make this work much easier and more intuitive. Using these tools we can select rows and columns by criteria, generate summary statistics, and plot variables.A simple way to introduce generative AI tools into your data exploration is with PandasAI. PandasAI uses the OpenAI API to translate natural language queries into data selection and operations that pandas can understand. As of July 2023, OpenAI is the only large language model API that can be used with PandasAI, though the developers of the library anticipate adding other

APIs.We can use PandasAI to substantially reduce the lines of code we need to write to produce some of the tabulations and visualizations we have created so far in this chapter. The steps in this recipe show how you can do that.

Getting ready...

You need to install PandasAI to run the code in this recipe. You can do that with pip install pandasai. We will work with the COVID data again, which is available in the GitHub repository, as is the code.You will also need an API key from OpenAI. You can get one at platform.openai.com. You will need to setup an account and then click on your profile in the upper right corner and then View API keys.

How to do it...

We create a PandasAI instance in the following steps and use it to take a look at the COVID data.

1. We start by importing pandas and the PandasAI library.

```
import pandas as pd
from pandasai import PandasAI
from pandasai.llm.openai import OpenAI
```

1. Next, we load the COVID data and instantiate a PandasAI object.

```
covidtotals = pd.read_csv("data/covidtotals.csv",
  parse_dates=['lastdate'])
covidtotals.set_index("iso_code", inplace=True)
llm = OpenAI(api_token="Your API Key")
pandas_ai = PandasAI(llm)
```

1. Let’s start looking at the COVID data by displaying the first two rows and looking at the data types. We can do this by passing a DataFrame and natural language instructions to the PandasAI run method.

```
pandas_ai.run(covidtotals, "Show first two rows.").T
```

```
iso_code                         AFG
```

```
  ALB
lastdate    2020-07-12 00:00:00    2020-07-12 00:00:00
location    Afghanistan    Albania
total_cases    34,451.0    3,371.0
total_deaths    1,010.0    89.0
total_cases_pm    885.0    1,171.4
total_deaths_pm    25.9    30.9
population    38,928,341.0    2,877,800.0
pop_density    54.4    104.9
median_age    18.6    38.0
gdp_per_capita    1,804.0    11,803.4
hosp_beds    0.5    2.9
region    South Asia    Eastern Europe
```

```
pandas_ai.run(covidtotals, "Show column types.")
```

```
lastdate    datetime64[ns]
location    object
total_cases    float64
total_deaths    float64
total_cases_pm    float64
total_deaths_pm    float64
population    float64
pop_density    float64
median_age    float64
gdp_per_capita    float64
hosp_beds    float64
region    object
dtype: object
```

1. We can see which locations (countries) have the highest total cases.

```
pandas_ai.run(covidtotals, "Show total cases for locations with
the most.")
```

```
location
United States    3,247,684
Brazil    1,839,850
India    849,553
Russia    720,547
Peru    322,710
Name: total_cases, dtype: float64
```

1. We can show the highest total cases per million as well, and also show other columns.

```
pandas_ai.run(covidtotals, "Show total cases pm, total deaths pm
, and location for locations with the 10 highest total cases pm.
")

          total_cases_pm  total_deaths_pm    location
iso_code
QAT    35,795.2    50.7    Qatar
SMR    21,038.4    1,237.6    San Marino
BHR    19,082.2    61.1    Bahrain
CHL    16,322.7    360.0    Chile
VAT    14,833.1    0.0    Vatican
KWT    12,658.3    90.4    Kuwait
AND    11,065.8    673.0    Andorra
OMN    10,711.0    48.6    Oman
ARM    10,593.8    188.6    Armenia
PAN    10,274.5    207.0    Panama
```

1. We can also create a DataFrame with selected columns. This is easy because the PandasAI `run` method returns a pandas DataFrame.

```
covidtotalsabb = pandas_ai.run(covidtotals, "Select total cases
pm, total deaths pm, and location.")
covidtotalsabb
```

```
          total_cases_pm  total_deaths_pm        location
iso_code
AFG    885.0    25.9    Afghanistan
ALB    1,171.4    30.9    Albania
DZA    426.7    22.9    Algeria
AND    11,065.8    673.0    Andorra
AGO    14.7    0.8    Angola
                 ...              ...              ...
VNM    3.8    0.0    Vietnam
ESH    868.9    1.7    Western Sahara
YEM    46.6    12.2    Yemen
ZMB    103.1    2.3    Zambia
ZWE    66.1    1.2    Zimbabwe
[209 rows x 3 columns]
```

1. We do not need to be very precise in the language we pass to PandasAI. Instead of writing `Select`, we could have written `Get`. We get the same results. Even `Grab` will work (not shown).

```
covidtotalsabb = pandas_ai.run(covidtotals, "Get total cases pm,
 total deaths pm, and location.")
covidtotalsabb
```

```
          total_cases_pm  total_deaths_pm        location
iso_code
AFG    885.0    25.9    Afghanistan
ALB    1,171.4    30.9    Albania
DZA    426.7    22.9    Algeria
AND    11,065.8    673.0    Andorra
AGO    14.7    0.8    Angola
                 ...               ...              ...
VNM    3.8    0.0    Vietnam
ESH    868.9    1.7    Western Sahara
YEM    46.6    12.2    Yemen
ZMB    103.1    2.3    Zambia
ZWE    66.1    1.2    Zimbabwe
[209 rows x 3 columns]
```

1. We can select rows by summary statistic. For example, we can choose those rows where total cases per million is greater than the 95th percentile. (This might take a little while to run on your machine.)

```
pandas_ai.run(covidtotals, "Show total cases pm and location whe
re total cases pm greater than 95th percentile.")
          total_cases_pm       location
iso_code

AND    11,065.8    Andorra
ARM    10,593.8    Armenia
BHR    19,082.2    Bahrain
CHL    16,322.7    Chile
KWT    12,658.3    Kuwait
OMN    10,711.0    Oman
PAN    10,274.5    Panama
QAT    35,795.2    Qatar
SMR    21,038.4    San Marino
USA    9,811.7    United States
VAT    14,833.1    Vatican
```

1. We can see how continuous variables are distributed by asking for their distribution.

```
pandas_ai.run(covidtotals, "Show the distribution of total cases
 pm and total deaths pm.")

       total_cases_pm                 total_deaths_pm
count    209.0    209.0
mean    2,297.0    73.6
std    4,039.8    156.3
```

```
min    1.2    0.0
25%    202.8    2.8
50%    868.9    15.2
75%    2,784.9    58.4
max    35,795.2    1,237.6
```

1. We can also generate summary statistics. This reveals that there were 12,698,299 cases and 564,917 deaths worldwide as of the time this dataset was downloaded.

```
pandas_ai.run(covidtotals, "Show sum of total cases and total de
aths.")
```

```
'12698299 564917'
```

1. We can get group totals. Let's get the total cases and deaths by region.

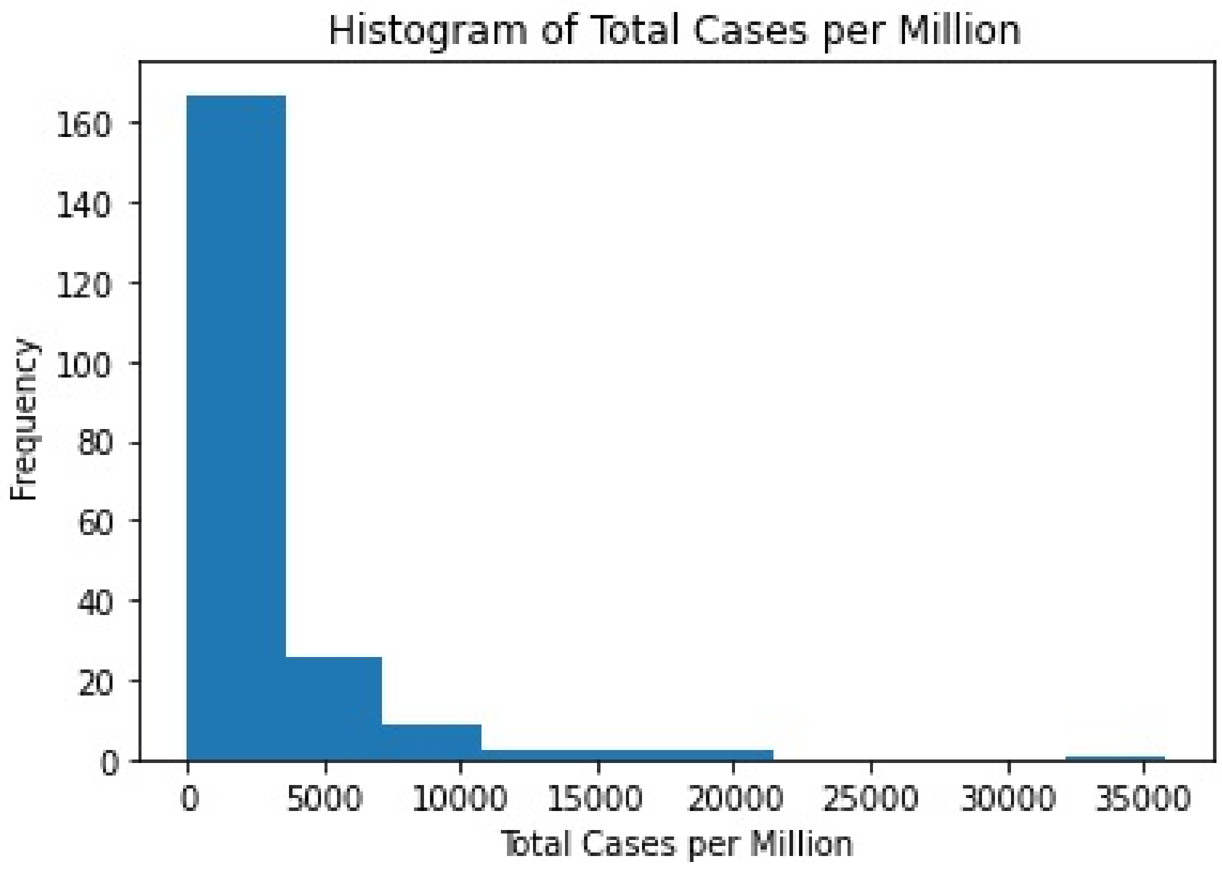
```
pandas_ai.run(covidtotals, "Show sum of total cases and total de
aths by region.")
```

```
    total_cases    total_deaths
region
Caribbean    68,689    1,325
Central Africa    41,848    862
Central America    120,018    3,211
Central Asia    142,814    1,365
East Africa    44,604    1,219
East Asia    307,282    11,041
Eastern Europe    996,788    19,805
North Africa    118,046    5,106
North America    3,650,287    178,317
Oceania / Aus    11,207    136
South America    2,845,898    103,917
South Asia    1,368,451    31,352
Southern Africa    272,934    4,094
West Africa    101,128    1,708
West Asia    1,088,492    25,165
Western Europe    1,519,813    176,294
```

1. We can easily generate plots on the COVID data.

```
pandas_ai.run(covidtotals, "Plot a histogram of total cases pm")
```

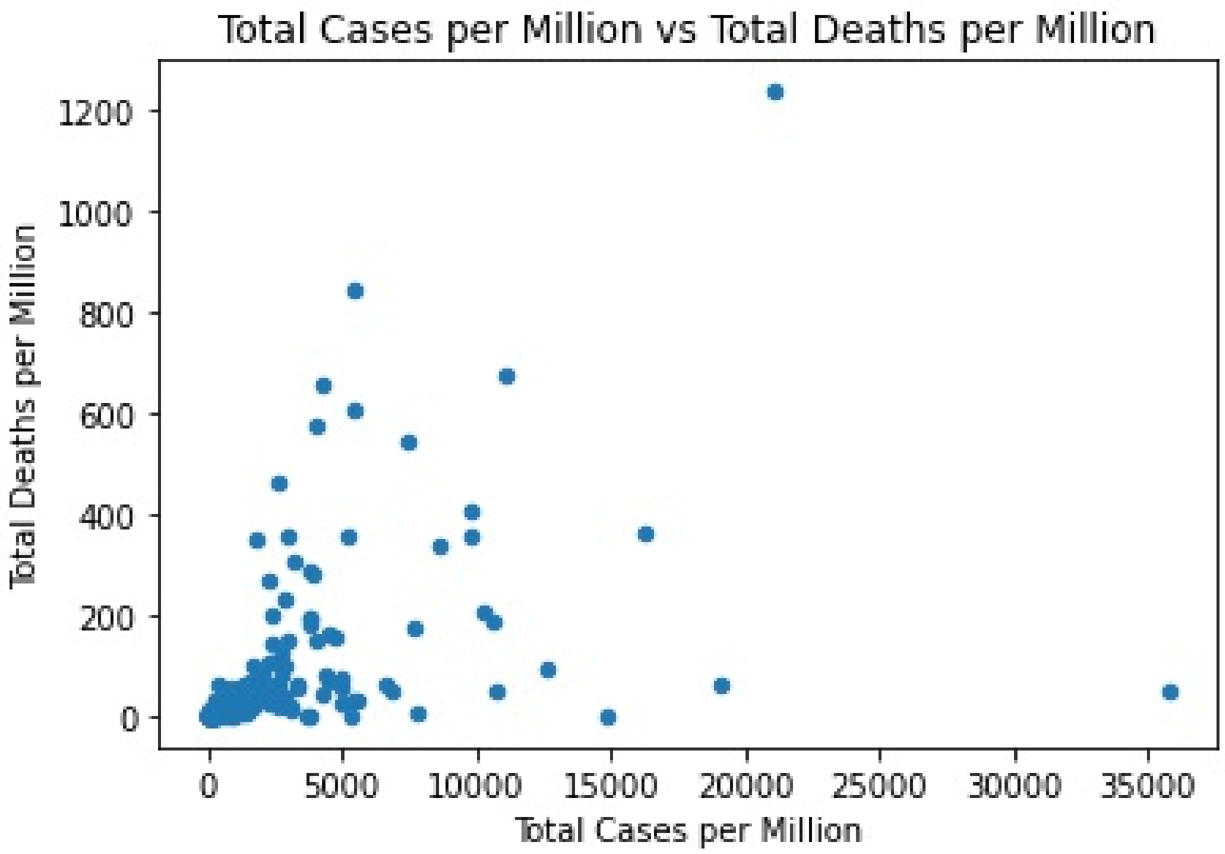
This generates the following plot:



1. We can also generate a scatterplot. Let's look at total cases per million by total deaths per million.

```
pandas_ai.run(covidtotals, "Plot total cases pm by total deaths
pm")
```

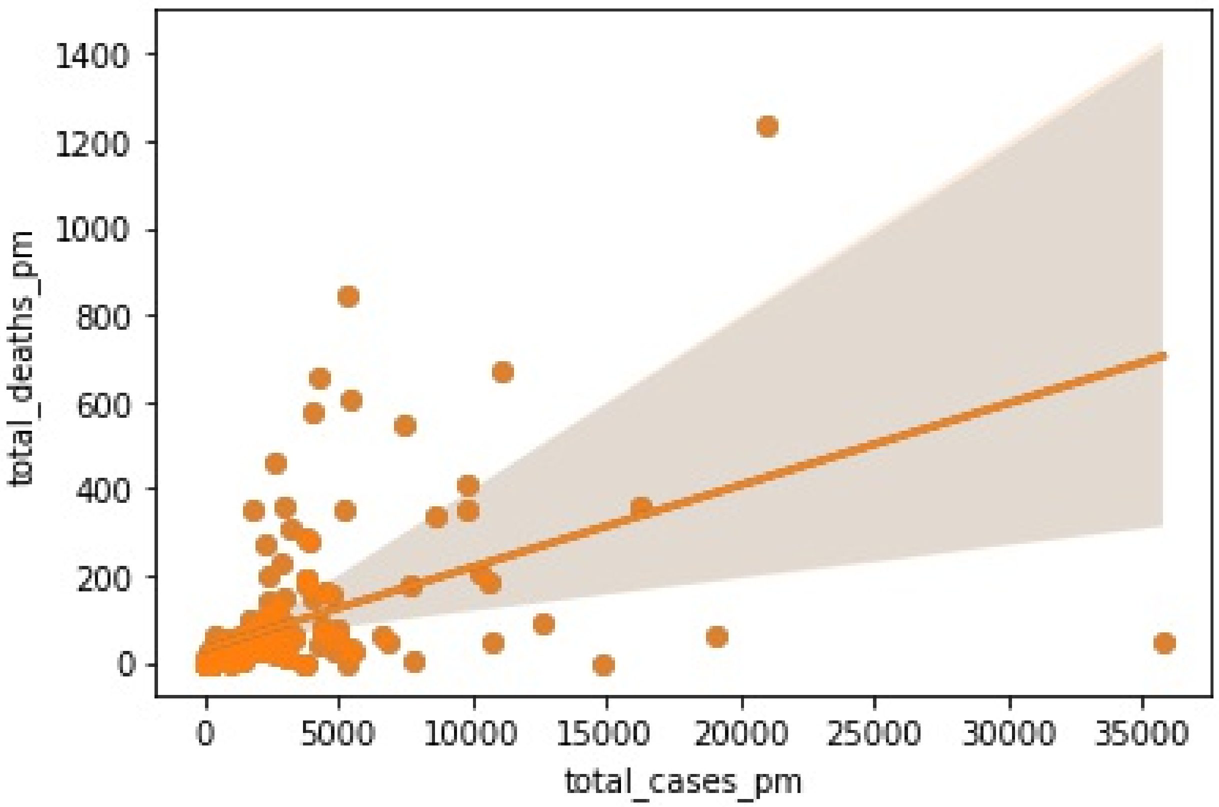
This produces the following plot:



1. We can indicate which plotting tool we want to use. A `regplot` here might be helpful. It might give us a better sense of the relationship between cases and deaths.

```
pandas_ai.run(covidtotals, "Use regplot to show total deaths pm
by total cases pm")
```

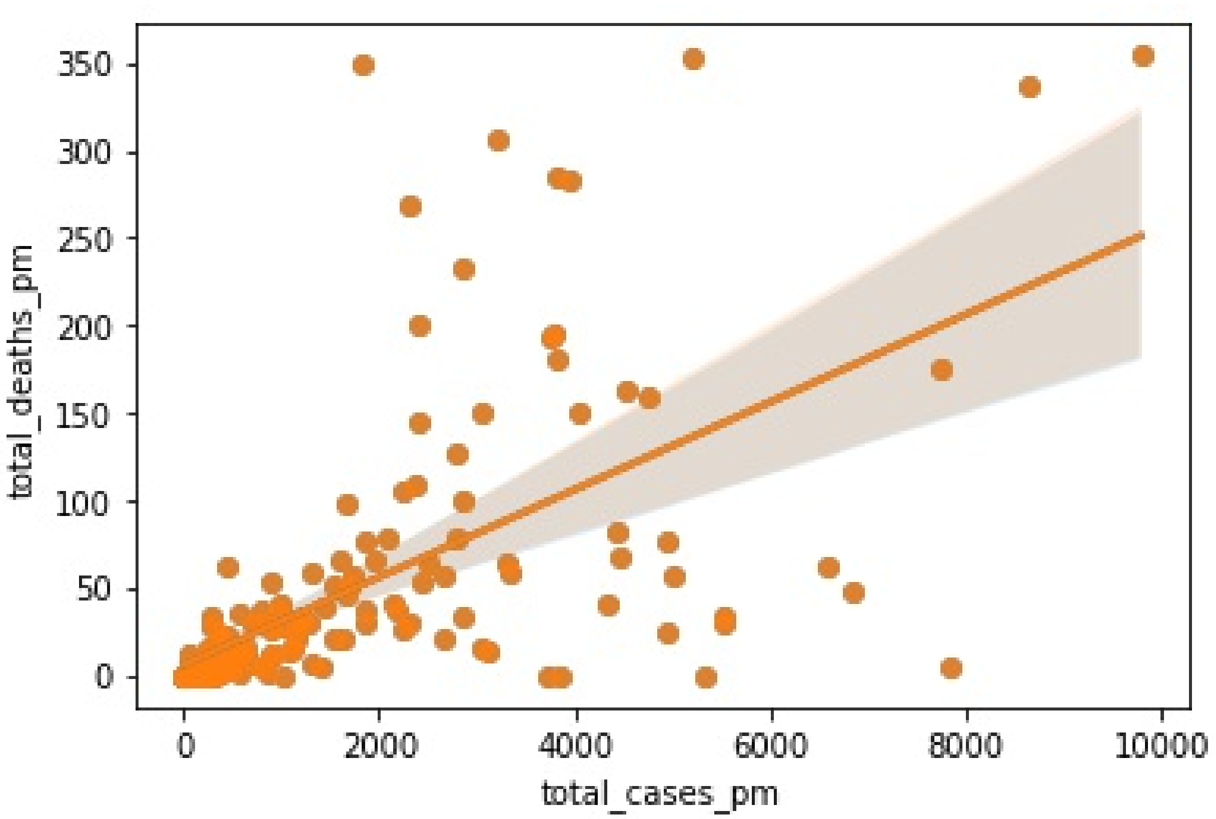
This produces the following plot:



1. The extreme values for cases or deaths make it harder to visualize the relationship between the two. Let's also ask PandasAI to remove extreme values.

```
pandas_ai.run(covidtotals, "Use regplot to show total deaths pm
by total cases pm without extreme values")
```

This produces the following plot:



This removed deaths per million above 350 and cases per million above 10000. It is easier to see the slope of the relationship over much of the data. We will work more with `regplot` and many other plotting tools in *Chapter 5*, *Using Visualizations for the Identification of Unexpected Values*.

How it works...

These examples demonstrate how intuitive it is to use PandasAI. Generative AI tools like PandasAI have the potential to improve our exploratory work, by making it possible to interact with the data nearly as quickly as we can imagine new analyses. We only need to pass natural language queries to the PandasAI object to get the results we want.The queries we pass are not commands. We can use any language we want that conveys our intent. Recall, for example, that we were able write `select` , or `get` , or even `grab` to choose columns. OpenAI's large language model is generally very good at understanding what we mean.A tool that helps us move more swiftly from question to answer can improve our thinking and analysis. It is definitely worth experimenting with if you have not done so already, even if you have well established routines for looking at your data.

See also

The PandasAI GitHub repository is a great place to go for more information and to keep apprised of updates in the library. You can get to it here: https://github.com/gventuri/pandas-ai. We will return to the PandasAI library in *Chapter 5, Using Visualizations for the Identification of Unexpected Values* and *Chapter 8, Fixing Messy Data When Aggregating*.

4 Identifying Missing Values and Outliers in Subsets of Data

Outliers and unexpected values may not be errors. They often are not. Individuals and events are complicated and surprise the analyst. Some people really are 7’4” tall and some really have $50 million salaries. Sometimes, data are messy because people and situations are messy; however, extreme values can have an out-sized impact on our analysis, particularly when we are using parametric techniques that assume a normal distribution.These issues may become even more apparent when working with subsets of data. That is not just because extreme or unexpected values have more weight with smaller samples. It is also because they may make less sense when bivariate and multivariate relationships are considered. When the 7’4” person, or the person making $50 million, is 10 years old, the red flag gets even redder. We take these complications into account in this chapter when considering strategies for detecting outliers, unexpected values, and missing values.Specifically, the recipes in this chapter examine the following:

- Finding missing values
- Identifying outliers with one variable
- Identifying outliers and unexpected values in bivariate relationships
- Using subsetting to examine logical inconsistencies in variable relationships
- Using linear regression to identify data points with significant influence

- Using k-nearest neighbor to find outliers
- Using Isolation Forest to find anomalies

Finding missing values

Before starting any analysis, we need to have a good sense of the number of missing values for each variable, and why those values are missing. We also want to know which rows in our data frame are missing values for several key variables. We can get this information with just a couple of statements in pandas.We also need good strategies for dealing with missing values before we begin statistical modeling, since those models do not typically handle missing values flexibly. We introduce imputation strategies in this recipe and go into more detail in subsequent recipes in this chapter.

Getting ready

We will work with cumulative data on coronavirus cases and deaths by country. The data frame has other relevant information, including population density, age, and GDP.

Note

> Our World in Data provides Covid-19 public use data at https://ourworldindata.org/coronavirus-source-data. The data used in this recipe was downloaded on June 1, 2020. The Covid case and death data were missing for Hong Kong as of this date, but this problem was rectified in files after that.

We will also be doing some routine plotting with Matplotlib in this recipe to help us visualize the distributions of Covid cases and deaths. You can install Matplotlib using `pip install matplotlib`.

How to do it…

We make good use of the `isnull` and `sum` functions to count the number of missing values for selected columns and the number of rows that have missing values for several key variables. We then use the very handy data frame `fillna` method to impute missing values:

1. Load the `pandas` library, along with the Covid case data file.

Also, set up the Covid case and demographic columns:

```
import pandas as pd
covidtotals = pd.read_csv("data/covidtotalswithmissings.csv")
totvars = ['location','total_cases',
...   'total_deaths','total_cases_pm',
...   'total_deaths_pm']
demovars = ['population','pop_density',
...   'median_age','gdp_per_capita',
...    'hosp_beds']
```

1. Check the demographic columns for missing data.

Set the axis to `0` (the default) to check for the count of countries that are missing values for each of the demographic variables (missing values down columns). Notice that 46 out of 210 countries, more than 20 percent of countries, are missing `hosp_beds`. Set the axis to `1` to check for the number of demographic variables that are missing for each country (missing values across rows). Next, get `value_counts` on the resulting `demovarsmisscnt` series to see whether some countries have missing values for much of the demographic data. Notice that 10 countries are missing values for 3 out of the 5 demographic variables, while 8 countries are missing values for 4 out of 5 demographic variables:

```
covidtotals[demovars].isnull().sum(axis=0)

population    0
pop_density    12
median_age    24
gdp_per_capita    28
hosp_beds    46
dtype: int64

demovarsmisscnt = covidtotals[demovars].isnull().sum(axis=1)
demovarsmisscnt.value_counts()

0    156
1    24
2    12
```

```
3    10
4    8
dtype: int64
```

1. List the countries with three or more missing values for the demographic data.

Index alignment and Boolean indexing allow us to use the count of missing values (`demovarsmisscnt`) to select rows. Append the location to the `demovars` list to see the country. (We only show the first five of these countries here.):

```
covidtotals.loc[demovarsmisscnt>=3, ['location'] + demovars].hea
d(5).T
```

```
iso_code    AND    AIA    BES  \
location    Andorra    Anguilla    Bonaire ...
population    77,265    15,002    26,221
pop_density    164    NaN    NaN
median_age    NaN    NaN    NaN
gdp_per_capita    NaN    NaN    NaN
hosp_beds    NaN    NaN    NaN
iso_code    VGB    FRO
location    British Vi ...    Faeroe Islands
population    30,237    48,865
pop_density    208    35
median_age    NaN    NaN
gdp_per_capita    NaN    NaN
hosp_beds    NaN    NaN
```

```
type(demovarsmisscnt)
```

```
<class 'pandas.core.series.Series'>
```

1. Check the Covid case data for missing values.

Notice that only one country has missing values for any of this data:

```
covidtotals[totvars].isnull().sum(axis=0)
```

```
location    0
total_cases    0
total_deaths    0
total_cases_pm    1
total_deaths_pm    1
```

```
dtype: int64

totvarsmisscnt = covidtotals[totvars].isnull().sum(axis=1)
totvarsmisscnt.value_counts()

0    209
2    1

dtype: int64
covidtotals.loc[totvarsmisscnt>0].T

iso_code    HKG
lastdate    2020-05-26 00:00:00
location    Hong Kong
total_cases    0
total_deaths    0
total_cases_pm    NaN
total_deaths_pm    NaN
population    7,496,988
pop_density    7,040
median_age    45
gdp_per_capita    56,055
hosp_beds    NaN
```

1. Use the `fillna` method to fix the missing cases data for the one country affected (Hong Kong).

We could just set the values to `0`, since the numerator is `0` in both cases. However, it is helpful in terms of code reuse to use the correct logic:

```
covidtotals.total_cases_pm. \
...   fillna(covidtotals.total_cases/
...   (covidtotals.population/1000000),
...   inplace=True)
covidtotals.total_deaths_pm. \
...   fillna(covidtotals.total_deaths/
...   (covidtotals.population/1000000),
...   inplace=True)
covidtotals[totvars].isnull().sum(axis=0)

location    0
total_cases    0
total_deaths    0
total_cases_pm    0
total_deaths_pm    0
dtype: int64
```

These steps give us a good sense of the number of missing values that we have for each column, and which countries have many missing values.

How it works...

Step 2 shows that there is a fair bit of missing data for the demographic variables, particularly for the number of hospital beds. 18 countries have at least 3 of the 5 demographic variables missing. We will either have to exclude those variables from any multivariate analyses we will do in the future or impute values for those variables. We make no attempt to fix those values here. We look more at fixing missing values, including by imputing values, in subsequent chapters.The key Covid case data is relatively free of missing values. We have one country with missing cases or death data, which we resolve in *step 5*. We use `fillna` to fix the missing value. We could have also used `fillna` to set the missing value to `0`.We should not gloss over the little bit of pandas magic in *steps 2 and 3*. We create a series, `demovarsmisscnt`, which has the count of demographic columns that have missing values for each country. We are able to use that series, along with the three or more test series (`demovarsmisscnt>=3`), because of pandas index alignment and Boolean indexing. That’s magic I say!

See also

We examine other pandas techniques for fixing missing values in *Chapter 6, Cleaning and Wrangling Data with Pandas Series Operations*.

Identifying outliers with one variable

The concept of an outlier is somewhat subjective but is closely tied to the properties of a particular distribution; to its central tendency, spread, and shape. We make assumptions about whether a value is expected or unexpected based on how likely we are to get that value given the variable’s distribution. We are more inclined to view a value as an outlier if it is multiple standard deviations away from the mean and it is from a distribution that is approximately normal; one that is symmetrical (has low skew) and has relatively skinny tails (low kurtosis).This becomes clear if we imagine trying to identify outliers from a uniform distribution. There is no central tendency and there are no tails. Each value is equally likely. If, for example, Covid cases per country were uniformly distributed, with a minimum of 1 and a maximum of 10,000,000, neither 1 nor 10,000,000 would be considered an outlier.We need to understand how a variable is distributed, then, before we can identify outliers. Several Python libraries provide tools to help us understand how variables of interest are distributed. We use a couple of them in this recipe to identify when a value is sufficiently out of range to be of concern.

Getting ready

You will need the `matplotlib`, `statsmodels`, and `scipy` libraries, in addition to `pandas` and `numpy`, to run the code in this recipe. You can install `matplotlib`, `statsmodels`, and `scipy` by entering `pip install matplotlib`, `pip install statsmodels`, and `pip install scipy` in a terminal client or `powershell` (in Windows).We continue to work with the Covid cases data.

How to do it…

We take a good look at the distribution of some of the key continuous variables in the Covid data. We examine the central tendency and shape of the distribution, generating measures and visualizations of normality:

1. Load the `pandas`, `numpy`, `matplotlib`, `statsmodels`, and `scipy` libraries, and the Covid case data file.

Also, set up the Covid case and demographic columns:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm
import scipy.stats as scistat
covidtotals = pd.read_csv("data/covidtotals720.csv")
covidtotals.set_index("iso_code", inplace=True)
totvars = ['location','total_cases',
...   'total_deaths','total_cases_pm',
...   'total_deaths_pm']
demovars = ['population','pop_density',
...   'median_age','gdp_per_capita',
...    'hosp_beds']
```

1. Get descriptive statistics for the Covid case data.

Create a data frame with just the key case data:

```
covidtotalsonly = covidtotals.loc[:, totvars]
covidtotalsonly.describe()
       total_cases  total_deaths  total_cases_pm  \

count    209    209    209
mean    60,757    2,703    2,297
std    272,440    11,895    4,040
min    3    0    1
25%    342    9    203
50%    2,820    53    869
75%    25,611    386    2,785
max    3,247,684    134,814    35,795
       total_deaths_pm
count    209
mean    74
std    156
min    0
25%    3
50%    15
75%    58
max    1,238
```

1. Show more detailed percentile data:

```
covidtotalsonly.quantile(np.arange(0.0, 1.1, 0.1))

      total_cases  total_deaths  total_cases_pm  \
0.00    3.00    0.00    1.23
0.10    63.60    0.00    63.33
0.20    231.20    3.60    144.82
0.30    721.60    14.40    261.51
0.40    1,324.40    28.40    378.78
0.50    2,820.00    53.00    868.87
0.60    6,695.60    116.60    1,398.33
0.70    14,316.40    279.00    2,307.93
0.80    40,245.40    885.20    3,492.31
0.90    98,632.80    4,719.00    5,407.65
```

```
1.00    3,247,684.00    134,814.00    35,795.16
      total_deaths_pm
0.00    0.00
0.10    0.00
0.20    1.24
0.30    3.76
0.40    7.02
0.50    15.22
0.60    29.37
0.70    47.73
0.80    76.28
0.90    201.42
1.00    1,237.55
```

Also show skewness and kurtosis. Skewness and kurtosis describe how symmetrical the distribution is and how fat the tails of the distribution are, respectively. Both measures are significantly higher than we would expect if our variables were distributed normally:

```
covidtotalsonly.skew()

total_cases    9.33
total_deaths    8.13
total_cases_pm    4.28
total_deaths_pm    3.91
dtype: float64
covidtotalsonly.kurtosis()
total_cases    99.15
total_deaths    79.38
total_cases_pm    26.14
total_deaths_pm    19.44
dtype: float64
```

1. Test the Covid data for normality.

Use the Shapiro-Wilk test from the `scipy` library. Print out the p-value from the test. (The `null` hypothesis of a normal distribution can be rejected at the 95% level at any p-value below 0.05.):

```
def testnorm(var, df):
  stat, p = scistat.shapiro(df[var])
  return p
print("total cases: %.5f" % testnorm("total_cases", covidtotalso
nly))
print("total deaths: %.5f" % testnorm("total_deaths", covidtotal
sonly))
print("total cases pm: %.5f" % testnorm("total_cases_pm", covidt
otalsonly))
print("total deaths pm: %.5f" % testnorm("total_deaths_pm", covi
dtotalsonly))
```

```
total cases: 0.00000
total deaths: 0.00000
total cases pm: 0.00000
total deaths pm: 0.00000
```

1. Show normal quantile-quantile plots (`qqplots`) of total cases and total cases per million.

The straight lines show what the distributions would look like if they were normal:

```
sm.qqplot(covidtotalsonly[['total_cases']]. \
...   sort_values(['total_cases']), line='s')
plt.title("QQ Plot of Total Cases")
sm.qqplot(covidtotals[['total_cases_pm']]. \
...   sort_values(['total_cases_pm']), line='s')
plt.title("QQ Plot of Total Cases Per Million")
plt.show()
```

This results in the following scatter plots:

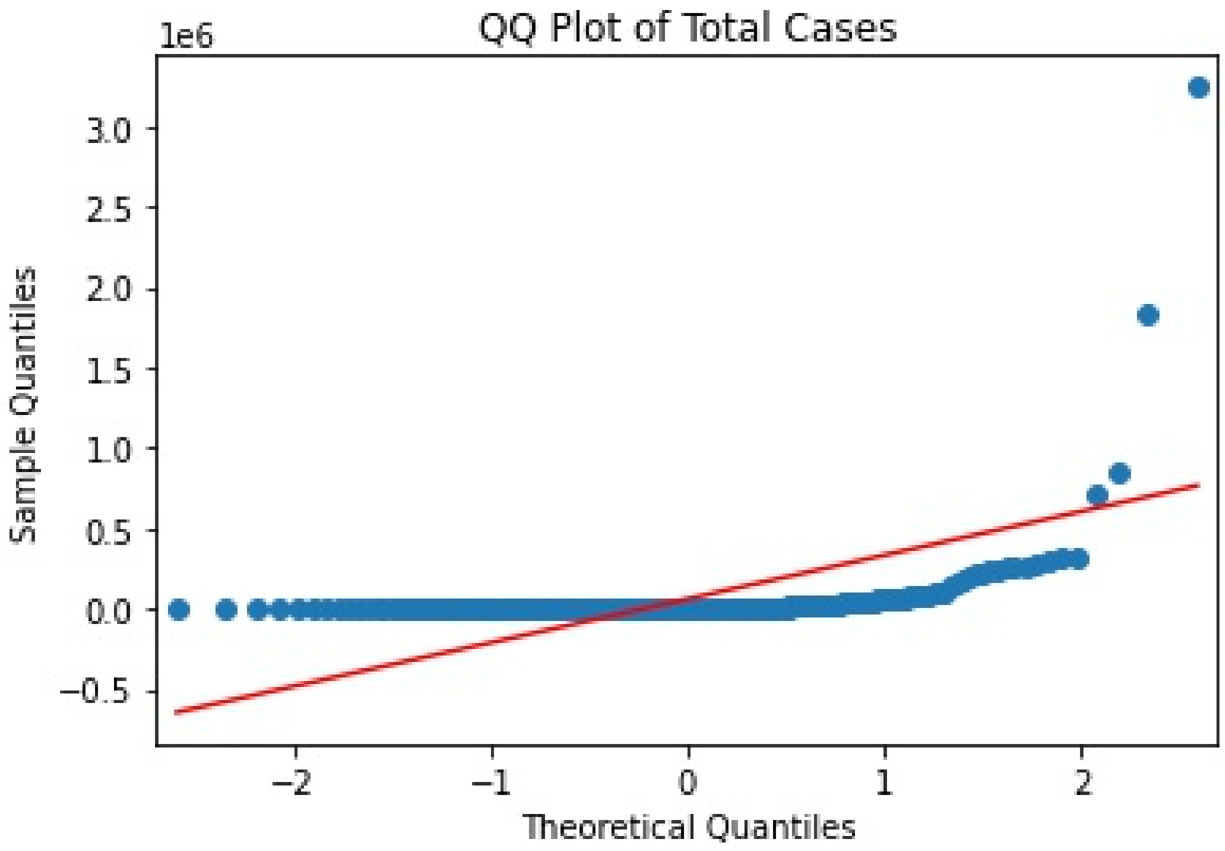


Figure 4.1: Distribution of Covid cases compared with a normal distribution

Even when adjusted by population with the total cases per million column, the distribution is substantially different from normal:

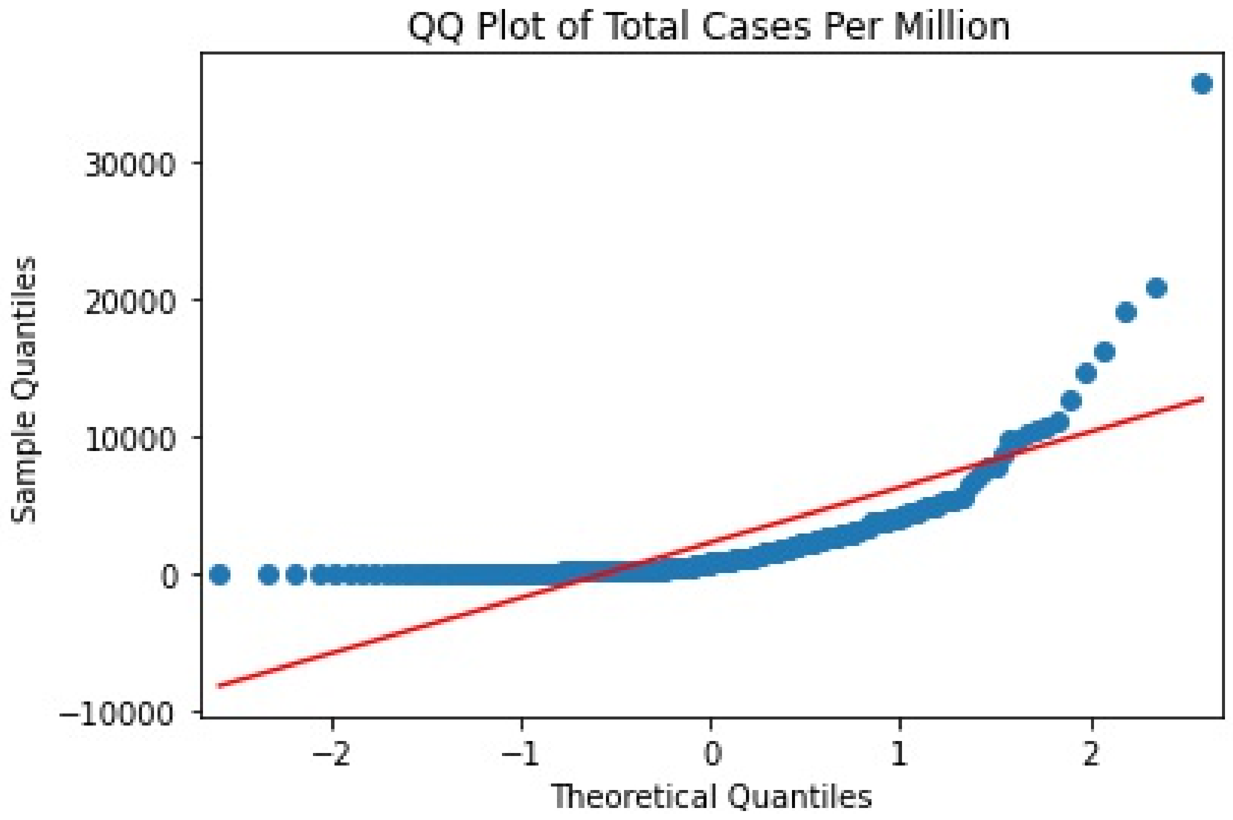


Figure 4.2: Distribution of Covid cases per million compared with a normal distribution

1. Show the outlier range for total cases.

One way to define an outlier for a continuous variable is by distance above the third quartile or below the first quartile. If that distance is more than 1.5 times the interquartile range (the distance between the first and third quartiles), that value is considered an outlier. In this case, since only 0 or positive values are possible, any total cases value above 25,028 is considered an outlier:

```
thirdq, firstq = covidtotalsonly.total_cases.quantile(0.75), cov
idtotalsonly.total_cases.quantile(0.25)
interquartilerange = 1.5*(thirdq-firstq)
outlierhigh, outlierlow = interquartilerange+thirdq, firstq-inte
rquartilerange
print(outlierlow, outlierhigh, sep=" <--> ")
```

```
-37561.5 <--> 63514.5
```

1. Generate a data frame of outliers and write it to Excel.

Iterate over the four Covid case columns. Calculate the outlier thresholds for each column as we did in the previous step. Select from the data frame those rows above the high threshold or below the low threshold. Add columns that indicate the variable examined (`varname`) for outliers and the threshold levels:

```
def getoutliers():
...   dfout = pd.DataFrame(columns=covidtotals. \
...     columns, data=None)
...   for col in covidtotalsonly.columns[1:]:
...     thirdq, firstq = covidtotalsonly[col].\
...       quantile(0.75),covidtotalsonly[col].\
...       quantile(0.25)
...     interquartilerange = 1.5*(thirdq-firstq)
...     outlierhigh, outlierlow = \
...       interquartilerange+thirdq, \
...       firstq-interquartilerange
...     df = covidtotals.loc[(covidtotals[col]> \
...       outlierhigh) | (covidtotals[col]< \
...       outlierlow)]
...     df = df.assign(varname = col,
...       threshlow = outlierlow,
...       threshhigh = outlierhigh)
...     dfout = pd.concat([dfout, df])
...   return dfout
...
outliers = getoutliers()
outliers.varname.value_counts()
```

```
total_deaths    36
total_cases    33
total_deaths_pm    28
total_cases_pm    17
Name: varname, dtype: int64
```

```
outliers.to_excel("views/outlierscases.xlsx")
```

1. Look a little more closely at outliers for cases per million.

Use the `varname` column we created in the previous step to select the outliers for `total_cases_pm`. Also show columns (`pop_density` and `gdp_per_capita`) that might help to explain the extreme values and the interquartile range for those columns:

```
outliers.loc[outliers.varname=="total_cases_pm",
...   ['location','total_cases_pm','pop_density',
...   'gdp_per_capita']].\
...   sort_values(['total_cases_pm'], ascending=False)

          location  total_cases_pm  pop_density  \
QAT    Qatar    35,795    227
SMR    San Marino    21,038    557
BHR    Bahrain    19,082    1,936
CHL    Chile    16,323    24
VAT    Vatican    14,833    NaN
KWT    Kuwait    12,658    232
AND    Andorra    11,066    164
OMN    Oman    10,711    15
ARM    Armenia    10,594    103
PAN    Panama    10,274    55
USA    United States    9,812    36
PER    Peru    9,787    25
BRA    Brazil    8,656    25
SGP    Singapore    7,826    7,916
LUX    Luxembourg    7,735    231
SWE    Sweden    7,416    25
BLR    Belarus    6,854    47
     gdp_per_capita
QAT    116,936
SMR    56,861
BHR    43,291
CHL    22,767
VAT    NaN
KWT    65,531
AND    NaN
OMN    37,961
ARM    8,788
PAN    22,267
USA    54,225
PER    12,237
BRA    14,103
SGP    85,535
LUX    94,278
SWE    46,949
BLR    17,168

covidtotals[['pop_density','gdp_per_capita']].quantile([0.25,0.5
,0.75])
      pop_density  gdp_per_capita
```

```
0.25    37.42    4,485.33
0.50    87.25    13,031.53
0.75    213.54    27,882.13
```

1. Show a histogram of total cases:

```
plt.hist(covidtotalsonly['total_cases']/1000, bins=7)
plt.title("Total Covid Cases (thousands)")
plt.xlabel('Cases')
plt.ylabel("Number of Countries")
plt.show()
```

This code produces the following plot:

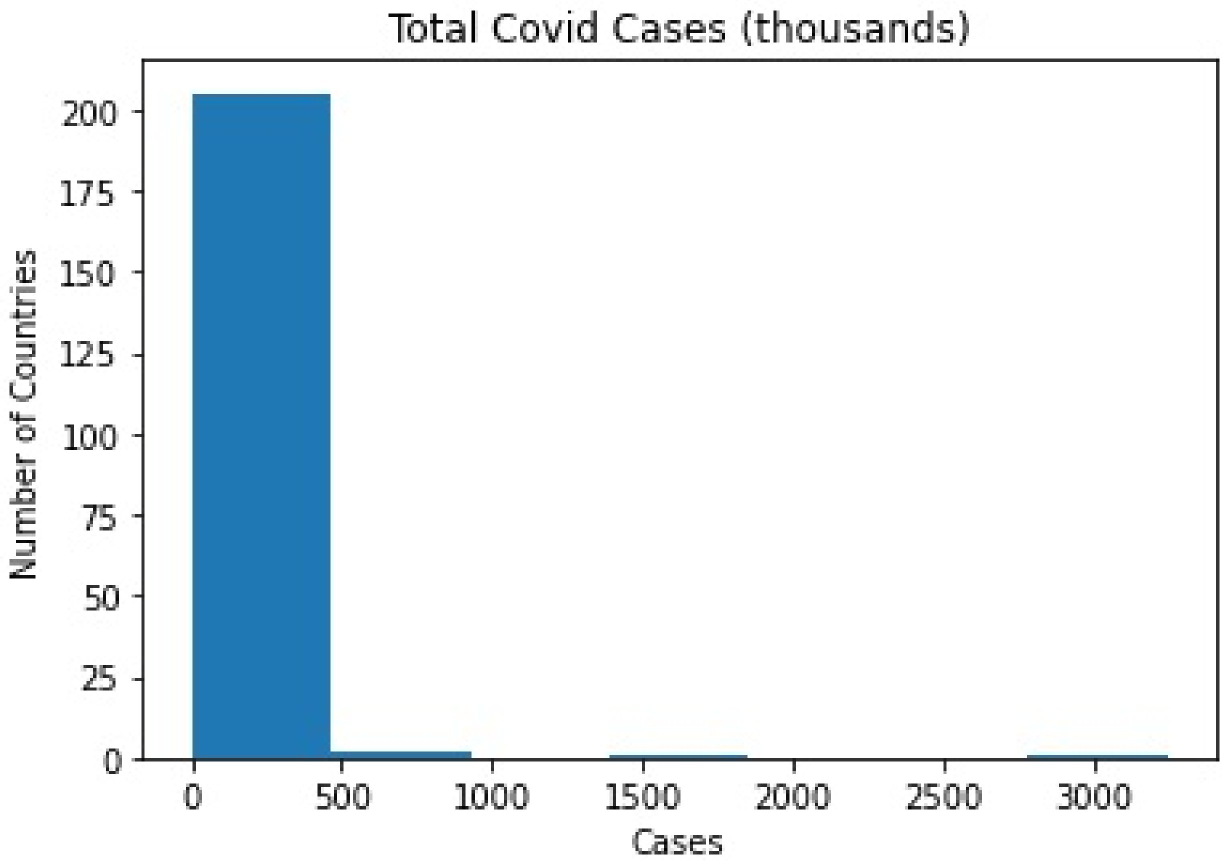


Figure 4.3: Histogram of total Covid cases

1. Perform a log transformation of the Covid data. Show a histogram of the log transformation of total cases:

```
covidlogs = covidtotalsonly.copy()
```

```
for col in covidtotalsonly.columns[1:]:
...   covidlogs[col] = np.log1p(covidlogs[col])
plt.hist(covidlogs['total_cases'], bins=7)
plt.title("Total Covid Cases (log)")
plt.xlabel('Cases')
plt.ylabel("Number of Countries")
plt.show()
```

This code produces the following:

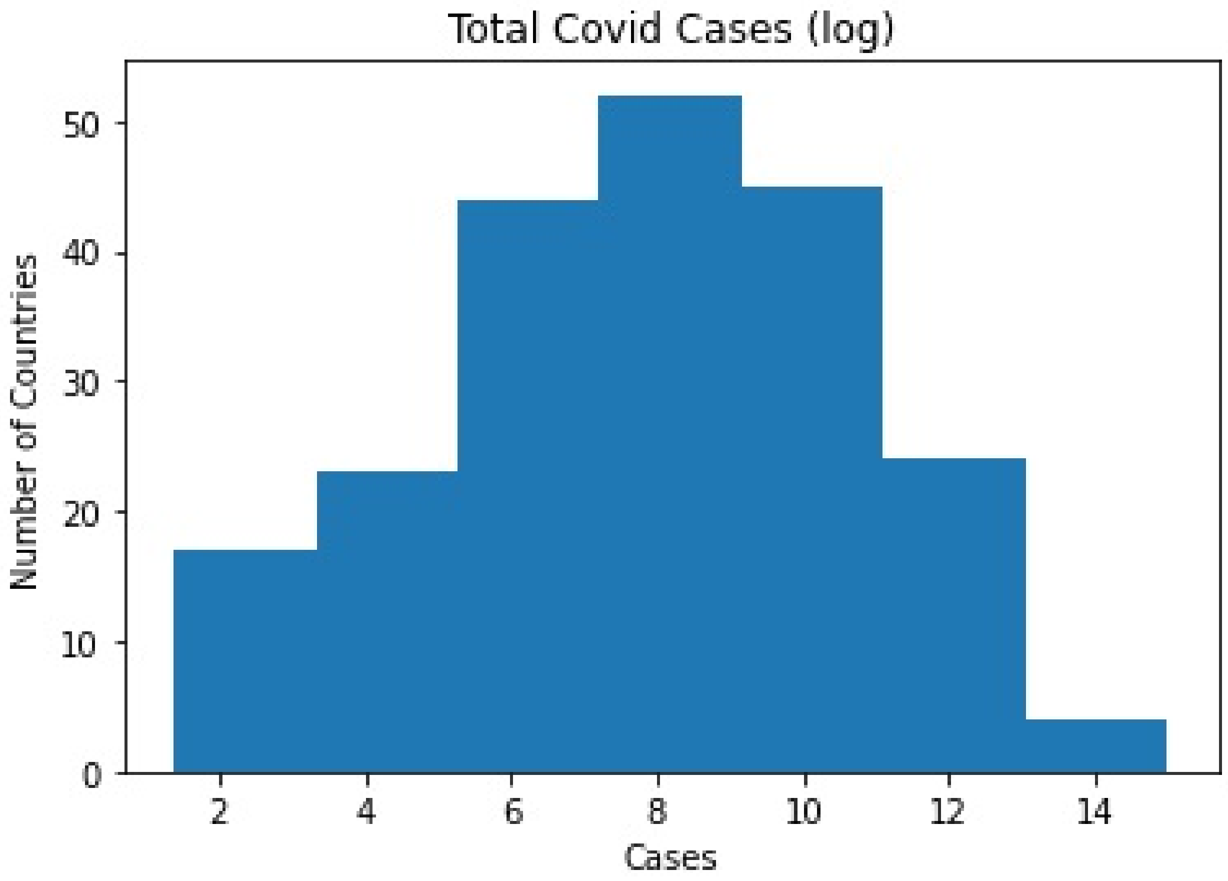


Figure 4.4: Histogram of total Covid cases with log transformation

The tools we used in the preceding steps tell us a fair bit about how Covid cases and deaths are distributed, and about where outliers are located.

How it works...

The percentile data shown in *step 3* reflect the skewness of the cases and deaths data. If, for example, we look at the range of values between the 20th and 30th percentiles, and compare it with the range from the 70th to the 80th percentiles, we see that the range is much greater in the higher percentiles for each variable. This is confirmed by the very high values for skewness and kurtosis, compared with normal distribution values of `0` and `3`, respectively. We run formal tests of normality in *step 4*, which indicate that the distributions of the Covid variables are not normal at high levels of significance.This is consistent with the `qqplots` we run in *step 5*. The distributions of both total cases and total cases per million differ significantly from normal, as represented by the straight line. Many cases hover around zero, and there is a dramatic increase in slope at the right tail.We identify outliers in *steps 6 and 7*. Using 1.5 times the interquartile range to determine outliers is a reasonable rule of thumb. I like to output those values to an Excel file, along with associated data, to see what patterns I can detect in the data. This often leads to more questions, of course. We will try to answer some of them in the next recipe, but one question we can consider now is what accounts for the countries with high cases per million, displayed in *step 8*. Some of the countries with extreme values are very small, in terms of land mass, so perhaps population density matters. But half of the countries on this list are near or below the 75th percentile in population density. On the other hand, most countries on this list are above the 75th percentile in GDP per capita. It is worth exploring these bivariate relationships further, which we do in subsequent recipes.Our identification of outliers in *step 7* assumes a normal distribution, an assumption that we have shown to be unwarranted. Looking again at the distribution in *step 9*, it seems much more like a log-normal distribution, with values clustered around `0` and a right skew. We transform the data in *step 10* and plot the results of the transformation.

There's more...

We could have also used standard deviation, rather than interquartile ranges, to identify outliers in *steps 6 and 7*.I should add here that outliers are not necessarily data collection or measurement errors, and we may or may not need to make adjustments to the data. However, extreme values can have a meaningful and persistent impact on our analysis, particularly with small datasets like this one.The overall impression we should have of the Covid case data is that it is relatively clean; that is, there are not many invalid values, narrowly defined. Looking at each variable independently of how it moves with other variables does not identify much that screams out as a clear data error. However, the distribution of the variables is quite problematic statistically. Building statistical models dependent on these variables will be complicated, as we might have to rule out parametric tests.It is also worth remembering that our sense of what constitutes an outlier is shaped by our assumption of a normal distribution. If, instead, we allow our expectations to be guided by the actual distribution of the data, we have a different understanding of extreme values. If our data reflects a social, or biological, or physical process that is inherently not normally distributed (uniform, logarithmic, exponential, weibull, Poisson, and so on), our sense of what constitutes an outlier should adjust accordingly.

See also

Box plots might have also been illuminating here. We do a few box plots on this data in *Chapter 5*, *Using Visualizations for Exploratory Data Analysis*.We explore bivariate relationships in this same dataset in the next recipe for any insights they might provide about outliers and unexpected values. In subsequent chapters, we consider strategies for imputing values for missing data and for making adjustments to extreme values.

Identifying outliers and unexpected values in bivariate relationships

A value might be unexpected, even if it is not an extreme value, when it does not deviate significantly from the distribution mean. Some values for a variable are unexpected when a second variable has certain values. This is easy to illustrate when one variable is categorical and the other is continuous.The following diagram illustrates the number of bird sightings per day over a several year period, but shows different distributions for each of the two sites. One site has a mean sightings per day of 33, and the other 52. (This is fictional data.) The overall mean (not shown) is 42. What should we make of a value of 58 for daily sightings? Is that an outlier? That clearly depends on which of the two sites was being observed. If there were 58 sightings on a day at site A, 58 would be an unusually high number. Not so for site B, where 58 sightings would not be very different from the mean for that site:

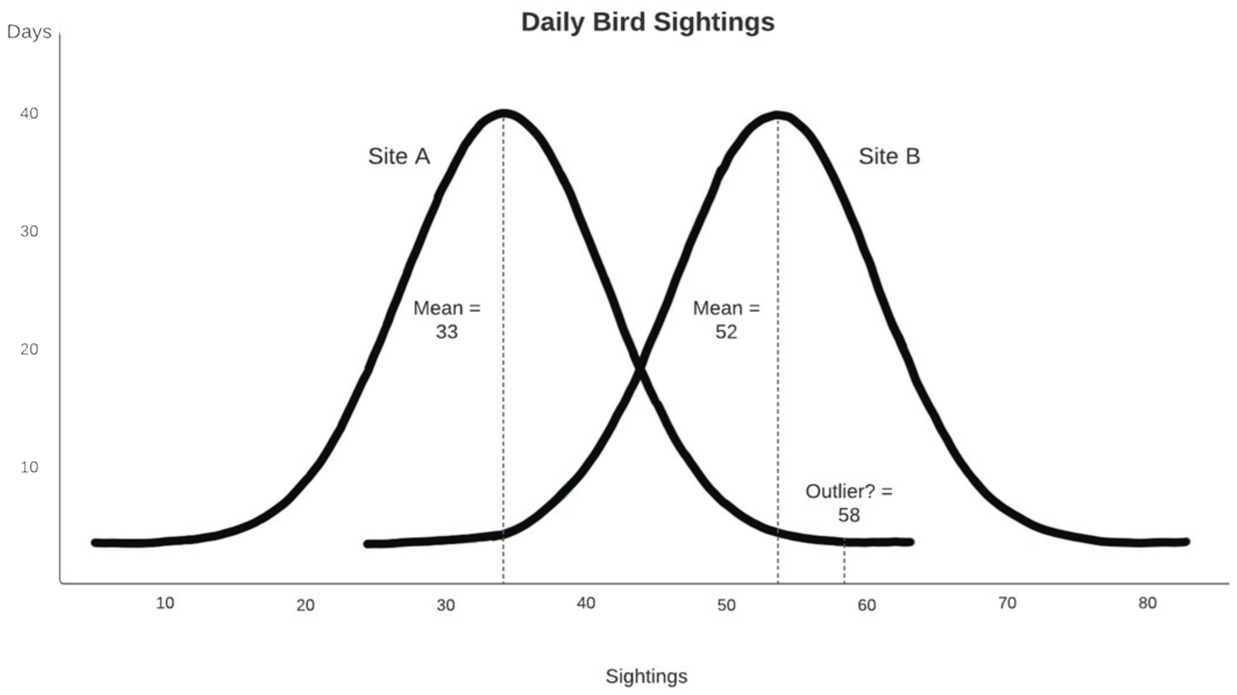


Figure 4.5: Daily bird sightings by site

This hints at a useful rule of thumb: whenever a variable of interest is significantly correlated with another variable, we should take that relationship into account when trying to identify outliers (or any statistical analysis with that variable actually). It is helpful to state this a little more precisely, and extend it to cases where both variables are continuous. If we assume a linear relationship between variable x and variable y, we can describe that relationship with the familiar $y = mx + b$ equation, where m is the slope and b is the y-intercept. We can then expect for y to increase by m for every 1 unit increase in x. Unexpected values are those that deviate substantially from this relationship, where the value of y is much higher or lower than what would be predicted given the value of x. This can be extended to multiple x, or predictor, variables.In this recipe, we demonstrate how to identify outliers and unexpected values by examining the relationship of a variable to one other variable. In subsequent recipes in this chapter, we use multivariate techniques to make additional improvements in our outlier detection.

Getting ready

We use the `matplotlib` and `seaborn` libraries in this recipe. You can install them with `pip` by entering `pip install matplotlib` and `pip install seaborn` with a terminal client or `powershell` (in Windows).

How to do it…

We examine the relationship between total cases and total deaths. We take a closer look at those countries where deaths are higher or lower than expected given the number of cases:

1. Load `pandas`, `matplotlib`, `seaborn`, and the Covid cumulative data:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
covidtotals = pd.read_csv("data/covidtotals720.csv")
covidtotals.set_index("iso_code", inplace=True)
totvars = ['location','total_cases',
...   'total_deaths','total_cases_pm',
...   'total_deaths_pm']
demovars = ['population','pop_density',
...   'median_age','gdp_per_capita',
...    'hosp_beds']
```

1. Generate a correlation matrix for the cumulative and demographic columns.

Unsurprisingly, there is a very high correlation (0.93) between total cases and total deaths, and a smaller (0.59) but still substantial one between total cases per million and total deaths per million. There is a strong (0.65) relationship between GDP per capita and cases per million:

```
covidtotals.corr(method="pearson")

                 total_cases  total_deaths  \
total_cases    1.00    0.93
total_deaths    0.93    1.00
total_cases_pm    0.23    0.20
total_deaths_pm    0.26    0.41
population    0.34    0.28
```

```
pop_density    -0.03    -0.03
median_age    0.12    0.17
gdp_per_capita    0.13    0.16
hosp_beds    -0.01    -0.01
                 total_cases_pm  ...  median_age  \
total_cases    0.23  ...    0.12
total_deaths    0.20  ...    0.17
total_cases_pm    1.00  ...    0.22
total_deaths_pm    0.49  ...    0.38
population    -0.04  ...    0.03
pop_density    0.08  ...    0.14
median_age    0.22  ...    1.00
gdp_per_capita    0.58  ...    0.64
hosp_beds    0.02  ..    0.66
                 gdp_per_capita  hosp_beds
total_cases    0.13    -0.01
total_deaths    0.16    -0.01
total_cases_pm    0.58    0.02
total_deaths_pm    0.37    0.09
population    -0.06    -0.04
pop_density    0.30    0.31
median_age    0.64    0.66
gdp_per_capita    1.00    0.30
hosp_beds    0.30    1.00
[9 rows x 9 columns]
```

1. Check to see whether some countries have unexpectedly high or low total deaths, given total cases.

First create a data frame with only the cases and deaths columns. Use `qcut` to create a column that breaks the data into quantiles. Show a crosstab of total cases quantiles by total deaths quantiles:

```
covidtotalsonly = covidtotals.loc[:, totvars]
covidtotalsonly['total_cases_q'] = pd.\
...   qcut(covidtotalsonly['total_cases'],
...   labels=['very low','low','medium',
...   'high','very high'], q=5, precision=0)
covidtotalsonly['total_deaths_q'] = pd.\
...   qcut(covidtotalsonly['total_deaths'],
...   labels=['very low','low','medium',
...   'high','very high'], q=5, precision=0)
pd.crosstab(covidtotalsonly.total_cases_q,
...   covidtotalsonly.total_deaths_q)
```

```
total_deaths_q  very low  low  medium  high  very high
```

```
total_cases_q
very low    35    7    0    0    0
low    7    25    10    0    0
medium    0    8    24    9    0
high    0    1    7    26    8
very high    0    1    0    7    34
```

1. Take a look at countries that do not fit along the diagonal.

There is one country with very high total cases but low total deaths. Also, look at countries with low cases but high deaths. (Since the `covidtotals` and `covidtotalsonly` DataFrames have the same index, we can use the Boolean series created from the latter to return selected rows from the former.):

```
covidtotals.loc[(covidtotalsonly. \
...   total_cases_q=="very high") & \
...   (covidtotalsonly.total_deaths_q=="low")].T

iso_code    SGP
lastdate    2020-07-12
location    Singapore
total_cases    45,783.00
total_deaths    26.00
total_cases_pm    7,825.69
total_deaths_pm    4.44
population    5,850,343.00
pop_density    7,915.73
median_age    42.40
gdp_per_capita    85,535.38
hosp_beds    2.40
region    East Asia
```

1. Do a scatter plot of total cases by total deaths.

Use Seaborn's `regplot` method to generate a linear regression line in addition to the scatter plot:

```
ax = sns.regplot(x=covidtotals.total_cases/1000, y=covidtotals.t
otal_deaths)
ax.set(xlabel="Cases (thousands)", ylabel="Deaths", title="Total
 Covid Cases and Deaths by Country")
plt.show()
```

This produces the following scatter plot:

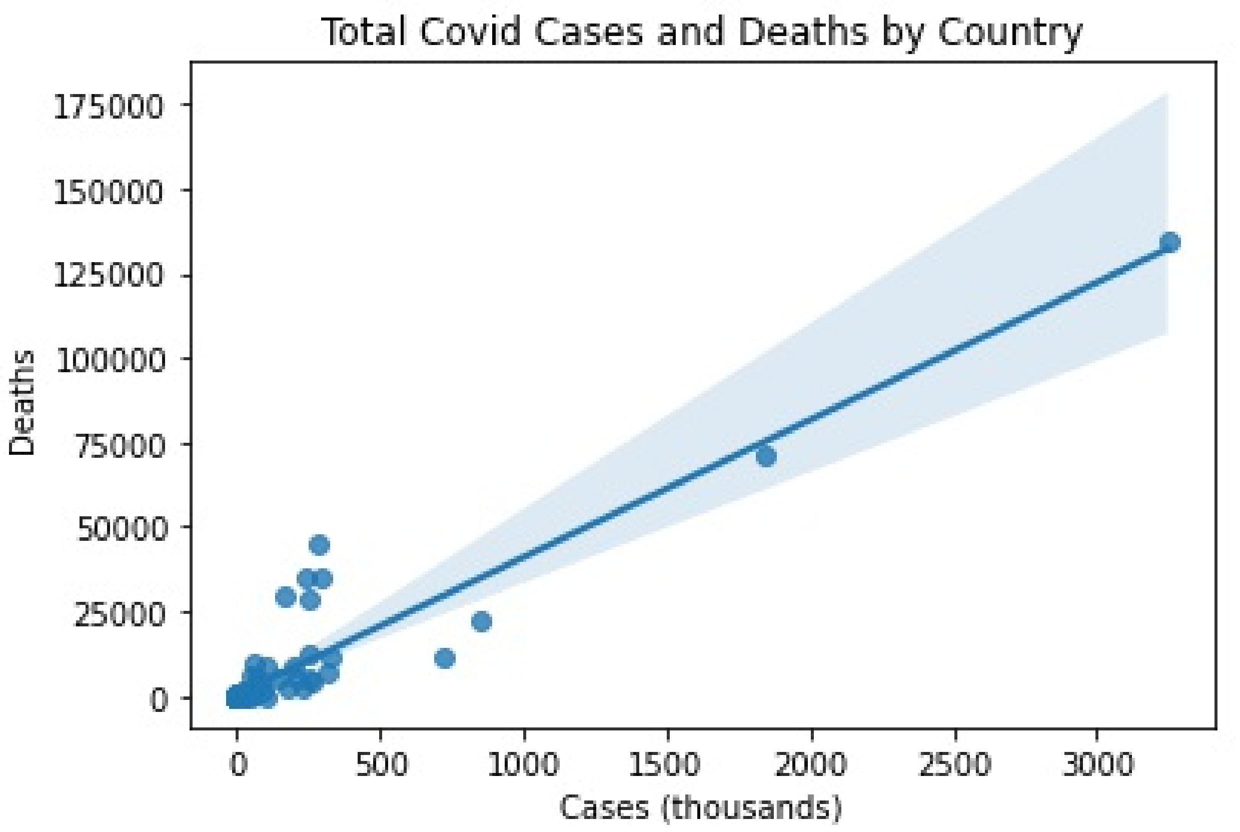


Figure 4.6: Scatter plot of total cases and deaths with a linear regression line

1. Examine unexpected values above the regression line.

It is good to take a closer look at countries with cases and deaths coordinates that are noticeably above or below the regression line through the data. There are five countries with fewer than 400,000 cases and more than 25,000 deaths:

```
covidtotals.loc[(covidtotals.total_cases<400000) \
  & (covidtotals.total_deaths>25000)].T

iso_code    FRA    ITA    MEX  \
lastdate    2020-07-12    2020-07-12    2020-07-12
location    France    Italy    Mexico
total_cases    170,752    242,827    295,268
total_deaths    30,004    34,945    34,730
total_cases_pm    2,616    4,016    2,290
total_deaths_pm    460    578    269
```

```
population    65,273,512    60,461,828    128,932,753
pop_density    123    206    66
median_age    42    48    29
gdp_per_capita    38,606    35,220    17,336
hosp_beds    6    3    1
region           Western Europe  Western Europe  North America

iso_code    ESP    GBR
lastdate    2020-07-11    2020-07-12
location    Spain    United Kingdom
total_cases    253,908    288,953
total_deaths    28,403    44,798
total_cases_pm    5,431    4,256
total_deaths_pm    607    660
population    46,754,783    67,886,004
pop_density    93    273
median_age    46    41
gdp_per_capita    34,272    39,753
hosp_beds    3    3
region           Western Europe  Western Europe
```

1. Examine unexpected values below the regression line.

There are two countries with more than 700,000 cases but fewer than 25,000 deaths:

```
covidtotals.loc[(covidtotals.total_cases>700000) \
  & (covidtotals.total_deaths<25000)].T

iso_code    IND    RUS
lastdate    2020-07-12    2020-07-12
location    India    Russia
total_cases    849,553    720,547
total_deaths    22,674    11,205
total_cases_pm    616    4,937
total_deaths_pm    16    77
population    1,380,004,385    145,934,460
pop_density    450    9
median_age    28    40
gdp_per_capita    6,427    24,766
hosp_beds    1    8
region    South Asia    Eastern Europe
```

1. Do a scatter plot of total cases per million by total deaths per million:

```
ax = sns.regplot(x="total_cases_pm", y="total_deaths_pm", data=c
ovidtotals)
ax.set(xlabel="Cases Per Million", ylabel="Deaths Per Million",
title="Total Covid Cases per Million and Deaths per Million by C
ountry")
plt.show()
```

This produces the following scatter plot:

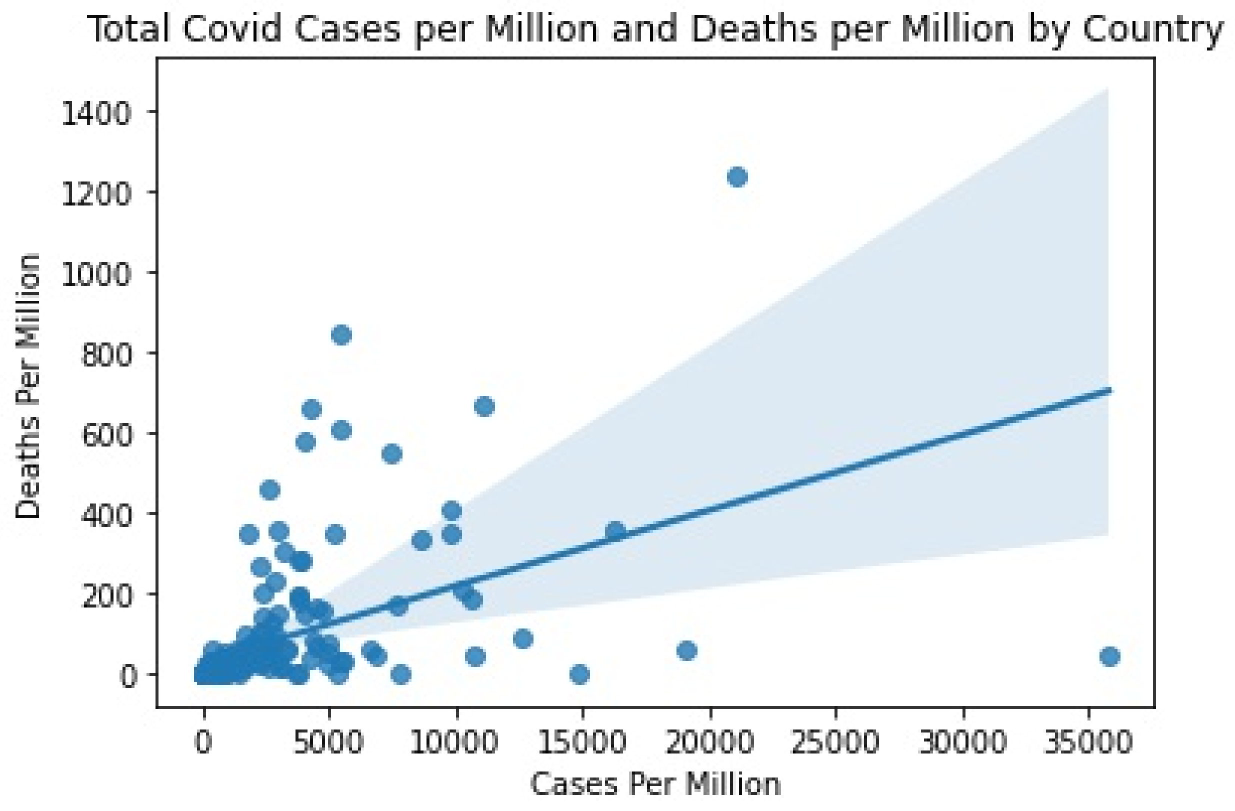


Figure 4.7: Scatter plot of cases and deaths per million with a linear regression line

1. Examine deaths per million above and below the regression line:

```
covidtotals.loc[(covidtotals.total_cases_pm<7500) \
  & (covidtotals.total_deaths_pm>600),\
  ['location','total_cases_pm','total_deaths_pm']]

                location  total_cases_pm  total_deaths_pm
iso_code
BEL    Belgium    5,402    844
ESP    Spain    5,431    607
GBR    United Kingdom    4,256    660
```

```
covidtotals.loc[(covidtotals.total_cases_pm>15000) \
  & (covidtotals.total_deaths_pm<=100), \
  ['location','total_cases_pm','total_deaths_pm']]
```

```
         location  total_cases_pm  total_deaths_pm
iso_code
BHR    Bahrain    19,082    61
QAT    Qatar    35,795    51
```

The preceding steps examined the relationship between variables in order to identify outliers.

How it works...

A number of questions are raised by looking at the bivariate relationships that did not surface in our univariate exploration in the previous recipe. There is confirmation of anticipated relationships, such as with total cases and total deaths, but this makes deviations from this all the more curious. There are possible substantive explanations for unusually high death rates, given a certain number of cases, but measurement error or poor reporting of cases cannot be ruled out either.*Step 2* shows a high correlation (0.93) between total cases and total deaths, but there is variation even there. We divide the cases and deaths into quantiles in *step 3* and then do a crosstab of the quantile values. Most countries are along the diagonal or close to it. However, one country has a very high number of cases but low deaths, Singapore. This is also a reminder that Singapore had a very high total cases per million, well into the 90th percentile. It is reasonable to wonder if there are potential reporting issues.One country, Yemen, had a low number of cases but a high number of deaths. This could perhaps be seen as consistent with the very low number of hospital beds per 100,000 people in Yemen. But it could also mean that coronavirus cases have been under-reported.We do a scatter plot in *step 5* of total cases and deaths. The strong upward sloping relationship between the two is confirmed, but there are a number of countries whose deaths are above the regression line. We can see that five countries (France, Italy, Mexico, Spain, and Great Britain) have higher deaths than would be predicted by the number of cases. Two countries, Russia and India, have a much lower number of deaths. It is at least worth wondering about whether this is a reporting problem, or reflects differences in how countries define a Covid death.Not surprisingly, there is even more scatter around the regression line in the scatter plot of cases per million and deaths per million. Countries such as Belgium, Spain, and the United Kingdom have much higher deaths per million than the number of cases per million would suggest. Bahrain and Qatar have significantly lower rates.

There's more...

We are beginning to get a good sense of what our data looks like, but the data in this form does not enable us to examine how the univariate distributions and bivariate relationships might change over time. For example, one reason why countries might have more deaths per million than the number of cases per million would indicate could be that more time has passed since the first confirmed cases. We are not able to explore that in the cumulative data. We need the daily data for that, which we look at in subsequent chapters.This recipe, and the previous one, show how much data cleaning can bleed into exploratory data analysis, even when you are first starting to get a sense of your data. I would definitely draw a distinction between data exploration and what we are doing here. We are trying to get a sense of how the data hangs together, why certain variables take on certain values in certain situations and not others. We want to get to the point where there are not huge surprises when we begin to do the analysis.I find it helpful to do small things to formalize this process. I use different naming conventions for files that are not quite ready for analysis. If nothing else, this helps remind me that any numbers produced at this point are far from ready for distribution.

See also

We still have not done much to examine possible data issues that only become apparent when examining subsets of data; for example, positive wage income values for people who say they are not working (both variables are on the National Longitudinal Survey). We do that in the next recipe.We do much more with Matplotlib and Seaborn in *Chapter 5*, *Using Visualizations for Exploratory Data Analysis*.

Using subsetting to examine logical inconsistencies in variable relationships

At a certain point, data issues come down to deductive logic problems, such as variable *x* has to be greater than some quantity *a* when variable *y* is less than some quantity *b*. Once we are through some initial data cleaning, it is important to check for logical inconsistencies. `pandas` makes this kind of error checking relatively straightforward with subsetting tools such as `loc` and Boolean indexing. This can be combined with summary methods on series and data frames to allow us to easily compare values for a particular row to values for the whole dataset or some subset of rows. We can also easily aggregate over columns. Just about any question we might have about the logical relationships between variables can be answered with these tools. We work through some examples in this recipe.

Getting ready

We will work with the National Longitudinal Survey of Youth (NLS), mainly with data on employment and education. We use `apply` and `lambda` functions several times in this recipe, but go into more detail on their use in *Chapter 7, Fixing Messy Data when Aggregating*. It is not necessary to review *chapter 7* to follow along, however, even if you have no experience with those tools.

Data note

> The NLS, administered by the United States Bureau of Labor Statistics, is a longitudinal survey of individuals who were in high school in 1997 when the survey started. Participants were surveyed each year through 2017.

How to do it...

We run a number of logical checks on the NLS data, such as individuals with post-graduate enrollment but no undergraduate enrollment, or having wage income but no weeks worked. We also check for large changes in key values for a given individual from one period to the next:

1. Import `pandas` and then load the NLS data:

```
import pandas as pd
```

```
nls97 = pd.read_csv("data/nls97.csv")
nls97.set_index("personid", inplace=True)
```

1. Look at some of the employment and education data.

The dataset has weeks worked each year from 2000 through 2017, and college enrollment status each month from February 1997 through October 2017. We use the ability of the `loc` accessor to choose all columns from the column indicated on the left of the colon through the column indicated on the right; for example, `nls97.loc[:, "colenroct09":"colenrfeb14"]`:

```
nls97[['wageincome','highestgradecompleted','highestdegree']].he
ad(3).T

personid    100061    100139    100284
wageincome    12,500    120,000    58,000
highestgradecompleted    13    12    7
highestdegree     2. High School    2. High School    0. None

nls97.loc[:, "weeksworked12":"weeksworked17"].head(3).T

personid    100061    100139    100284
weeksworked12    40    52    0
weeksworked13    52    52    nan
weeksworked14    52    52    11
weeksworked15    52    52    52
weeksworked16    48    53    47
weeksworked17    48    52    0

nls97.loc[:, "colenroct09":"colenrfeb14"].head(2).T

personid    100061    100139
colenroct09    1. Not enrolled    1. Not enrolled
colenrfeb10    1. Not enrolled    1. Not enrolled
colenroct10    1. Not enrolled    1. Not enrolled
colenrfeb11    1. Not enrolled    1. Not enrolled
colenroct11    3. 4-year college    1. Not enrolled
colenrfeb12    3. 4-year college    1. Not enrolled
colenroct12    3. 4-year college    1. Not enrolled
colenrfeb13    1. Not enrolled    1. Not enrolled
colenroct13    1. Not enrolled    1. Not enrolled
colenrfeb14    1. Not enrolled    1. Not enrolled
```

1. Show individuals with wage income but no weeks worked.

The wage income variable reflects wage income for 2016:

```
nls97.loc[(nls97.weeksworked16==0) & nls97.wageincome>0, ['weeks
worked16','wageincome']]

          weeksworked16  wageincome
personid
102625    0    1,200
109403    0    5,000
118704    0    25,000
130701    0    12,000
131151    0    65,000
...                     ...    ...
957344    0    90,000
966697    0    65,000
969334    0    5,000
991756    0    9,000
992369    0    35,000
[145 rows x 2 columns]
```

1. Check for whether an individual was ever enrolled in a 4-year college.

Chain several methods. First, create a data frame with columns that start with `colenr` (`nls97.filter(like="colenr")`). These are the college enrollment columns for October and February of each year. Then, use `apply` to run a `lambda` function that examines the first character of each `colenr` column (`apply(lambda x: x.str[0:1]=='3')`). This returns a value of `True` or `False` for all of the college enrollment columns; `True` if the first value of the string is `3`, meaning enrollment at a 4-year college. Finally, use the `any` function to test whether any of the values returned from the previous step has a value of `True` (`any(axis=1)`). This will identify whether the individual was enrolled in a 4-year college between February 1997 and October 2017. The first statement here shows the results of the first two steps for explanatory purposes only. Only the second statement needs to be run to get the desired results: whether the individual was enrolled at a 4-year college at some point:

```
nls97.filter(like="colenr").apply(lambda x: x.str[0:1]=='3').hea
d(2).T

personid     100061  100139
...
colenroct09    False    False
```

```
colenrfeb10    False    False
colenroct10    False    False
colenrfeb11    False    False
colenroct11    True    False
colenrfeb12    True    False
colenroct12    True    False
colenrfeb13    False    False
colenroct13    False    False
colenrfeb14    False    False
```

```
nls97.filter(like="colenr"). \
      apply(lambda x: x.str[0:1]=='3').\
...   any(axis=1).head(2)
```

```
personid
100061    True
100139    False
dtype: bool
```

1. Show individuals with post-graduate enrollment but no bachelor's enrollment.

We can use what we tested in *step 4* to do some checking. We want individuals who have a `4` (graduate enrollment) as the first character for `colenr` any month, but who never had a `3` (bachelor enrollment). Note the "~" before the second half of the test, for negation. There are 22 individuals who fall into this category:

```
nobach = nls97.loc[nls97.filter(like="colenr").\
...   apply(lambda x: x.str[0:1]=='4').\
...   any(axis=1) & ~nls97.filter(like="colenr").\
...   apply(lambda x: x.str[0:1]=='3').\
...   any(axis=1), "colenrfeb97":"colenroct17"]
len(nobach)
```

```
22
```

```
nobach.head(2).T
```

```
personid                  153051              154535
...
colenroct08    1. Not enrolled    1. Not enrolled
colenrfeb09    1. Not enrolled    1. Not enrolled
colenroct09    1. Not enrolled    1. Not enrolled
colenrfeb10    1. Not enrolled    1. Not enrolled
colenroct10    1. Not enrolled    4. Graduate program
```

```
colenrfeb11    1. Not enrolled    4. Graduate program
colenroct11    1. Not enrolled    4. Graduate program
colenrfeb12    1. Not enrolled    4. Graduate program
colenroct12    1. Not enrolled    4. Graduate program
colenrfeb13    4. Graduate program    4. Graduate program
colenroct13    1. Not enrolled    4. Graduate program
colenrfeb14    4. Graduate program    4. Graduate program
colenroct14    4. Graduate program    4. Graduate program
colenrfeb15    1. Not enrolled    4. Graduate program
colenroct16    1. Not enrolled    4. Graduate program
colenrfeb17    1. Not enrolled    4. Graduate program
colenroct15    1. Not enrolled    4. Graduate program
colenrfeb16    1. Not enrolled    4. Graduate program
colenroct17    1. Not enrolled    4. Graduate program
```

1. Show individuals with bachelor's degrees or more, but no 4-year college enrollment.

Use `isin` to compare the first character in `highestdegree` with all of the values in a list (`nls97.highestdegree.str[0:1].isin(['4','5','6','7'])`):

```
nls97.highestdegree.value_counts().sort_index()

0. None        953
1. GED    1146
2. High School    3667
3. Associates    737
4. Bachelors    1673
5. Masters    603
6. PhD    54
7. Professional    120
Name: highestdegree, dtype: int64

no4yearenrollment = \
...   nls97.loc[nls97.highestdegree.str[0:1].\
...   isin(['4','5','6','7']) & \
...   ~nls97.filter(like="colenr").\
...   apply(lambda x: x.str[0:1]=='3').\
...   any(axis=1), "colenrfeb97":"colenroct17"]
len(no4yearenrollment)

39

no4yearenrollment.head(3).T
```

```
personid    113486    118749    124616

no4yearenrollment.head(2).T

personid    113486    118749
colenroct01    2. 2-year college    1. Not enrolled
colenrfeb02    2. 2-year college    1. Not enrolled
colenroct02    2. 2-year college    1. Not enrolled
colenrfeb03    2. 2-year college    1. Not enrolled
colenroct03    2. 2-year college    1. Not enrolled
colenrfeb04    2. 2-year college    1. Not enrolled
colenroct04    1. Not enrolled    1. Not enrolled
colenrfeb05    1. Not enrolled    1. Not enrolled
colenroct05    1. Not enrolled    1. Not enrolled
colenrfeb06    1. Not enrolled    1. Not enrolled
colenroct06    1. Not enrolled    1. Not enrolled
colenrfeb07    1. Not enrolled    2. 2-year college
colenroct07    1. Not enrolled    2. 2-year college
colenrfeb08    1. Not enrolled    1. Not enrolled
```

1. Show individuals with a high wage income.

Define high wages as 3 standard deviations above the mean. It looks as though wage income values have been truncated at $235,884:

```
highwages = \
 nls97.loc[nls97.wageincome > nls97.wageincome.mean()+ \
 (nls97.wageincome.std()*3),['wageincome']]
highwages

          wageincome
personid
131858    235,884
133619    235,884
151863    235,884
164058    235,884
164897    235,884
...                ...
964406    235,884
966024    235,884
976141    235,884
983819    235,884
989896    235,884
[121 rows x 1 columns]
```

1. Show individuals with large changes in weeks worked for the most

recent year.

Calculate the average value for weeks worked between 2012 and 2016 for each person (`nls97.loc[:, "weeksworked12":"weeksworked16"].mean(axis=1)`). We indicate `axis=1` to calculate the mean across columns for each individual, rather than over individuals. We then check to see whether the mean is either less than 50% of the weeks worked in 2017 value or more than twice as much. We also indicate that we are not interested in rows that satisfy those criteria by being `null` for weeks worked in 2017. There are 1,160 individuals with sharp changes in weeks worked in 2017:

```
workchanges = nls97.loc[~nls97.loc[:,
...   "weeksworked12":"weeksworked16"].mean(axis=1).\
...   between(nls97.weeksworked17*0.5,\
...   nls97.weeksworked17*2) \
...   & ~nls97.weeksworked17.isnull(),
...   "weeksworked12":"weeksworked17"]
len(workchanges)
```

```
1160
```

```
workchanges.head(6).T
```

```
personid    100284    101526    ...    102228    102454
weeksworked12    0    0    ...    52    52
weeksworked13    nan    0    ...    52    52
weeksworked14    11    0    ...    17    7
weeksworked15    52    0    ...    0    0
weeksworked16    47    0    ...    0    0
weeksworked17    0    45    ...    0    0
```

1. Show inconsistencies in the highest grade completed and the highest degree.

Use the `crosstab` function to show `highestgradecompleted` by `highestdegree` for people with `highestgradecompleted` less than 12. A good number of these individuals indicate that they have completed high school, which is unusual in the United States if the highest grade completed is less than 12:

```
ltgrade12 = nls97.loc[nls97.highestgradecompleted<12, ['highestg
radecompleted','highestdegree']]
```

```
pd.crosstab(ltgrade12.highestgradecompleted, ltgrade12.highestde
gree)
```

```
highestdegree          0. None  1. GED  2. High School
highestgradecompleted
5    0    0    1
6    11    5    0
7    24    6    1
8    113    78    7
9    112    169    8
10    111    204    13
11    120    200    41
```

These steps reveal a number of logical inconsistences in the NLS data.

How it works...

The syntax required to do the kind of subsetting that we have done in this recipe may seem a little complicated if you are seeing it for the first time. You do get used to it, however, and it allows for quickly running any query against the data that you might imagine.Some of the inconsistencies or unexpected values suggest either respondent or entry error, so warrant further investigation. It is hard to explain positive values for wage income when weeks worked is `0`. Other unexpected values might not be data problems at all, but suggest that we should be careful about how we use that data. For example, we might not want to use the weeks worked in 2017 by itself. Instead, we might consider using three-year averages in many analyses.

See also

The *Selecting and organizing columns* and *Selecting rows* recipes in the *Taking the Measure of Your Data* chapter demonstrate some of the techniques for subsetting data used here. We examine `apply` functions in more detail in the *Fixing Messy Data when Aggregating* chapter.

Using linear regression to identify data points with significant influence

The remaining recipes in this chapter use statistical modeling to identify outliers. The advantage of these techniques is that they are less dependent on the distribution of the variable of concern, and take more into account than can be revealed in either univariate or bivariate analyses. This allows us to identify outliers that are not otherwise apparent. On the other hand, by taking more factors into account, multivariate techniques may provide evidence that a previously suspect value is actually within an expected range, and provides meaningful information.In this recipe, we use linear regression to identify observations (rows) that have an out-sized influence on models of a target or dependent variable. This can indicate that one or more values for a few observations are so extreme that they compromise model fit for all of the other observations.

Getting ready

The code in this recipe requires the `matplotlib` and `statsmodels` libraries. You can install Matplotlib and Statsmodels by entering `pip install matplotlib` and `pip install statsmodels` in a terminal window or `powershell` (in Windows).We will be working with data on total Covid cases and deaths per country.

How to do it…

We will use the statsmodels `OLS` method to fit a linear regression model of total cases per million of the population. We then identify those countries that have the greatest influence on that model:

1. Import pandas, Matplotlib, and Statsmodels, and load the Covid case data:

```
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm
covidtotals = pd.read_csv("data/covidtotals720.csv")
covidtotals.set_index("iso_code", inplace=True)
```

1. Create an analysis file and generate descriptive statistics.

Get just the columns required for analysis. Drop any row with missing data for the analysis columns:

```
xvars = ['pop_density','median_age','gdp_per_capita']
covidanalysis = covidtotals.loc[:,['total_cases_pm'] + xvars].dr
opna()
covidanalysis.describe()
```

```
total_cases_pm  pop_density  median_age gdp_per_capita
count    174    174    174    174
mean    2,200    208    30    18,795
std    3,964    642    9    19,527
min    1    2    15    661
25%    199    36    22    4,454
50%    768    82    30    12,623
75%    2,666    207    39    27,114
max    35,795    7,916    48    116,936
```

1. Fit a linear regression model.

There are good conceptual reasons to believe that population density, median age, and GDP per capita may be predictors of total cases per million. We use all three variables in our model:

```
def getlm(df):
...   Y = df.total_cases_pm
...   X = df[['pop_density',
        'median_age','gdp_per_capita']]
...   X = sm.add_constant(X)
...   return sm.OLS(Y, X).fit()
...
lm = getlm(covidanalysis)
lm.summary()
```

```
     coef    std err    t    P>|t|
-----------------------------------------------------------------
---------------
const    2591.8003    929.551    2.788    0.006
pop_density    -0.0364    0.394    -0.093    0.926
median_age    -104.0544    34.837    -2.987    0.003
gdp_per_capita    0.1482    0.017    8.869    0.000
```

1. Identify those countries with an out-sized influence on the model.

Cook’s Distance values of greater than 0.5 should be scrutinized closely:

```
influence = lm.get_influence().summary_frame()
influence.loc[influence.cooks_d>0.5, ['cooks_d']]

          cooks_d
iso_code
QAT    4.30
SGP    6.11

covidanalysis.loc[influence.cooks_d>0.5]
total_cases_pm  pop_density  median_age gdp_per_capita
iso_code

QAT    35,795    227    32    116,936
SGP    7,826    7,916    42    85,535
```

1. Do an influence plot.

Countries with higher Cook's Distance values have larger circles:

```
fig, ax = plt.subplots(figsize=(10,6))
sm.graphics.influence_plot(lm, ax = ax, criterion="cooks")
plt.show()
```

This produces the following plot:

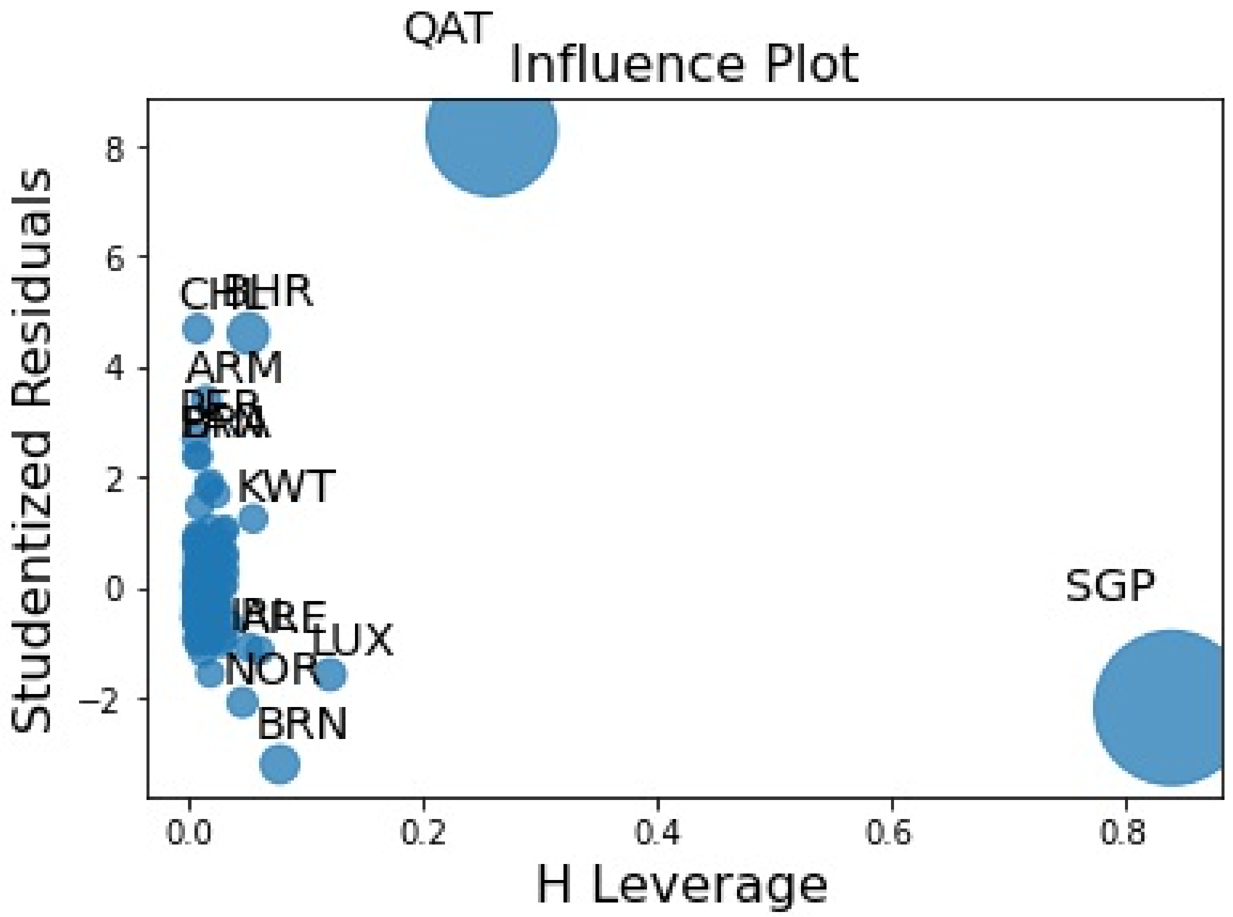


Figure 4.8: Influence plot, including countries with the highest Cook's Distance

1. Run the model without the two outliers.

Removing these outliers, particularly Qatar, has a dramatic effect on the model. The estimates for `median_age` and for the constant are no longer significant:

```
covidanalysisminusoutliers = covidanalysis.loc[influence.cooks_d
<0.5]
lm = getlm(covidanalysisminusoutliers)
lm.summary()
```

```
      coef    std err    t    P>|t|
------------------------------------------------------------------
---------------
const    901.1855    803.745    1.121    0.264
```

```
pop_density    1.8371    0.793    2.317    0.022
median_age    -23.2250    31.229    -0.744    0.458
gdp_per_capita    0.0828    0.016    5.079    0.000
```

This gives us a sense of the countries that are most unlike the others in terms of the relationship between demographic variables and total cases per million in population.

How it works...

Cook's Distance is a measure of how much each observation influences the model. The large impact of the two outliers is confirmed in *step 6* when we rerun the model without them. The question for the analyst is whether outliers such as these add important information or distort the model and limit its applicability. The coefficient of -49 for median age in the first regression results indicates that every one-year increase in median age is associated with a 49 point reduction in cases per million people. But this seems largely due to the model trying to fit a quite extreme total cases per million value for Qatar. Without Qatar, the coefficient on age is no longer significant.The `P>|t|` value in the regression output tells us whether the coefficient is significantly different from `0`. In the first regression, the coefficients for `median_age` and `gdp_per_capita` are significant at the 99% level; that is, the `P>|t|` value is less than 0.01. Only `gdp_per_capita` is significant when the model is run without the 2 outliers, though `pop_density` approaches significance here.

There's more...

We run a linear regression model in this recipe, not so much because we are interested in the parameter estimates of the model, but because we want to determine whether there are observations with potential out-sized influence on any multivariate analysis we might conduct. That definitely seems to be true in this case.Often, it makes sense to remove the outliers, as we have done here, but that is not always true. When we have independent variables that do a good job of capturing what makes outliers different, then the parameter estimates for the other independent variables are less vulnerable to distortion. We also might consider transformations, such as the log transformation we did in a previous recipe, and the scaling we will do in the next two recipes. An appropriate transformation, given your data, can reduce the influence of outliers by limiting the size of residuals at the extremes.

Using K-nearest neighbor to find outliers

Unsupervised machine learning tools can help us identify observations that are unlike others when we have unlabeled data; that is, when there is no target or dependent variable. (In the previous recipe, we used total cases per million as the dependent variable.) Even when selecting targets and factors is relatively straightforward, it might be helpful to identify outliers without making any assumptions about relationships between variables. We can use K-nearest neighbor to find observations that are most unlike others, those where there is the greatest difference between their values and their nearest neighbors' values.

Getting ready

You will need PyOD (Python outlier detection) and scikit-learn to run the code in this recipe. You can install both by entering `pip install pyod` and `pip install sklearn` in the terminal or `powershell` (in Windows).

How to do it...

We will use k-nearest neighbor to identify countries whose attributes indicate that they are most anomalous:

1. Load `pandas`, `pyod`, and `scikit-learn`, along with the Covid case data:

```
import pandas as pd
from pyod.models.knn import KNN
from sklearn.preprocessing import StandardScaler
covidtotals = pd.read_csv("data/covidtotals.csv")
covidtotals.set_index("iso_code", inplace=True)
```

1. Create a standardized data frame of the analysis columns:

```
standardizer = StandardScaler()
analysisvars = ['location','total_cases_pm',
...   'total_deaths_pm',  'pop_density',
...   'median_age','gdp_per_capita']
covidanalysis = covidtotals.loc[:, analysisvars].dropna()
covidanalysisstand = standardizer.fit_transform(covidanalysis.il
oc[:, 1:])
```

1. Run the KNN model and generate anomaly scores.

We create an arbitrary number of outliers by setting the contamination parameter to 0.1:

```
clf_name = 'KNN'
clf = KNN(contamination=0.1)
clf.fit(covidanalysisstand)
KNN(algorithm='auto', contamination=0.1, leaf_size=30, method='l
argest',
  metric='minkowski', metric_params=None, n_jobs=1, n_neighbors=
5, p=2,
  radius=1.0)
y_pred = clf.labels_
y_scores = clf.decision_scores_
```

1. Show the predictions from the model.

Create a data frame from the `y_pred` and `y_scores` NumPy arrays. Set the index to the `covidanalysis` data frame index so that we can easily combine it with that data frame later. Notice that the decision scores for outliers are all higher than those for the inliers (outlier = 0):

```
pred = pd.DataFrame(zip(y_pred, y_scores),
...   columns=['outlier','scores'],
...   index=covidanalysis.index)
pred.sample(10, random_state=1)

          outlier  scores
iso_code
LTU    0    0.29
NZL    0    0.61
BTN    0    0.20
HTI    0    0.49
EST    0    0.35
VCT    0    0.34
PHL    0    0.42
BRB    0    0.87
MNG    0    0.27
NPL    0    0.36
```

```
pred.outlier.value_counts()

0    156
1    18
Name: outlier, dtype: int64
pred.groupby(['outlier'])[['scores']].agg(['min','median','max']
)
        scores
           min median  max
outlier
0    0.10    0.44    1.74
1    1.76    1.95    11.86
```

1. Show Covid data for the outliers.

First, merge the `covidanalysis` and `pred` data frames:

```
covidanalysis.join(pred).\
...   loc[pred.outlier==1,\
...   ['location','total_cases_pm',
...   'total_deaths_pm','scores']].\
...   sort_values(['scores'],
...   ascending=False).head(10)

                location  total_cases_pm  \
iso_code
SGP    Singapore    7,825.69
QAT    Qatar    35,795.16
BHR    Bahrain    19,082.23
BEL    Belgium    5,401.90
LUX    Luxembourg    7,735.12
CHL    Chile    16,322.75
USA    United States    9,811.66
KWT    Kuwait    12,658.28
ITA    Italy    4,016.20
NLD    Netherlands    2,968.57
              total_deaths_pm    scores
iso_code
SGP    4.44    11.86
QAT    50.68    7.54
BHR    61.12    4.01
BEL    844.03    3.02
LUX    175.73    2.39
CHL    359.96    2.36
USA    407.29    2.21
KWT    90.39    2.17
ITA    577.97    1.95
```

```
NLD    357.63    1.95
```

These steps show how we can use k-nearest neighbor to identify outliers based on multivariate relationships.

How it works...

PyOD is a package of Python outlier detection tools. We use it here as a wrapper around scikit-learn’s KNN package. This simplifies some tasks.Our focus in this recipe is not on building a model, but on getting a quick sense of which observations (countries) are significant outliers once we take all the data we have into account. This analysis supports our developing sense that Singapore and Qatar are very different observations than the others in our dataset. They have very high decision scores. (The table in *step 5* is sorted in descending order of score.)Countries such as Belgium, Bahrain, and Luxembourg might also be considered outliers, though that is less clear cut. The previous recipe did not indicate that they had an overwhelming influence on a regression model. But that model did not take both cases per million and deaths per million into account at the same time. That could also explain why Singapore is even more of an outlier than Qatar here. It has both high cases per million and below-average deaths per million.scikit-learn makes scaling very easy. We use the standard scaler in *step 2*, which returns the *z*-score for each value in the data frame. The *z*-score subtracts the variable mean from each variable value and divides it by the standard deviation for the variable. Many machine learning tools require standardized data to run well.

There’s more...

K-nearest neighbor is a very popular machine learning algorithm. It is easy to run and interpret. Its main limitation is that it will run slowly on large datasets.We have skipped steps we might usually take when building machine learning models. We did not create separate training and test datasets, for example. PyOD allows this to be done easily, but this is not necessary for our purposes here.

See also

The PyOD toolkit has a large number of supervised and unsupervised learning techniques for detecting anomalies in data. You can get the documentation for this at https://pyod.readthedocs.io/en/latest/.

Using Isolation Forest to find anomalies

Isolation Forest is a relatively new machine learning technique for identifying anomalies. It has quickly become popular, partly because its algorithm is optimized to find anomalies, rather than normal values. It finds outliers by successive partitioning of the data until a data point has been isolated. Points that require fewer partitions to be isolated receive higher anomaly scores. This process turns out to be fairly easy on system resources. In this recipe, we demonstrate how to use it to detect outlier Covid cases and deaths.

Getting ready

You will need scikit-learn and Matplotlib to run the code in this recipe. You can install them by entering `pip install sklearn` and `pip install matplotlib` in the terminal or `powershell` (in Windows).

How to do it…

We will use Isolation Forest to find the countries whose attributes indicate that they are most anomalous:

1. Load pandas, Matplotlib, and the standard scaler and Isolation Forest modules from scikit-learn:

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import IsolationForest
from mpl_toolkits.mplot3d import Axes3D
covidtotals = pd.read_csv("data/covidtotals.csv")
covidtotals.set_index("iso_code", inplace=True)
```

1. Create a standardized analysis data frame.

First, remove all rows with missing data:

```
analysisvars = ['location','total_cases_pm','total_deaths_pm',
...   'pop_density','median_age','gdp_per_capita']
standardizer = StandardScaler()
covidtotals.isnull().sum()
```

```
lastdate    0
location    0
total_cases    0
total_deaths    0
total_cases_pm    0
total_deaths_pm    0
population    0
pop_density    11
median_age    24
gdp_per_capita    27
hosp_beds    45
region    0
dtype: int64
```

```
covidanalysis = covidtotals.loc[:, analysisvars].dropna()
covidanalysisstand = standardizer.fit_transform(covidanalysis.il
oc[:, 1:])
```

1. Run an Isolation Forest model to detect outliers.

Pass the standardized data to the `fit` method. 18 countries are identified as outliers. (These countries have anomaly values of -1.) This is determined by the contamination number of `0.1`:

```
clf=IsolationForest(n_estimators=100, max_samples='auto',
...   contamination=.1, max_features=1.0)
clf.fit(covidanalysisstand)
IsolationForest(behaviour='deprecated', bootstrap=False, contami
nation=0.1,
                max_features=1.0, max_samples='auto', n_estimato
rs=100,
                n_jobs=None, random_state=None, verbose=0, warm_
start=False)
covidanalysis['anomaly'] = clf.predict(covidanalysisstand)
covidanalysis['scores'] = clf.decision_function(covidanalysissta
nd)
covidanalysis.anomaly.value_counts()
```

```
1    156
```

```
-1    18
Name: anomaly, dtype: int64
```

1. Create outlier and inlier data frames.

List the top 10 outliers according to anomaly score:

```
inlier, outlier = covidanalysis.loc[covidanalysis.anomaly==1],\
...   covidanalysis.loc[covidanalysis.anomaly==-1]
outlier[['location','total_cases_pm','total_deaths_pm',\
...   'median_age','gdp_per_capita','scores']].\
...   sort_values(['scores']).\
...   head(10)
location  total_cases_pm  total_deaths_pm  median_age  \
iso_code

                location  total_cases_pm  ...  \
iso_code                                  ...
SGP    Singapore    7,825.69  ...
QAT    Qatar    35,795.16  ...
BHR    Bahrain    19,082.23  ...
BEL    Belgium    5,401.90  ...
ITA    Italy    4,016.20  ...
CHL    Chile    16,322.75  ...
ESP    Spain    5,430.63  ...
SWE    Sweden    7,416.18  ...
GBR    United Kingdom    4,256.44  ...
LUX    Luxembourg    7,735.12  ...
          gdp_per_capita  scores
iso_code
SGP    85,535.38    -0.23
QAT    116,935.60    -0.21
BHR    43,290.71    -0.14
BEL    42,658.58    -0.12
ITA    35,220.08    -0.08
CHL    22,767.04    -0.08
ESP    34,272.36    -0.06
SWE    46,949.28    -0.04
GBR    39,753.24    -0.04
LUX    94,277.96    -0.04
```

1. Plot the outliers and inliers:

```
ax = plt.axes(projection='3d')
ax.set_title('Isolation Forest Anomaly Detection')
ax.set_zlabel("Cases Per Million")
ax.set_xlabel("GDP Per Capita")
ax.set_ylabel("Median Age")
ax.scatter3D(inlier.gdp_per_capita, inlier.median_age, inlier.to
tal_cases_pm, label="inliers", c="blue")
ax.scatter3D(outlier.gdp_per_capita, outlier.median_age, outlier
.total_cases_pm, label="outliers", c="red")
ax.legend()
plt.tight_layout()
plt.show()
```

This produces the following plot:

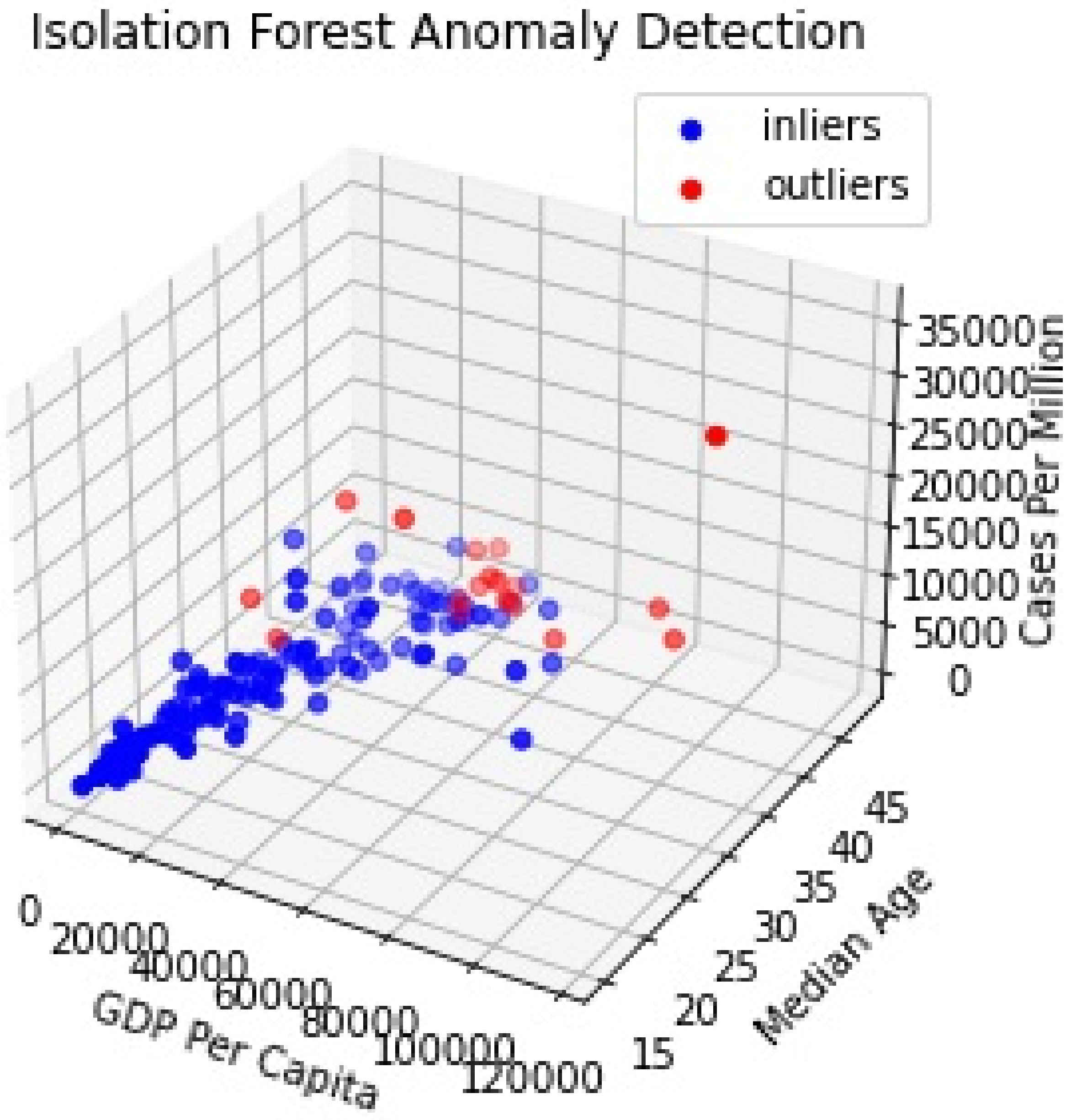


Figure 4.9: Inlier and outlier countries by GDP, median age, and cases per million

The preceding steps demonstrate the use of Isolation Forest as an alternative to k-nearest neighbor for anomaly detection.

How it works...

We use Isolation Forest in this recipe much like we used k-nearest neighbor in the previous recipe. In *step 3*, we pass a standardized dataset to the Isolation Forest `fit` method, and then use its `predict` and `decision_function` methods to get the anomaly flag and score, respectively. We use the anomaly flag in *step 4* to separate the data into inliers and outliers.We plot the inliers and outliers in *step 5*. Since there are only three dimensions in the plot, it does not quite capture all of the features in our Isolation Forest model, but the outliers (the red dots) clearly have higher GDP per capita and median age; these are typically to the right of, and behind, the inliers.The results from Isolation Forest are quite similar to the k-nearest neighbor results. Qatar, Singapore, and Hong Kong have the highest (most negative) anomaly scores. Belgium is not far behind, just as with the KNN model. This is most likely due to an exceptionally high total of deaths per million for Belgium, the highest in the dataset. We should consider removing these four observations from any multivariate analyses we conduct.

There's more...

Isolation Forest is a good alternative to k-nearest neighbor, particularly when working with large datasets. The efficiency of its algorithm allows it to handle large samples and a high number of features (variables).The anomaly detection techniques we have used in the last three recipes were designed to improve multivariate analyses and the training of machine learning models. However, we might want to exclude the outliers they help us identify much earlier in the analysis process. For example, if it makes sense to exclude Qatar from our modeling, it might also make sense to exclude Qatar from some descriptive statistics.

See also

In addition to being useful for anomaly detection, the Isolation Forest algorithm is quite satisfying intuitively. (I think the same could be said about k-nearest neighbor.) You can read more about Isolation Forest here: https://cs.nju.edu.cn/zhouzh/zhouzh.files/publication/icdm08b.pdf.

5 Using Visualizations for the Identification of Unexpected Values

We dipped our toes in the water on visualizations in several recipes in the previous chapter. We used histograms and QQ plots to examine the distribution of a single variable, and scatter plots to view how two variables are related. But we were just scratching the surface of the rich visualization tools available in the Matplotlib and Seaborn libraries. Getting comfortable with these tools, and their seemingly inexhaustible capabilities, can help us uncover patterns and oddities that are not obvious when we run the standard battery of descriptives.Boxplots, for example, are a great tool for visualizing values outside of a certain range. These can be extended with grouped boxplots or violin plots that allow us to compare distributions across subsets of data. We can also do much more with scatter plots than we did in the last chapter, including getting some sense of multivariate relationships. Histograms, too, can sometimes offer additional insight if we display several histograms on one plot or create a stacked histogram. We explore all of these capabilities in this chapter.Specifically, the recipes in this chapter demonstrate the following topics:

- Using histograms to examine the distribution of continuous variables
- Using boxplots to identify outliers for continuous variables
- Using grouped boxplots to uncover unexpected values in a particular group

- Examining both distribution shape and outliers with violin plots
- Using scatter plots to view bivariate relationships
- Using line plots to examine trends in continuous variables
- Generating a heat map based on a correlation matrix

Using histograms to examine the distribution of continuous variables

The go-to visualization tool for statisticians trying to understand how single variables are distributed is the histogram. Histograms plot a continuous variable on the *x* axis, in bins determined by the researcher, and the frequency of occurrence on the *y* axis.Histograms provide a clear and meaningful illustration of the shape of a distribution, including central tendency, skewness (symmetry), excess kurtosis (relatively fat tails), and spread. This matters for statistical testing, as many tests make assumptions about a variable's distribution. Moreover, our expectation of what data values to expect should be guided by our understanding of the distribution's shape. For example, a value at the 90th percentile has very different implications when it comes from a normal distribution rather than from a uniform distribution.One of the first tasks I ask introductory statistics students to do is construct a histogram manually from a small sample. We do boxplots in the following class. Together, histograms and boxplots provide a solid foundation for subsequent analysis. In my data science work, I try to remember to construct histograms and boxplots on all continuous variables of interest shortly after the initial importing and cleaning of data. We create histograms in this recipe, and boxplots in the following two recipes.

Getting ready

We will use the matplotlib library to generate histograms. Some tasks can be done quickly and straightforwardly in matplotlib. Histograms are one of those tasks. We will switch between matplotlib and seaborn (which is built on matplotlib) in this chapter, based on which tool gets us to the required graphic more easily.We will also use the statsmodels library. You can install matplotlib and statsmodels with pip using `pip install matplotlib` and `pip install statsmodels`.We will work with data on land temperature and on coronavirus cases in this recipe. The land temperature DataFrame has one row per weather station. The coronavirus DataFrame has one row per country and reflects totals as of July 18, 2020.

Data note

The land temperature DataFrame has the average temperature reading (in °C) in 2019 from over 12,000 stations across the world, though a majority of the stations are in the United States. The raw data was retrieved from the *Global Historical Climatology Network* integrated database. It is made available for public use by the United States National Oceanic and Atmospheric Administration at https://www.ncdc.noaa.gov/data-access/land-based-station-data/land-based-datasets/global-historical-climatology-network-monthly-version-4.

Our World in Data provides Covid-19 public use data at https://ourworldindata.org/coronavirus-source-data. The data used in this recipe was downloaded on June 1, 2020. Some of the data was missing for Hong Kong as of this date, but this problem was fixed in files after that.

How to do it...

We take a close look at the distribution of land temperatures by weather station in 2019 and total coronavirus cases per million in population for each country. We start with a few descriptive statistics before doing a QQ plot, histograms, and stacked histograms.

1. Import the pandas, matplotlib, and statsmodels libraries.

Also, load data on land temperatures and Covid cases:

```
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm
landtemps = pd.read_csv("data/landtemps2019avgs.csv")
covidtotals = pd.read_csv("data/covidtotals.csv", parse_dates=["
lastdate"])
```

```
covidtotals.set_index("iso_code", inplace=True)
```

1. Show some of the station temperature rows.

The `latabs` column is the value of latitude without the north or south indicators; so, Cairo, Egypt at approximately 30 degrees north, and Porto Alegre, Brazil at about 30 degrees south have the same value:

```
landtemps[['station','country','latabs',
...   'elevation','avgtemp']].\
...   sample(10, random_state=1)

                    station            country      latabs
10526    NEW_FORK_LAKE    United States    43
1416    NEIR_AGDM    Canada    51
2230    CURICO    Chile    35
6002    LIFTON_PUM...    United States    42
2106    HUAILAI    China    40
2090    MUDANJIANG    China    45
7781    CHEYENNE_6S...     United States    36
10502    SHARKSTOOTH    United States    38
11049    CHALLIS_AP    United States    45
2820    METHONI    Greece    37
    elevation     avgtemp
10526    2,542    2
1416    1,145    2
2230    225    16
6002    1,809    4
2106    538    11
2090    242    6
7781    694    15
10502    3,268    4
11049    1,534    7
2820    52    18
```

1. Show some descriptive statistics.

Also, look at the skew and the kurtosis:

```
landtemps.describe()

    latabs    elevation    avgtemp
count      12,095    12,095    12,095
mean    40    589    11
std    13    762    9
```

```
min    0    -350    -61
25%    35    78    5
50%    41    271    10
75%    47    818    17
max    90    9,999    34
```

```
landtemps.avgtemp.skew()
```

```
-0.2678382583481769
```

```
landtemps.avgtemp.kurtosis()
```

```
2.1698313707061074
```

1. Do a histogram of average temperatures.

Also, draw a line at the overall mean:

```
plt.hist(landtemps.avgtemp)
plt.axvline(landtemps.avgtemp.mean(), color='red', linestyle='da
shed', linewidth=1)
plt.title("Histogram of Average Temperatures (Celsius)")
plt.xlabel("Average Temperature")
plt.ylabel("Frequency")
plt.show()
```

This results in the following histogram:

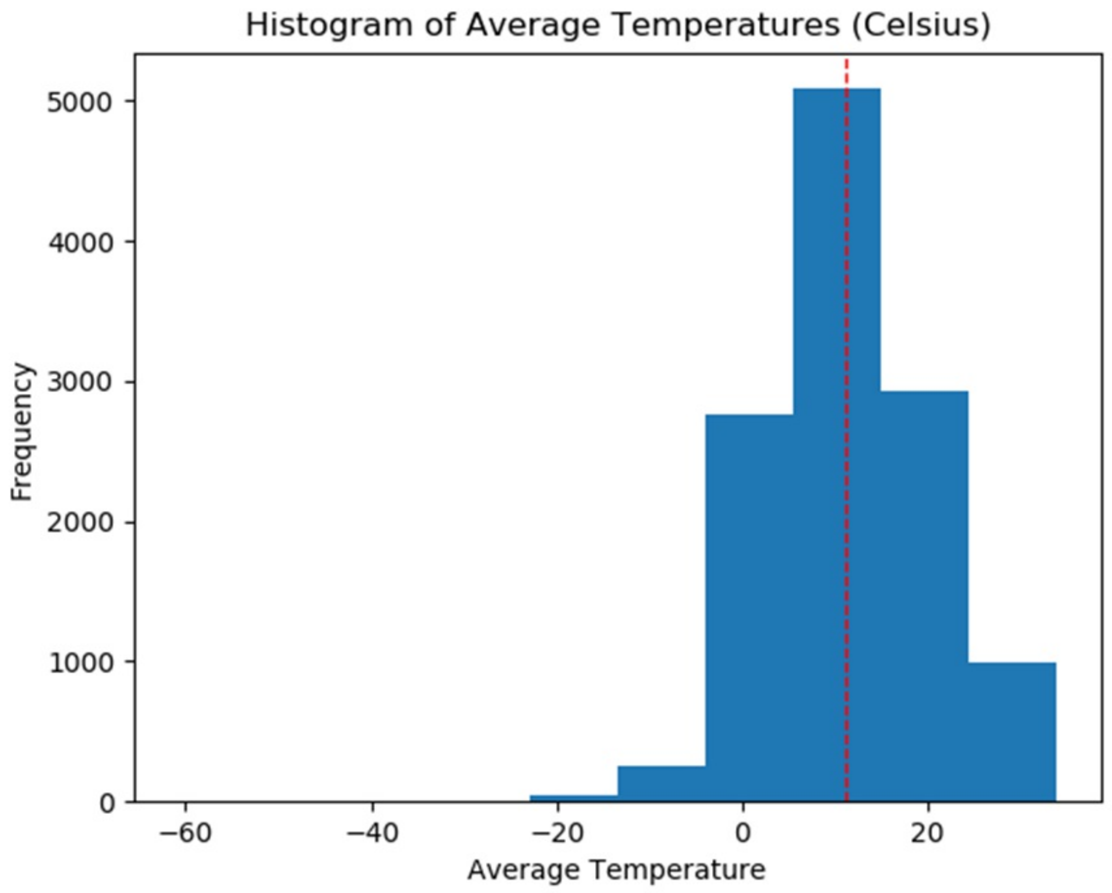


Figure 5.1: Histogram of average temperatures across weather stations in 2019

1. Run a QQ plot to examine where the distribution deviates from a normal distribution.

Notice that much of the distribution of temperatures falls along the red line (all dots would fall on the red line if the distribution were perfectly normal, but the tails fall off dramatically from normal):

```
sm.qqplot(landtemps[['avgtemp']].sort_values(['avgtemp']), line=
's')
plt.title("QQ Plot of Average Temperatures")
plt.show()
```

This results in the following QQ plot:

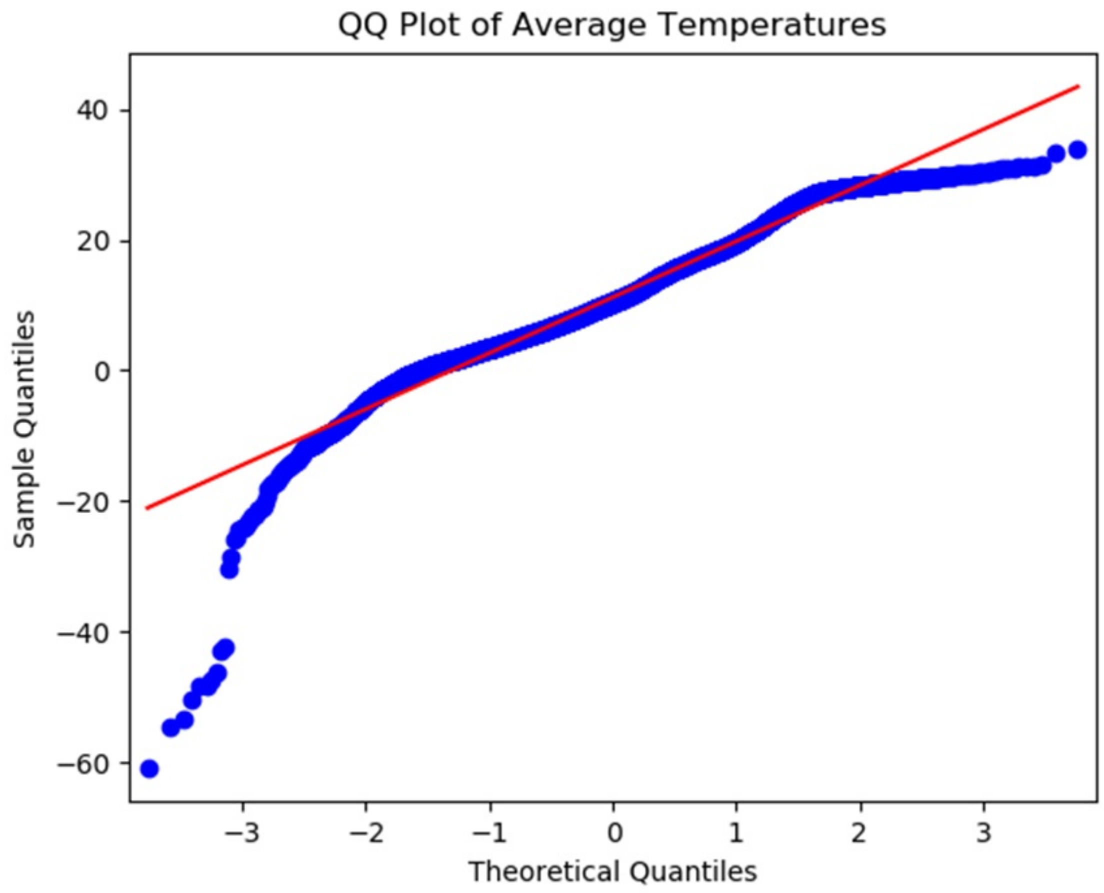


Figure 5.2: Plot of average temperature by station compared with a normal distribution

1. Show skewness and kurtosis for total Covid cases per million.

This is from the coronavirus DataFrame, which has one row for each country:

```
covidtotals.total_cases_pm.skew()

4.284484653881833

covidtotals.total_cases_pm.kurtosis()

26.137524276840452
```

1. Do a stacked histogram of the Covid case data.

Select data from four of the regions. (Stacked histograms can get messy with any more categories than that.) Define a `getcases` function that returns a series for `total_cases_pm` for the countries of a region. Pass those series to the `hist` method (`[getcases(k) for k in showregions]`) to create the stacked histogram. Notice that much of the distribution—almost 40 countries out of the 65 countries in these regions—has cases per million below 2,000:

```
showregions = ['Oceania / Aus','East Asia','Southern Africa',
...   'Western Europe']
def getcases(regiondesc):
...   return covidtotals.loc[covidtotals.\
...     region==regiondesc,
...     'total_cases_pm']
...
plt.hist([getcases(k) for k in showregions],\
...   color=['blue','mediumslateblue','plum','mediumvioletred'],
\
...   label=showregions,\
...   stacked=True)
plt.title("Stacked Histogram of Cases Per Million for Selected R
egions")
plt.xlabel("Cases Per Million")
plt.ylabel("Frequency")
plt.xticks(np.arange(0, 22500, step=2500))
plt.legend()
plt.show()
```

This results in the following stacked histogram:

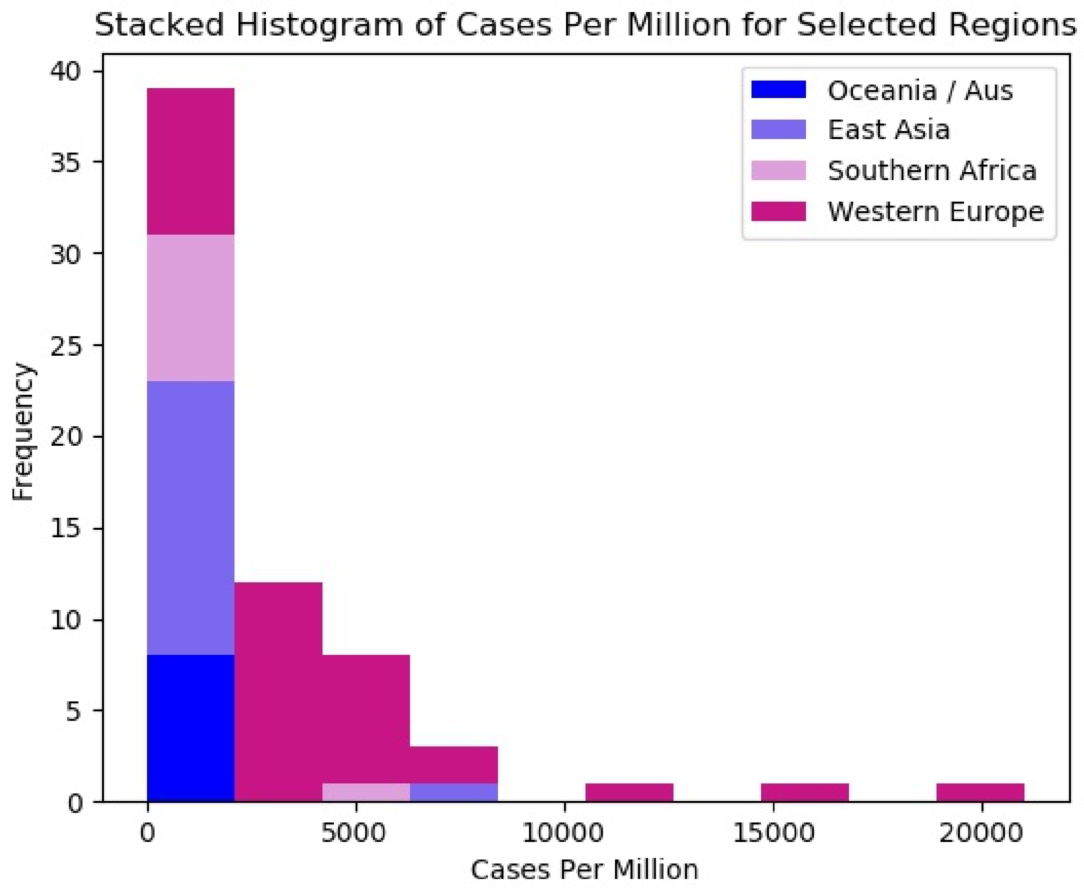


Figure 5.3: Stacked histogram of number of countries per region at different cases per million levels

1. Show multiple histograms on one figure.

This allows different *x* and *y* axis values. We need to loop through each axis and select a different region from `showregions` for each subplot:

```
fig, axes = plt.subplots(2, 2)
fig.subtitle("Histograms of Covid Cases Per Million by Selected
Regions")
axes = axes.ravel()
for j, ax in enumerate(axes):
...   ax.hist(covidtotals.loc[covidtotals.region==showregions[j]
].\
...     total_cases_pm, bins=5)
...   ax.set_title(showregions[j], fontsize=10)
...   for tick in ax.get_xticklabels():
```

```
...     tick.set_rotation(45)
...
plt.tight_layout()
fig.subplots_adjust(top=0.88)
plt.show()
```

This results in the following histograms:

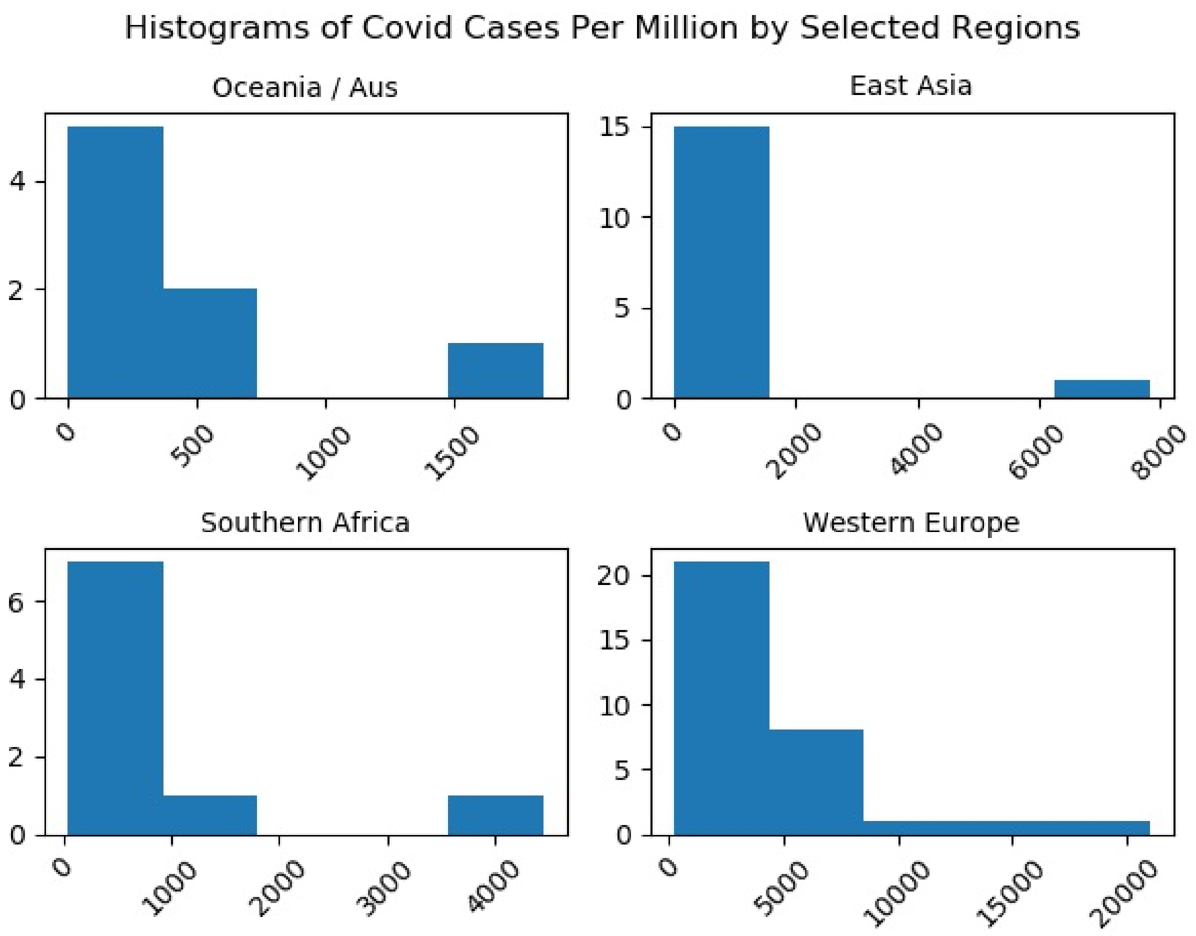


Figure 5.4: Histograms by region of number of countries at different cases per million levels

The preceding steps demonstrated how to visualize the distribution of a continuous variable using histograms and QQ plots.

How it works...

Step 4 shows how easy it is to display a histogram. This can be done by passing a series to the `hist` method of Matplotlib's `pyplot` module. (We use an alias of `plt` for matplotlib.) We could have also passed any `ndarray`, or even a list of data series.We also get great access to the attributes of the figure and its axes. We can set the labels for each axis, as well as the tick marks and tick labels. We can also specify the content and look and feel of the legend. We will be taking advantage of this functionality often in this chapter.We pass multiple series to the `hist` method in *Step 7* to produce the stacked histogram. Each series is the `total_cases_pm` (cases per million of population) value for the countries in a region. To get the series for each region, we call the `getcases` function for each item in `showregions`. We choose colors for each series rather than allowing that to happen automatically. We also use the `showregions` list to select labels for the legend.In *Step 8*, we start by indicating that we want four subplots, in two rows and two columns. That is what we get with `plt.subplots(2, 2)`, which returns both a figure and the four axes. We loop through the axes with `for j, ax in enumerate(axes)`. Within each loop, we select a different region for the histogram from `showregions`. Within each axis, we loop through the tick labels and change the rotation. We also adjust the start of the subplots to make enough room for the figure title. Note that we need to use `suptitle` to add a title in this case. Using `title` would add the title to a subplot.

There's more...

The land temperature data is not quite normally distributed, as the histograms and the skew and kurtosis measures show. It is skewed to the left (skew of `-0.26`) and actually has somewhat skinnier tails than normal (kurtosis of 2.17, compared with 3). Although there are some extreme values, there are not that many of them relative to the overall size of the dataset. While it is not perfectly bell-shaped, the land temperature DataFrame is a fair bit easier to deal with than the Covid case data.The skew and kurtosis of the Covid `cases per million` variable show that it is some distance from normal. The skew of 4 and kurtosis of 26 indicates a high positive skew and much fatter tails than with a normal distribution. This is also reflected in the histograms, even when we look at the numbers by region. There are a number of countries at very low levels of cases per million in most regions, and just a few countries with high levels of cases. The *Using grouped boxplots to uncover unexpected values in a particular group* recipe in this chapter shows that there are outliers in almost every region.If you work through all of the recipes in this chapter, and you are relatively new to matplotlib and seaborn, you will find those libraries either usefully flexible or confusingly flexible. It is difficult to even pick one strategy and stick with it because you might need to set up your figure and axes in a particular way to get the visualization you want. It is helpful to keep two things in mind when working through these recipes: first, you will generally need to create a figure and one or more subplots; and second, the main plotting functions work similarly regardless, so `plt.hist` and `ax.hist` will both often work.

Using boxplots to identify outliers for continuous variables

Boxplots are essentially a graphical representation of our work in the *Identifying outliers with one variable* recipe in *Chapter 4*, *Identifying Missing Values and Outliers in Subsets of Data*. There, we used the concept of **interquartile range** (**IQR**)—the distance between the value at the first quartile and the value at the third quartile—to determine outliers. Any value greater than (`1.5 * IQR`) + the third quartile value, or less than the first quartile value – (`1.5 * IQR`), was considered an outlier. That is precisely what is revealed in a boxplot.

Getting ready

We will work with cumulative data on coronavirus cases and deaths by country, and the **National Longitudinal Surveys** (**NLS**) data. You will need the matplotlib library to run the code on your computer.

How to do it…

We use boxplots to show the shape and spread of **Scholastic Assessment Test** (**SAT**) scores, weeks worked, and Covid cases and deaths.

1. Load the pandas and matplotlib libraries.

Also, load the NLS and Covid data:

```
import pandas as pd
import matplotlib.pyplot as plt
nls97 = pd.read_csv("data/nls97.csv")
nls97.set_index("personid", inplace=True)
covidtotals = pd.read_csv("data/covidtotals.csv", parse_dates=["
lastdate"])
covidtotals.set_index("iso_code", inplace=True)
```

1. Do a boxplot of SAT verbal scores.

Produce some descriptives first. The `boxplot` method produces a rectangle that represents the IQR, the values between the first and third quartile. The whiskers go from that rectangle to 1.5 times the IQR. Any values above or below the whiskers (what we have labeled the outlier threshold) are considered outliers (we use `annotate` to point to the first and third quartile points, the median, and to the outlier thresholds):

```
nls97.satverbal.describe()

count    1,406
mean    500
std    112
min    14
25%    430
50%    500
75%    570
max    800
Name: satverbal, dtype: float64

plt.boxplot(nls97.satverbal.dropna(), labels=['SAT Verbal'])
plt.annotate('outlier threshold', xy=(1.05,780), xytext=(1.15,78
0), size=7, arrowprops=dict(facecolor='black', headwidth=2, widt
h=0.5, shrink=0.02))
plt.annotate('3rd quartile', xy=(1.08,570), xytext=(1.15,570), s
ize=7, arrowprops=dict(facecolor='black', headwidth=2, width=0.5
, shrink=0.02))
plt.annotate('median', xy=(1.08,500), xytext=(1.15,500), size=7,
 arrowprops=dict(facecolor='black', headwidth=2, width=0.5, shri
nk=0.02))
plt.annotate('1st quartile', xy=(1.08,430), xytext=(1.15,430), s
ize=7, arrowprops=dict(facecolor='black', headwidth=2, width=0.5
, shrink=0.02))
plt.annotate('outlier threshold', xy=(1.05,220), xytext=(1.15,22
0), size=7, arrowprops=dict(facecolor='black', headwidth=2, widt
h=0.5, shrink=0.02))
#plt.annotate('outlier threshold', xy=(1.95,15), xytext=(1.55,15
), size=7, arrowprops=dict(facecolor='black', headwidth=2, width
=0.5, shrink=0.02))
plt.show()
```

This results in the following boxplot:

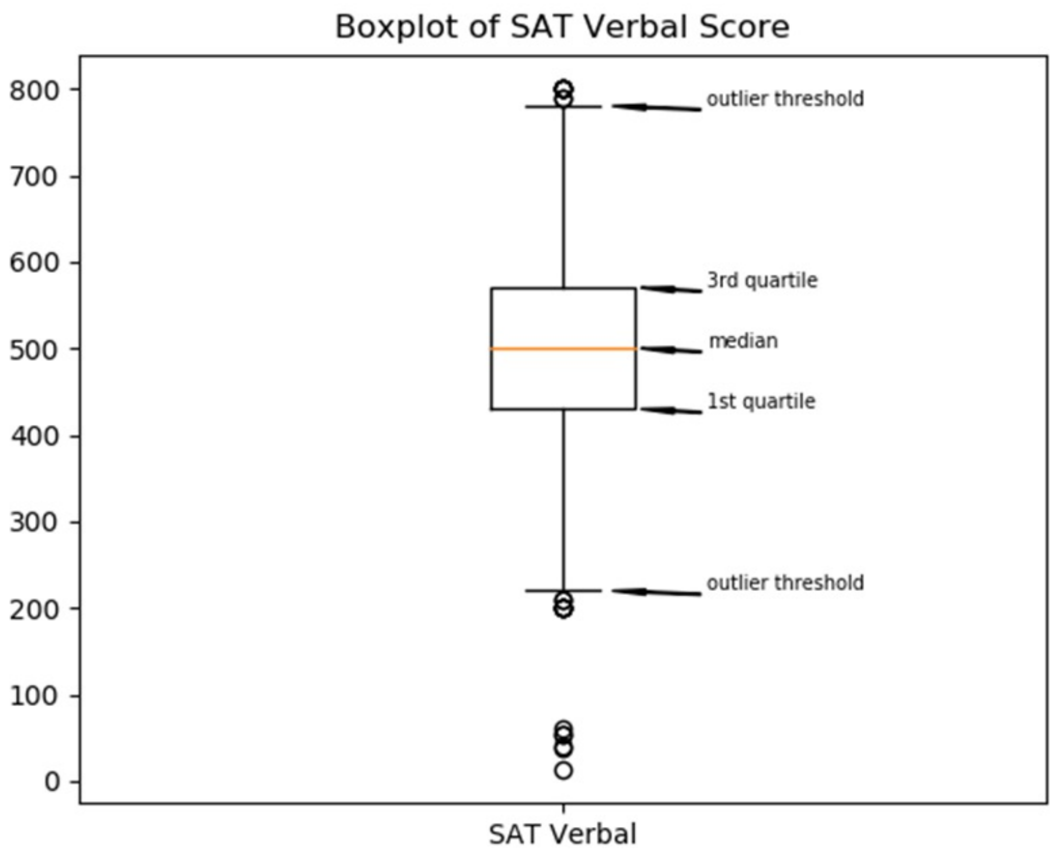


Figure 5.5: Boxplot of SAT verbal scores with labels for interquartile range and outliers

1. Show some descriptives on weeks worked:

```
weeksworked = nls97.loc[:, ['highestdegree',
...    'weeksworked16','weeksworked17']]
weeksworked.describe()
```

```
       weeksworked16  weeksworked17
count    7,068    6,670
mean    39    39
std    21    19
min    0    0
25%    23    37
50%    53    49
75%    53    52
max    53    52
```

1. Do boxplots of weeks worked:

```
plt.boxplot([weeksworked.weeksworked16.dropna(),
...   weeksworked.weeksworked17.dropna()],
...   labels=['Weeks Worked 2016','Weeks Worked 2017'])
plt.title("Boxplots of Weeks Worked")
plt.tight_layout()
plt.show()
```

This results in the following boxplots:

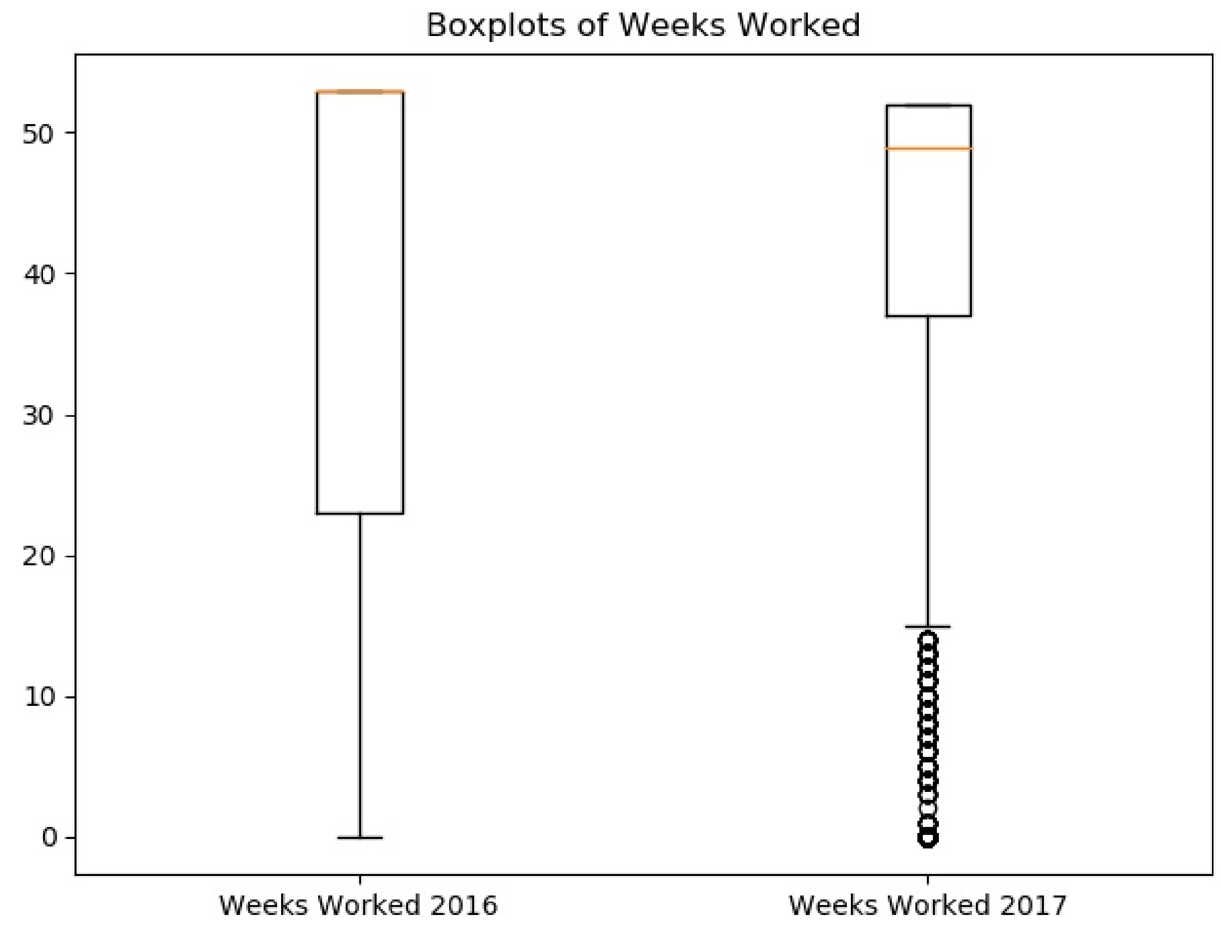


Figure 5.6: Boxplots of two variables side by side

1. Show some descriptives for the Covid data.

Create a list of labels (`totvarslabels`) for columns to use in a later step:

```
totvars = ['total_cases','total_deaths',
...   'total_cases_pm','total_deaths_pm']
totvarslabels = ['cases','deaths',
...   'cases per million','deaths per million']
covidtotalsonly = covidtotals[totvars]
covidtotalsonly.describe()
```

```
       total_cases  total_deaths  total_cases_pm  \
count    209    209    209
mean    60,757    2,703    2,297
std    272,440    11,895    4,040
min    3    0    1
```

```
25%    342    9    203
50%    2,820    53    869
75%    25,611    386    2,785
max    3,247,684    134,814    35,795
       total_deaths_pm
count    209
mean    74
std    156
min    0
25%    3
50%    15
75%    58
max    1,238
```

1. Do a boxplot of cases and deaths per million:

```
fig, ax = plt.subplots()
plt.title("Boxplots of Covid Cases and Deaths Per Million")
ax.boxplot([covidtotalsonly.total_cases_pm,covidtotalsonly.total
_deaths_pm],\
...   labels=['cases per million','deaths per million'])
plt.tight_layout()
plt.show()
```

This results in the following boxplots:

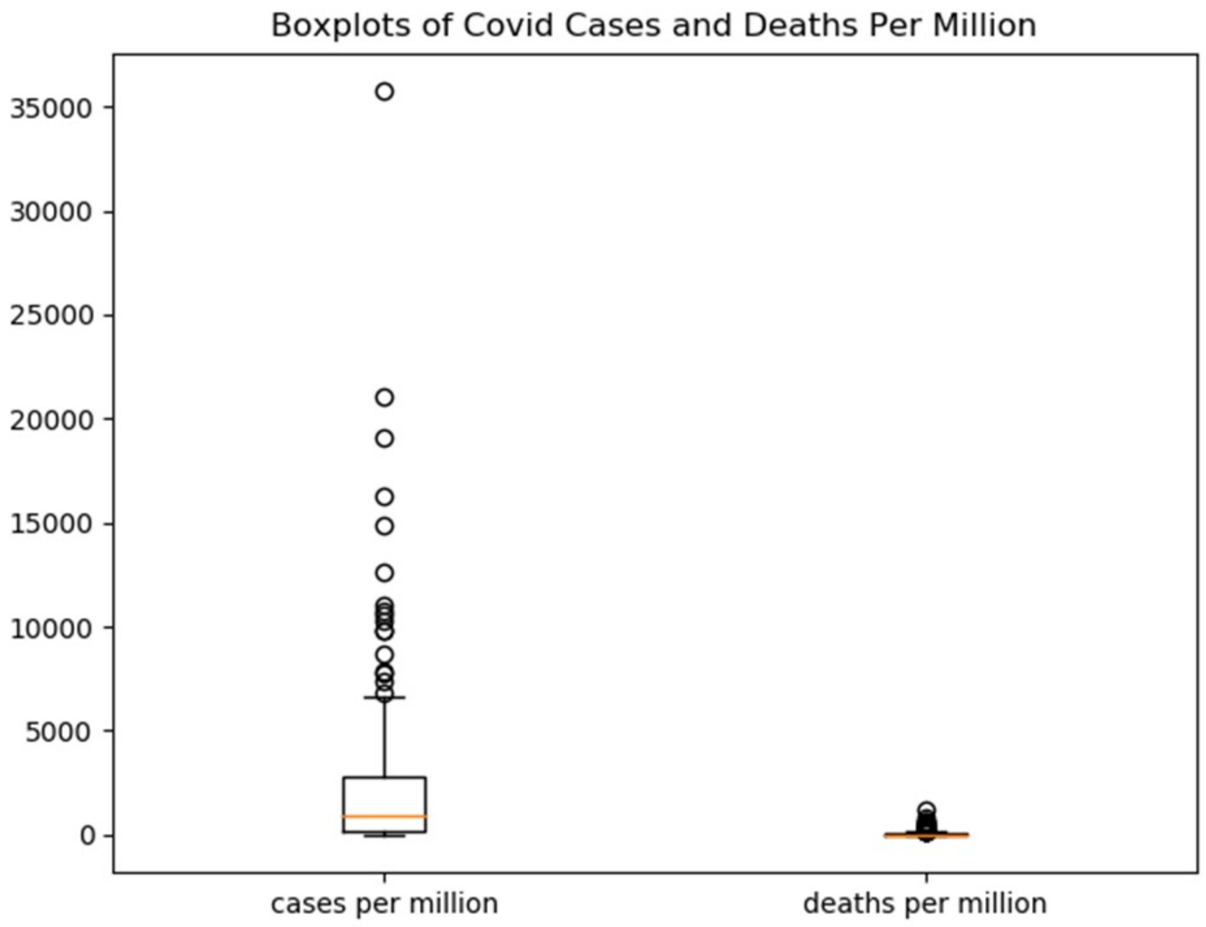


Figure 5.7: Boxplots of two variables side by side

1. Show boxplots as separate subplots on one figure.

It is hard to view multiple boxplots on one figure when the variable values are very different, as is true for Covid cases and deaths. Fortunately, matplotlib allows us to create multiple subplots on each figure, each of which can use different *x* and *y* axes:

```
fig, axes = plt.subplots(2, 2)
fig.suptitle("Boxplots of Covid Cases and Deaths")
axes = axes.ravel()
for j, ax in enumerate(axes):
...   ax.boxplot(covidtotalsonly.iloc[:, j], labels=[totvarslabe
ls[j]])
...
plt.tight_layout()
```

```
fig.subplots_adjust(top=0.94)
plt.show()
```

This results in the following boxplots:

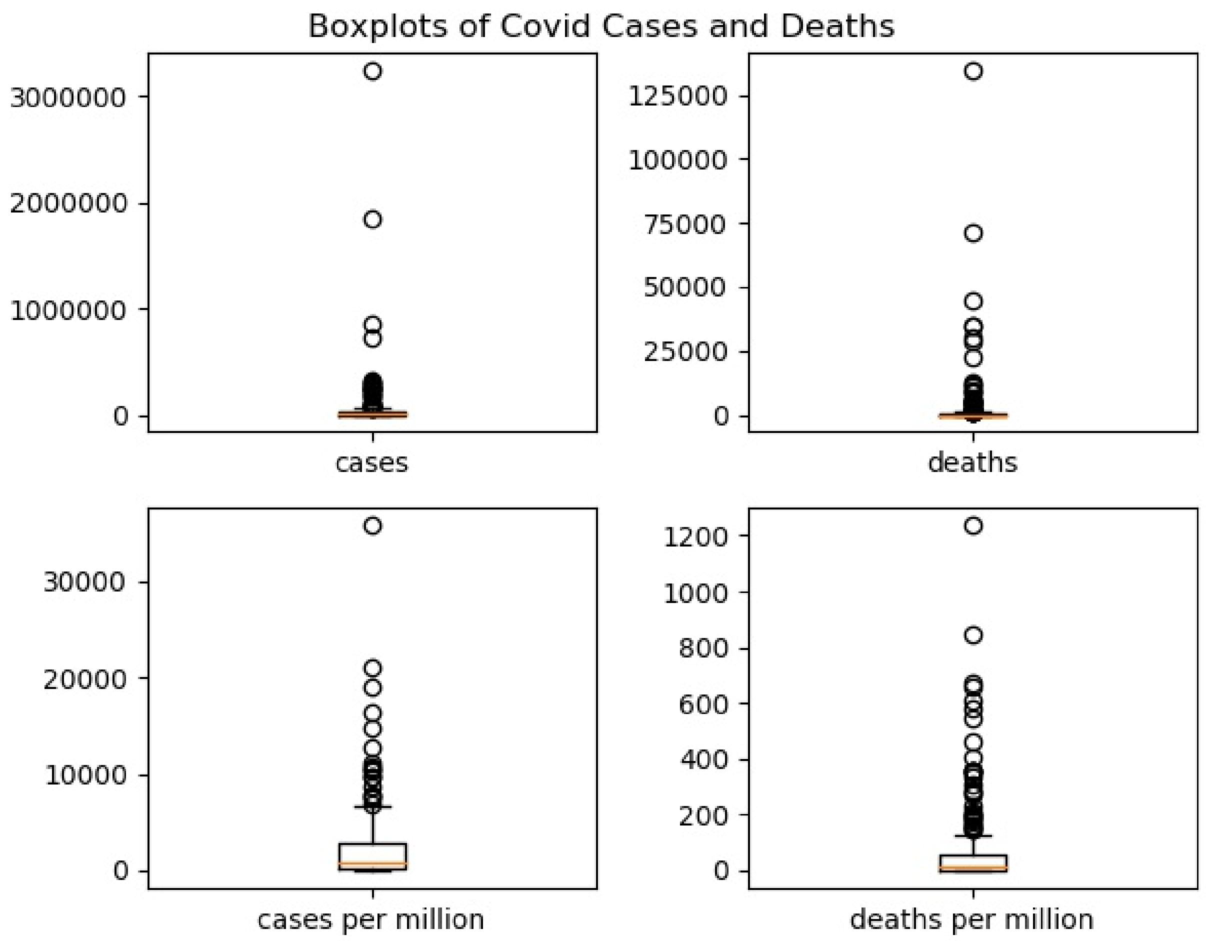


Figure 5.8: Boxplots with different y axes

Boxplots are a relatively straightforward but exceedingly useful way to view how variables are distributed. They make it easy to visualize spread, central tendency, and outliers, all in one graphic.

How it works...

It is fairly easy to create a boxplot with matplotlib, as *Step 2* shows. Passing a series to pyplot is all that is required (we use the `plt` alias). We call pyplot's `show` method to show the figure. This step also demonstrates how to use annotate to add text and symbols to your figure. We show multiple boxplots in *Step 4* by passing multiple series to pyplot.It can be difficult to show multiple boxplots in a single figure when the scales are very different, as is the case with the Covid outcome data (cases, deaths, cases per million, and deaths per million). *Step 7* shows one way to deal with that. We can create several subplots on one plot. We start by indicating that we want four subplots, in two columns and two rows. That is what we get with `plt.subplots(2, 2)`, which returns both a figure and the four axes. We can then loop through the axes, calling `boxplot` on each one. Nifty!However, it is still hard to see the IQR for cases and deaths because of some of the extreme values. In the next recipe, we remove some of the extreme values to give us a better visualization of the remaining data.

There's more...

The boxplot of SAT verbal scores in *Step 2* suggests a relatively normal distribution. The median is close to the center of the IQR. This is not surprising given that the descriptives we ran show that the mean and median have the same value. There is, however, substantially more room for outliers at the lower end than at the upper end. (Indeed, the very low SAT verbal scores seem implausible and should be checked.)The boxplots of weeks worked in 2016 and 2017 in *Step 4* show variables that are distributed much differently than SAT scores. The medians are near the top of the IQR and are much greater than the means. This suggests a negative skew. Also, notice that there are no whiskers or outliers at the upper end of the distribution as the median value is at, or near, the maximum.

See also

Some of these boxplots suggest that the data we are examining is not normally distributed. The *Identifying outliers with one variable* recipe in *Chapter 4* covers some normal distribution tests. It also shows how to take a closer look at the values outside of the outlier thresholds: the circles in the boxplots.

Using grouped boxplots to uncover unexpected

values in a particular group

We saw in the previous recipe that boxplots are a great tool for examining the distribution of continuous variables. They can also be useful when we want to see if those variables are distributed differently for parts of our dataset: salaries for different age groups; number of children by marital status; litter size for different mammal species. Grouped boxplots are a handy and intuitive way to view differences in variable distribution by categories in our data.

Getting ready

We will work with the NLS and the Covid case data. You will need matplotlib and seaborn installed on your computer to run the code in this recipe.

How to do it…

We generate descriptive statistics of weeks worked by highest degree earned. We then use grouped boxplots to visualize the spread of the weeks worked distribution by degree, and of Covid cases by region:

1. Import the pandas, matplotlib, and seaborn libraries:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
nls97 = pd.read_csv("data/nls97.csv")
nls97.set_index("personid", inplace=True)
covidtotals = pd.read_csv("data/covidtotals.csv", parse_dates=["
lastdate"])
covidtotals.set_index("iso_code", inplace=True)
```

1. View the median, and first and third quartile values for weeks worked for each degree attainment level.

First, define a function that returns those values as a series, then use `apply` to call it for each group:

```
def gettots(x):
...   out = {}
...   out['min'] = x.min()
...   out['qr1'] = x.quantile(0.25)
...   out['med'] = x.median()
...   out['qr3'] = x.quantile(0.75)
...   out['max'] = x.max()
...   out['count'] = x.count()
...   return pd.Series(out)
...
nls97.groupby(['highestdegree'])['weeksworked17'].\
...   apply(gettots).unstack()
```

```
                 min  qr1  med  qr3  max  count
highestdegree
0. None    0    0    40    52    52    510
1. GED    0    8    47    52    52    848
2. High School    0    31    49    52    52    2,665
3. Associates    0    42    49    52    52    593
4. Bachelors    0    45    50    52    52    1,342
5. Masters    0    46    50    52    52    538
6. PhD    0    46    50    52    52    51
7. Professional    0    47    50    52    52    97
```

1. Do a boxplot of weeks worked by highest degree earned.

Use Seaborn for these boxplots. First, create a subplot and name it `myplt`. This makes it easier to access subplot attributes later. Use the `order` parameter of `boxplot` to order by highest degree earned. Notice that there are no outliers or whiskers at the lower end for individuals with no degree ever received. This is because the IQR for those individuals covers the whole range of values; that is, the value at the 25th percentile is 0 and the value at the 75th percentile is 52:

```
myplt = sns.boxplot('highestdegree','weeksworked17', data=nls97,
...   order=sorted(nls97.highestdegree.dropna().unique()))
myplt.set_title("Boxplots of Weeks Worked by Highest Degree")
myplt.set_xlabel('Highest Degree Attained')
myplt.set_ylabel('Weeks Worked 2017')
myplt.set_xticklabels(myplt.get_xticklabels(), rotation=60, hori
zontalalignment='right')
plt.tight_layout()
plt.show()
```

This results in the following boxplots:

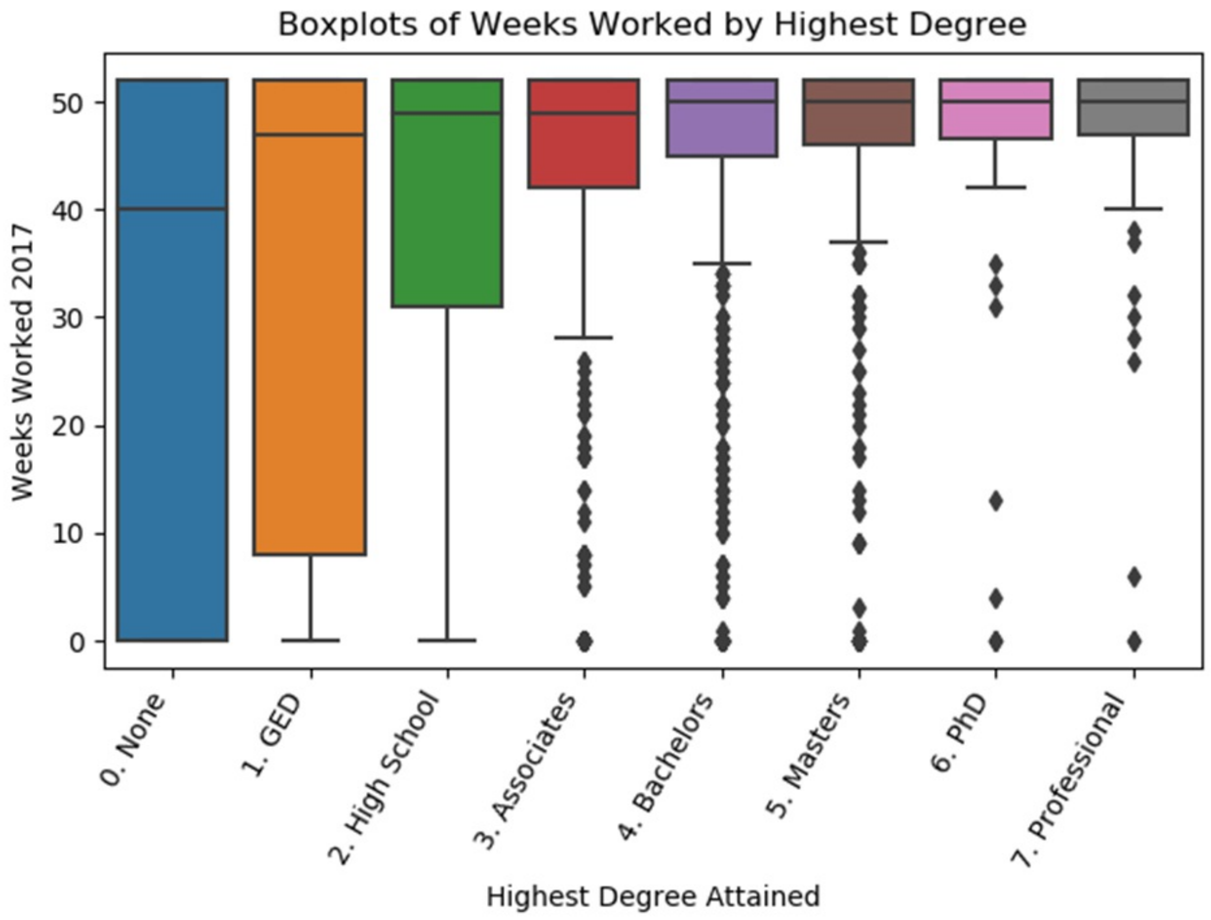


Figure 5.9: Boxplots of weeks worked with IQR and outliers by highest degree

1. View the minimum, maximum, median, and first and third quartile values for total cases per million by region.

Use the `gettots` function defined in *Step 2*:

```
covidtotals.groupby(['region'])['total_cases_pm'].\
...   apply(gettots).unstack()

                      min   qr1   med   qr3    max  count
region
Caribbean              95   252   339 1,726  4,435     22
Central Africa         15    71   368 1,538  3,317     11
Central America    93   925 1,448 2,191 10,274      7
Central Asia      374   919 1,974 2,907 10,594      6
East Africa         9    65   190   269  5,015     13
East Asia           3    16    65   269  7,826     16
Eastern Europe    347   883 1,190 2,317  6,854     22
North Africa      105   202   421   427    793      5
North America   2,290 2,567 2,844 6,328  9,812      3
Oceania / Aus       1    61   234   424  1,849      8
South America     284   395 2,857 4,044 16,323     13
South Asia        106   574   885 1,127 19,082      9
Southern Africa    36    86   118   263  4,454      9
West Africa        26   114   203   780  2,862     17
West Asia          23   273 2,191 5,777 35,795     16
Western Europe    200 2,193 3,769 5,357 21,038     32
```

1. Do boxplots of cases per million by region.

Flip the axes since there are a large number of regions. Also, do a swarm plot to give some sense of the number of countries by region. The swarm plot displays a dot for each country in each region. Some of the IQRs are hard to see because of the extreme values:

```
sns.boxplot('total_cases_pm', 'region', data=covidtotals)
sns.swarmplot(y="region", x="total_cases_pm", data=covidtotals,
size=2, color=".3", linewidth=0)
plt.title("Boxplots of Total Cases Per Million by Region")
plt.xlabel("Cases Per Million")
plt.ylabel("Region")
plt.tight_layout()
plt.show()
```

This results in the following boxplots:

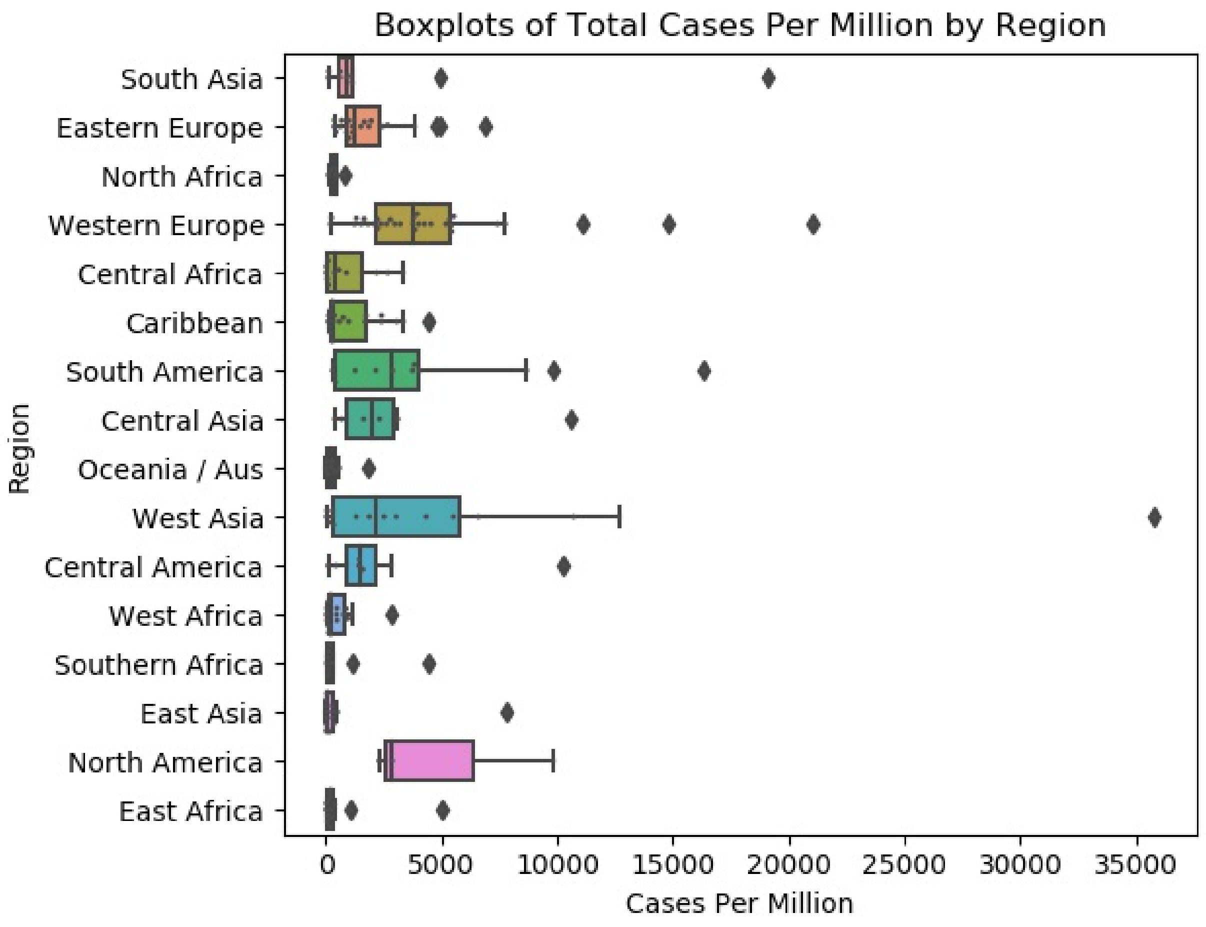


Figure 5.10: Boxplots and swarm plots of cases per million by region, with IQR and outliers

1. Show the most extreme values for cases per million:

```
covidtotals.loc[covidtotals.total_cases_pm>=14000,\
...   ['location','total_cases_pm']]

            location  total_cases_pm
iso_code
BHR          Bahrain          19,082
CHL            Chile          16,323
QAT            Qatar          35,795
SMR       San Marino          21,038
VAT          Vatican          14,833
```

1. Redo the boxplots without the extreme values:

```
sns.boxplot('total_cases_pm', 'region', data=covidtotals.loc[cov
idtotals.total_cases_pm<14000])
sns.swarmplot(y="region", x="total_cases_pm", data=covidtotals.l
oc[covidtotals.total_cases_pm<14000], size=3, color=".3", linewi
dth=0)
plt.title("Total Cases Without Extreme Values")
plt.xlabel("Cases Per Million")
plt.ylabel("Region")
plt.tight_layout()
plt.show()
```

This results in the following boxplots:

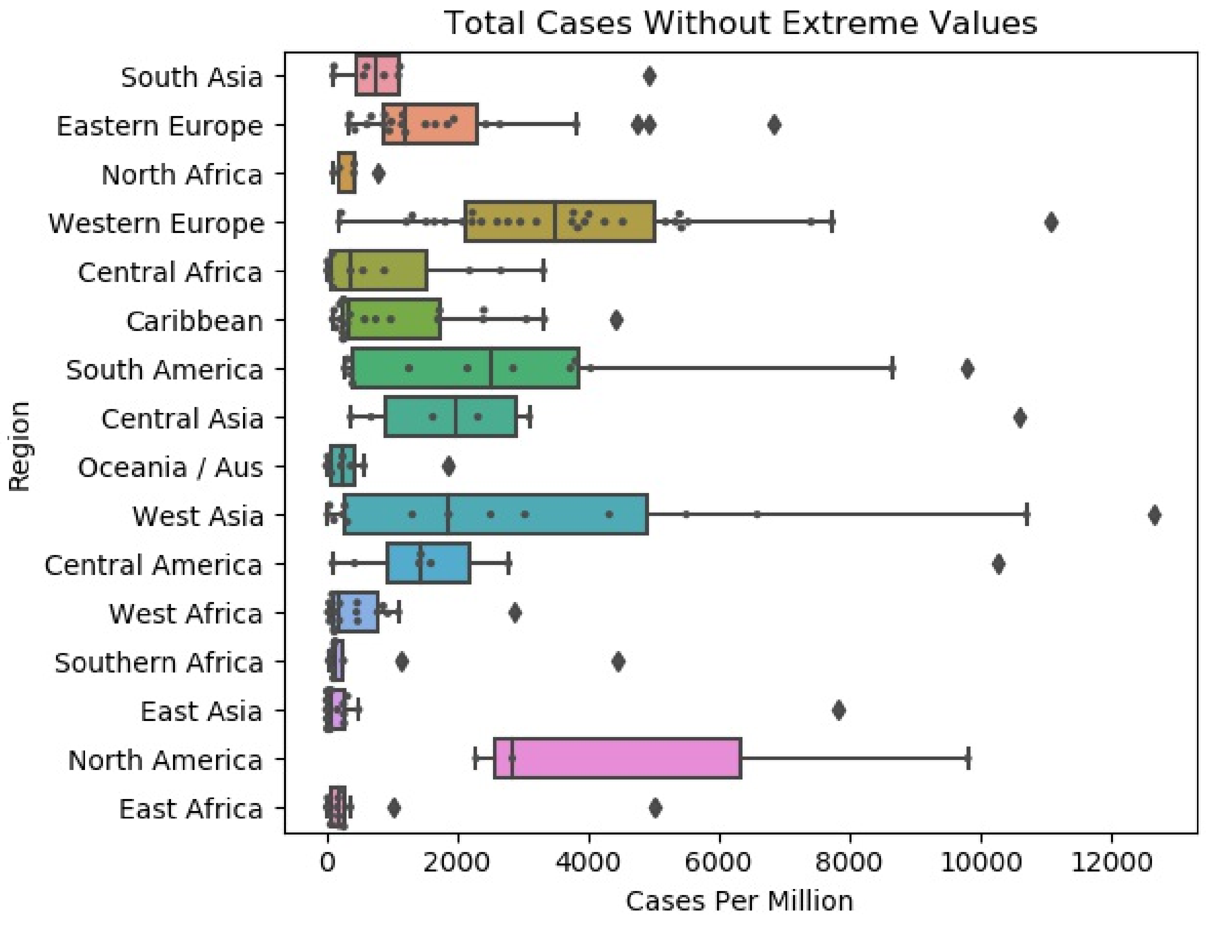


Figure 5.11: Boxplots of cases per million by region without the extreme values

These grouped boxplots reveal how much the distribution of cases, adjusted by population, varies by region.

How it works...

We use seaborn for the figures we create in this recipe. We could have also used matplotlib. Seaborn is actually built on top of matplotlib, extending it in some areas, and making some things easier. It sometimes produces more aesthetically pleasing figures with the default settings than matplotlib does.It is a good idea to have some descriptives in front of us before creating figures with multiple boxplots. In *Step 2*, we get the first and third quartile values, and the median, for each degree attainment level. We do this by first creating a function called `gettots`, which returns a series with those values. We apply `gettots` to each group in the data frame with the following statement:

```
nls97.groupby(['highestdegree'])['weeksworked17'].apply(gettots)
.unstack()
```

The `groupby` method creates a data frame with grouping information, which is passed to the `apply` function. `gettots` then calculates summary values for each group. `unstack` reshapes the returned rows, from multiple rows per group (one for each summary statistic) to one row per group, with columns for each summary statistic.In *Step 3*, we generate a boxplot for each degree attainment level. We do not normally need to name the subplot object we create when we use seaborn's `boxplot` method. We do so in this step, naming it `myplt`, so that we can easily change attributes—such as tick labels—later. We rotate the labels on the *x* axis using `set_xticklabels` so that the labels do not run into each other.We flip the axes for the boxplots in *Step 5* since there are more group levels (regions) than there are ticks for the continuous variable, cases per million. We do that by making `total_cases_pm` the value for the first argument, rather than the second. We also do a swarm plot to give some sense of the number of observations (countries) in each region.Extreme values can sometimes make it difficult to view a boxplot. Boxplots show both the outliers and the IQR, but the IQR rectangle will be so small that it is not viewable when outliers are several times the third or first quartile value. In *Step 5*, we remove all values of `total_cases_pm` greater than or equal to 14000. This improves the presentation of each IQR.

There's more...

The boxplots of weeks worked by educational attainment in *Step 3* reveal high variation in weeks worked, something that is not obvious in univariate analysis. The lower the educational attainment level, the greater the spread in weeks worked. There is substantial variability in weeks worked in 2017 for individuals with less than a high school degree, and very little variability for individuals with college degrees.This is quite relevant, of course, to our understanding of what is an outlier in terms of weeks worked. For example, someone with a college degree who worked 20 weeks is an outlier, but they would not be an outlier if they had less than a high school diploma.The `Cases Per Million` boxplots also invite us to think more flexibly about what an outlier is. For example, none of the outliers for cases per million in East Africa would have been identified as an outlier in the dataset as a whole. In addition, those values are all lower than the third quartile value for North America. But they definitely are outliers for East Africa.One of the first things I notice when looking at a boxplot is where the median is in the IQR. When the median is not at all close to the center, I know I am not dealing with a normally distributed variable. It also gives me a good sense of the direction of the skew. If it is near the bottom of the IQR, meaning that the median is much closer to the first quartile than the third, then there is positive skew. Compare the boxplot for the Caribbean to that of Western Europe. A large number of low values and a few high values bring the median close to the first quartile value for the Caribbean.

See also

We work much more with `groupby` in *Chapter 7*, *Fixing Messy Data When Aggregating*. We work more with `stack` and `unstack` in *Chapter 9*, *Tidying and Reshaping Data*.

Examining both distribution shape and outliers with violin plots

Violin plots combine histograms and boxplots in one plot. They show the IQR, median, and whiskers, as well as the frequency of observations at all ranges of values. It is hard to visualize how that is possible without seeing an actual violin plot. We generate a few violin plots on the same data we used for boxplots in the previous recipe, to make it easier to grasp how they work.

Getting ready

We will work with the NLS and the Covid case data. You need matplotlib and seaborn installed on your computer to run the code in this recipe.

How to do it...

We do violin plots to view both the spread and shape of the distribution on the same graphic. We then do violin plots by groups.

1. Load pandas, matplotlib, and seaborn, and the Covid case and NLS data:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
nls97 = pd.read_csv("data/nls97.csv")
nls97.set_index("personid", inplace=True)
covidtotals = pd.read_csv("data/covidtotals.csv", parse_dates=["
lastdate"])
covidtotals.set_index("iso_code", inplace=True)
```

1. Do a violin plot of the SAT verbal score:

```
sns.violinplot(nls97.satverbal, color="wheat", orient="v")
plt.title("Violin Plot of SAT Verbal Score")
plt.ylabel("SAT Verbal")
plt.text(0.08, 780, "outlier threshold", horizontalalignment='ce
nter', size='x-small')
plt.text(0.065, nls97.satverbal.quantile(0.75), "3rd quartile",
horizontalalignment='center', size='x-small')
plt.text(0.05, nls97.satverbal.median(), "Median", horizontalali
gnment='center', size='x-small')
plt.text(0.065, nls97.satverbal.quantile(0.25), "1st quartile",
horizontalalignment='center', size='x-small')
plt.text(0.08, 210, "outlier threshold", horizontalalignment='ce
nter', size='x-small')
plt.text(-0.4, 500, "frequency", horizontalalignment='center', s
ize='x-small')
plt.show()
```

This results in the following violin plot:

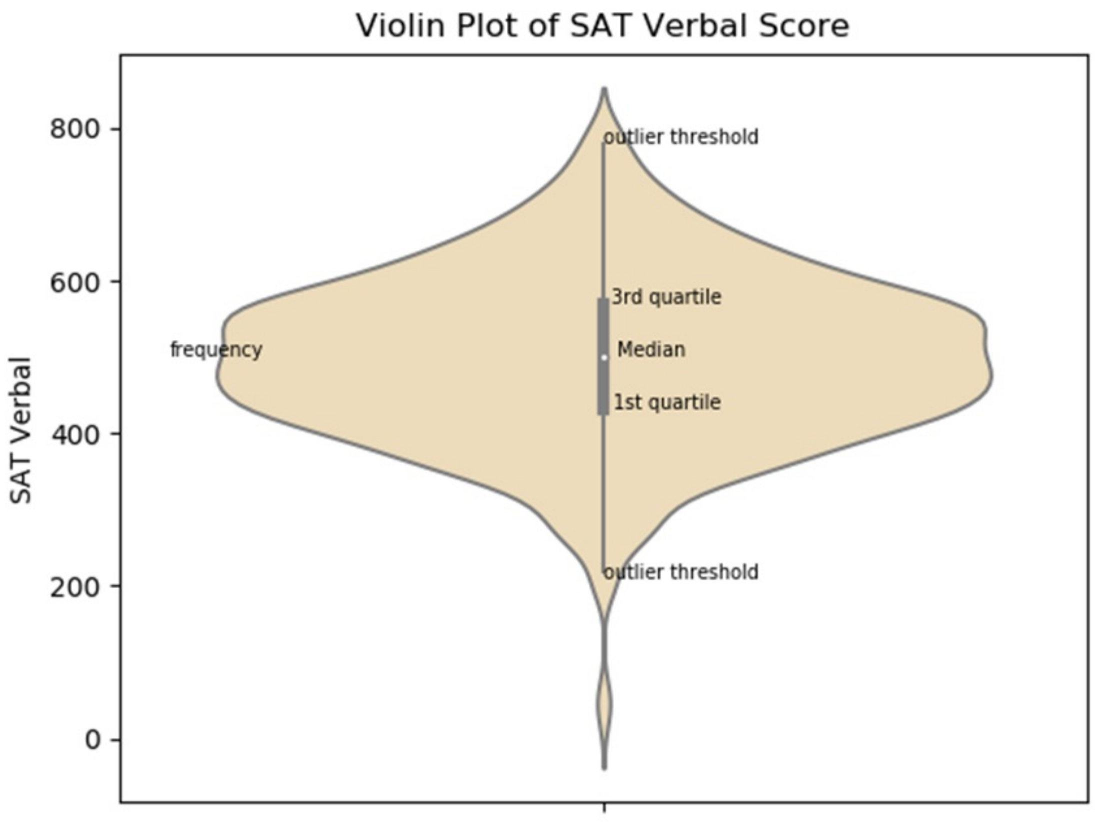


Figure 5.12: Violin plot of SAT verbal score with labels for IQR and outlier threshold

1. Get some descriptives for weeks worked:

```
nls97.loc[:, ['weeksworked16','weeksworked17']].describe()

       weeksworked16  weeksworked17
count          7,068          6,670
mean              39             39
std               21             19
min                0              0
25%               23             37
50%               53             49
75%               53             52
max               53             52
```

1. Show weeks worked for 2016 and 2017.

Use a more object-oriented approach to make it easier to access some axes attributes. Notice that the `weeksworked` distributions are bimodal, with bulges near the top and the bottom of the distribution. Also, note the very different IQR for 2016 and 2017:

```
myplt = sns.violinplot(data=nls97.loc[:, ['weeksworked16','weeks
worked17']])
myplt.set_title("Violin Plots of Weeks Worked")
myplt.set_xticklabels(["Weeks Worked 2016","Weeks Worked 2017"])
plt.show()
```

This results in the following violin plots:

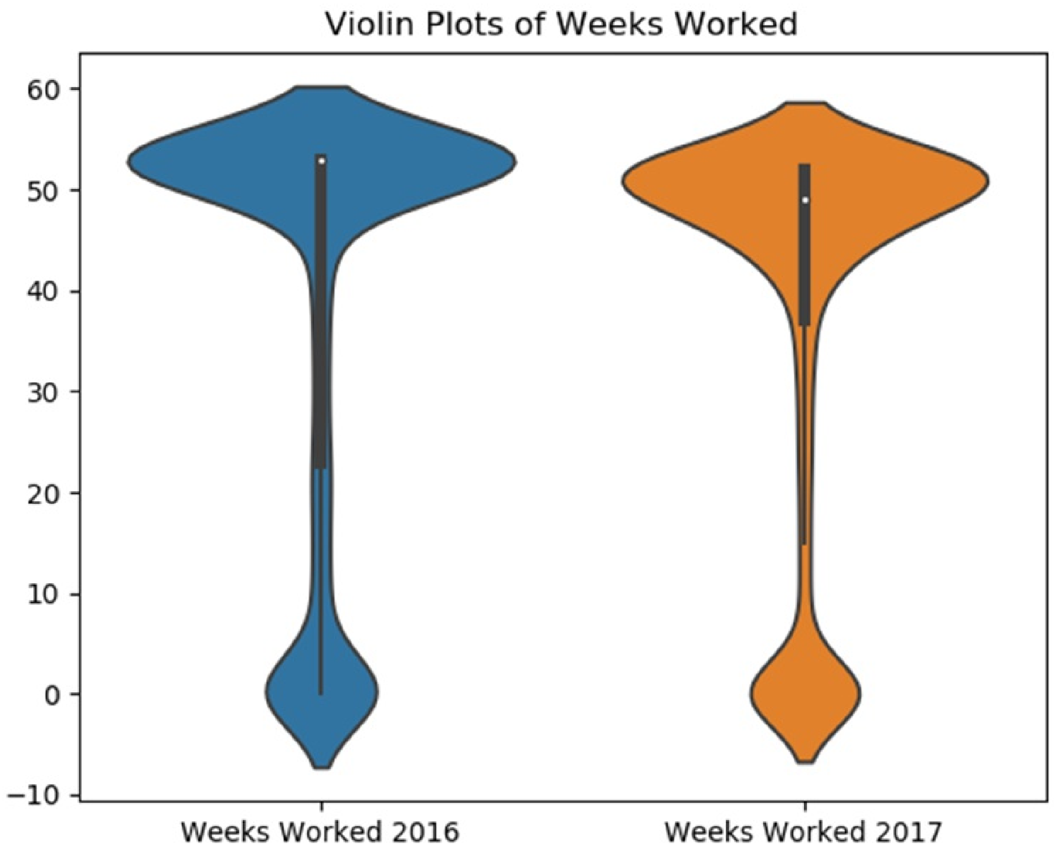


Figure 5.13: Violin plots showing spread and shape of distribution for two variables side by side

1. Do a violin plot of wage income by gender and marital status.

First, create a collapsed marital status column. Specify gender for the *x* axis, salary for the *y* axis, and a new collapsed marital status column for `hue`. The `hue` parameter is used for grouping, which will be added to any grouping already used for the *x* axis. We also indicate `scale="count"` to generate violin plots sized according to the number of observations in each category:

```
nls97["maritalstatuscollapsed"] = nls97.maritalstatus.\
...   replace(['Married','Never-married','Divorced','Separated',
'Widowed'],\
...   ['Married','Never Married','Not Married','Not Married','No
t Married'])
sns.violinplot(nls97.gender, nls97.wageincome, hue=nls97.marital
statuscollapsed, scale="count")
plt.title("Violin Plots of Wage Income by Gender and Marital Sta
tus")
plt.xlabel('Gender')
plt.ylabel('Wage Income 2017')
plt.legend(title="", loc="upper center", framealpha=0, fontsize=
8)
plt.tight_layout()
plt.show()
```

This results in the following violin plots:

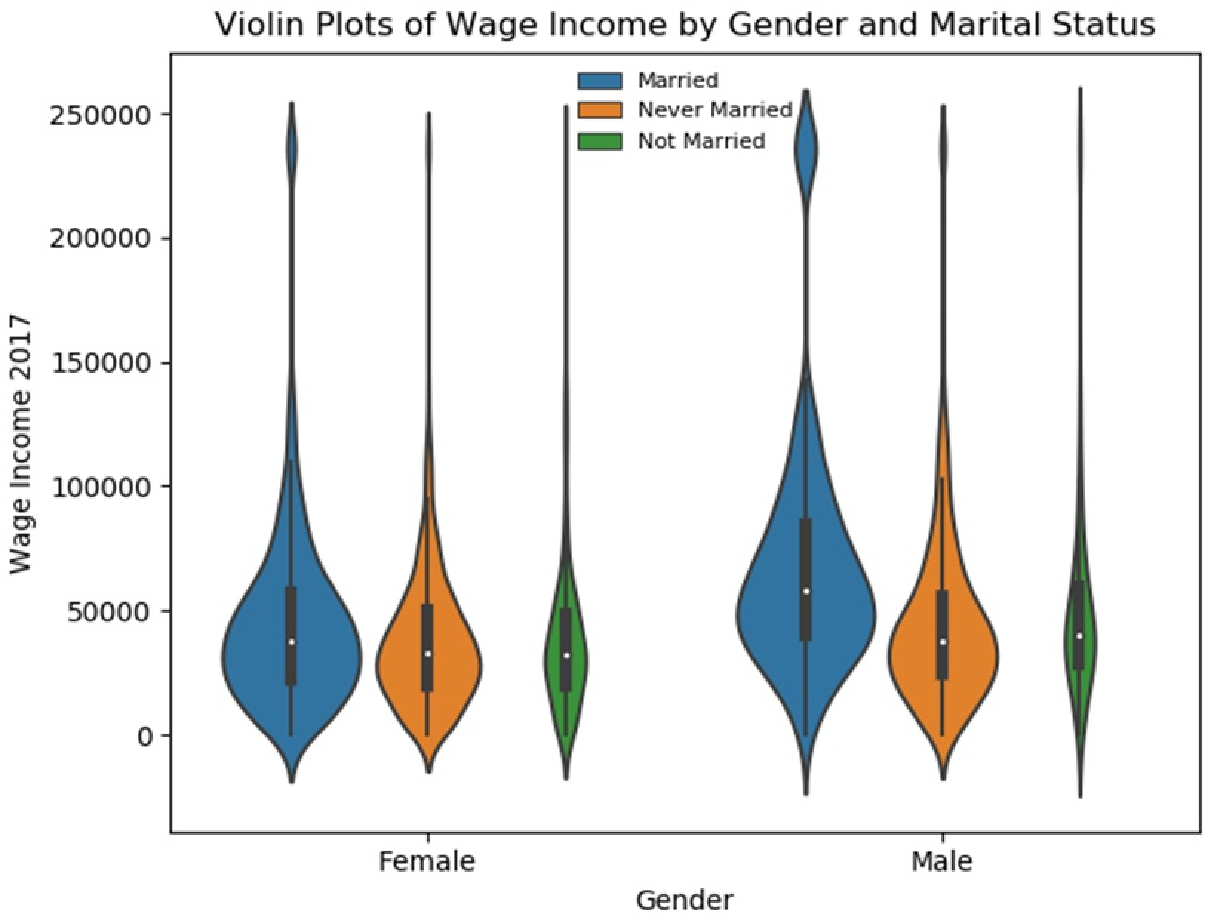


Figure 5.14: Violin plots showing spread and shape of distribution by two different groups

1. Do violin plots of weeks worked by highest degree attained:

```
myplt = sns.violinplot('highestdegree','weeksworked17', data=nls
97, rotation=40)
myplt.set_xticklabels(myplt.get_xticklabels(), rotation=60, hori
zontalalignment='right')
myplt.set_title("Violin Plots of Weeks Worked by Highest Degree"
)
myplt.set_xlabel('Highest Degree Attained')
myplt.set_ylabel('Weeks Worked 2017')
plt.tight_layout()
plt.show()
```

This results in the following violin plots:

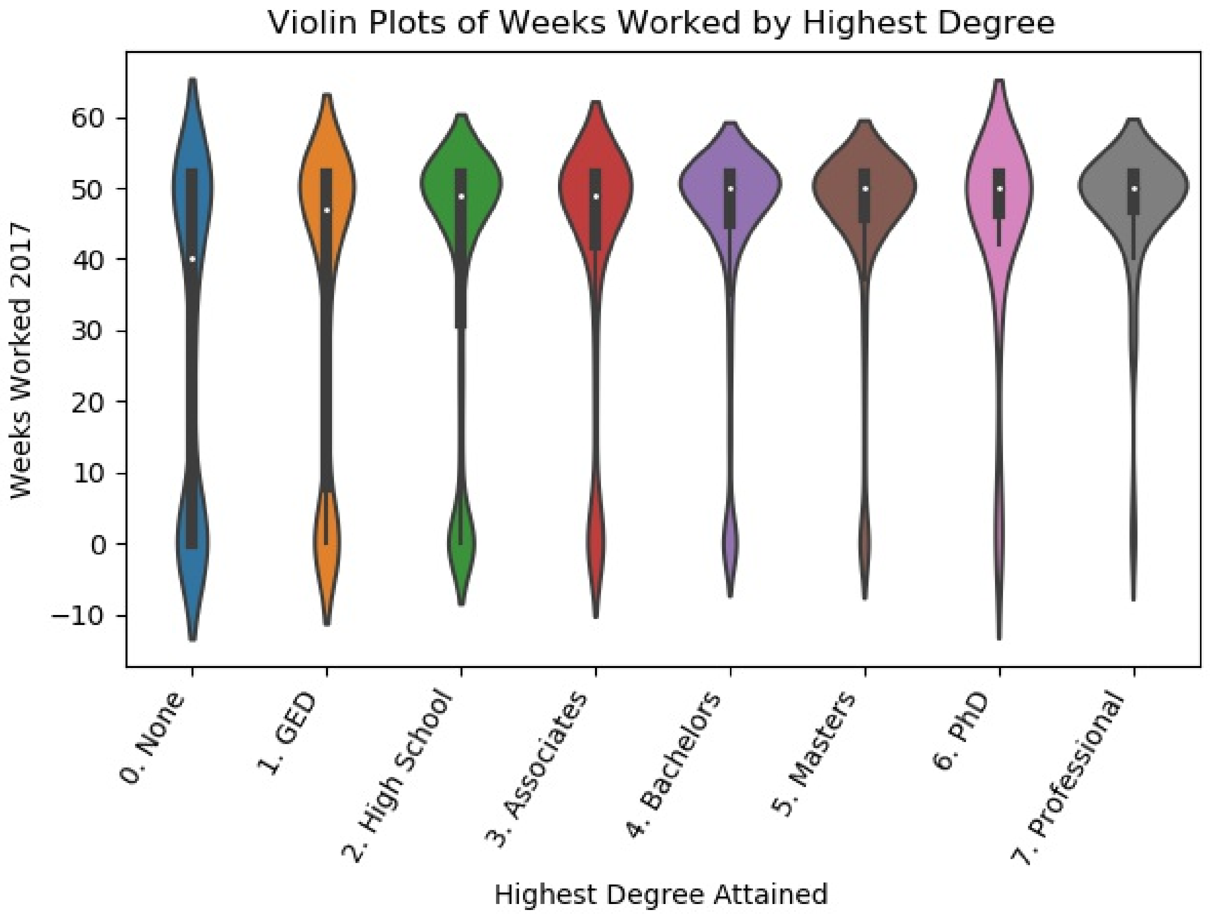


Figure 5.15: Violin plots showing spread and shape of distribution by group

These steps show just how much violin plots can tell us about how continuous variables in our DataFrame are distributed, and how that might vary by group.

How it works...

Similar to boxplots, violin plots show the median, first and third quartiles, and the whiskers. They also show the relative frequency of variable values. (When the violin plot is displayed vertically, the relative frequency is the width at a given point.) The violin plot produced in *Step 2*, and the associated annotations, provide a good illustration. We can tell from the violin plot that the distribution of SAT verbal scores is not dramatically different from normal, other than the extreme values at the lower end. The greatest bulge (greatest width) is at the median, declining fairly symmetrically from there. The median is relatively equidistant from the first and third quartiles.We can create a violin plot in seaborn by passing one or more data series to the `violinplot` method. We can also pass a whole data frame of one or more columns. We do that in *Step 4* because we want to plot more than one continuous variable.We sometimes need to experiment with the legend a bit to get it to be both informative and unobtrusive. In *Step 5*, we used the following command to remove the legend title (since it is clear from the values), locate it in the best place on the figure, and make the box transparent (`framealpha=0`):

```
plt.legend(title="", loc="upper center", framealpha=0, fontsize=
8)
```

We can pass data series to `violinplot` in a variety of ways. If you do not indicate an axis with `"x="` or `"y="`, or grouping with `"hue="`, seaborn will figure that out based on order. For example, in *Step 5*, we did the following:

```
sns.violinplot(nls97.gender, nls97.wageincome, hue=nls97.marital
statuscollapsed, scale="count")
```

We would have gotthe same results if we had done the following:

```
sns.violinplot(x=nls97.gender, y=nls97.wageincome, hue=nls97.mar
italstatuscollapsed, scale="count")
```

We could have also done this to obtain the same result:

```
sns.violinplot(y=nls97.wageincome, x=nls97.gender, hue=nls97.mar
italstatuscollapsed, scale="count")
```

Although I have highlighted this flexibility in this recipe, these techniques for sending data to matplotlib and seaborn apply to all of the plotting methods discussed in this chapter (though not all of them have a `hue` parameter).

There's more...

Once you get the hang of violin plots, you will appreciate the enormous amount of information they make available on one figure. We get a sense of the shape of the distribution, its central tendency, and its spread. We can also easily show that information for different subsets of our data.The distribution of weeks worked in 2016 is different enough from weeks worked in 2017 to give the careful analyst pause. The IQR is quite different—30 for 2016 (23 to 53), and 15 for 2017 (37 to 52).An unusual fact about the distribution of wage income is revealed when examining the violin plots produced in *Step 5*. There is a bunching-up of incomes at the top of the distribution for married males, and somewhat for married females. That is quite unusual for a wage income distribution. As it turns out, it looks like there is a ceiling on wage income of $235,884. This is something that we definitely want to take into account in future analyses that include wage income.The income distributions have a similar shape across gender and marital status, with bulges slightly below the median and extended positive tails. The IQRs have relatively similar lengths. However, the distribution for married males is noticeably higher (or to the right, depending on chosen orientation) than that for the other groups.The violin plots of weeks worked by degree attained show very different distributions by group, as we also discovered in the boxplots of the same data in the previous recipe. What is more clear here, though, is the bimodal nature of the distribution at lower levels of education. There is a bunching at low levels of weeks worked for individuals without college degrees. Individuals without high school diplomas or a **GED** (a **Graduate Equivalency Diploma**) were nearly as likely to work 5 or fewer weeks in 2017 as they were to work 50 or more weeks.We used seaborn exclusively to produce violin plots in this recipe. Violin plots can also be produced with matplotlib. However, the default graphics in matplotlib for violin plots look very different from those for seaborn.

See also

It might be helpful to compare the violin plots in this recipe to histograms, boxplots, and grouped boxplots in the previous recipes in this chapter.

Using scatter plots to view bivariate relationships

My sense is that there are few plots that data analysts rely more on than scatter plots, with the possible exception of histograms. We are all very used to looking at relationships that can be illustrated in two dimensions. Scatter plots capture important real-world phenomena (the relationship between variables) and are quite intuitive for most people. This makes them a valuable addition to our visualization toolkit.

Getting ready

You will need matplotlib and seaborn for this recipe. We will be working with the `landtemps` dataset, which provides the average temperature in 2019 for 12,095 weather stations across the world.

How to do it…

We level up our scatter plot skills from the previous chapter and visualize more complicated relationships. We display the relationship between average temperature, latitude, and elevation by showing multiple scatter plots on one chart, creating 3D scatter plots, and showing multiple regression lines.

1. Load pandas, NumPy, matplotlib, the Axes3D module, and seaborn:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns
landtemps = pd.read_csv("data/landtemps2019avgs.csv")
```

1. Run a scatter plot of latitude (`latabs`) by average temperature:

```
plt.scatter(x="latabs", y="avgtemp", data=landtemps)
plt.xlabel("Latitude (N or S)")
plt.ylabel("Average Temperature (Celsius)")
plt.yticks(np.arange(-60, 40, step=20))
plt.title("Latitude and Average Temperature in 2019")
plt.show()
```

This results in the following scatter plot:

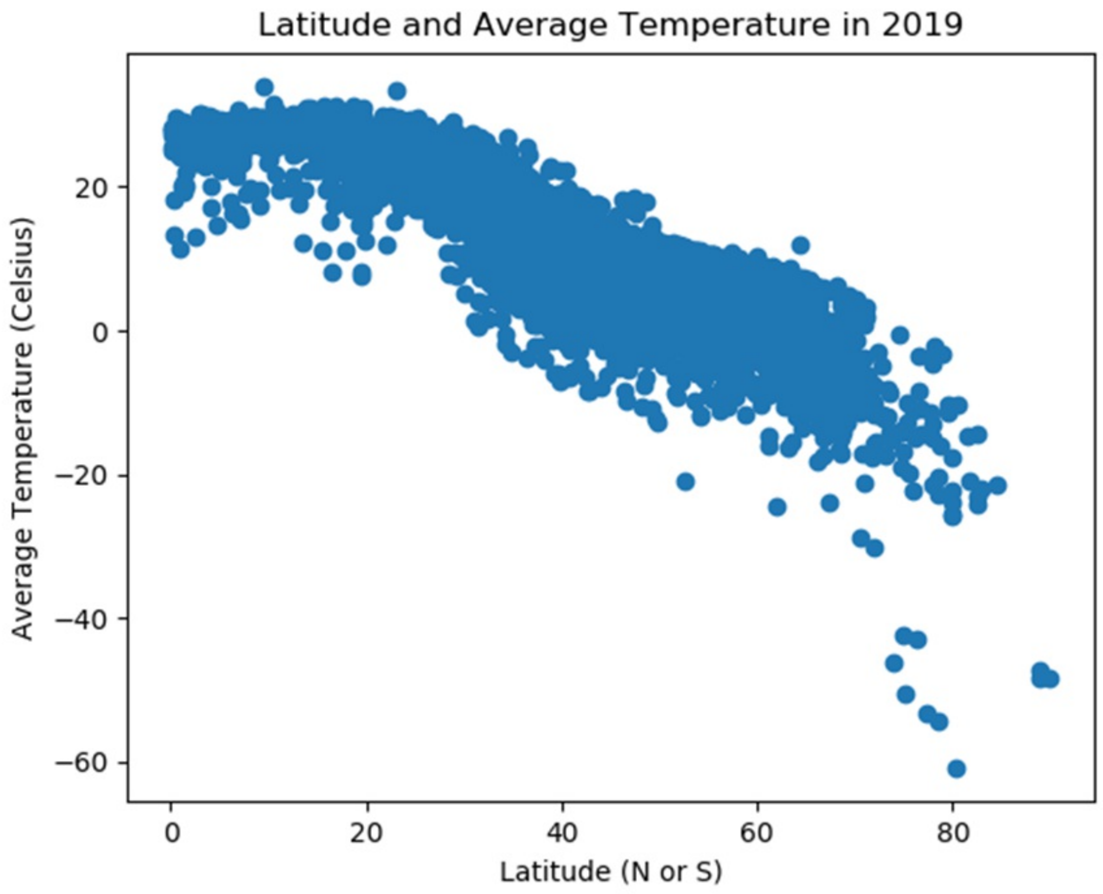


Figure 5.16: Scatter plot of latitude by average temperature

1. Show the high elevation points in red.

Create low and high elevation data frames. Notice that the high elevation points are generally lower (that is, cooler) on the figure at each latitude:

```
low, high = landtemps.loc[landtemps.elevation<=1000], landtemps.
loc[landtemps.elevation>1000]
plt.scatter(x="latabs", y="avgtemp", c="blue", data=low)
plt.scatter(x="latabs", y="avgtemp", c="red", data=high)
plt.legend(('low elevation', 'high elevation'))
plt.xlabel("Latitude (N or S)")
plt.ylabel("Average Temperature (Celsius)")
plt.title("Latitude and Average Temperature in 2019")
plt.show()
```

This results in the following scatter plot:

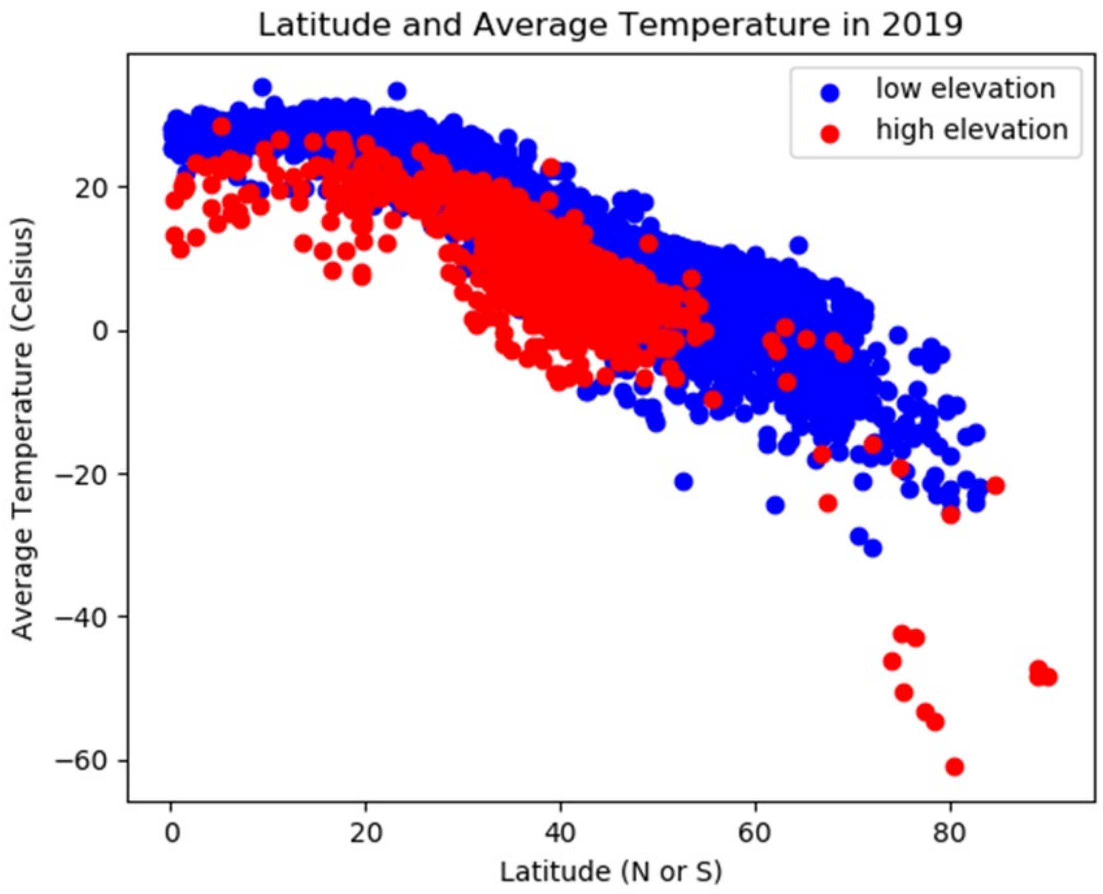


Figure 5.17: Scatter plot of latitude by average temperature and elevation

1. View a three-dimensional plot of temperature, latitude, and elevation.

It looks like there is a somewhat steeper decline in temperature, with increases in latitude for high elevation stations:

```
fig = plt.figure()
plt.suptitle("Latitude, Temperature, and Elevation in 2019")
ax.set_title('Three D')
ax = plt.axes(projection='3d')
ax.set_xlabel("Elevation")
ax.set_ylabel("Latitude")
ax.set_zlabel("Avg Temp")
ax.scatter3D(low.elevation, low.latabs, low.avgtemp, label="low
elevation", c="blue")
ax.scatter3D(high.elevation, high.latabs, high.avgtemp, label="h
igh elevation", c="red")
```

```
ax.legend()
plt.show()
```

This results in the following scatter plot:

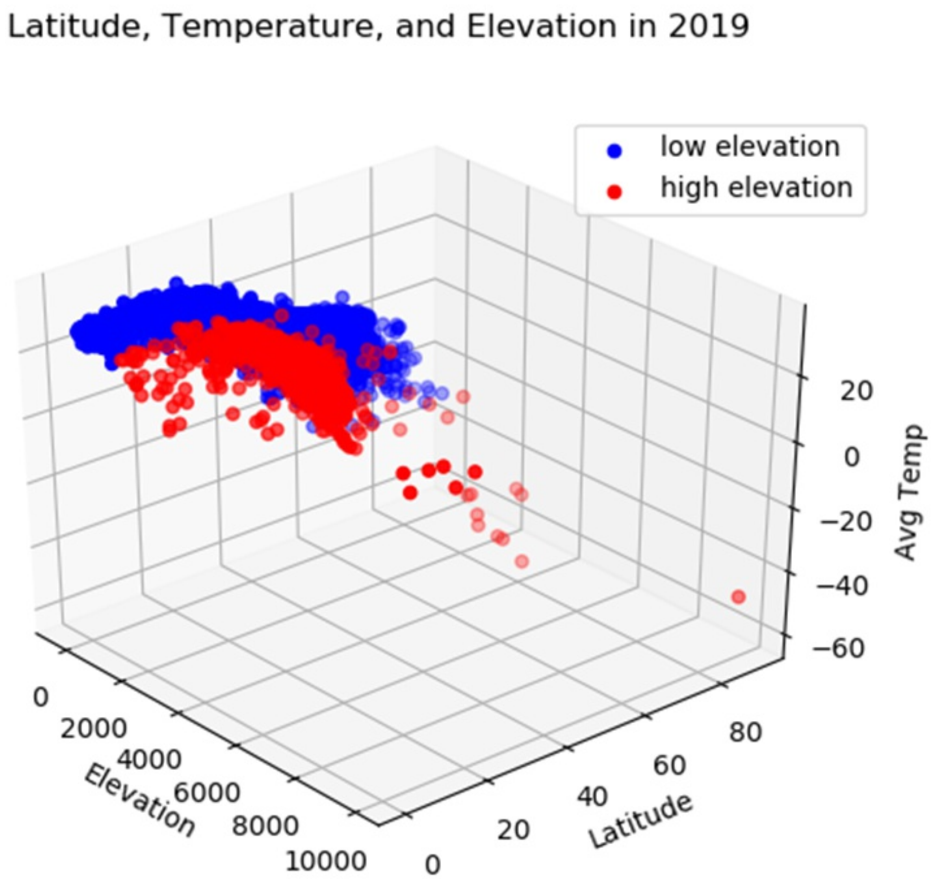


Figure 5.18: 3D scatter plot of latitude and elevation by average temperature

1. Show a regression line of latitude on temperature.

Use `regplot` to get a regression line:

```
sns.regplot(x="latabs", y="avgtemp", color="blue", data=landtemp
s)
plt.title("Latitude and Average Temperature in 2019")
plt.xlabel("Latitude (N or S)")
plt.ylabel("Average Temperature")
plt.show()
```

This results in the following scatter plot:

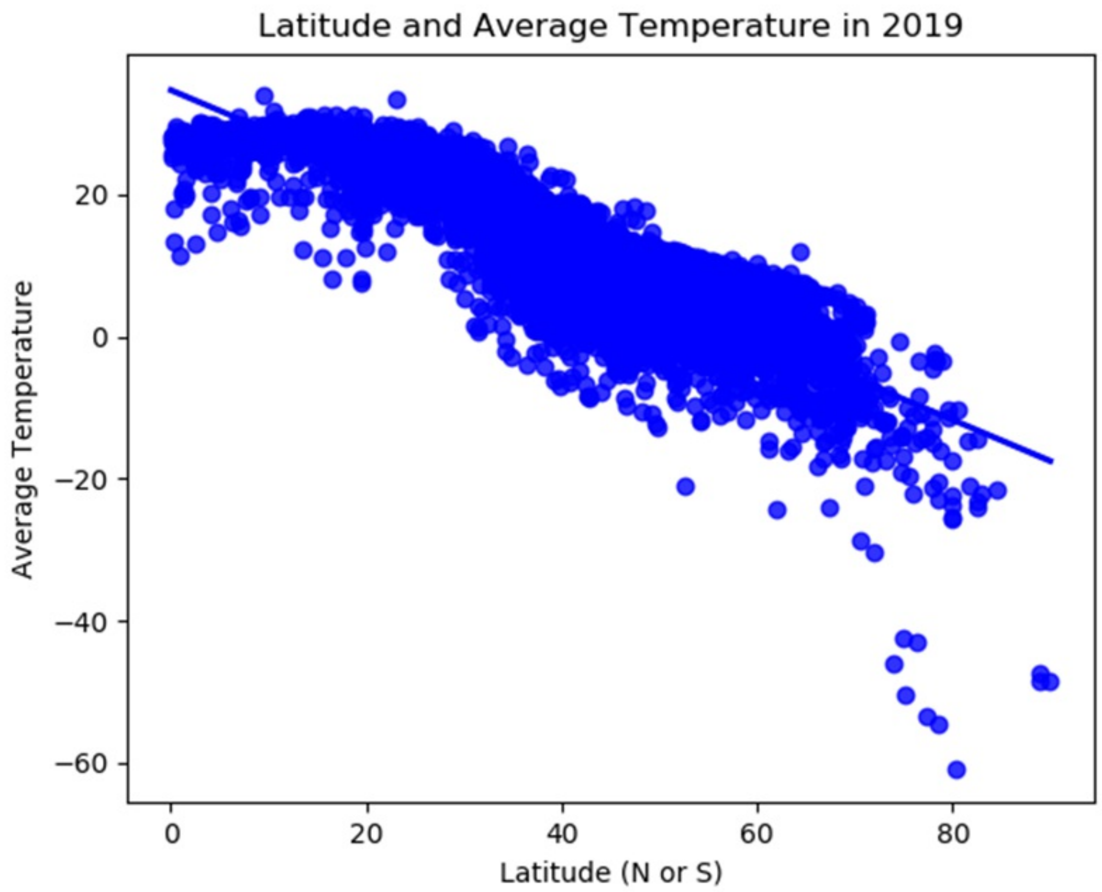


Figure 5.19: Scatter plot of latitude by average temperature with regression line

1. Show separate regression lines for low and high elevation stations.

We use `lmplot` this time instead of `regplot`. The two methods have similar functionality. Unsurprisingly, high elevation stations appear to have both lower intercepts (where the line crosses the *y* axis) and steeper negative slopes:

```
landtemps['elevation_group'] = np.where(landtemps.elevation<=100
0,'low','high')
sns.lmplot(x="latabs", y="avgtemp", hue="elevation_group", palet
te=dict(low="blue", high="red"), legend_out=False, data=landtemp
s)
```

```
plt.xlabel("Latitude (N or S)")
plt.ylabel("Average Temperature")
plt.legend(('low elevation', 'high elevation'), loc='lower left'
)
plt.yticks(np.arange(-60, 40, step=20))
plt.title("Latitude and Average Temperature in 2019")
plt.tight_layout()
plt.show()
```

This results in the following scatter plot:

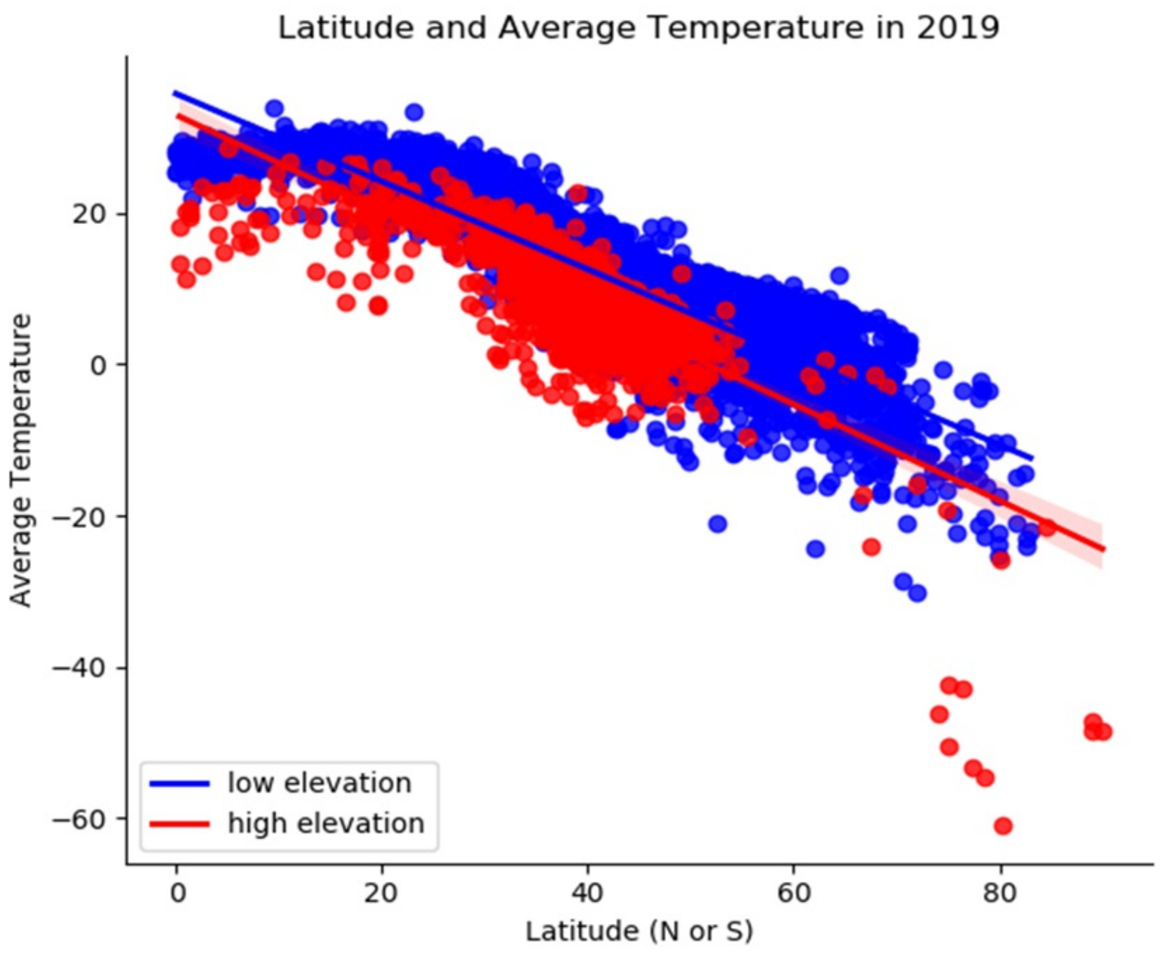


Figure 5.20: Scatter plot of latitude by temperature with separate regression lines for elevation

1. Show some stations above the low and high elevation regression lines:

```
high.loc[(high.latabs>38) & \
...   (high.avgtemp>=18),
...   ['station','country','latabs',
...   'elevation','avgtemp']]

    station    country    latabs  \
3943    LAJES_AB    Portugal    39
5805    WILD_HORSE_6N      United States    39
      elevation  avgtemp
3943      1,016       18
5805      1,439       23

low.loc[(low.latabs>47) & \
...   (low.avgtemp>=14),
...   ['station','country','latabs',
...   'elevation','avgtemp']]

     station    country    latabs  \
1048    SAANICHTON_CDA    Canada    49
1146    CLOVERDALE_EAST    Canada    49
6830    WINNIBIGOSHISH_DAM    United States    47
7125    WINIFRED    United States    48
     elevation    avgtemp
1048    61    18
1146    50    15
6830    401    18
7125    988    16
```

1. Show some stations below the low and high elevation regression lines:

```
high.loc[(high.latabs<5) & \
...   (high.avgtemp<18),
...   ['station','country','latabs',
...   'elevation','avgtemp']]

     station    country    latabs  elevation  \
2250    BOGOTA_ELDORADO    Colombia    5    2,548
2272    SAN_LUIS    Colombia    1    2,976
2303    IZOBAMBA    Ecuador    0    3,058
2306    CANAR    Ecuador    3    3,083
2307    LOJA_LA_ARGELIA    Ecuador    4    2,160
      avgtemp
2250       15
2272       11
2303       13
2306       13
2307       17

low.loc[(low.latabs<50) & \
...   (low.avgtemp<-9),
...   ['station','country','latabs',
...   'elevation','avgtemp']]

     station    country    latabs  \
1189    FT_STEELE_DANDY_CRK    Canada    50
1547    BALDUR    Canada    49
1833    POINTE_CLAVEAU    Canada    48
1862    CHUTE_DES_PASSES    Canada    50
6544    PRESQUE_ISLE    United States    47
     elevation    avgtemp
1189    856    -12
1547    450    -11
1833    4    -11
1862    398    -13
6544    183    -10
```

Scatter plots are a great way to view the relationship between two variables. These steps also show how we can display that relationship for different subsets of our data.

How it works...

We can run a scatter plot by just providing column names for `x` and `y` and a DataFrame. Nothing more is required. We get the same access to the attributes of the figure and its axes that we get when we run histograms and boxplots—titles, axis labels, tick marks and labels, and so on. Note that to access attributes such as labels on an axis (rather than on the figure), we use `set_xlabels` or `set_ylabels`, not `xlabels` or `ylabels`.3D plots are a little more complicated. First, we need to have imported the `axes3d` module. Then, we set the projection of our axes to `3d` — `plt.axes(projection='3d')`, as we do in *Step 4*. We can then use the `scatter3D` method for each subplot.Since scatter plots are designed to illustrate the relationship between a regressor (the `x` variable) and a dependent variable, it is quite helpful to see a least-squares regression line on the scatter plot. Seaborn provides two methods for doing that: `regplot` and `lmplot`. I use `regplot` typically, since it is less resource-intensive. But sometimes, I need the features of `lmplot`. We use `lmplot` and its `hue` attribute in *Step 6* to generate separate regression lines for each elevation level.In *Steps 7* and *8*, we view some of the outliers: those stations with temperatures much higher or lower than the regression line for their group.

We would want to investigate the data for the `LAJES_AB` station in Portugal and the `WILD_HORSE_6N` station in the United States (`(high.latabs>38) & (high.avgtemp>=18)`). The average temperatures are higher than would be predicted at the latitude and elevation level. Similarly, there are four stations in Canada and one in the United States that are at low elevation and have lower average temperatures than would be expected (`low.latabs<50) & (low.avgtemp<-9)`).

There's more...

We see the expected relationship between latitude and average temperatures. Temperatures fall as latitude increases. But elevation is another important factor. Being able to visualize all three variables at once helps us identify outliers more easily. Of course, there are additional factors that matter for temperatures, such as warm ocean currents. That data is not in this dataset, unfortunately.Scatter plots are great for visualizing the relationship between two continuous variables. With some tweaking, matplotlib's and seaborn's scatter plot tools can also provide some sense of relationships between three variables—by adding a third dimension, creative use of colors (when the third dimension is categorical), or changing the size of the dots (the *Using linear regression to identify data points with high influence* recipe in *Chapter 4, Identifying Missing Values and Outliers in Subsets of Data*, provides an example of that).

See also

This is a chapter on visualization, and identifying unexpected values through visualizations. But these figures also scream out for the kind of multivariate analyses we did in *Chapter 4, Identifying missing values and outliers in Subsets of Data*. In particular, linear regression analysis, and a close look at the residuals, would be useful for identifying outliers.

Using line plots to examine trends in continuous variables

A typical way to visualize values for a continuous variable over regular intervals of time is through a line plot, though sometimes bar charts are used for small numbers of intervals. We will use line plots in this recipe to display variable trends, and examine sudden deviations in trends and differences in values over time by groups.

Getting ready

We will work with daily Covid case data in this recipe. In previous recipes, we have used totals by country. The daily data provides us with the number of new cases and new deaths each day by country, in addition to the same demographic variables we used in other recipes. You will need matplotlib installed to run the code in this recipe.

How to do it...

We use line plots to visualize trends in daily coronavirus cases and deaths. We create line plots by region, and stacked plots to get a better sense of how much one country can drive the number of cases for a whole region.

1. Import pandas, matplotlib, and the matplotlib `dates` and date formatting utilities:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from matplotlib.dates import DateFormatter
coviddaily = pd.read_csv("data/coviddaily720.csv", parse_dates=[
"casedate"])
```

1. View a couple of rows of the Covid daily data:

```
coviddaily.sample(2, random_state=1).T

                             2478             9526
iso_code                      BRB              FRA
casedate               2020-06-11       2020-02-16
location                 Barbados           France
continent           North America           Europe
new_cases                       4                0
new_deaths                      0                0
```

```
population             287,371      65,273,512
pop_density                664             123
median_age                  40              42
gdp_per_capita          16,978          38,606
hosp_beds                    6               6
region               Caribbean  Western Europe
```

1. Calculate new cases and deaths by day.

Select dates between 2020-02-01 and 2020-07-12, and then use `groupby` to summarize cases and deaths across all countries for each day:

```
coviddailytotals = coviddaily.loc[coviddaily.casedate.between('2
020-02-01','2020-07-12')].\
...   groupby(['casedate'])[['new_cases','new_deaths']].\
...   sum().\
...   reset_index()
coviddailytotals.sample(7, random_state=1)

       casedate  new_cases  new_deaths
44   2020-03-16     12,386         757
47   2020-03-19     20,130         961
94   2020-05-05     77,474       3,998
78   2020-04-19     80,127       6,005
160  2020-07-10    228,608       5,441
11   2020-02-12      2,033          97
117  2020-05-28    102,619       5,168
```

1. Show line plots for new cases and new deaths by day.

Show cases and deaths on different subplots:

```
fig = plt.figure()
plt.suptitle("New Covid Cases and Deaths By Day Worldwide in 202
0")
ax1 = plt.subplot(2,1,1)
ax1.plot(coviddailytotals.casedate, coviddailytotals.new_cases)
ax1.xaxis.set_major_formatter(DateFormatter("%b"))
ax1.set_xlabel("New Cases")
ax2 = plt.subplot(2,1,2)
ax2.plot(coviddailytotals.casedate, coviddailytotals.new_deaths)
ax2.xaxis.set_major_formatter(DateFormatter("%b"))
ax2.set_xlabel("New Deaths")
plt.tight_layout()
fig.subplots_adjust(top=0.88)
plt.show()
```

This results in the following line plots:

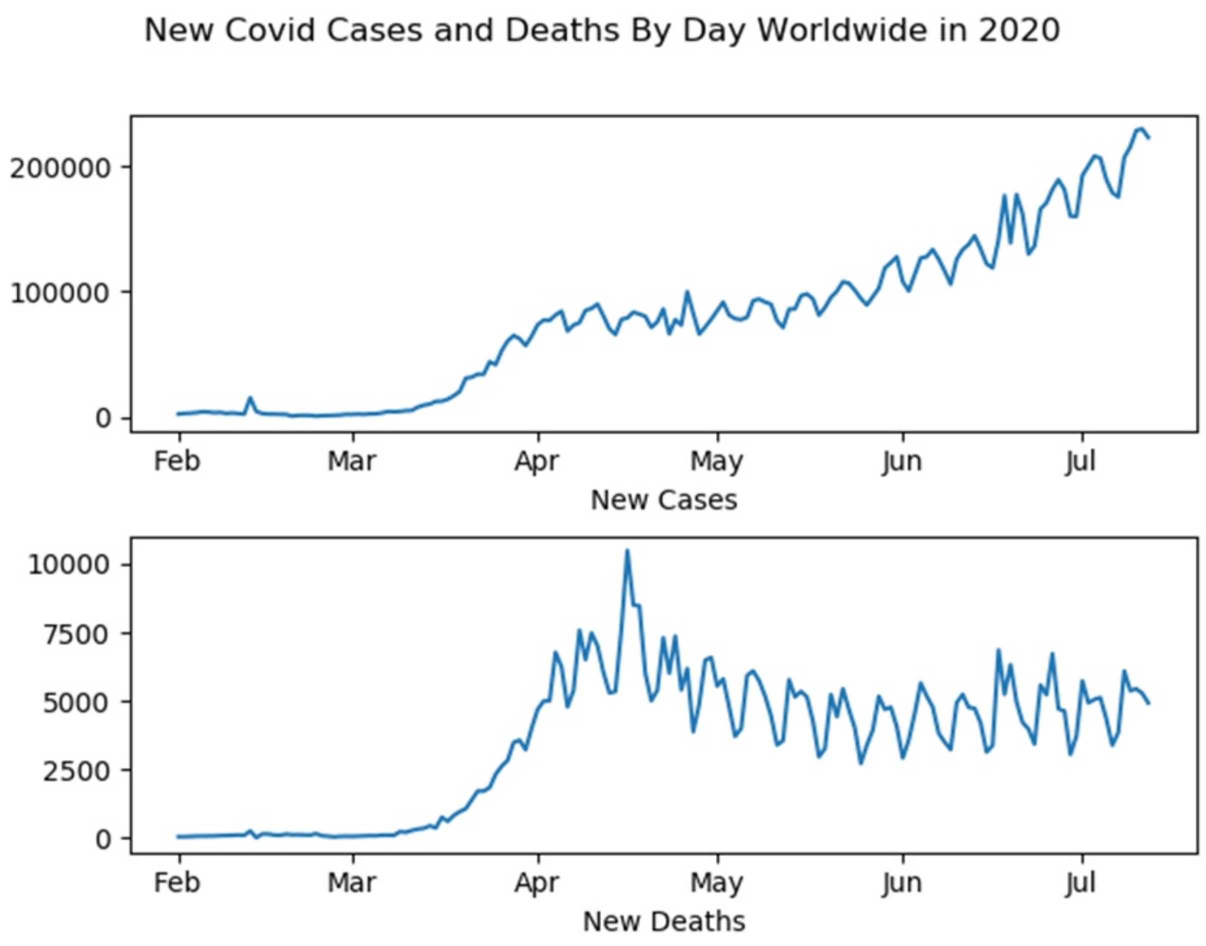


Figure 5.21: Daily trend lines of worldwide Covid cases and deaths

1. Calculate new cases and deaths by day and region:

```
regiontotals = coviddaily.loc[coviddaily.casedate.between('2020-
02-01','2020-07-12')].\
...   groupby(['casedate','region'])[['new_cases','new_deaths']]
.\
...   sum().\
...   reset_index()
regiontotals.sample(7, random_state=1)

       casedate          region  new_cases  new_deaths
1518 2020-05-16    North Africa        634          28
2410 2020-07-11    Central Asia      3,873          26
870  2020-04-05  Western Europe     30,090       4,079
```

```
1894 2020-06-08  Western Europe      3,712         180
790  2020-03-31  Western Europe     30,180       2,970
2270 2020-07-02    North Africa      2,006          89
306  2020-02-26   Oceania / Aus          0           0
```

1. Show line plots of new cases by selected regions.

Loop through the regions in `showregions`. Do a line plot of the total `new_cases` by day for each region. Use the `gca` method to get the *x* axis and set the date format:

```
showregions = ['East Asia','Southern Africa',
...   'North America','Western Europe']
for j in range(len(showregions)):
...   rt = regiontotals.loc[regiontotals.\
...     region==showregions[j],
...     ['casedate','new_cases']]
...   plt.plot(rt.casedate, rt.new_cases,
...     label=showregions[j])

plt.title("New Covid Cases By Day and Region in 2020")
plt.gca().get_xaxis().set_major_formatter(DateFormatter("%b"))
plt.ylabel("New Cases")
plt.legend()
plt.show()
```

This results in the following line plots:

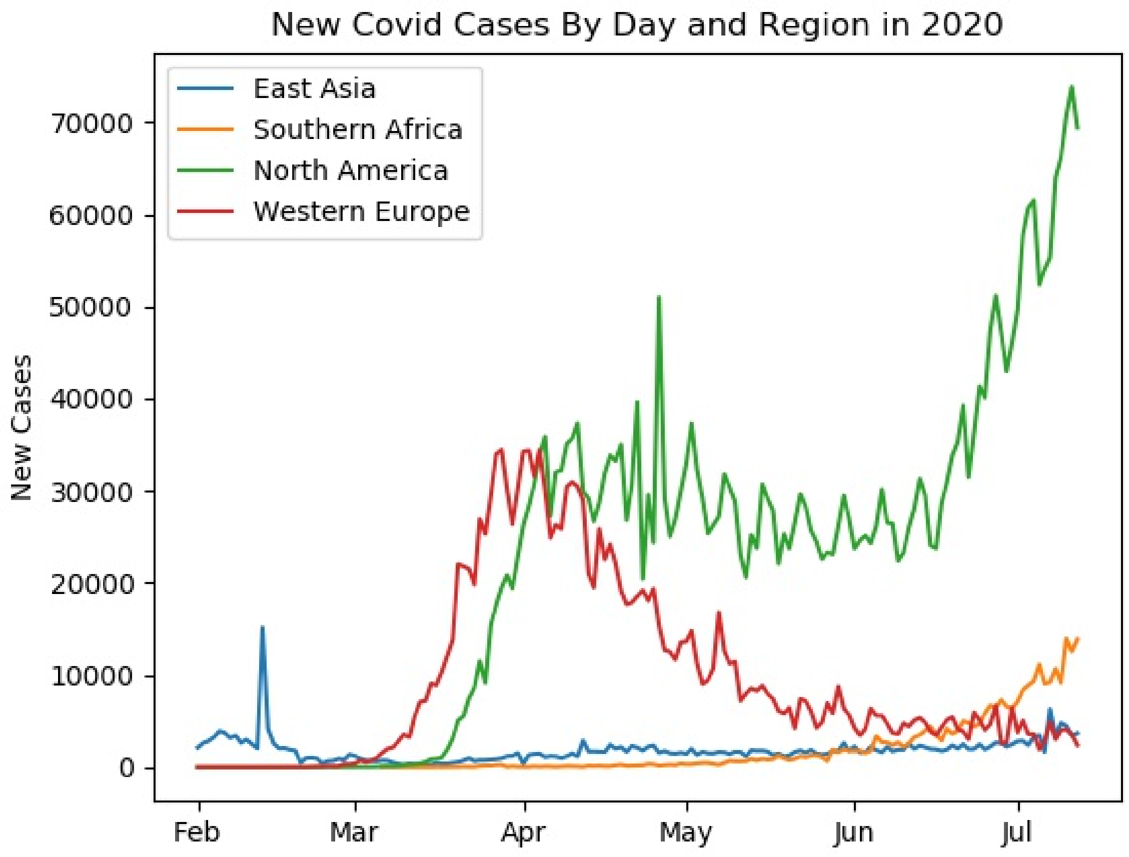


Figure 5.22: Daily trend lines of Covid cases by region

1. Use a stacked plot to examine the uptick in Southern Africa more closely.

See if one country (South Africa) in Southern Africa is driving the trend line. Create a DataFrame (`af`) for `new_cases` by day for Southern Africa (the region). Add a series for `new_cases` in South Africa (the country) to the `af` DataFrame. Then, create a new series in the `af` DataFrame for Southern Africa cases minus South Africa cases (`afcasesnosa`). Select only data in April or later, since that is when we start to see an increase in new cases:

```
af = regiontotals.loc[regiontotals.\
...   region=='Southern Africa',
...   ['casedate','new_cases']].\
...   rename(columns={'new_cases':'afcases'})
sa = coviddaily.loc[coviddaily.\
...   location=='South Africa',
...   ['casedate','new_cases']].\
...   rename(columns={'new_cases':'sacases'})
af = pd.merge(af, sa, left_on=['casedate'], right_on=['casedate'
], how="left")
af.sacases.fillna(0, inplace=True)
af['afcasesnosa'] = af.afcases-af.sacases
afabb = af.loc[af.casedate.between('2020-04-01','2020-07-12')]
fig = plt.figure()
ax = plt.subplot()
ax.stackplot(afabb.casedate, afabb.sacases, afabb.afcasesnosa, l
abels=['South Africa','Other Southern Africa'])
ax.xaxis.set_major_formatter(DateFormatter("%m-%d"))
plt.title("New Covid Cases in Southern Africa")
plt.tight_layout()
plt.legend(loc="upper left")
plt.show()
```

This results in the following stacked plot:

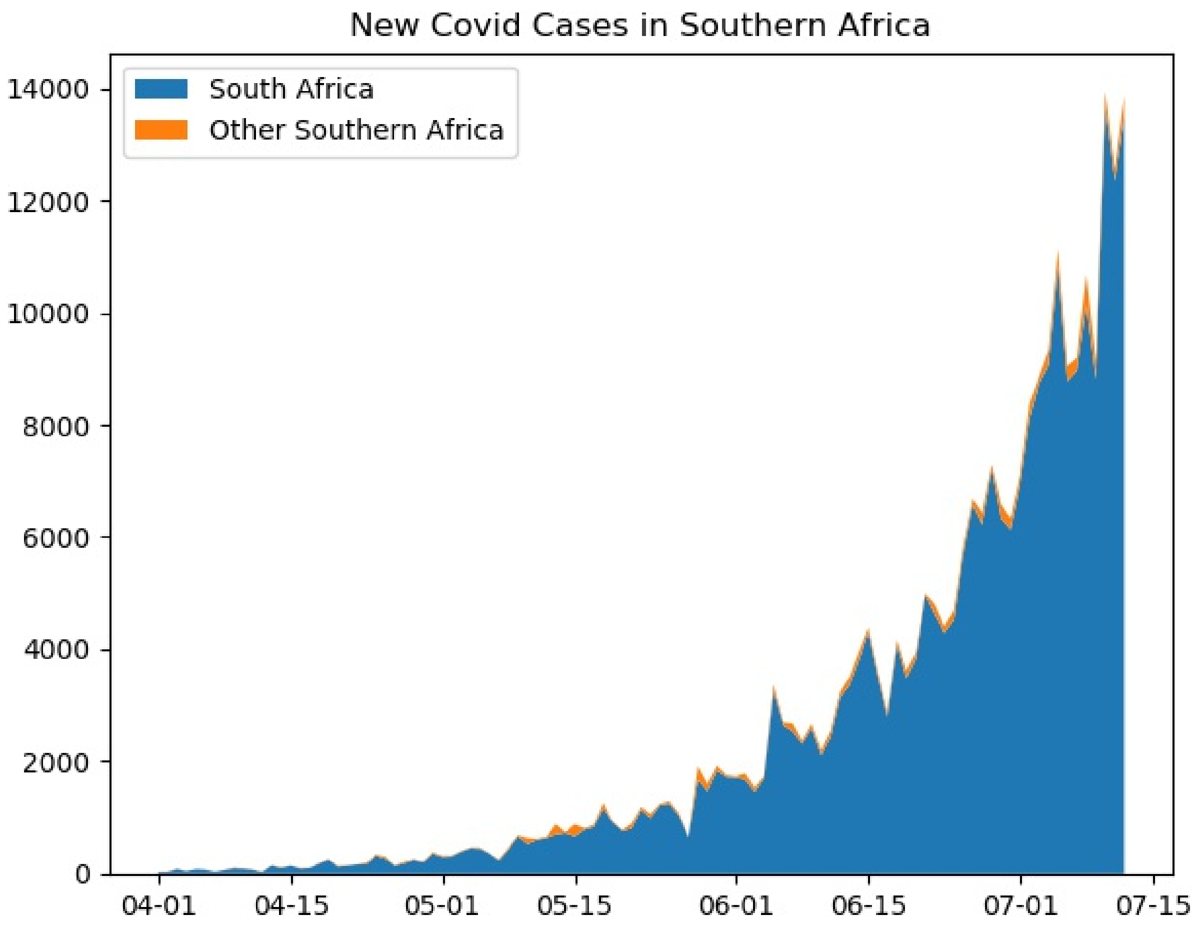


Figure 5.23: Stacked daily trends of cases in South Africa and the rest of that region (Southern Africa)

These steps show how to use line plots to examine trends in a variable over time, and how to display trends for different groups on one figure.

How it works...

We need to do some manipulation of the daily Covid data before we do the line charts. We use `groupby` in *Step 3* to summarize new cases and deaths over all countries for each day. We use `groupby` in *Step 5* to summarize cases and deaths for each region and day.In *Step 4*, we set up our first subplot with `plt.subplot(2,1,1)`. That will give us a figure with two rows and one column. The `1` for the third argument indicates that this subplot will be the first, or top, subplot. We can pass a data series for date and for the values for the *y* axis. So far, this is pretty much what we have done with the `hist`, `scatterplot`, `boxplot`, and `violinplot` methods. But since we are working with dates here, we take advantage of matplotlib's utilities for date formatting and indicate that we want only the month to show, with `xaxis.set_major_formatter(DateFormatter("%b"))`. Since we are working with subplots, we use `set_xlabel` rather than `xlabel` to indicate the label we want for the *x* axis.We show line plots for four selected regions in *Step 6*. We do this by calling `plot` for each region that we want plotted. We could have done it for all of the regions, but it would have been too difficult to view.We have to do some additional manipulation in *Step 7* to pull the South Africa (the country) cases out of the cases for Southern Africa (the region). Once we do that, we can do a stacked plot with the Southern Africa cases (minus South Africa) and South Africa. This figure suggests that the increase in cases in Southern Africa is almost completely driven by increases in South Africa.

There's more...

The figure produced in *Step 6* reveals a couple of potential data issues. There are unusual spikes in mid-February in East Asia and in late April in North America. It is important to examine these anomalies to see if there is a data collection error.It is difficult to miss how much the trends differ by region. There are substantive reasons for this, of course. The different lines reflect what we know to be reality about different rates of spread by country and region. However, it is worth exploring any significant change in the direction or slope of trend lines to make sure that we can confirm that the data is accurate. We want to be able to explain what happened in Western Europe in early April and in North America and Southern Africa in early June. One question is whether the trends reflect changes in the whole region (such as with the decline in Western Europe in early April) or for one or two large countries in the region (the United States in North America and South Africa in Southern Africa).

See also

We cover `groupby` in more detail in *Chapter 7*, *Fixing Messy Data When Aggregating*. We go over merging data, as we did in *Step 7*, in *Chapter 8*, *Addressing Data Issues When Combining Data Frames*.

Generating a heat map based on a correlation matrix

The correlation between two variables is a measure of how much they move together. A correlation of 1 means that the two variables are perfectly positively correlated. As one variable increases in size, so does the other. A value of -1 means that they are perfectly negatively correlated. As one variable increases in size, the other decreases. Correlations of 1 or -1 only rarely happen, but correlations above 0.5 or below -0.5 might still be meaningful. There are several tests that can tell us whether the relationship is statistically significant (Pearson, Spearman, Kendall). Since this is a chapter on visualizations, we will focus on viewing important correlations.

Getting ready

You will need Matplotlib and Seaborn installed to run the code in this recipe. Both can be installed by using `pip`, with the `pip install matplotlib` and `pip install seaborn` commands.

How to do it...

We first show part of a correlation matrix of the Covid data, and scatter plots of some key relationships. We then show a heat map of the correlation matrix to visualize the correlations between all variables.

1. Import matplotlib and seaborn, and load the Covid totals data:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
covidtotals = pd.read_csv("data/covidtotals.csv", parse_dates=["
lastdate"])
```

1. Generate a correlation matrix.

View part of the matrix:

```
corr[['total_cases','total_deaths',
...   'total_cases_pm','total_deaths_pm']]

                 total_cases  total_deaths  \
total_cases             1.00          0.93
total_deaths            0.93          1.00
total_cases_pm          0.23          0.20
total_deaths_pm         0.26          0.41
population              0.34          0.28
pop_density            -0.03         -0.03
median_age              0.12          0.17
gdp_per_capita          0.13          0.16
hosp_beds              -0.01         -0.01
                 total_cases_pm  total_deaths_pm
total_cases                0.23             0.26
total_deaths               0.20             0.41
total_cases_pm             1.00             0.49
total_deaths_pm            0.49             1.00
population                -0.04            -0.00
pop_density                0.08             0.02
median_age                 0.22             0.38
gdp_per_capita             0.58             0.37
hosp_beds                  0.02             0.09
```

1. Show scatter plots of median age and **gross domestic product** (**GDP**) per capita by cases per million.

Indicate that we want the subplots to share *y* axis values with `sharey=True`:

```
fig, axes = plt.subplots(1,2, sharey=True)
sns.regplot(covidtotals.median_age, covidtotals.total_cases_pm,
ax=axes[0])
sns.regplot(covidtotals.gdp_per_capita, covidtotals.total_cases_
pm, ax=axes[1])
axes[0].set_xlabel("Median Age")
axes[0].set_ylabel("Cases Per Million")
axes[1].set_xlabel("GDP Per Capita")
axes[1].set_ylabel("")
plt.suptitle("Scatter Plots of Age and GDP with Cases Per Millio
n")
plt.tight_layout()
fig.subplots_adjust(top=0.92)
plt.show()
```

This results in the following scatter plots:

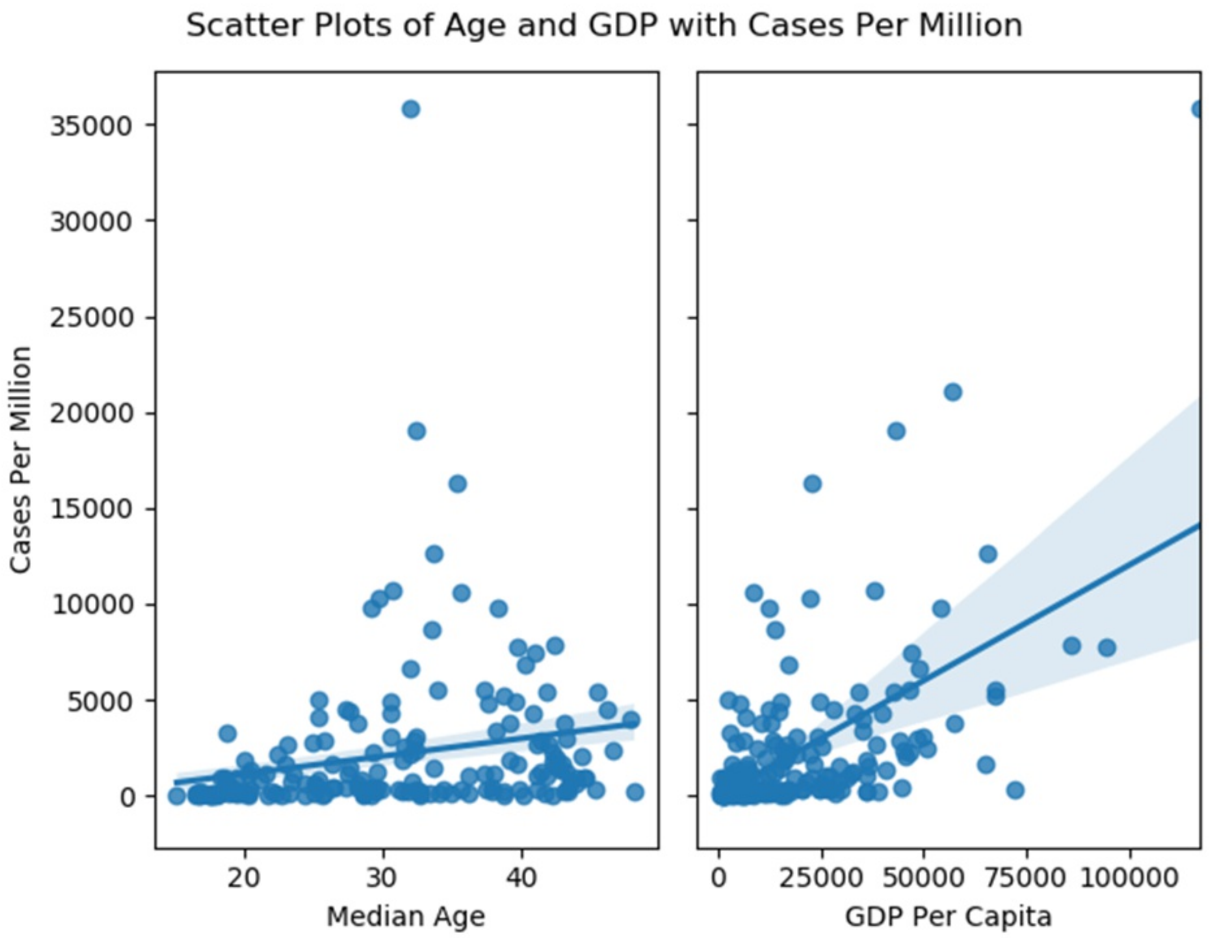


Figure 5.24: Scatter plots of median age and GDP by cases per million side by side

1. Generate a heat map of the correlation matrix:

```
sns.heatmap(corr, xticklabels=corr.columns, yticklabels=corr.col
umns, cmap="coolwarm")
plt.title('Heat Map of Correlation Matrix')
plt.tight_layout()
plt.show()
```

This results in the following heat map:

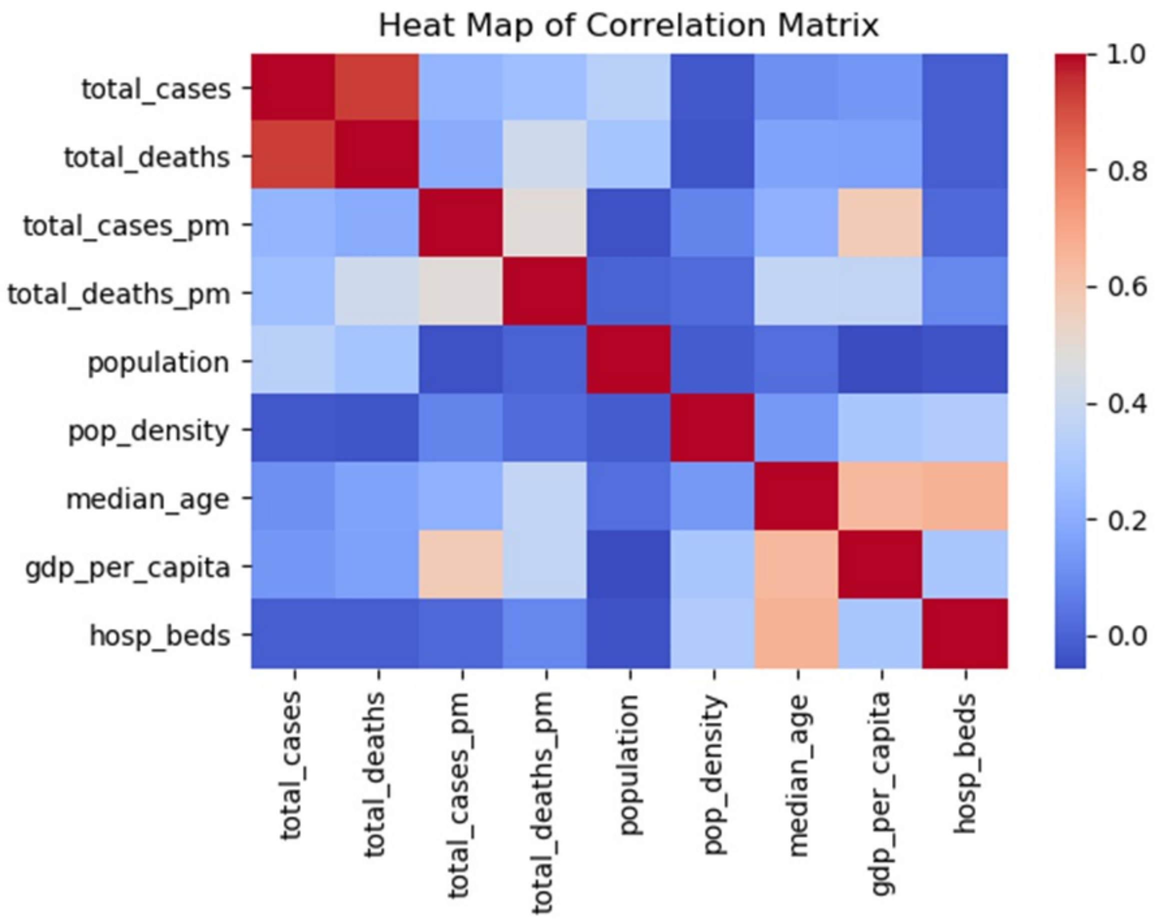


Figure 5.25: Heat map of Covid data, with strongest correlations in red and peach

Heat maps are a great way to visualize how all key variables in our DataFrame are correlated with one another.

How it works...

The `corr` method of a DataFrame generates correlation coefficients of all numeric variables by all other numeric variables. We display part of that matrix in *Step 2*. In *Step 3*, we do scatter plots of median age by cases per million, and GDP per capita by cases per million. These plots give a sense of what it looks like when the correlation is 0.22 (median age and cases per million) and when it is 0.58 (GDP per capita and cases per million). There is not much of a relationship between median age and cases per million. There is more of a relationship between GDP per capita and cases per million.The heat map provides a visualization of the correlation matrix we created in *Step 2*. All of the red squares are correlations of 1.0 (which is the correlation of the variable with itself). The slightly lighter red squares are between `total_cases` and `total_deaths` (0.93). The peach squares (those with correlations between 0.55 and 0.65) are also interesting. GDP per capita, median age, and hospital beds per 1,000 people are positively correlated with each other, and GDP per capita is positively correlated with cases per million.

There's more...

I find it helpful to always have a correlation matrix or heat map close by when I am doing exploratory analysis or statistical modeling. I understand the data much better when I am able to keep these bivariate relationships in mind.

See also

We go over tools for examining the relationship between two variables in more detail in the *Identifying outliers and unexpected values in bivariate relationships* recipe in *Chapter 4, Identifying Issues in Subsets of Data*.