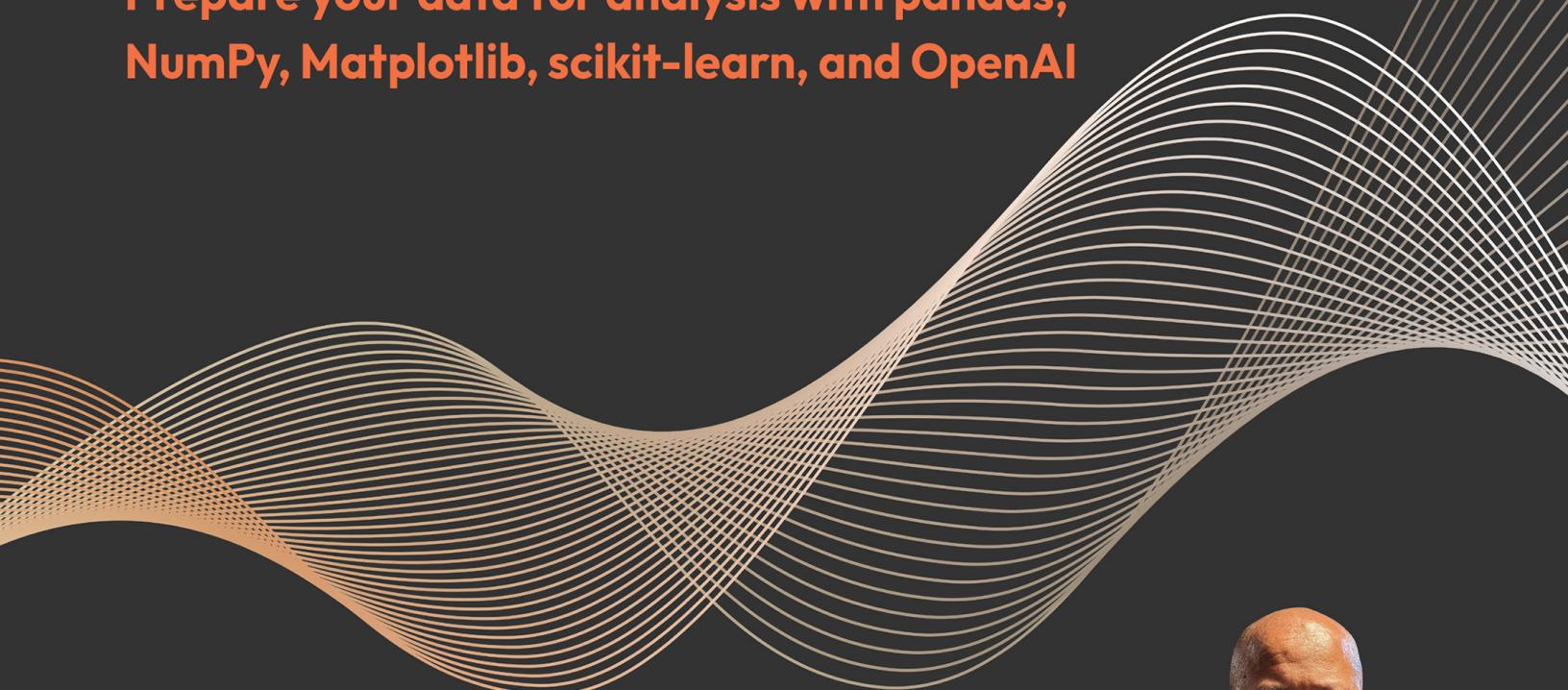


EXPERT INSIGHT

---

# Python Data Cleaning Cookbook

**Prepare your data for analysis with pandas,  
NumPy, Matplotlib, scikit-learn, and OpenAI**



**Second Edition**



---

**Michael Walker**

**<packt>**

# **Python Data Cleaning Cookbook**

**Second Edition**

**Prepare your data for analysis with pandas,  
NumPy, Matplotlib, scikit-learn, and OpenAI**

Michael Walker



# Python Data Cleaning Cookbook

Second Edition

Copyright © 2024 Packt Publishing

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Senior Publishing Product Manager:** Gebin George

**Acquisition Editor – Peer Reviews:** Gaurav Gavas

**Project Editor:** Namrata Katare

**Content Development Editor:** Deepayan Bhattacharjee

**Copy Editor:** Safis Editing

**Technical Editor:** Karan Sonawane

**Proofreader:** Safis Editing

**Indexer:** Subalakshmi Govindhan

**Presentation Designer:** Ajay Patule

**Developer Relations Marketing Executive:** Vignesh Raju

First published: December 2020

Second edition: May 2024

Production reference: 1230524

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN: 978-1-80323-987-3

[www.packt.com](http://www.packt.com)

# Contributors

## About the author

**Michael Walker** has worked as a data analyst for 37 years at various educational institutions. He has also taught data science, research methods, statistics, and computer programming to undergraduates since 2006. He is currently the Chief Information Officer at College Unbound in Providence, Rhode Island. Michael is also the author of *Data Cleaning and Exploration with Machine Learning*.

*I deeply appreciate the editors of this text, particularly Namrata Katare and Deepayan Bhattacharjee, and the technical reviewer, Kalyana Bedhu. They made this text so much better than it would have been otherwise. Any remaining issues are completely the fault of the author. I would also like to thank my partner over the last several decades, Karen Walker. She has had to listen to me go on and on about statistics and data, and the teaching of both topics, for all those years. Along the way, she patiently and lovingly taught me most of what I know about how to teach and be in community with other humans, even when that means putting aside some modeling or programming task for a few hours to focus on what matters most. For the sake of the reader, I hope this book reflects some fraction of her thoughtfulness and patience.*

# About the reviewer

**Kalyana Bedhu**, a recipient of patents and an author of award-winning papers and data science courses, is an Engineering Leader and Architect of Enterprise AI/ML at Fannie Mae. He has over 20 years of experience in applied AI/ML across various companies like Microsoft, Ericsson, Sony, Bosch, Fidelity, and Oracle. As a practitioner of Data Science at Ericsson, he helped set up a data science lab. He played a pivotal role in transforming a central IT organization into an AI and data science engine.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.gg/p8uSgEAETX>



# Preface

This book is a practical guide to data cleaning, broadly defined as all tasks necessary to prepare data for analysis. It is organized by the tasks usually completed during the data-cleaning process: importing data, viewing data diagnostically, identifying outliers and unexpected values, imputing values, tidying data, and so on. Each recipe walks the reader from raw data through the completion of a specific data-cleaning task.

There are already a number of very good pandas books. Unsurprisingly, there is some overlap between those texts and this one. However, the emphasis here is different. I focus as much on the why as on the how in this book.

Since pandas is still relatively new, the lessons I have learned about cleaning data have been shaped by my experiences with other tools. Before settling into my current work routine with Python and R about 10 years ago, I relied mostly on C# and T-SQL in the early 2000s, SAS and Stata in the 90s, and FORTRAN and Pascal in the 80s. Most readers of this text probably have experience with a variety of data-cleaning and analysis tools. In many ways the specific tool is less significant than the data preparation task and the attributes of the data. I would have covered pretty much the same topics if I had been asked to write *The SAS Data Cleaning Cookbook* or *The R Data Cleaning Cookbook*. I just take a Python/pandas-specific approach to the same data-cleaning challenges that analysts have faced for decades.

I start each chapter with how to think about the particular data-cleaning task at hand before discussing how to approach it with a tool from the Python ecosystem—pandas, NumPy, Matplotlib, and so on. This is reinforced in each recipe by a discussion of the implications of what we are uncovering in the data. I try to connect tool to purpose. For example, concepts like skewness and kurtosis matter as much for handling outliers as does knowing how to update pandas Series values.

## New in the Second Edition

Readers of the first edition will recognize that this book is substantially longer than that one. That is partly because there are two new chapters—a chapter devoted to treating missing values and another one on pre-processing data for predictive analysis. The insufficient coverage of missing values, and the absence of coverage of pre-processing data for machine learning applications were important omissions. The pre-processing coverage is further improved by new recipes on data pipelines in the final chapter that take the reader from raw data to model evaluation.

The recipes in all chapters have been revised. This is to make sure that they all work well with the most recent versions of pandas. pandas went from version 1.5.3 to 2.2.1 during the writing of this book. I have tried to make sure that all code works fine on all versions of pandas released from January 2023 through February 2024. Since AI tools are becoming increasingly common in our work, I have included discussion of OpenAI tools in four of the chapters. Altogether, 22 of the 82 recipes are new. All of the datasets used have also been updated.

# Who this book is for

I had multiple audiences in mind as I wrote this book, but I most consistently thought about a dear friend of mine who bought a Transact-SQL book 30 years ago and quickly developed great confidence in her database work, ultimately building a career around those skills. I would love it if someone just starting their career as a data scientist or analyst worked through this book and had a similar experience as my friend. More than anything else, I want you to feel good and excited about what you can do as a result of reading this book.

I also hope this book will be a useful reference for folks who have been doing this kind of work for a while. Here, I imagine someone opening the book and wondering to themselves, “What’s an approach to handling missing data that maintains the variance of my variable?”

In keeping with the hands-on nature of this text, every bit of output is reproducible with code in this book. I also stuck to a rule throughout, even when it was challenging. Every recipe starts with raw data largely unchanged from the original downloaded file. You go from data file to better prepared data in each recipe. If you have forgotten how a particular object was created, all you will ever need to do is turn back a page or two to see.

Readers who have some knowledge of pandas and NumPy will have an easier time with some code blocks, as will folks with some knowledge of Python and introductory statistics. None of that is essential though. There are just some recipes you might want to pause over longer.

# What this book covers

*Chapter 1, Anticipating Data Cleaning Issues When Importing Tabular Data with pandas*, explores tools for loading CSV files, Excel files, relational database tables, SAS, SPSS, Stata, and R files into pandas DataFrames.

*Chapter 2, Anticipating Data Cleaning Issues When Working with HTML, JSON, and Spark Data*, discusses techniques for reading and normalizing JSON data, web scraping, and working with big data using Spark. It also explores techniques for persisting data, including with versioning.

*Chapter 3, Taking the Measure of Your Data*, introduces common techniques for navigating around a DataFrame, selecting columns and rows, and generating summary statistics. The use of OpenAI tools for examining dataset structure and generating statistics is introduced.

*Chapter 4, Identifying Outliers in Subsets of Data*, explores a wide range of strategies to identify outliers across a whole DataFrame and by selected groups.

*Chapter 5, Using Visualizations for the Identification of Unexpected Values*, demonstrates the use of the Matplotlib and Seaborn tools to visualize how key variables are distributed, including with histograms, boxplots, scatter plots, line plots, and violin plots.

*Chapter 6, Cleaning and Exploring Data with Series Operations*, discusses updating pandas Series with scalars, arithmetic operations, and conditional statements based on the values of one or more Series.

*Chapter 7, Identifying and Fixing Missing Values*, goes over strategies for identifying missing values across rows and columns, and over subsets of data. It explores strategies for imputing values, such as setting values to the overall mean or the mean for a given category and forward filling. It also examines multivariate techniques for imputing values for missing values and discusses when they are appropriate.

*Chapter 8, Encoding, Transforming, and Scaling Features*, covers a range of variable transformation techniques to prepare features and targets for predictive analysis. This includes the most common kinds of encoding—one-hot, ordinal, and hashing encoding; transformations to improve the distribution of variables; and binning and scaling approaches to address skewness, kurtosis, and outliers and to adjust for features with widely different ranges.

*Chapter 9, Fixing Messy Data When Aggregating*, demonstrates multiple approaches to aggregating data by group, including looping through data with `itertuples` or NumPy arrays, dropping duplicate rows, and using pandas' groupby and pivot tables. It also discusses when to choose one approach over the others.

*Chapter 10, Addressing Data Issues When Combining DataFrames*, examines different strategies for concatenating and merging data, and how to anticipate common data challenges when combining data.

*Chapter 11, Tidying and Reshaping Data*, introduces several strategies for de-duplicating, stacking, melting, and pivoting data.

*Chapter 12, Automate Data Cleaning with User-Defined Functions and Classes and Pipelines*, examines how to turn many of the techniques from the first 11 chapters into reuseable code.

# Download the example code files

The code bundle for the book is hosted on GitHub at

<https://github.com/PacktPublishing/Python-Data-Cleaning-Cookbook-Second-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

# Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/gbp/9781803239873>.

# Conventions used

There are a number of text conventions used throughout this book.

`code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system.”

A block of code is set as follows:

```
import pandas as pd
import os
import sys
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index('personid', inplace=True)
```

Any output from the code will appear like this:

|       | satverbal | satmath |
|-------|-----------|---------|
| min   | 14        | 7       |
| per15 | 390       | 390     |
| qr1   | 430       | 430     |
| med   | 500       | 500     |
| qr3   | 570       | 580     |
| per85 | 620       | 621     |
| max   | 800       | 800     |
| count | 1,406     | 1,407   |
| mean  | 500       | 501     |
| iqr   | 140       | 150     |

**Bold:** Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: “Select **System info** from the **Administration** panel.”



Warnings or important notes appear like this.



Tips and tricks appear like this.

## Get in touch

Feedback from our readers is always welcome.

**General feedback:** Email [feedback@packtpub.com](mailto:feedback@packtpub.com) and mention the book’s title in the subject of your message. If you have questions about any aspect of this book, please email us at [questions@packtpub.com](mailto:questions@packtpub.com).

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit

<http://www.packtpub.com/submit-errata>, click **Submit Errata**, and fill in the form.

**Piracy:** If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the material.

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com>.

# Share your thoughts

Once you've read *Python Data Cleaning Cookbook, Second Edition*, we'd love to hear your thoughts! Please [click here to go straight to the Amazon review page](#) for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

# Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781803239873>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.

# 1

## Anticipating Data Cleaning Issues When Importing Tabular Data with 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. The recipes in this book demonstrate how many common data preparation tasks can be done more easily with pandas than with other tools. While we work with pandas extensively in this book, we also use other 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 import 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 to import 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

## Technical requirements

The code and notebooks for this chapter are available on GitHub at <https://github.com/michaelbwalker/Python-Data-Cleaning-Cookbook-Second-Edition>. You can use any **IDE (Integrated Development Environment)** of your choice – IDLE, Visual Studio, Sublime, Spyder, and so on – or Jupyter Notebook to work with any of the code in this chapter, or any chapter in this book. A good guide to get started with Jupyter Notebook can be found here:

<https://www.dataquest.io/blog/jupyter-notebook-tutorial/>. I used the Spyder IDE to write the code in this chapter.

I used pandas 2.2.1 and NumPy version 1.24.3 for all of the code in this chapter and subsequent chapters. I have also tested all code with pandas 1.5.3.

# 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 import 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 then create a new Python script or **Jupyter Notebook** file in that folder. Create a data subfolder, and then place the `landtempssample.csv` file in that subfolder. Alternatively, you could retrieve all of the files from the GitHub repository, including the data files. 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
USS0009J085,2015,2,-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
JMLT656433,1868,1,21.09,18.1,-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

## 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.ncei.noaa.gov/products/land-based-station/global-historical-climatology-network-monthly>. I used the data from version 4. The data in this recipe uses 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)
```

2. 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`. This will cause pandas to load all of the data into memory at once, rather than in chunks. We do this so that pandas can identify the data type of each column automatically. In the *There's more...* section, we see how to set the data type for each column manually:

```
landtemps = pd.read_csv('data/landtempssample.csv',
...     names=['stationid', 'year', 'month', 'avgtemp', 'latit
...         'longitude', 'elevation', 'station', 'countryid', 'c
...     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`.

### 3. 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      col
0 2000-04-01    USS0010K01S  ...       US    United St
1 1940-05-01    CI000085406  ...       CI        C
2 2013-12-01    USC00036376  ...       US    United St
3 1963-02-01    ASN00024002  ...       AS    Austria
4 2001-11-01    ASN00028007  ...       AS    Austria
5 1991-04-01    USW00024151  ...       US    United St
6 1993-12-01    RSM00022641  ...       RS        Ru
[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
```

```
(1000000, 9)
```

4. Give the date column a more appropriate name and view the summary statistics for average monthly temperature:

```
landtemps.rename(columns={'month_year':'measuredate'}  
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
```

## 5. 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 missing values 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
stationed        0
avgtemp         14446
latitude         0
longitude        0
elevation        0
station          0
countryid        0
country          5
dtype: int64
```

## 6. 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 seven rows. We use the `dtypes` attribute of the DataFrame 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 DataFrame (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 used the `rename` method in *Step 4* to give that column a more appropriate 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 missing values. 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` was run are index values. Any number of rows can be specified for `head`. The default value is `5`.

Instead of setting `low_memory` to `False`, to get pandas to make good guesses regarding data types, we could have set data types manually:

```
landtemps = pd.read_csv('data/landtempssample.csv',
    names=['stationid','year','month','avgtemp','latitude',
        'longitude','elevation','station','countryid','country'],
    skiprows=1,
    parse_dates=[['month','year']],
    dtype={'stationid':'object', 'avgtemp':'float64',
        'latitude':'float64','longitude':'float64',
        'elevation':'float64','station':'object',
        'countryid':'object','country':'object'},
    )
landtemps.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 9 columns):
 #   Column      Non-Null Count   Dtype  
 ---  --          --          --      
 0   month_year  100000 non-null   datetime64[ns]
 1   stationid   100000 non-null   object  
 2   avgtemp     85554 non-null   float64 
 3   latitude    100000 non-null   float64 
 4   longitude   100000 non-null   float64 
 5   elevation   100000 non-null   float64 
 6   station     100000 non-null   object  
 7   countryid   100000 non-null   object  
 8   country     99995 non-null   object  
dtypes: datetime64[ns](1), float64(4), object(4)
memory usage: 6.9+ MB
```

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`. The `sum` function 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 chaining helpful when the overall operation feels like a single step, even if it's two or more steps mechanically. 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 11, 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, select particular columns, and pull data from a particular sheet or sheets.

# Getting ready

You can download the `GDPpercapita22b.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 (some columns were hidden for display purposes):

| Dataset: Metropolitan areas |  |   |       |       |       |       |
|-----------------------------|--|---|-------|-------|-------|-------|
| Variables                   |  | GDP per capita (USD, constant prices, constant PPP, base year 2015) |       |       |       |       |
| Unit                        |  | US Dollar   |       |       |       |       |
| Year                        |  | 2016  | 2017  | 2018  | 2019  | 2020  |
| Metropolitan areas          |  |   |       |       |       |       |
| AUS: Australia              |  | ..  | ..    | ..    | ..    | ..    |
| AUS01: Greater Sydney       |  | 47582   | 47569 | 47171 | 45576 | 45152 |
| AUS02: Greater Melbourne    |  | 43394   | 43772 | 43237 | 42299 | 40848 |
| AUS03: Greater Brisbane     |  | 43547   | 44663 | 44328 | 42145 | 40741 |
| AUS04: Greater Perth        |  | 62433   | 62686 | 67313 | 70970 | 78489 |
| AUS05: Greater Adelaide     |  | 38984   | 39302 | 39055 | 38314 | 39181 |

Figure 1.2: View of the dataset

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

|  |       |       |       |       |       |
|--|-------|-------|-------|-------|-------|
| USA155: Benton (WA)  | 49868 | 49591 | 49851 | 51145 | 49867 |
| USA156: Weld   | 53186 | 59487 | 65664 | 60528 | 46089 |
| USA157: Kalamazoo  | 48700 | 50434 | 51147 | 51694 | 49169 |
| USA158: Butte  | 37570 | 38534 | 39029 | 41680 | 41317 |
| USA160: Yakima   | 37706 | 38410 | 39815 | 40211 | 40104 |
| USA161: Brazos   | 45336 | 47559 | 49483 | 49113 | 45939 |
| USA162: Tuscaloosa   | 41260 | 43149 | 43973 | 45120 | 42095 |
| USA170: Benton (AR)  | 49784 | 51043 | 51328 | 51739 | 51423 |
| Data extracted on 15 Jan<br>2024 01:48 UTC (GMT)<br>from OECD.Stat |       |       |       |       |       |

*Figure 1.3: View of the dataset*



### Data 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
```

## 2. 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 `W` (column `B` is blank). Use `head` to view the first few rows and `shape` to get the number of rows and columns:

```
percapitaGDP = pd.read_excel("data/GDPpercapita22b.xlsx",
...     sheet_name="OECD.Stat export",
...     skiprows=4,
...     skipfooter=1,
...     usecols="A,C:W")
percapitaGDP.head()
```

```
          Year 2000 ...
0 Metropolitan areas ... NaN ..
1 AUS: Australia .. ...
2 AUS01: Greater Sydney ...
3 AUS02: Greater Melbourne ...
4 AUS03: Greater Brisbane ...
[5 rows x 22 columns]
```

```
percapitaGDP.shape
```

```
(731, 22)
```

 **Note**



You may encounter a problem with `read_excel` if the Excel file does not use utf-8 encoding. One way to resolve this is to save the Excel file as a CSV file, reopen it, and then save it with utf-8 encoding.

3. Use the `info` method of the DataFrame to view data types and the `non-null` count. Notice that all columns have the `object` data type:

```
percapitaGDP.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 731 entries, 0 to 730
Data columns (total 22 columns):
 #   Column    Non-Null Count  Dtype  
--- 
 0   Year       731 non-null   object 
 1   2000      730 non-null   object 
 2   2001      730 non-null   object 
 3   2002      730 non-null   object 
 4   2003      730 non-null   object 
 5   2004      730 non-null   object 
 6   2005      730 non-null   object 
 7   2006      730 non-null   object 
 8   2007      730 non-null   object 
 9   2008      730 non-null   object 
 10  2009      730 non-null   object 
 11  2010      730 non-null   object 
 12  2011      730 non-null   object 
 13  2012      730 non-null   object 
 14  2013      730 non-null   object 
 15  2014      730 non-null   object 
 16  2015      730 non-null   object 
 17  2016      730 non-null   object 
 18  2017      730 non-null   object 
 19  2018      730 non-null   object 
 20  2019      730 non-null   object 
 21  2020      730 non-null   object
```

```
dtypes: object(22)
memory usage: 125.8+ KB
```

4. 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. 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()
```

```
False
```

```
percapitaGDP.metro = percapitaGDP.metro.str.strip()
percapitaGDP.metro.str.startswith(' ').any()
```

```
False
```

5. Convert the data columns to numeric.

Iterate over all of the GDP year columns (2000–2020) 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)
    ...
percapitaGDP.head()
```

```
          metro  pcGDP2000  pcGDP2001  ...  \
0  Metropolitan areas      NaN      NaN  ...
1        AUS: Australia      NaN      NaN  ...
2  AUS01: Greater Sydney      NaN  41091  ...
3  AUS02: Greater Melbourne      NaN  40488  ...
4  AUS03: Greater Brisbane      NaN  35276  ...
pcGDP2018  pcGDP2019  pcGDP2020
0        NaN        NaN        NaN
1        NaN        NaN        NaN
2      47171      45576      45152
3      43237      42299      40848
4      44328      42145      40741
[5 rows x 22 columns]
```

```
percapitaGDP.dtypes
```

```
metro          object
pcGDP2000      float64
```

```
pcGDP2001      float64  
abbreviated to save space  
pcGDP2019      float64  
pcGDP2020      float64  
dtype: object
```

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

```
percapitaGDP.describe()
```

```
          pcGDP2000  pcGDP2001  pcGDP2002  ...  pcGDP2018  \  
count        158       450       479  ...       692  
mean      33961     38874     39621  ...     41667  
std       15155     13194     13061  ...     17440  
min       2686      7805      7065  ...      5530  
25%      21523     30790     31064  ...     31322  
50%      35836     38078     39246  ...     41428  
75%      42804     46576     47874  ...     51130  
max      95221     96941     98929  ...    147760  
          pcGDP2019  pcGDP2020  
count        596       425  
mean      42709     39792  
std       18893     19230  
min       5698      5508  
25%      29760     24142  
50%      43505     41047  
75%      53647     51130  
max      146094    131082  
[8 rows x 21 columns]
```

7. 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) and going through to 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='percapitaGDP.shape
```

```
(692, 22)
```

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

Confirm that there are 692 valid values for `metro` and that there are 692 unique values, before setting the index:

```
percapitaGDP.metro.count()
```

```
692
```

```
percapitaGDP.metro.nunique()
```

```
692
```

```
percapitaGDP.set_index('metro', inplace=True)  
percapitaGDP.head()
```

```
          pcGDP2000  pcGDP2001  ...  \nmetro\nAUS01: Greater Sydney      NaN    41091  ... \nAUS02: Greater Melbourne     NaN    40488  ... \nAUS03: Greater Brisbane     NaN    35276  ...
```

```
AUS04: Greater Perth           NaN    43355 ...
AUS05: Greater Adelaide       NaN    36081 ...
                                         pcGDP2019  pcGDP2020
metro
AUS01: Greater Sydney          45576    45152
AUS02: Greater Melbourne       42299    40848
AUS03: Greater Brisbane        42145    40741
AUS04: Greater Perth            70970    78489
AUS05: Greater Adelaide        38314    39181
[5 rows x 21 columns]
percapitaGDP.loc['AUS02: Greater Melbourne']
pcGDP2000      NaN
pcGDP2001      40488
...
pcGDP2019      42299
pcGDP2020      40848
Name: AUS02: Greater Melbourne, dtype: float64
```

We have now imported the Excel data into a pandas DataFrame 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. We could have just used `lstrip` to remove leading blanks, or `rstrip` if there had been trailing blanks. It is a good idea to assume that there might be leading or trailing blanks in any character data, cleaning 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 type in *Step 5*. This also results in the original values of `..` being replaced with `NaN` (not a number), how pandas represents 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 for some years. When we drop all rows that have missing values for all per capita GDP columns in *step 7*, we end up with 692 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 Series. Either method makes it possible to use 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 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 Series operations. We will try many of them out in subsequent recipes. This recipe showed those that I find most useful when importing Excel data.

## See also

There are good reasons to consider reshaping this data. Instead of 21 columns of GDP per capita data for each metropolitan area, we should have 21 rows of data for each metropolitan area, with columns for year and GDP per capita. Recipes for reshaping data can be found in *Chapter 11, 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 such as these 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 merchandise at a retail store chain might be different in 1998 than they are in 2024. Similarly, frequently there are codes for missing values, such as 99,999, 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. However, 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`. We will work with data on educational attainment in this recipe.



### Data note

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

## How to do it...

We import SQL Server and MySQL data tables into a pandas DataFrame, 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
```

2. 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 DataFrame by passing the `SELECT` statement and connection

object to `read_sql`. Use `close` to return the connection to the pool on the server:

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

### Note

Although tools such as `pymssql` make connecting to a SQL Server instance relatively straightforward, the syntax still might take a little time to get used to if it is unfamiliar. The previous step shows the parameter values you will typically need to pass to a connection object – the name of the server, the name of a user with credentials on the server, the password for that user, and the name of a SQL database on the server.

3. 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
gout             int64
gradeperiod1    int64
gradeperiod2    int64
gradeperiod3    int64
dtype: object
```

```
studentmath.head()
```

```
   studentid    school  ...  gradeperiod2  gradeperi
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]
```

4. Connecting to a MySQL server is not very different from connecting to a SQL Server instance. We can use the `connect` method of the `mysql` connector to do that and then use `read_sql` to load the data.

Create a connection to the `mysql` data, pass that connection to `read_sql` to retrieve the data, and load it into a pandas DataFrame (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()
```

## 5. 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()
```

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

## 6. Use the DataFrame'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
```

## 7. 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)
```

## 8. 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
```

9. 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=True)
```

```
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
```

10. 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 |      |
| 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. However, 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, as well as 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 did 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 further 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 that there are no missing values – for non-missing values, the count 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 prevent `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`. The `0` values for `mothereducation` and `fathereducation`

seem to fall into that category. We fix this problem in the `setvalues` dictionary by indicating that the `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 *SPSS*, *SAS*, and *R* will notice the difference between this approach and value labels in SPSS and R, as well as the `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, keeping them at the forefront of your mind. It is also helpful to keep similar columns together. In this example, personal demographic variables (sex and age) are next to one another, as are family variables (`mothereducation` and `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 10, Addressing Data Issues When Combining DataFrames*, go into detail on merging data. We will take a closer look at bivariate and multivariate relationships between variables in *Chapter 4, Identifying Outliers in Subsets of Data*. We will 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 `overall satisfaction with presentation` variable label and `1–5` value labels, 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 to represent 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 Surveys (NLS)** of Youth.



### Data note

The NLS 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 2023. 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.

The original NLS data can be downloaded from  
<https://www.nlsinfo.org/investigator/pages/search>, along with code for creating SPSS, Stata, or

SAS files from the ASCII data files included in the download.

## 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
```

2. 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. Note 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
```

### 3. 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)
```

#### 4. 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=True)
```

```
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
".... abbreviated to save space"
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]', '', regex=True)
nls97spss.set_index('pubid_yth_id_code_1997', inplace=True)
```

## 5. 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')
nls97spss.columns = metaspss.column_labels
nls97spss.columns = nls97spss.columns.\
...      str.lower().\
...      str.replace(' ', '_').\
...      str.replace('[^a-z0-9]', '', regex=True)
```

## 6. 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_199      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_199
0           nan         52
1            6          0
2            6          0
3            6          4
4            5         12
[5 rows x 42 columns]
```

7. 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)
```

8. That demonstrated how to convert data from SPSS. Let's try that with Stata data.
9. Import the Stata data, apply value labels, and improve the column headings.

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

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

```
pubid_yth_id_code_1997 float64
keysex_symbol_1997          category
keybdate_my_symbol_1997 float64
keybdate_my_symbol_1997 float64
hrsnights_r_sleeps_2017 float64
cvc_wkswk_yr_all_199     float64
dtype: object
```

10. View a few rows of the data and run frequencies:

```
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_199
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
```

11. Fix the logical missing values that show up with the Stata data and set an index. We can use the `replace` method to set any value that is between `-9` and `-1` in any column to missing:

```
nls97stata.min(numeric_only=True)
```

```
pubid_yth_id_code_1997          1
keybdate_my_symbol_1997          1
keybdate_my_symbol_1997         1,980
trans_sat_verbal_hstr           -4
trans_sat_math_hstr             -4
trans_crd_gpa_overall_hstr      -9
trans_crd_gpa_eng_hstr          -9
trans_crd_gpa_math_hstr         -9
trans_crd_gpa_lp_sci_hstr       -9
cv_ba_credits_l1_2011            5
cv_bio_child_hh_2017             5
cv_bio_child_nr_2017             5
hrsnight_r_sleeps_2017           5
cvc_wkswk_yr_all_199             4
dtype: float64
```

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

```
pubid_yth_id_code_1997          1
keybdate_my_symbol_1997          1
keybdate_my_symbol_1997         1,980
trans_sat_verbal_hstr           14
trans_sat_math_hstr              7
trans_crd_gpa_overall_hstr      10
trans_crd_gpa_eng_hstr           0
trans_crd_gpa_math_hstr          0
trans_crd_gpa_lp_sci_hstr        0
cv_ba_credits_l1_2011             0
cv_bio_child_hh_2017              0
cv_bio_child_nr_2017              0
hrsnight_r_sleeps_2017             0
cvc_wkswk_yr_all_199              0
dtype: float64
```

```
nls97stata.set_index('pubid_yth_id_code_1997', inplace=True)
```

The process is fairly similar when working with SAS data files, as the next few steps illustrate.

## 12. 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.sas7bdat')
nls97sas.columns = metasas.column_labels
nls97sas.columns = nls97sas.columns.\n    ... str.lower().\\
... str.replace(' ', '_').\\
... str.replace('[^a-zA-Z0-9]', '', regex=True)
nls97sas.head()
```

```
      pubkey_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_199
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 12*).

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 DataFrame 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 looked at some of the SPSS data before 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. However, 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 NLS has nine 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 may combine individual and event-level data – a person and hospital visits, a brown bear and the date it 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 11, 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. However, 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 it 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 NLS 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
```

2. 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]
```

### 3. 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'}}}
...abbreviated to save space
```

```
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']
```

4. Set value labels and missing values, and change selected columns to the `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
```

## 5. 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 loaded a dictionary that maps our variables to value labels, and a list for our preferred column headings. In *Step 4* we applied the value labels. We also changed the data type to `category` for the columns where we applied the values. We did this by generating a list of the keys in 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 work.

## There's more...

`Pyreadr` is able to return multiple DataFrames. 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.

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    | Z96 |
|--------------------------|----------|----------|-----|-------------|-----|
| 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 replacement

of that number with `NaN`, after confirming that that is not a valid value, would be a good next step.

## 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.

## 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 DataFrame 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 as all these file types in this recipe.



### Note

Pickle and Feather are binary file formats that can be used to store pandas DataFrames.

## 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.

### 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.ncei.noaa.gov/products/land-based-station/global-historical-climatology-network-monthly>. I used the data from version 4. The data in this recipe uses 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 a Feather file. We will also save subsets of the data to the CSV and Excel formats:

1. Import `pandas` and `pyarrow`.

`pyarrow` needs to be imported in order to save pandas to Feather:

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

2. 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'})
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=True)
```

3. Write extreme values for `temperature` to CSV and Excel files.

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

```
extremevals = landtemps[(landtemps.avgtemp < landtemps.avgtemp.quantile(0.001)) | (landtemps.avgtemp > landtemps.avgtemp.quantile(0.999))]
```

```
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')
```

#### 4. 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")
```

#### 5. Load the Pickle and Feather files we just saved.

Note 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 demonstrated how to serialize pandas DataFrames using two different formats, Pickle and Feather.

## How it works...

Persisting pandas data is quite straightforward. DataFrames have the `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 global 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 also using R. When I am wrapping up a project, I pickle the DataFrames.

## Summary

Our Python data projects typically start with raw data stored in a range of formats and exported from a variety of software tools. Among the most popular tabular formats and tools are CSV and Excel files, SQL tables, and SPSS, Stata, SAS, and R datasets. We converted data from all of these sources into a pandas DataFrame in this chapter, and addressed the most

common challenges. We also explored approaches to persisting tabular data. We will work with data in other formats in the next chapter.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.gg/p8uSgEAETX>



# 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. 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 will 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 will 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 improvements 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 Spark data
- Persisting JSON data
- Versioning data

## Technical requirements

You will need pandas, NumPy, and Matplotlib to complete the recipes in this chapter. I used pandas 2.1.4, but the code will run on pandas 1.5.3 or later.

The code in this chapter can be downloaded from the book's GitHub repository, <https://github.com/PacktPublishing/Python-Data-Cleaning-Cookbook-Second-Edition>.

## 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 local storage and then converts it from something like an SQL Server, MySQL, or PostgreSQL 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 will 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/0ZLHOK](https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/0ZLHOK). 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 comprehension in this recipe. *DataCamp* has good guides on list comprehension (<https://www.datacamp.com/community/tutorials/python-list-comprehension>) and dictionary comprehension (<https://www.datacamp.com/community/tutorials/python-dictionary-comprehension>) if you are feeling a little rusty or have limited or no experience with list and dictionary comprehension.

## 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
```

2. 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',
 'url': 'https://www.nbcnews.com/politics/2020-election/t...'},
 {'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 Plan to End ICE",
 'url': 'https://www.townandcountrymag.com/society/politics...'}]
```

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

```
'NBC News'
```

### 3. Check for differences in the structure of the dictionaries.

Use `Counter` to check for any dictionaries in the list with fewer or more than the 9 keys that are 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 candidate_news])
```

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

```
pprint pprint(next(item for item in candidatenews if len(:
```

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

```
pprint pprint(next(item for item in candidatenews if len(:
```

```
{'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 Crc
          'Hickenlooper Photo \n'
          'In Elaborate Grid Of Rivals',
'url': 'https://politics.theonion.com/and-then-there-were
```

```
pprint pprint([item for item in candidatenews if len(item)
```

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

```
3candidatenews = [item for item in candidatenews if len(it
```

```
len(candidatenews)
```

```
57618
```

#### 4. 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 Super Tuesday',  
 'url': 'https://www.politico.com/story/2019/05/09/mariar...'},  
 {'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': 'Julian Castro to drop out of 2020 presidential race',  
 'url': 'https://www.politico.com/story/2018/12/27/julian...'}]
```

```
'title': "O'Rourke and Castro on collision course in Te  
'url': 'https://www.politico.com/story/2018/12/27/orourk
```

## 5. 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.c
```

```
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)]
```

## 6. 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 newsdi  
...         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)]
```

## 7. 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
```

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

Also, rename the `date` column:

```
candidatenewsdf.rename(columns={'date':'storydate'}, inplace=True)
candidatenewsdf.storydate = candidatenewsdf.storydate.astype(str)
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 or more than 9 keys to get a sense of the structure of those items. We use slicing to show 2 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 `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 *Step 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 `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 but 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 manipulate the data in a very different way than we are used to doing or convert the JSON to a tabular form. We examine the first approach in the *Classes that handle non-tabular data structures* recipe in *Chapter 12, Automate Data Cleaning with User-Defined Functions and Classes*. 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_artis": "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"
    }
]
```



## Data 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/>.

Since the call to the API retrieves real-time data, you may get different output from running the code in this recipe.

## 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
```

## 2. 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/collections")
camcollections = json.loads(response.text)
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)...', 'name': 'Jacob Lawrence'}]}
```

```
'birth_year': '1917',
'description': 'Jacob Lawrence (American...)',
'role': 'artist'}],
'type': 'Painting'}
```

### 3. 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                      |                      |
|------------------|------------------------|----------------------|
| citation         | Annual Exhibiti...     | "Moscow to See Moder |
| page_number      | Unpaginated,           | F                    |
| url              | None                   |                      |
| accession_number | 2007.158               | 2007                 |
| title            | Fulton and No...       | Fulton and N         |
| creation_date    | 1958                   |                      |
| collection       | American - Pa...       | American - F         |
| creators         | [{'description':'J...] | [{'description':'    |
| type             | Painting               | Pair                 |

### 4. Pull the `birth_year` value from `creators`:

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

```
dict
```

```
pprint.pprint(creator)
```

```
[{'biography': 'Jacob Lawrence (born 1917) has been a promi',
 '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.\n...     creators.apply(lambda x: x[0]['birth_year'])\ncamcollectionsdf.birthyear.value_counts().\n...     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 the 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 the first two observations. `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 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 4*, 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 of 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 to 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 on 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 that are unique to web scraping, including HTML elements that house the data 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 will 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 on that page, and load it into a pandas DataFrame:

| Lowest Cases Per Million |            |             |              |                |                 |             |                |            |
|--------------------------|------------|-------------|--------------|----------------|-----------------|-------------|----------------|------------|
| Country                  | Last Date  | Total Cases | Total Deaths | Total Cases PM | Total Deaths PM | Population  | GDP Per Capita | Median Age |
| <b>Yemen</b>             | 11/6/2022  | 11,945      | 2,159        | 354            | 64              | 33,696,612  | 1,479          | 20.3       |
| <b>Niger</b>             | 7/2/2023   | 9,515       | 315          | 363            | 12              | 26,207,982  | 926            | 15.1       |
| <b>Chad</b>              | 4/30/2023  | 7,698       | 194          | 434            | 11              | 17,723,312  | 1,768          | 16.7       |
| <b>Tanzania</b>          | 12/10/2023 | 43,223      | 846          | 660            | 13              | 65,497,752  | 2,683          | 17.7       |
| <b>Sierra Leone</b>      | 12/24/2023 | 7,779       | 125          | 904            | 15              | 8,605,723   | 1,390          | 19.1       |
| <b>Burkina Faso</b>      | 12/17/2023 | 22,109      | 400          | 975            | 18              | 22,673,764  | 1,703          | 17.6       |
| <b>DR Congo</b>          | 12/17/2023 | 99,338      | 1,468        | 1,003          | 15              | 99,010,216  | 808            | 17         |
| <b>Nigeria</b>           | 10/29/2023 | 267,173     | 3,155        | 1,223          | 14              | 218,541,216 | 5,338          | 18.1       |
| <b>Sudan</b>             | 4/16/2023  | 63,993      | 5,046        | 1,365          | 108             | 46,874,200  | 4,467          | 19.7       |
| <b>Mali</b>              | 12/17/2023 | 33,164      | 743          | 1,468          | 33              | 22,593,598  | 2,014          | 16.4       |

Figure 2.1: COVID-19 data from countries with the lowest cases per million in population

### Data note



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

## How to do it...

We scrape the COVID-19 data from the website and do some routine data checks:

1. Import the `pprint`, `requests`, and `BeautifulSoup` libraries:

```
import pandas as pd
import numpy as np
```

```
import json
import pprint
import requests
from bs4 import BeautifulSoup
```

## 2. 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/tblLowCases")
bs = BeautifulSoup(webpage.text, 'html.parser')
theadrows = bs.find('table', {'id':'tblLowCases'}).thead.findAll('th')
type(theadrows)
```

```
<class 'bs4.element.ResultSet'>
```

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

```
['rowheadings',
 'Last Date',
 'Total Cases',
 'Total Deaths',
 'Total Cases PM',
 'Total Deaths PM',
 'Population',
 'GDP Per Capita',
 'Median Age']
```

### 3. 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':'tblLowCases'}).tbody.find_
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])
```

```
[['11/6/2022', '11,945', '2,159', '354', '64', '33,696,612',
 ['7/2/2023', '9,515', '315', '363', '12', '26,207,982', '12,159']]
```

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

```
['Yemen', 'Niger']
```

```
for i in range(len(datarows)):
...     datarows[i].insert(0, labelrows[i])
...
pprint.pprint(datarows[0:2])
```

```
[['Yemen',
  '11/6/2022',
  '11,945',
  '2,159',
  '354',
  '64',
  '33,696,612',
  '1,479',
  '20.3'],
 ['Niger',
  '7/2/2023',
  '9,515',
  '315',
  '363',
  '12',
  '26,207,982',
  '926',
  '15.1']]
```

#### 4. 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):

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

|   | Last Date | Total Cases | Total Deaths | Total Cases |
|---|-----------|-------------|--------------|-------------|
| 0 | 11/6/2022 | 11,945      | 2,159        | 3           |

|   |            |        |     |   |
|---|------------|--------|-----|---|
| 1 | 7/2/2023   | 9,515  | 315 | 3 |
| 2 | 4/30/2023  | 7,698  | 194 | 2 |
| 3 | 12/10/2023 | 43,223 | 846 | € |
| 4 | 12/24/2023 | 7,779  | 125 | £ |

```
lowcases.dtypes
```

|                 |        |
|-----------------|--------|
| rowheadings     | object |
| Last Date       | object |
| Total Cases     | object |
| Total Deaths    | object |
| Total Cases PM  | object |
| Total Deaths PM | object |
| Population      | object |
| GDP Per Capita  | object |
| Median Age      | object |
| dtype:          | object |

## 5. Fix the column names and convert the data to numeric values.

Remove spaces from the column names. Remove all non-numeric data from the first columns with data, including the commas (`str.replace("[^0-9]", "")`). Convert to numeric values, handling most columns as integers, the `last_date` column as a datetime, `median_age` as float, and leaving `rowheadings` as an object:

```
lowcases.columns = lowcases.columns.str.replace(" ", "_")
for col in lowcases.columns[2:-1]:
    lowcases[col] = lowcases[col].\
        str.replace("[^0-9]", "", regex=True).astype('int64')
lowcases['last_date'] = pd.to_datetime(lowcases.last_date)
```

```
lowcases['median_age'] = lowcases['median_age'].astype('f')
lowcases.dtypes
```

```
rowheadings          object
last_date           datetime64[ns]
total_cases         int64
total_deaths        int64
total_cases_pm      int64
total_deaths_pm     int64
population          int64
gdp_per_capita      int64
median_age          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':'tblLowCases'}).thead.find_all
labelcols = [j.get_text() for j in theadrows]
```

The first statement finds all the `th` elements we want and creates a BeautifulSoup result set called `theadrows` from the elements it found. The second statement iterates over the `theadrows` BeautifulSoup result set using the `get_text` method to get the text from each element and then stores it in the `labelcols` list.

*Step 3* is a little more involved but makes use of the same BeautifulSoup methods. We find all of the table rows (`tr`) in the target table (`rows = bs.find('table', {'id': 'tblLowCases'}).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 table of interest, 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 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:

1. Install `Java` with `conda install openjdk`.
2. Install `PySpark` with `conda install pyspark` or `conda install -c conda-forge pyspark`.
3. Install `findspark` with `conda install -c conda-forge findspark`.



## Note

Installation of PySpark can be tricky, particularly setting the necessary environment variables. While `findspark` helps with this, a common problem is that the Java installation is not recognized when running PySpark commands. If you get the dreaded `JAVA_HOME is not set` error when attempting to run the code in this recipe, then there is a good chance that that is your problem.

*Step 3* at the following link shows you how to set the environment variables for Linux, macOS, and Windows machines:

<https://www.dei.unipd.it/~capri/BDC/PythonInstructions.html>.

We will work with the land temperature data from *Chapter 1, Anticipating Data Cleaning Issues When Importing Tabular Data with pandas*, 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.



## 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.ncei.noaa.gov/data/global-historical-climatology-network-integrated-database>

## [1-historical-climatology-network-monthly/v4/](#)

We will use PySpark in this recipe to read data we have in local storage.

## How to do it...

To read and explore the data, follow these steps:

1. Let's start a Spark session and load the land temperature 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.tar.gz")
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.

2. 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 a 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)
```

3. 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', '
.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
```

4. We should fix 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')]
```

5. 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|
+-----+-----+
```

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

7. 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)
```

8. 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, 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 processor 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:

```
allcandidatenewddf = allcandidatenews.toPandas()
allcandidatenewddf.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. We often reach a point in our data cleaning work where it makes sense to preserve the results of that cleaning by persisting data. At the end of *Chapter 1, Anticipating Data Cleaning Issues When Importing Tabular Data with pandas*, 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 the 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 the *Importing more complicated JSON data from an API* recipe earlier 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_artis  
"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
```

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

```
response = requests.get("https://openaccess-api.cleve  
camcollections = json.loads(response.text)  
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'}
```

### 3. 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(cmcollections['data'][0]['creators'])
```

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

#### 4. Save and reload the JSON file using `msgpack`:

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

```
1586507
```

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

```
[{'biography': 'Jacob Lawrence (born 1917) has been a prominent figure in the art world for decades.',  
 '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 12, Automate Data Cleaning with User-Defined Functions, Classes and Pipelines*.

## Versioning data

There may be times when we want to persist data without overwriting a prior version of the data file. This can be accomplished by appending a time stamp to a filename or a unique identifier. However, there are more elegant solutions available. One such solution is the Delta Lake library, which we will explore in this recipe.

We will work with the land temperature data again in this recipe. We will load the data, save it to a data lake, and then save an altered version to the same data lake.

## Getting ready

We will be using the Delta Lake library in this recipe, which can be installed with `pip install deltalake`. We will also need the `os` library so that we can make a directory for the data lake.

## How to do it...

You can get started with the data and version it as follows:

1. We start by importing the Delta Lake library. We also create a folder called `temps_lake` for our data versions:

```
import pandas as pd
from deltalake.writer import write_deltalake
from deltalake import DeltaTable
import os
os.makedirs("data/temps_lake", exist_ok=True)
```

2. Now, let's load the land temperature data:

```
landtemps = pd.read_csv('data/landtempssample.csv',
    names=['stationid', 'year', 'month', 'avgtemp', 'latitude',
           'longitude', 'elevation', 'station', 'countryid'],
    skiprows=1,
    parse_dates=[['month', 'year']])
landtemps.shape
```

```
(100000, 9)
```

3. We save the `landtemps` DataFrame to the data lake:

```
write_deltalake("data/temps_lake", landtemps)
```

4. Let's now retrieve the data we just saved. We specify that we want the first version, though that is not necessary as the most recent version will be retrieved when the version is not indicated. This returns a `DeltaTable` type, which we can convert into a pandas DataFrame:

```
tempdelta = DeltaTable("data/temp_lake", version=0)
type(tempdelta)
```

```
deltalake.table.DeltaTable
```

```
tempdfv1 = tempdelta.to_pandas()
tempdfv1.shape
```

```
(100000, 9)
```

5. Let's persist only the first 1,000 rows of the land temperature data to the data lake, without replacing the existing data. We pass a value of `overwrite` to the `mode` parameter. This saves a new dataset to the data lake. It does not replace the previous one. The `overwrite` parameter value is a little confusing here. This will be clearer when we use the `append` parameter value later:

```
write_deltalake("data/temp_lake", landtemps.head(100
```

6. Let's now retrieve this latest version of our data. Notice that this only has 1,000 rows:

```
tempdfv2 = DeltaTable("data/temp_lake", version=1).
```

```
tempssdfv2.shape
```

```
(1000, 9)
```

7. If we specify `append` instead, we add the rows of the DataFrame in the second argument of `write_deltalake` to the rows of the previous version:

```
write_deltalake("data/temps_lake", landtemps.head(100)
tempssdfv3 = DeltaTable("data/temps_lake", version=2).
tempssdfv3.shape
```

```
(2000, 9)
```

8. Let's confirm that our first version of the dataset in the data lake is still accessible and has the number of rows we expect:

```
DeltaTable("data/temps_lake", version=0).to_pandas().
```

```
(100000, 9)
```

## How it works...

The nomenclature regarding overwrite and append is a little confusing, but it might make more sense if you think of overwrite as a logical deletion of the previous dataset, not a physical deletion. The most recent version has all new data, but the previous versions are still stored.

# Summary

The recipes in this chapter examined importing and data preparation of non-tabular data in a variety of forms, including JSON and HTML. We introduced Spark for working with big data and discussed how to persist tabular and non-tabular data. We also examined how to create a data lake for versioning. We will learn how to take the measure of our data in the next chapter.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.gg/p8uSgEAETX>



# 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; it's difficult 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 and follow them, 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 summary statistics for continuous variables
- Using generative AI to display descriptive statistics

## Technical requirements

You will need pandas, Numpy, and Matplotlib to complete the recipes in this chapter. I used pandas 2.1.4, but the code will run on pandas 1.5.3 or later.

The code in this chapter can be downloaded from the book's github repository, <https://github.com/PacktPublishing/Python-Data-Cleaning-Cookbook-Second-Edition>.

## 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 2023; and the counts of COVID-19 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 COVID-19 case data.

## 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 to 2023.

For this recipe, I pulled 89 variables on grades, employment, income, and attitudes toward government from the hundreds of data items in the survey. Separate files for SPSS, Stata, and SAS can be downloaded from the repository. The NLS data can be downloaded from <https://www.nlsinfo.org>. You must create an investigator account to download the data, but there is no charge.

Our World in Data provides COVID-19 public use data at <https://ourworldindata.org/covid-cases>.

The dataset includes total cases and deaths, tests administered, hospital beds, and demographic data such as median age, gross domestic product, and a human development index, which is a composite measure of standard of living, educational levels, and life expectancy.



The dataset used in this recipe was downloaded on March 3, 2024

## How to do it...

We will get an initial look at the NLS and COVID-19 data, including the number of rows and columns, and the data types:

1. Import the required 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'])
```

2. 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
```

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

```
nls97.shape
```

```
(8984, 88)
```

```
nls97.index.nunique()
```

```
8984
```

3. 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    obj    
 1   birthmonth      8984 non-null    int64  
 2   birthyear       8984 non-null    int64  
 3   highestgradecompleted  6663 non-null  float64
 4   maritalstatus   6672 non-null    obj    
 5   childathome     4791 non-null    float64
 6   childnotathome 4791 non-null    float64
 7   wageincome      5091 non-null    object  
 8   weeklyhrscomputer 6710 non-null  object  
 9   weeklyhrstv     6711 non-null    obj    
 10  nightlyhrssleep 6706 non-null    float64
 11  satverbal       1406 non-null    float64
 12  satmath         1407 non-null    float64
 ...
 83  colenroct15    7469 non-null    obj    
 84  colenrfeb16    7036 non-null    obj    
 85  colenroct16    6733 non-null    obj    
 86  colenrfeb17    6733 non-null    obj    
 87  colenroct17    6734 non-null    obj    
dtypes: float64(29), int64(2), object(57)
memory usage: 6.1+ MB
```

4. Show the first 2 rows of the nls97 data.

Use transpose to show a little more of the output:

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

|                       |                 |            |
|-----------------------|-----------------|------------|
| personid              | 100061          | 10         |
| gender                | Female          | Male       |
| birthmonth            | 5               | 9          |
| birthyear             | 1980            | 19         |
| highestgradecompleted | 13              | 12         |
| maritalstatus         | Married         | Married    |
| ...                   | ...             | ..         |
| colenroct15           | 1. Not enrolled | 1. Not enr |
| colenrfeb16           | 1. Not enrolled | 1. Not enr |
| colenroct16           | 1. Not enrolled | 1. Not enr |
| colenrfeb17           | 1. Not enrolled | 1. Not enr |
| colenroct17           | 1. Not enrolled | 1. Not enr |

5. Set and show the index and size for the COVID-19 data.

Also, check to see whether the index values are unique:

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

```
Index(['AFG', 'ALB', 'DZA', 'ASM', 'AND', 'AGO', 'AIA', 'ARM',  
       ...  
      'URY', 'UZB', 'VUT', 'VAT', 'VEN', 'VNM', 'WLF', 'Y  
      'ZWE'],  
      dtype='object', name='iso_code', length=231)
```

```
covidtotals.shape
```

```
(231, 16)
```

```
covidtotals.index.unique()
```

```
231
```

## 6. Show the data types and non-null value counts:

```
covidtotals.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 231 entries, AFG to ZWE
Data columns (total 16 columns):
 #   Column           Non-Null Count   Dtype  
--- 
 0   lastdate        231 non-null     datetime64[ns]
 1   location         231 non-null     object  
 2   total_cases      231 non-null     float64
 3   total_deaths     231 non-null     float64
 4   total_cases_pm   231 non-null     float64
 5   total_deaths_pm  231 non-null     float64
 6   population       231 non-null     int64  
 7   pop_density      209 non-null     float64
 8   median_age       194 non-null     float64
 9   gdp_per_capita   191 non-null     float64
 10  hosp_beds        170 non-null     float64
 11  vac_per_hund     13 non-null      float64
 12  aged_65_older    188 non-null     float64
 13  life_expectancy  227 non-null     float64
 14  hum_dev_ind      187 non-null     float64
 15  region           231 non-null     object  
dtypes: datetime64[ns](1), float64(12), int64(1), object(2)
memory usage: 38.8+ KB
```

7. Show a sample of 2 rows of the COVID-19 data:

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

|                 |             |               |
|-----------------|-------------|---------------|
| iso_code        | GHA         | NIU           |
| lastdate        | 2023-12-03  | 2023-12-31    |
| location        | Ghana       | Niue          |
| total_cases     | 171,834     | 993           |
| total_deaths    | 1,462       | 0             |
| total_cases_pm  | 5,133       | 508,709       |
| total_deaths_pm | 44          | 0             |
| population      | 33475870    | 1952          |
| pop_density     | 127         | NaN           |
| median_age      | 21          | NaN           |
| gdp_per_capita  | 4,228       | NaN           |
| hosp_beds       | 1           | NaN           |
| vac_per_hund    | NaN         | NaN           |
| aged_65_older   | 3           | NaN           |
| life_expectancy | 64          | 74            |
| hum_dev_ind     | 1           | NaN           |
| region          | West Africa | Oceania / Aus |

This has given us a good foundation for understanding our DataFrames, including their sizes 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 `nls97.loc[personid==1000061]`

statement to get the row for that person, we can use `nls97.loc[1000061]`.

We will try this out in the next recipe.

pandas makes it easy to view the number of rows and columns, the data type and numbers of non-missing values for each column, and the values of the columns for a few rows of your data. This can be accomplished by using the `shape` attribute and calling the `info` method, followed by either the `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 231 rows and 16 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 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 26-year span, from 1997 till 2023. 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 any 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 with 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 11, 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 `loc` and `iloc` data accessors.

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 in the NLS data of the object data type to the category data type. Do this by selecting object data type columns with `select_dtypes` and using `transform` 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[nls97.select_dtypes(['object']).columns] = \
    nls97.select_dtypes(['object']). \
    transform(lambda x: x.astype('category'))
```

2. 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'>
```

### 3. 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()
```

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

#### 4. Select multiple columns based on a list of columns.

If you are selecting more than a few columns, it is helpful to create a 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   object 
 1   maritalstatus   8984 non-null   object 
 2   highestgradecompleted  8984 non-null   object 
 3   wageincome     8984 non-null   float64
 4   gpaoverall     8984 non-null   float64
 5   weeksworked17  8984 non-null   int64  
 6   colenroct17    8984 non-null   int64 
```

```
0   gender                  8984 non-null      cat
1   maritalstatus            6672 non-null      cat
2   highestgradecompleted    6663 non-null      cat
3   wageincome               5091 non-null      float64
4   gpaoverall               6004 non-null      float64
5   weeksworked17             6670 non-null      float64
6   colenroct17                6734 non-null      cat
dtypes: category(3), float64(4)
memory usage: 377.7 KB
```

## 5. Select one or more columns by filtering by 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
```

## 6. Select all columns of 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, 1000061 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
```

## 7. 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',... 'co]
```

## 8. 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 `transform` and a `lambda` function `(transform(lambda x: x.astype('category')))` to change those columns to category. We use the `loc` accessor to only update columns with the object data type (`nls97.loc[:, nls97.dtypes == 'object']`). We go into much more detail on the use of `transform`, `apply` (which works similarly to `transform`), and `lambda` functions in *Chapter 6, Cleaning and Exploring Data with Series Operations*.

We select columns by data type in *Step 6*. `select_dtypes` becomes quite useful when passing columns to methods such as `describe` or `value_counts`, when you want to limit the analysis to continuous or categorical variables.

In Step 8, 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
 ---  -- 
 0   highestgradecompleted    6663 non-null
 1   childathome            4791 non-null
 2   childnotathome          4      791 non-null
 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
 26  weeksworked13            7680 non-null
 27  weeksworked14            7612 non-null
 28  weeksworked15            7389 non-null
 29  weeksworked16            7068 non-null
 30  weeksworked17            6670 non-null
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 constantly zoom in and out and look 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 pandas tools to detect 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)
```

2. Use slicing to start at the 1001<sup>st</sup> row and go to the 1004<sup>th</sup> 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 1001<sup>st</sup> 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 |
| gender   | Male   | Male   | Female |

|                       |      |                 |               |
|-----------------------|------|-----------------|---------------|
| birthmonth            | 12   | 9               | 3             |
| birthyear             | 1981 | 1980            | 1982          |
| highestgradecompleted | NaN  | 12              | 17            |
| maritalstatus         | NaN  | Never-married   | Never-married |
| ...                   | ...  | ...             | ...           |
| colenrfeb16           | NaN  | 1. Not enrolled | 1. Not enr    |
| colenroct16           | NaN  | 1. Not enrolled | 1. Not enr    |
| colenrfeb17           | NaN  | 1. Not enrolled | 1. Not enr    |
| colenroct17           | NaN  | 1. Not enrolled | 1. Not enr    |

3. Use slicing to start at the 1001<sup>st</sup> row and go to the 1004<sup>th</sup> 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 enr    |
| colenrfeb16           | NaN  |        | 1. Not enr    |
| colenroct16           | NaN  |        | 1. Not enr    |
| colenrfeb17           | NaN  |        | 1. Not enr    |
| colenroct17           | NaN  |        | 1. Not enr    |

4. Select the first three rows using [] operator slicing.

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[: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 |        |

Note that `nls97[:3]` returns the same DataFrame as `nls97.head(3)` would have returned.

5. Select the last three rows using `[]` operator slicing:

```
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 |        |

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

## 6. 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, since those rows are contiguous:

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

| personid              | 195884 | 195891          | 195970          |
|-----------------------|--------|-----------------|-----------------|
| gender                | Male   | Male            | Female          |
| birthmonth            | 12     | 9               | 3               |
| birthyear             | 1981   | 1980            | 1981            |
| 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          | 1981          |
| highestgradecompleted | NaN    | 12            | 17            |
| maritalstatus         | NaN    | Never-married | Never-married |

```
...      ...      ...      ...
colenroct15      NaN      1. Not enrolled 1. Not en
colenrfeb16      NaN      1. Not enrolled 1. Not en
colenroct16      NaN      1. Not enrolled 1. Not en
colenrfeb17      NaN      1. Not enrolled 1. Not en
colenroct17      NaN      1. Not enrolled 1. Not en
```

7. 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
```

8. 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`:

```
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 enrc |
| colenrfeb16 | 1. Not enrolled | 1. Not enrolled | 1. Not enrc |
| colenroct16 | 1. Not enrolled | 1. Not enrolled | 1. Not enrc |
| colenrfeb17 | 1. Not enrolled | 1. Not enrolled | 1. Not enrc |
| colenroct17 | 1. Not enrolled | 1. Not enrolled | 1. Not enrc |

We would get the same result if we passed `[0:3]` to the accessor.

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

Use `nls97.iloc[[-3, -2, -1]]` to retrieve the last three rows 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 enrc |
| colenrfeb16 | 1. Not enrolled | 1. Not enrolled | 1. Not enrc |
| colenroct16 | 1. Not enrolled | 1. Not enrolled | 1. Not enrc |

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

We would have gotten the same result with `nls97.iloc[-3:]`. 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.

## 10. 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 of 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)
```

## 11. 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 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:

```
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<=
lowsleep3pluschildren.shape
```

```
(82, 88)
```

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.
```

## 12. 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<=
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 given row to be selected. We would have used `|` for `or` if we had wanted to select the rows where *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),
```

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, *Selecting and organizing columns*, 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 of object data type to the category data type:

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

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

Note that there are no missing values for `gender` and only a 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
```

### 3. 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
```

### 4. Turn off sorting by frequency:

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

```
Divorced         663  
Married          3066  
Never-married    2766
```

```
Separation           154  
Widowed              23  
Name: maritalstatus, dtype: int64
```

5. Show percentages instead of counts:

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

| Marital Status | Percentage |
|----------------|------------|
| Divorced       | 0.10       |
| Married        | 0.46       |
| Never-married  | 0.41       |
| Separated      | 0.02       |
| Widowed        | 0.00       |

```
Name: maritalstatus, dtype: float64
```

6. 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.Series.value_counts, normalize=True)
```

| Column                | Value             | Percentage |
|-----------------------|-------------------|------------|
| govprovidejobs        | 1. Definitely     | 0.25       |
|                       | 2. Probably       | 0.34       |
|                       | 3. Probably not   | 0.25       |
|                       | 4. Definitely not | 0.16       |
| govdecenthousing      | 1. Definitely     | 0.44       |
|                       | 2. Probably       | 0.43       |
|                       | 3. Probably not   | 0.09       |
|                       | 4. Definitely not | 0.04       |
| govprotectenvironment | 1. Definitely     | 0.67       |
|                       | 2. Probably       | 0.29       |
|                       | 3. Probably not   | 0.04       |
|                       | 4. Definitely not | 0.04       |

|                   |      |      |
|-------------------|------|------|
| 3. Probably not   | 0.10 | 0.03 |
| 4. Definitely not | 0.02 | 0.02 |

7. 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.Series.value_counts, normalize=True)
```

|                   | govprovidejobs   | govpricecc            |
|-------------------|------------------|-----------------------|
| 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                  |

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

First, open a file to write out the frequencies:

```
for col in nls97.\n    select_dtypes(include=["category"]):\n        print(col, "-----",\n              "frequencies",\n              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 star of the show for this recipe is the `value_counts` method. It can generate frequencies for a Series, as we did with

`nls97.maritalstatus.value_counts`. It can also be run on a whole

DataFrame as we did with

```
nls97.filter(like="gov").apply(pd.Series.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 iterated over all of the columns with the category data type:

```
for col in nls97.select_dtypes(include=["category"]).
```

For each of those columns, we ran `value_counts` to get frequencies and `value_counts` again to get percentages. We used a `print` function to 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 generated 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 stopped short of doing two-way frequencies in

this recipe, we did start the process of splitting up the data for investigation in *Step 7*. In that step, we looked at government responsibility responses for married individuals and saw that those responses differed 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-19 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 in the terminal by entering `pip install matplotlib`.

## How to do it...

Let's take a look at the distribution of a few key continuous variables:

1. Import `pandas`, `numpy`, and `matplotlib`, and load the COVID-19 case totals data:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
covidtotals = pd.read_csv("data/covidtotals.csv",
...    parse_dates=['lastdate'])
covidtotals.set_index("iso_code", inplace=True)
```

2. Let's remind ourselves of the structure of the data:

```
covidtotals.shape
```

```
(231, 16)
```

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

```
iso_code          GHA \
lastdate         2023-12-03 00:00:00
location        Ghana
total_cases     171,834.00
total_deaths    1,462.00
total_cases_pm  5,133.07
total_deaths_pm 43.67
population      33475870
pop_density     126.72
median_age      21.10
gdp_per_capita  4,227.63
hosp_beds       0.90
vac_per_hund    NaN
aged_65_older   3.38
life_expectancy 64.07
hum_dev_ind     0.61
region          West Africa
```

```
covidtotals.dtypes
```

```
lastdate           datetime64[ns]
location          object
total_cases       float64
total_deaths      float64
total_cases_pm    float64
total_deaths_pm   float64
population        int64
pop_density       float64
median_age        float64
gdp_per_capita   float64
hosp_beds         float64
vac_per_hund      float64
aged_65_older    float64
life_expectancy   float64
hum_dev_ind       float64
```

```
region          object  
dtype: object
```

3. Get the descriptive statistics on the COVID-19 totals columns:

```
totvars = ['total_cases',  
          'total_deaths', 'total_cases_pm',  
          'total_deaths_pm']  
covidtotals[totvars].describe()
```

|       | total_cases   | total_deaths | total_cases_pm | total_deaths_pm |
|-------|---------------|--------------|----------------|-----------------|
| count | 231.0         | 231.0        | 231.0          | 231.0           |
| mean  | 3,351,598.6   | 30,214.2     | 206,177.8      | 203,858.1       |
| std   | 11,483,211.8  | 104,778.9    | 203,858.1      | 203,858.1       |
| min   | 4.0           | 0.0          | 354.5          | 354.5           |
| 25%   | 25,671.5      | 177.5        | 21,821.9       | 21,821.9        |
| 50%   | 191,496.0     | 1,937.0      | 133,946.3      | 133,946.3       |
| 75%   | 1,294,286.0   | 14,150.0     | 345,689.8      | 345,689.8       |
| max   | 103,436,829.0 | 1,127,152.0  | 763,475.4      | 763,475.4       |

4. 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:

```
covidtotals[totvars].\n    quantile(np.arange(0.0, 1.1, 0.1))
```

|     | total_cases | total_deaths |
|-----|-------------|--------------|
| 0.0 | 4.0         | 0.0          |
| 0.1 | 8,359.0     | 31.0         |
| 0.2 | 17,181.0    | 126.0        |
| 0.3 | 38,008.0    | 294.0        |

|     |                 |             |
|-----|-----------------|-------------|
| 0.4 | 74,129.0        | 844.0       |
| 0.5 | 191,496.0       | 1,937.0     |
| 0.6 | 472,755.0       | 4,384.0     |
| 0.7 | 1,041,111.0     | 9,646.0     |
| 0.8 | 1,877,065.0     | 21,218.0    |
| 0.9 | 5,641,992.0     | 62,288.0    |
| 1.0 | 103,436,829.0   | 1,127,152.0 |
|     | total_deaths_pm |             |
| 0.0 | 0.0             |             |
| 0.1 | 32.9            |             |
| 0.2 | 105.3           |             |
| 0.3 | 210.5           |             |
| 0.4 | 498.8           |             |
| 0.5 | 827.0           |             |
| 0.6 | 1,251.3         |             |
| 0.7 | 1,697.6         |             |
| 0.8 | 2,271.7         |             |
| 0.9 | 3,155.9         |             |
| 1.0 | 6,507.7         |             |

## 5. View the distribution of total cases:

```
plt.hist(covidtotals['total_cases']/1000, bins=12)
plt.title("Total COVID-19 Cases (in thousands)")
plt.xlabel('Cases')
plt.ylabel("Number of Countries")
plt.show()
```

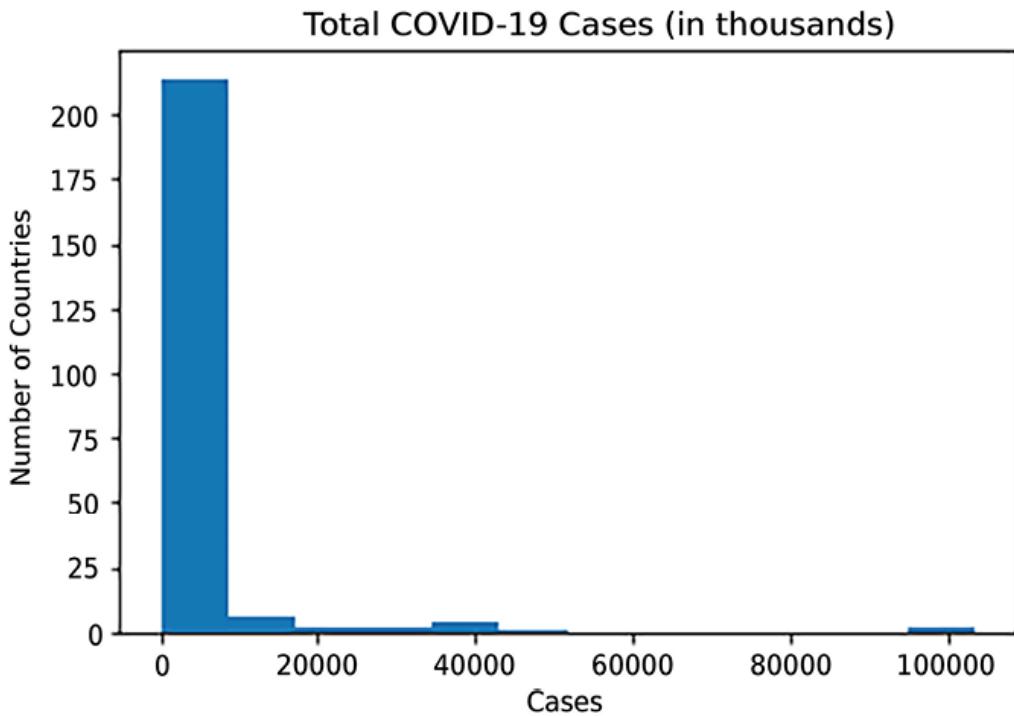


Figure 3.1: A plot of total COVID-19 cases

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 used 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 (the value at the 50<sup>th</sup> percentile) 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 90<sup>th</sup>-percentile and 100<sup>th</sup>-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.

The histogram of total cases confirms that much of the distribution is between 0 and 100,000, 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 the Identification of Unexpected Values*.

## Using generative AI to display descriptive statistics

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-19 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](https://platform.openai.com). You will need to set up an account and then click on your profile in the upper-right corner, followed by **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-19 data:

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

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

2. Next, we load the COVID-19 data and instantiate a `PandasAI SmartDataframe` object. The `SmartDataframe` object will allow us to work with our data using natural language:

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

3. Let's look at the structure of the COVID-19 data. We can do this by passing natural language instructions to the `SmartDataframe`'s `chat` method:

```
covidtotalssdf.chat("Show me some information about t
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 231 entries, 0 to 230
Data columns (total 17 columns):
 #   Column           Non-Null Count  Dtype  
 --- 
 0   iso_code         231 non-null    object  
 1   lastdate        231 non-null    datetime64[ns]
 2   location         231 non-null    object  
 3   total_cases     231 non-null    float64 
 4   total_deaths    231 non-null    float64 
 5   total_cases_pm  231 non-null    float64 
 6   total_deaths_pm 231 non-null    float64 
 7   population      231 non-null    int64   
 8   pop_density     209 non-null    float64 
 9   median_age      194 non-null    float64 
 10  gdp_per_capita  191 non-null    float64 
 11  hosp_beds       170 non-null    float64 
 12  vac_per_hund    13 non-null    float64 
 13  aged_65_older   188 non-null    float64
```

```
14 life_expectancy    227 non-null      float64
15 hum_dev_ind        187 non-null      float64
16 region             231 non-null      object
dtypes: datetime64[ns](1), float64(12), int64(1), object(3)
memory usage: 30.8+ KB
```

4. It is also straightforward to see the first few rows:

```
covidtotalssdf.chat("Show first five rows.")
```

```
      lastdate      location      tot
iso_code
AFG      2024-02-04  Afghanistan  231,539
ALB      2024-01-28   Albania     334
DZA      2023-12-03   Algeria     272
ASM      2023-09-17 American Samoa  8,3
AND      2023-05-07  Andorra     48,
          life_expectancy  hum_dev_ind  reg
iso_code
AFG      65           1           Sol
ALB      79           1           East
DZA      77           1           Nor
ASM      74           NaN          Oce
AND      84           1           Wes
[5 rows x 16 columns]
```

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

```
covidtotalssdf.chat("Show total cases for locations w")
```

```
      location      total_cases
iso_code
USA      United States  103,436,829
CHN      China          99,329,249
IND      India          45,026,139
```

|     |         |            |
|-----|---------|------------|
| FRA | France  | 38,997,490 |
| DEU | Germany | 38,437,756 |

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

Notice that we do not need to add the underscores in `total_cases_pm` or `total_deaths_pm`. The `chat` method figures out that that is what we meant:

```
covidtotalssdf.chat("Show total cases pm, total deaths pm,
```

| iso_code | total_cases_pm | total_deaths_pm | location |
|----------|----------------|-----------------|----------|
| BRN      | 763,475        | 396             | Br       |
| CYP      | 760,161        | 1,523           | Cy       |
| SMR      | 750,727        | 3,740           | Sr       |
| AUT      | 680,263        | 2,521           | Al       |
| KOR      | 667,207        | 693             | Sc       |
| FRO      | 652,484        | 527             | Fr       |
| SVN      | 639,408        | 4,697           | Sj       |
| GIB      | 628,883        | 3,458           | Gi       |
| MTQ      | 626,793        | 3,004           | Mq       |
| LUX      | 603,439        | 1,544           | Lx       |

7. We can also create a `SmartDataframe` with selected columns. When we use the `chat` method in this step, it figures out that a `SmartDataframe` should be returned:

```
covidtotalsabb = covidtotalssdf.chat("Select total ca
type(covidtotalsabb)
```

```
pandasai.smart_dataframe.SmartDataframe
```

```
covidtotalsabb
```

| iso_code | total_cases_pm | total_deaths_pm | loc |
|----------|----------------|-----------------|-----|
| AFG      | 5,630          | 194             | Afç |
| ALB      | 117,813        | 1,268           | Alk |
| DZA      | 6,058          | 153             | Alç |
| ASM      | 188,712        | 768             | Ame |
| AND      | 601,368        | 1,991           | Anc |
|          | ...            | ...             | ... |
| VNM      | 118,387        | 440             | Viç |
| WLF      | 306,140        | 690             | Wal |
| YEM      | 354            | 64              | Yem |
| ZMB      | 17,450         | 203             | Zan |
| ZWE      | 16,315         | 352             | Zin |

[231 rows x 3 columns]

8. We do not need to be very precise in the language we pass to PandasAI. Instead of writing `Select`, we could have written `Get` or `Grab`:

```
covidtotalsabb = covidtotalssdf.chat("Grab total case  
covidtotalsabb")
```

| iso_code | total_cases_pm | total_deaths_pm | location |
|----------|----------------|-----------------|----------|
| AFG      | 5,630          | 194             | Afç      |
| ALB      | 117,813        | 1,268           | Alk      |
| DZA      | 6,058          | 153             | Alç      |
| ASM      | 188,712        | 768             | Ame      |
| AND      | 601,368        | 1,991           | Anc      |
|          | ...            | ...             | ...      |
| VNM      | 118,387        | 440             | Viç      |

```
WLF      306,140          690          Wal  
YEM      354              64           Yen  
ZMB      17,450           203          Zam  
ZWE      16,315           352          Zin  
[231 rows x 3 columns]
```

9. We can select rows by summary statistic. For example, we can choose those rows where the total cases per million is greater than the 95<sup>th</sup> percentile. Note that this might take a little while to run on your machine:

```
covidtotalssdf.chat("Show total cases pm and location")
```

| iso_code | location       | total_cases_pm |
|----------|----------------|----------------|
| AND      | Andorra        | 601,368        |
| AUT      | Austria        | 680,263        |
| BRN      | Brunei         | 763,475        |
| CYP      | Cyprus         | 760,161        |
| FRO      | Faeroe Islands | 652,484        |
| FRA      | France         | 603,428        |
| GIB      | Gibraltar      | 628,883        |
| LUX      | Luxembourg     | 603,439        |
| MTQ      | Martinique     | 626,793        |
| SMR      | San Marino     | 750,727        |
| SVN      | Slovenia       | 639,408        |
| KOR      | South Korea    | 667,207        |

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

```
covidtotalssdf.chat("Summarize values for total cases")
```

|       | total_cases_pm | total_deaths_pm |
|-------|----------------|-----------------|
| count | 231            | 231             |
| mean  | 206,178        | 1,262           |
| std   | 203,858        | 1,315           |
| min   | 354            | 0               |
| 25%   | 21,822         | 141             |
| 50%   | 133,946        | 827             |
| 75%   | 345,690        | 1,998           |
| max   | 763,475        | 6,508           |

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

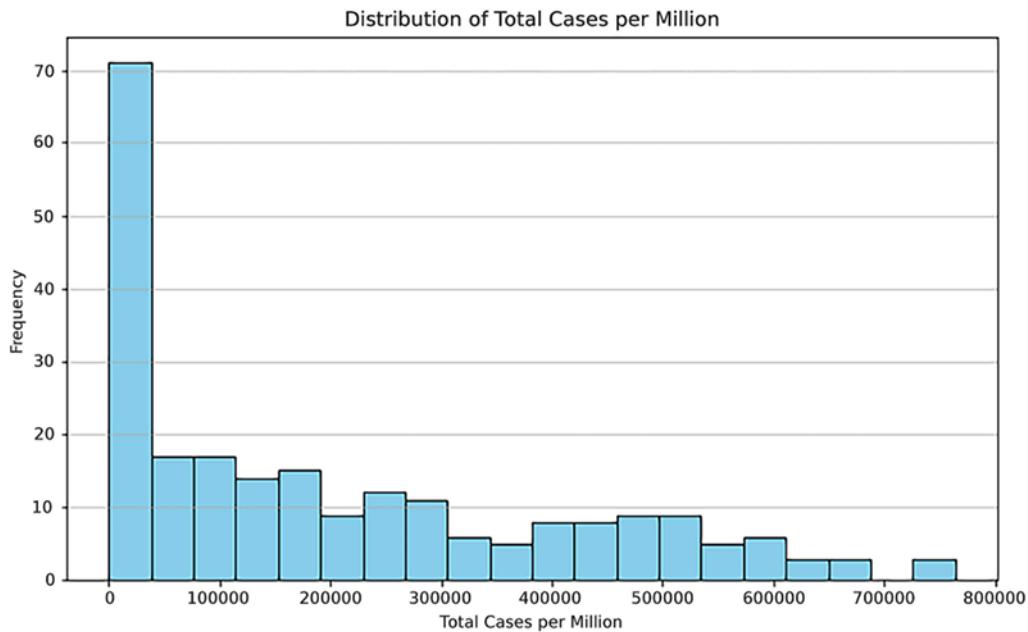
```
covidtotalssdf.chat("Show sum of total cases and tota
```

|    | region          | total_cases | total_deat |
|----|-----------------|-------------|------------|
| 0  | Caribbean       | 4,258,031   | 32,584     |
| 1  | Central Africa  | 640,579     | 8,128      |
| 2  | Central America | 4,285,644   | 54,500     |
| 3  | Central Asia    | 3,070,921   | 40,365     |
| 4  | East Africa     | 2,186,107   | 28,519     |
| 5  | East Asia       | 205,704,775 | 604,355    |
| 6  | Eastern Europe  | 62,360,832  | 969,011    |
| 7  | North Africa    | 3,727,507   | 83,872     |
| 8  | North America   | 115,917,286 | 1,516,239  |
| 9  | Oceania / Aus   | 14,741,706  | 31,730     |
| 10 | South America   | 68,751,186  | 1,354,440  |
| 11 | South Asia      | 51,507,806  | 632,374    |
| 12 | Southern Africa | 5,627,277   | 126,376    |
| 13 | West Africa     | 953,756     | 12,184     |
| 14 | West Asia       | 41,080,675  | 360,258    |
| 15 | Western Europe  | 189,405,185 | 1,124,545  |

12. We can easily generate plots on the COVID-19 data:

```
covidtotalssdf.chat("Plot the total_cases_pm column d
```

This code generates the following plot:

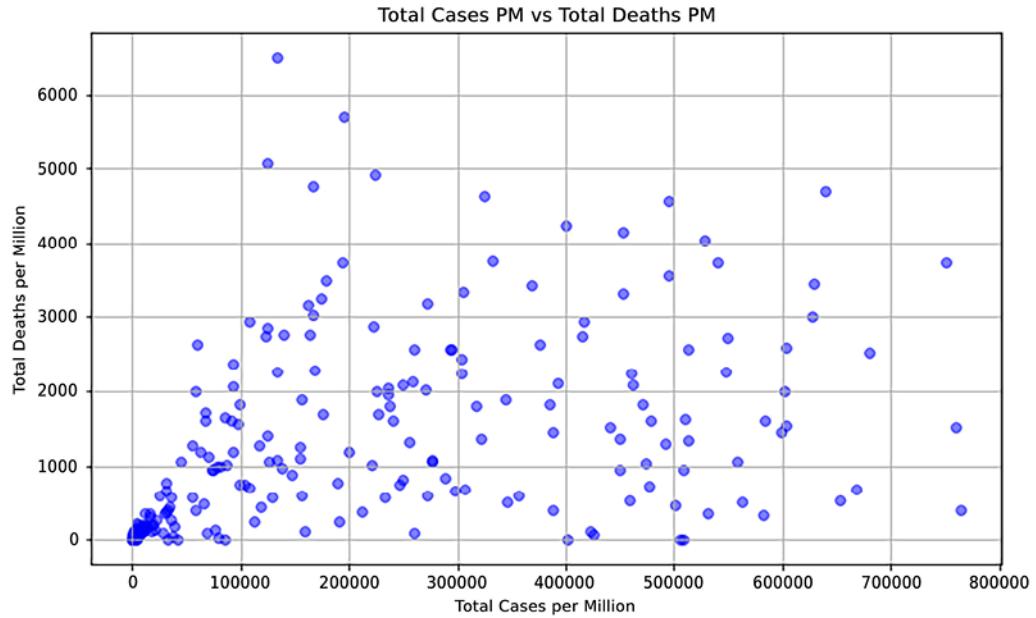


*Figure 3.2: Distribution of total cases per million*

13. We can also generate a scatterplot. Let's look at total cases per million against total deaths per million:

```
covidtotalssdf.chat( "Plot total cases pm values agai
```

This code produces the following plot:



*Figure 3.3: Scatterplot of total cases per million against total deaths per million*

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

```
covidtotalssdf.chat("Use regplot to show total deaths")
```

This code produces the following plot:

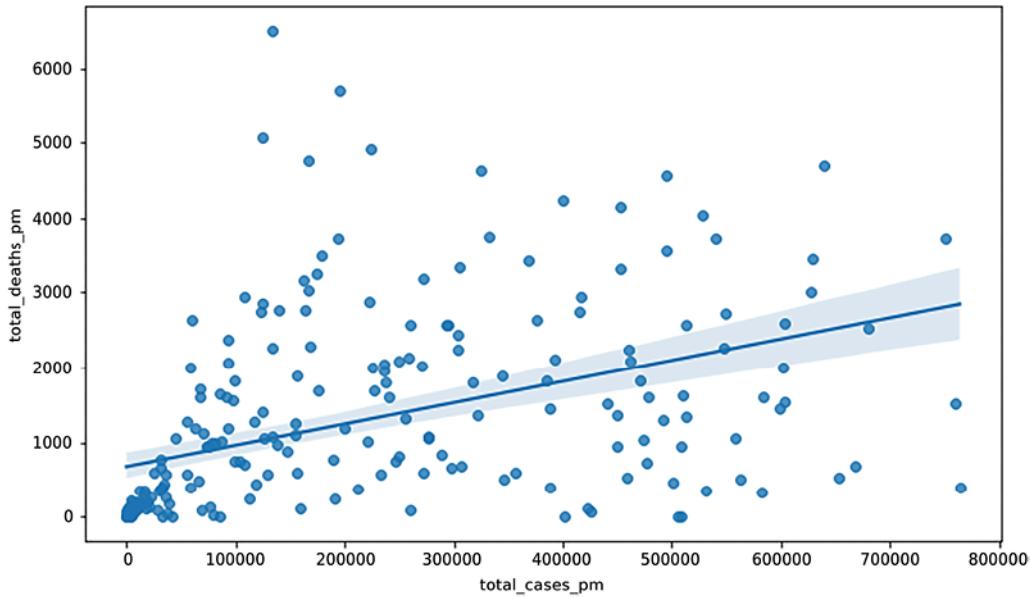


Figure 3.4: Regression plot

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

```
covidtotalssdf.chat("Use regplot to show total deaths")
```

This produces the following plot:

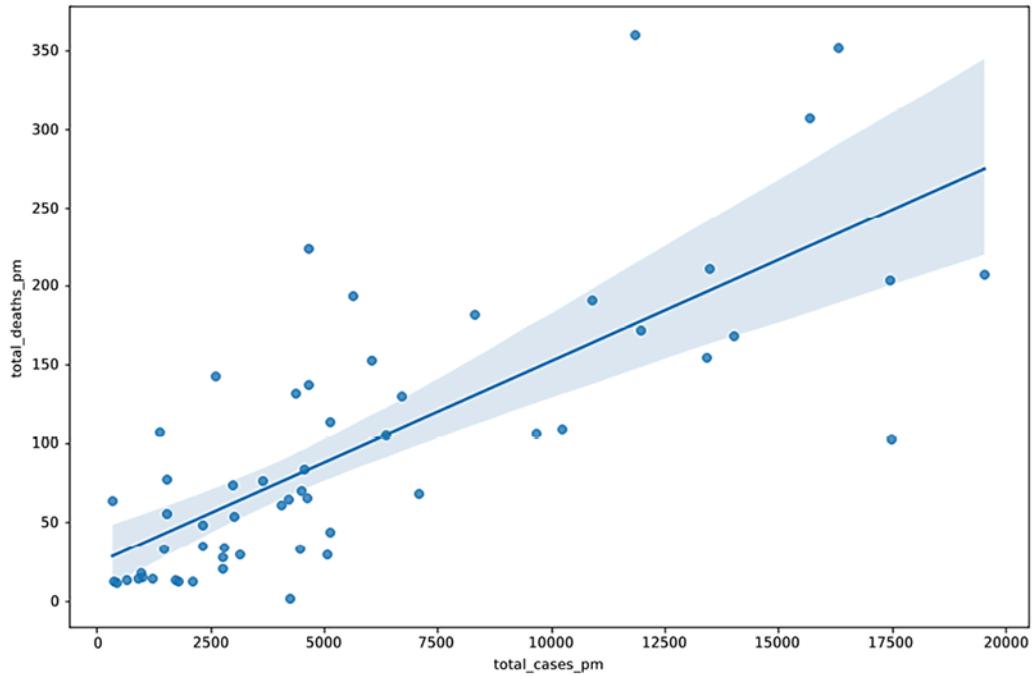


Figure 3.5: Regression plot without the extreme values

This removed deaths per million above 350 and cases per million above 20,000. 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 to write

`select`, `get`, or even `grab` to choose columns. OpenAI's large language model is generally very good at understanding what we mean.

It is a good idea to check the PandasAI log file to see the code that is generated when you pass instructions to the SmartDataframe `chat` method. The `pandasai.log` file will be in the same folder as your Python code.

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 recipes throughout this book.

## Summary

This chapter covered key steps we need to take the day after we convert our raw data into a pandas DataFrame. We explored techniques for examining the structure of our data, including the number of rows and columns, and data types. We also learned how to generate frequencies for categorical variables, and began to look at how values for one variable change with the values of another variable. Finally, we saw how to examine the distribution of continuous variables, including with sample statistics such as the mean, minimum, and max, and by plotting. This sets us up for the topics in the next chapter, where we will use techniques to identify outliers in our data.

# Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.gg/p8uSgEAETX>



# 4

## Identifying 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 is 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. This may suggest some measurement or data collection error.

But the key issue is the undue influence that outliers can have on the inferences we draw from our data. Indeed, it may be helpful to think of an outlier as an observation with variable values, or relationships between variable values, that are so unusual that they cannot help to explain relationships in the rest of the data. This matters for statistical inference because we cannot assume a neutral impact of outliers on our summary

statistics or parameter estimates. Sometimes our models work so hard to construct parameter estimates that can account for patterns in outlier observations that we compromise the model’s explanatory or predictive power for all other observations. Raise your hand if you have ever spent days trying to interpret a model only to discover that your coefficients and predictions completely changed once you removed a few outliers.

The identification and handling of outliers is among the most important data preparation tasks we have in a data analysis project. We explore a range of strategies for detecting and treating outliers in this chapter. Specifically, the recipes in this chapter examine the following:

- 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 neighbors (KNN) to find outliers
- Using Isolation Forest to find anomalies
- Using PandasAI to identify outliers

## Technical requirements

You will need pandas, NumPy, and Matplotlib to complete the recipes in this chapter. I used pandas 2.1.4, but the code will run on pandas 1.5.3 or later.

The code in this chapter can be downloaded from the book’s GitHub repository, <https://github.com/PacktPublishing/Python->

[Data-Cleaning-Cookbook-Second-Edition.](#)

# 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-19 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). You may also need to install `openpyxl` to save Excel files.

We will work with COVID-19 cases data in this recipe. This dataset has one observation for each country with total COVID-19 cases and deaths.

### Data note

Our World in Data provides COVID-19 public use data at <https://ourworldindata.org/covid-cases>.

The dataset includes total cases and deaths, tests administered, hospital beds, and demographic data such as median age, gross domestic product, and a human development index, which is a composite measure of standard of living, educational levels, and life expectancy. The dataset used in this recipe was downloaded on March 3, 2024.



## How to do it...

We take a good look at the distribution of some of the key continuous variables in the COVID-19 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-19 case data file.

Also, set up the COVID-19 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/covidtotals.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', 'hum_dev_ind']
```

2. Get descriptive statistics for the COVID-19 case data.

Create a DataFrame with just the key case data:

```
covidtotalsonly = covidtotals.loc[:, totvars]
covidtotalsonly.describe()
```

|       | total_cases | total_deaths | total_cases_pm | tot |
|-------|-------------|--------------|----------------|-----|
| count | 231         | 231          | 231            | 231 |
| mean  | 3,351,599   | 30,214       | 206,178        |     |
| std   | 11,483,212  | 104,779      | 2 03,858       |     |
| min   | 4           | 0            | 354            |     |
| 25%   | 25,672      | 178          | 21,822         |     |
| 50%   | 191,496     | 1,937        | 133,946        |     |
| 75%   | 1,294,286   | 14,150       | 345,690        |     |
| max   | 103,436,829 | 1,127,152    | 763,475        |     |

3. Show more detailed percentile data. We indicate that we only want to do this for numeric values so that the location column is skipped:

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

|     | total_cases   | total_deaths | total_cases_pm | total_deat |
|-----|---------------|--------------|----------------|------------|
| 0.0 | 4.0           | 0.0          | 354.5          |            |
| 0.1 | 8,359.0       | 31.0         | 3,138.6        |            |
| 0.2 | 17,181.0      | 126.0        | 10,885.7       |            |
| 0.3 | 38,008.0      | 294.0        | 35,834.6       |            |
| 0.4 | 74,129.0      | 844.0        | 86,126.2       |            |
| 0.5 | 191,496.0     | 1,937.0      | 133,946.3      |            |
| 0.6 | 472,755.0     | 4,384.0      | 220,429.4      | 1,         |
| 0.7 | 1,041,111.0   | 9,646.0      | 293,737.4      | 1,         |
| 0.8 | 1,877,065.0   | 21,218.0     | 416,608.1      | 2,         |
| 0.9 | 5,641,992.0   | 62,288.0     | 512,388.4      | 3,         |
| 1.0 | 103,436,829.0 | 1,127,152.0  | 763,475.4      | 6,         |

### Note



Starting with pandas version 2.0.0, the default value for the `numeric_only` parameter is `False` for the `quantile` function. We needed to set the `numeric_only` value to `True` to get `quantile` to skip the `location` column.

You should 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, for `total_cases` and `total_deaths`, are significantly higher than we would expect if our variables were distributed normally:

```
covidtotalsonly.skew(numeric_only=True)
```

```
total_cases      6.3
total_deaths     7.1
total_cases_pm    0.8
total_deaths_pm   1.3
dtype: float64
```

```
covidtotalsonly.kurtosis(numeric_only=True)
```

```
total_cases      47.1
total_deaths     61.7
total_cases_pm    -0.4
total_deaths_pm   1.3
dtype: float64
```

The prototypical normal distribution has a skewness of 0 and a kurtosis of 3.

#### 4. Test the COVID-19 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", covidt
print("total deaths: %.5f" % testnorm("total_deaths", covi
print("total cases pm: %.5f" % testnorm("total_cases_pm",
print("total deaths pm: %.5f" % testnorm("total_deaths_pm"
```

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

5. 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 scatterplots:

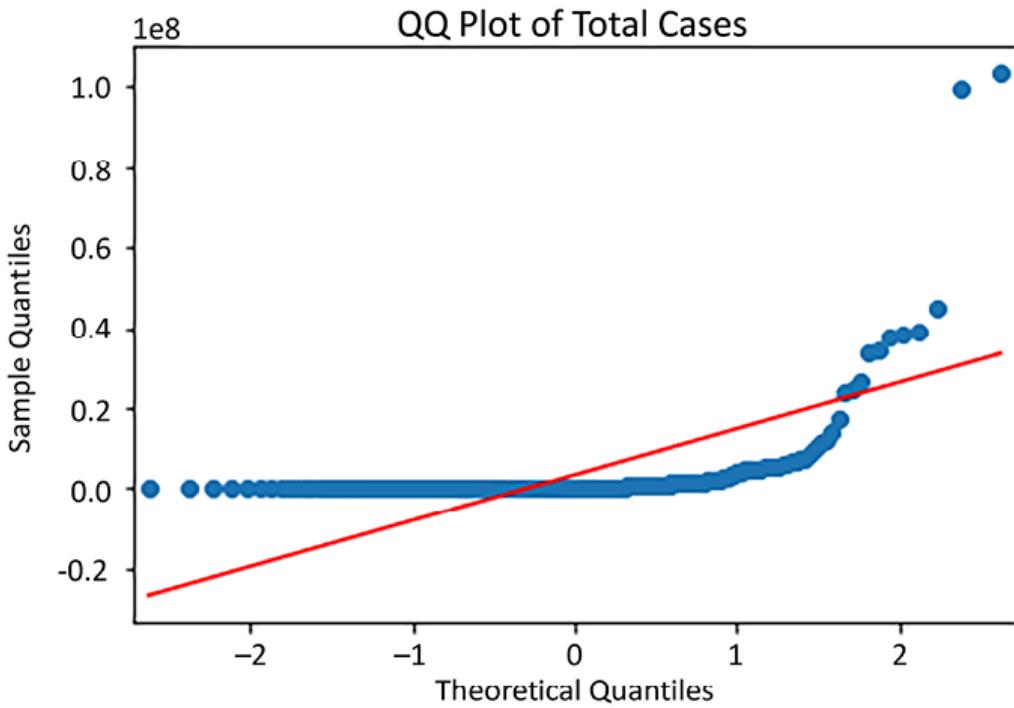


Figure 4.1: Distribution of COVID-19 cases compared with a normal distribution

When adjusted by population with the total cases per million column, the distribution is closer to normal:

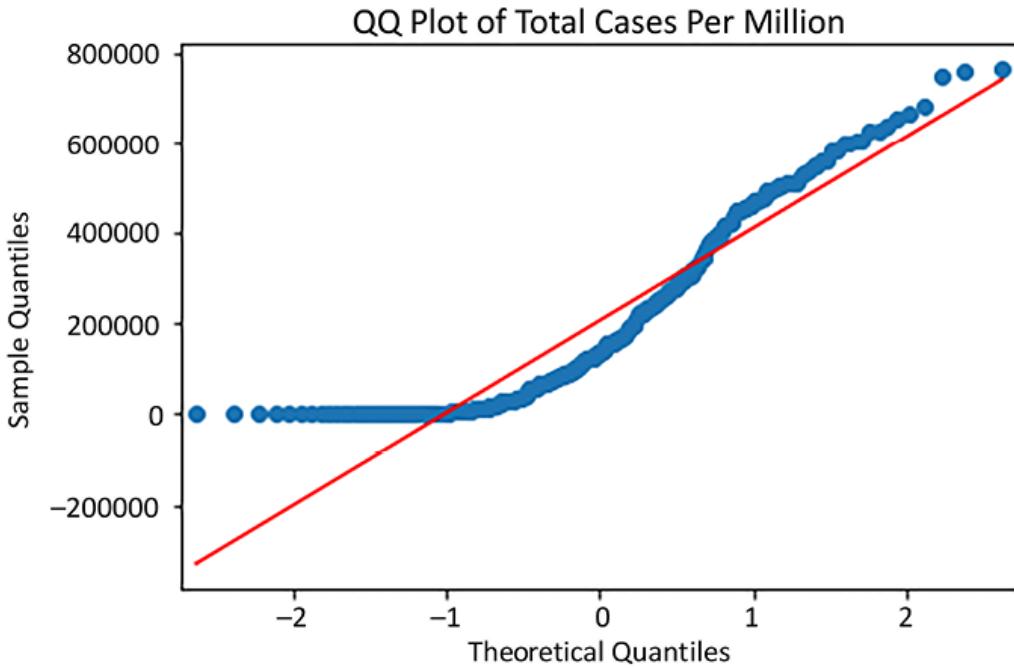


Figure 4.2: Distribution of COVID-19 cases per million compared with a normal distribution

## 6. Show the outlier range for total cases.

One way to define an outlier for a continuous variable is by the 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. The calculation in this step indicates that values above 3,197,208 can be considered outliers. In this case, we can ignore an outlier threshold that is less than 0, as that is not possible:

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

```
-1877250 <--> 3197208
```

## 7. Generate a DataFrame of outliers and write it to Excel.

Iterate over the four COVID-19 case columns. Calculate the outlier thresholds for each column as we did in the previous step. From the DataFrame, select 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      39
total_cases       33
total_deaths_pm    4
Name: varname, dtype: int64
```

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

This produces the following Excel file (some columns are hidden to save space):

| A  | B                     | C        | D           | E            | F              | G               | J          | K              | P                       | Q           | R          | S         | T          |
|----|-----------------------|----------|-------------|--------------|----------------|-----------------|------------|----------------|-------------------------|-------------|------------|-----------|------------|
| 1  | lastdate              | location | total_cases | total_deaths | total_cases_pm | total_deaths_pm | median_age | gdp_per_capita | hum_dev_ind             | region      | varname    | threshlow | threshhigh |
| 2  | 2024-01-26Argentina   | 10094643 | 130733      | 221809.957   | 2872.601       | 31.9            | 18933.907  | 0.845          | South Amer total_cases  | -1877250.25 | 3197207.75 |           |            |
| 3  | 2024-02-04Australia   | 11769858 | 24566       | 449618.889   | 938.443        | 37.9            | 44648.71   | 0.944          | Oceania / A total_cases | -1877250.25 | 3197207.75 |           |            |
| 4  | 2023-07-02Austria     | 6081287  | 22534       | 680262.588   | 2520.69        | 44.4            | 45436.686  | 0.922          | Western Eu total_cases  | -1877250.25 | 3197207.75 |           |            |
| 5  | 2024-01-26Belgium     | 4855952  | 34339       | 416608.106   | 2946.056       | 41.8            | 42658.576  | 0.931          | Western Eu total_cases  | -1877250.25 | 3197207.75 |           |            |
| 6  | 2023-10-01Brazil      | 37519960 | 702116      | 174257.347   | 3260.901       | 33.5            | 14103.452  | 0.765          | South Amer total_cases  | -1877250.25 | 3197207.75 |           |            |
| 7  | 2024-02-04Canada      | 4774222  | 54127       | 124153.047   | 1407.566       | 41.4            | 44017.591  | 0.929          | North Amer total_cases  | -1877250.25 | 3197207.75 |           |            |
| 8  | 2024-01-26Chile       | 5345008  | 62288       | 272652.519   | 3177.354       | 35.4            | 22767.037  | 0.851          | South Amer total_cases  | -1877250.25 | 3197207.75 |           |            |
| 9  | 2024-02-04China       | 99329249 | 121933      | 69661.357    | 85.514         | 38.7            | 15308.712  | 0.761          | East Asia total_cases   | -1877250.25 | 3197207.75 |           |            |
| 10 | 2024-01-26Colombia    | 6393550  | 142727      | 123251.466   | 2751.415       | 32.2            | 13254.949  | 0.767          | South Amer total_cases  | -1877250.25 | 3197207.75 |           |            |
| 11 | 2024-01-26Czechia     | 4756085  | 43478       | 453219.891   | 4143.133       | 43.3            | 32605.906  | 0.9            | Western Eu total_cases  | -1877250.25 | 3197207.75 |           |            |
| 12 | 2024-01-07Denmark     | 3433033  | 9436        | 583624.93    | 1604.146       | 42.3            | 46682.515  | 0.94           | Western Eu total_cases  | -1877250.25 | 3197207.75 |           |            |
| 13 | 2023-06-25France      | 38997490 | 167985      | 603427.621   | 2599.316       | 42              | 38605.671  | 0.901          | Western Eu total_cases  | -1877250.25 | 3197207.75 |           |            |
| 14 | 2023-07-02Germany     | 38437756 | 174979      | 461051.095   | 2098.829       | 46.6            | 45229.245  | 0.947          | Western Eu total_cases  | -1877250.25 | 3197207.75 |           |            |
| 15 | 2024-01-26Greece      | 5611832  | 38806       | 540380.08    | 3736.746       | 45.3            | 24574.382  | 0.888          | Eastern Eur total_cases | -1877250.25 | 3197207.75 |           |            |
| 16 | 2024-02-04India       | 45026139 | 533454      | 31771.799    | 376.421        | 28.2            | 6426.674   | 0.645          | South Asia total_cases  | -1877250.25 | 3197207.75 |           |            |
| 17 | 2024-02-04Indonesia   | 6828268  | 162054      | 24784.881    | 588.215        | 29.3            | 11188.744  | 0.718          | East Asia total_cases   | -1877250.25 | 3197207.75 |           |            |
| 18 | 2024-02-04Iran        | 7626527  | 146799      | 86126.235    | 1657.799       | 32.4            | 19082.62   | 0.783          | West Asia total_cases   | -1877250.25 | 3197207.75 |           |            |
| 19 | 2023-10-25Israel      | 4841558  | 12707       | 512388.401   | 1344.798       | 30.6            | 33132.32   | 0.919          | West Asia total_cases   | -1877250.25 | 3197207.75 |           |            |
| 20 | 2024-01-26Italy       | 26699442 | 195996      | 452245.686   | 3319.858       | 47.9            | 35220.084  | 0.892          | Western Eu total_cases  | -1877250.25 | 3197207.75 |           |            |
| 21 | 2023-05-14Japan       | 33803572 | 74694       | 272715.688   | 602.606        | 48.2            | 39002.223  | 0.919          | East Asia total_cases   | -1877250.25 | 3197207.75 |           |            |
| 22 | 2024-01-26Malaysia    | 5269967  | 37340       | 155281.203   | 1100.235       | 29.9            | 26808.164  | 0.81           | East Asia total_cases   | -1877250.25 | 3197207.75 |           |            |
| 23 | 2023-12-31Mexico      | 7702809  | 334958      | 60412.236    | 2627.037       | 29.3            | 17336.469  | 0.779          | North Amer total_cases  | -1877250.25 | 3197207.75 |           |            |
| 24 | 2024-01-26Netherlands | 8633769  | 22986       | 491559.962   | 1308.698       | 43.2            | 48472.545  | 0.944          | Western Eu total_cases  | -1877250.25 | 3197207.75 |           |            |
| 25 | 2023-12-31Peru        | 4536733  | 221583      | 133238.998   | 6507.656       | 29.1            | 12236.706  | 0.777          | South Amer total_cases  | -1877250.25 | 3197207.75 |           |            |
| 26 | 2024-01-21Philippines | 4140383  | 66864       | 35829.167    | 578.613        | 25.2            | 7599.188   | 0.718          | East Asia total_cases   | -1877250.25 | 3197207.75 |           |            |
| 27 | 2024-01-26Poland      | 6653365  | 120577      | 166930.3     | 3025.229       | 41.8            | 27216.445  | 0.88           | Eastern Eur total_cases | -1877250.25 | 3197207.75 |           |            |

Figure 4.3 Excel file with outlier cases

There were 39 countries identified as outliers in the `total_deaths` values according to the interquartile method, and 33 `total_cases` outliers. Notice that there were no outliers for `total_cases_pm`.

## 8. Look a little more closely at outliers for total deaths per million.

Use the `varname` column we created in the previous step to select the outliers for `total_deaths_pm`. Show the columns (`median_age` and `hum_dev_ind`) that might help to explain the extreme values for those columns. We also show the 25<sup>th</sup>, 50<sup>th</sup>, and 75<sup>th</sup> percentile for those columns for the whole dataset for comparison:

```
outliers.loc[outliers.varname=="total_deaths_pm",
['location','total_deaths_pm','total_cases_pm',
'median_age','hum_dev_ind']]. \
sort_values(['total_deaths_pm'], ascending=False)
```

|     | location               | total_deaths_pm |
|-----|------------------------|-----------------|
| PER | Peru                   | 6,507.7         |
| BGR | Bulgaria               | 5,703.5         |
| BIH | Bosnia and Herzegovina | 5,066.3         |

|     | Hungary        | 4,918.3    |             |
|-----|----------------|------------|-------------|
|     | total_cases_pm | median_age | hum_dev_ind |
| HUN |                |            |             |
| PER | 133,239.0      | 29.1       | 0.8         |
| BGR | 195,767.9      | 44.7       | 0.8         |
| BIH | 124,806.3      | 42.5       | 0.8         |
| HUN | 223,685.2      | 43.4       | 0.9         |

```
covidtotals[['total_deaths_pm', 'median_age',
 'hum_dev_ind']]. \
quantile([0.25, 0.5, 0.75])
```

|      | total_deaths_pm | median_age | hum_dev_ind |
|------|-----------------|------------|-------------|
| 0.25 | 141.18          | 22.05      | 0.60        |
| 0.50 | 827.05          | 29.60      | 0.74        |
| 0.75 | 1,997.51        | 38.70      | 0.83        |

All four countries are well beyond the 75<sup>th</sup> percentile for deaths per million. Three of the four countries are near or above the 75<sup>th</sup> percentile for both median age and the human development index. Surprisingly, there is a positive correlation between the human development index and deaths per million. We display a correlation matrix in the next recipe.

## 9. Show a histogram of total cases:

```
plt.hist(covidtotalsonly['total_cases']/1000, bins=7)
plt.title("Total COVID-19 Cases (thousands)")
plt.xlabel('Cases')
plt.ylabel("Number of Countries")
plt.show()
```

This code produces the following plot:

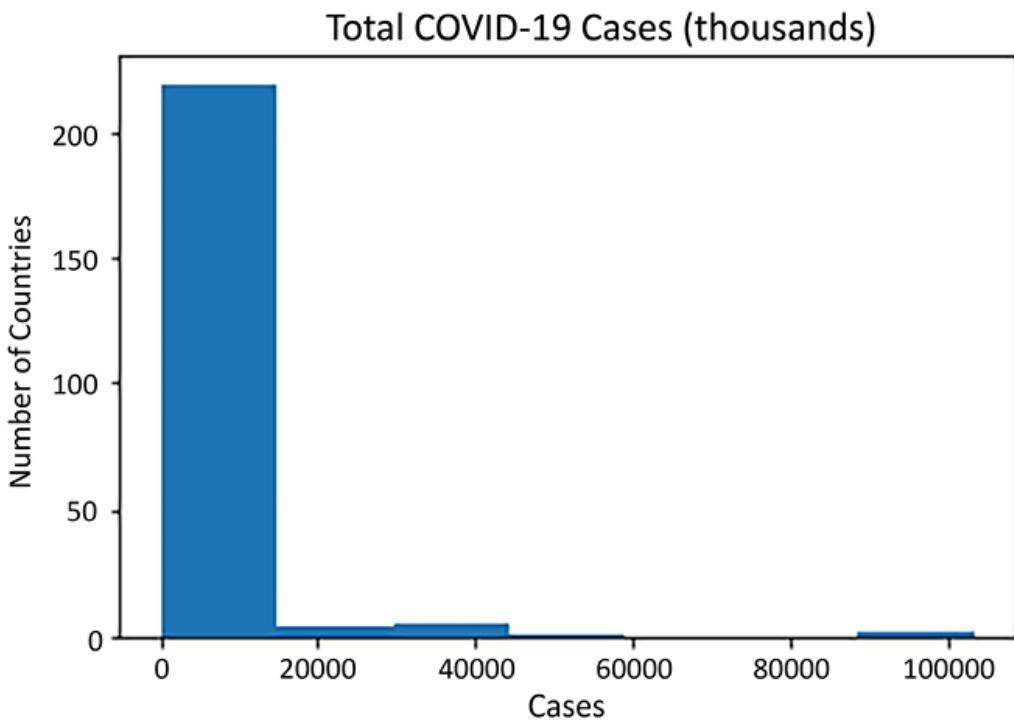


Figure 4.4: Histogram of total COVID-19 cases

10. Perform a log transformation of the COVID-19 data. Show a histogram of the log transformation of total cases:

```
covidlogs = covidtotalsonly.copy()
for col in covidlogs.columns[1:]:
    ...    covidlogs[col] = np.log1p(covidlogs[col])
plt.hist(covidlogs['total_cases'], bins=7)
plt.title("Total COVID-19 Cases (log)")
plt.xlabel('Cases')
plt.ylabel("Number of Countries")
plt.show()
```

This code produces the following:

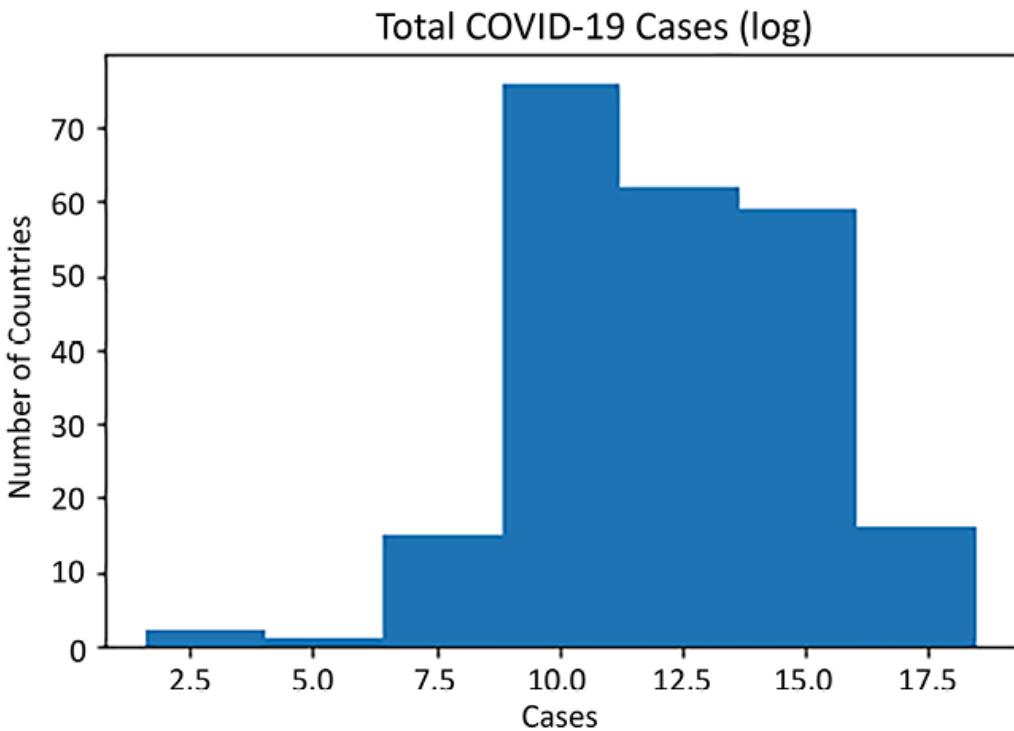


Figure 4.5: Histogram of total COVID-19 cases with log transformation

The tools we used in the preceding steps tell us a fair bit about how COVID-19 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 20<sup>th</sup> and 30<sup>th</sup> percentiles, and compare it with the range from the 70<sup>th</sup> to the 80<sup>th</sup> 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-19 variables are not normal at high levels of significance.

This is consistent with the `qqplots` that we ran 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 the 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 deaths per million, as displayed in *step 8*. Median age and the human development index seem like they might be a part of the story. 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 at the distribution of total cases 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-19 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 statistically quite problematic. 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

Boxplots might have also been illuminating here. We do a box plot on this data in *Chapter 5, Using Visualizations for the Identification of Unexpected Values*. We examine variable transformations in more detail in *Chapter 8, Encoding, Transforming, and Scaling Features*.

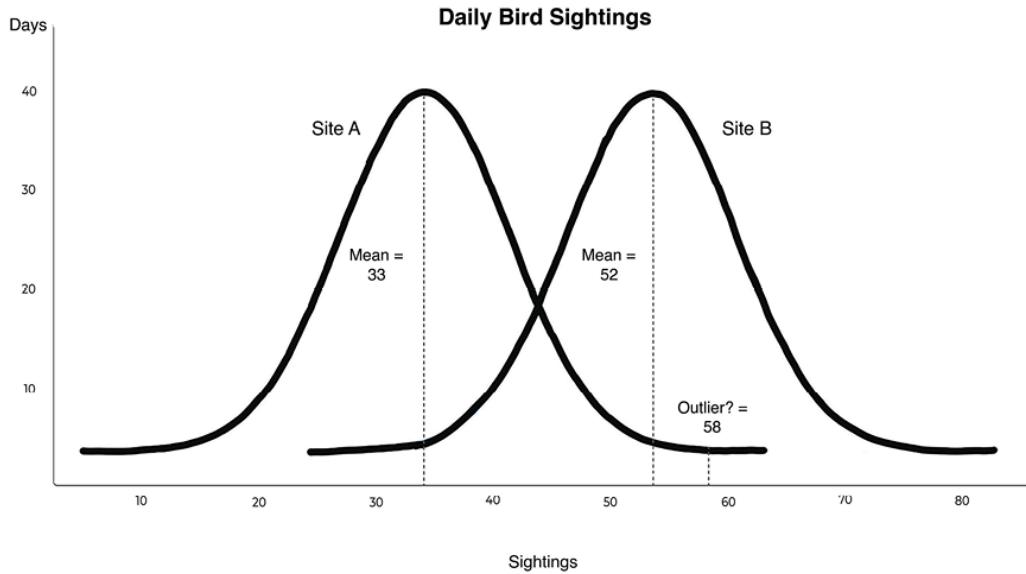
We explore bivariate relationships in this same dataset in the next recipe for any insights they might provide about outliers and unexpected values. In subsequent chapters, we consider strategies for imputing values for missing data and for making adjustments to extreme values.

# 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 period of several years, 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 4.6: 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  $y$  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 in the COVID-19 database. 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-19 cumulative data:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
covidtotals = pd.read_csv("data/covidtotals.csv")
covidtotals.set_index("iso_code", inplace=True)
```

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

Unsurprisingly, there is a high correlation (`0.76`) between total cases and total deaths and less of a correlation (`0.44`) between total cases per million and total deaths per million. There is a strong (`0.66`) relationship between GDP per capita and cases per million (Note that not all of the correlations are shown):

```
covidtotals.corr(method="pearson", numeric_only=True)
```

|                 | total_cases     | total_deaths | \               |
|-----------------|-----------------|--------------|-----------------|
| total_cases     | 1.00            | 0.76         |                 |
| total_deaths    | 0.76            | 1.00         |                 |
| total_cases_pm  | 0.10            | 0.01         |                 |
| total_deaths_pm | 0.15            | 0.27         |                 |
| population      | 0.70            | 0.47         |                 |
| pop_density     | -0.03           | -0.04        |                 |
| median_age      | 0.29            | 0.19         |                 |
| gdp_per_capita  | 0.19            | 0.13         |                 |
| hosp_beds       | 0.21            | 0.05         |                 |
| vac_per_hund    | 0.02            | -0.07        |                 |
| aged_65_older   | 0.29            | 0.19         |                 |
| life_expectancy | 0.19            | 0.11         |                 |
| hum_dev_ind     | 0.26            | 0.21         |                 |
|                 | total_cases_pm  | ...          | aged_65_older \ |
| total_cases     | 0.10            | ...          | 0.29            |
| total_deaths    | 0.01            | ...          | 0.19            |
| total_cases_pm  | 1.00            | ...          | 0.72            |
| total_deaths_pm | 0.44            | ...          | 0.68            |
| population      | -0.13           | ...          | -0.01           |
| pop_density     | 0.19            | ...          | 0.07            |
| median_age      | 0.74            | ...          | 0.92            |
| gdp_per_capita  | 0.66            | ...          | 0.51            |
| hosp_beds       | 0.48            | ...          | 0.65            |
| vac_per_hund    | 0.24            | ...          | 0.35            |
| aged_65_older   | 0.72            | ...          | 1.00            |
| life_expectancy | 0.69            | ...          | 0.73            |
| hum_dev_ind     | 0.76            | ...          | 0.78            |
|                 | life_expectancy | hum_dev_ind  |                 |
| total_cases     | 0.19            | 0.26         |                 |
| total_deaths    | 0.11            | 0.21         |                 |
| total_cases_pm  | 0.69            | 0.76         |                 |
| total_deaths_pm | 0.49            | 0.60         |                 |
| population      | -0.04           | -0.02        |                 |
| pop_density     | 0.20            | 0.14         |                 |
| median_age      | 0.83            | 0.90         |                 |
| gdp_per_capita  | 0.68            | 0.75         |                 |
| hosp_beds       | 0.46            | 0.57         |                 |

```

vac_per_hund          0.67          0.51
aged_65_older         0.73          0.78
life_expectancy       1.00          0.91
hum_dev_ind           0.91          1.00
[13 rows x 13 columns]

```

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

Use `qcut` to create a column that breaks the data into quantiles. Show a crosstab of total cases quantiles by total deaths quantiles:

```

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

```

| total_deaths_q | very low | low | medium | high | very |
|----------------|----------|-----|--------|------|------|
| total_cases_q  |          |     |        |      |      |
| very low       | 36       | 10  | 1      | 0    |      |
| low            | 11       | 26  | 8      | 1    |      |
| medium         | 0        | 9   | 27     | 10   |      |
| high           | 0        | 1   | 8      | 31   |      |
| very high      | 0        | 0   | 2      | 4    |      |

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

There is one country with high total cases but low total 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[(covidtotals. \
    total_cases_q=="high") & \
    (covidtotals.total_deaths_q=="low")].T
```

```
iso_code          QAT
lastdate   2023-06-25
location      Qatar
total_cases   514,524.00
total_deaths     690.00
total_cases_pm  190,908.72
total_deaths_pm    256.02
population      2695131
pop_density       227.32
median_age        31.90
gdp_per_capita   116,935.60
hosp_beds         1.20
vac_per_hund      NaN
aged_65_older      1.31
life_expectancy    80.23
hum_dev_ind        0.85
region        West Asia
```

## 5. Make a scatterplot of total cases by total deaths.

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

```
ax = sns.regplot(x=covidtotals.total_cases/1000, y=covidtotals.total_deaths)
ax.set(xlabel="Cases (thousands)", ylabel="Deaths", title="")
```

```
plt.show()
```

This produces the following scatterplot:

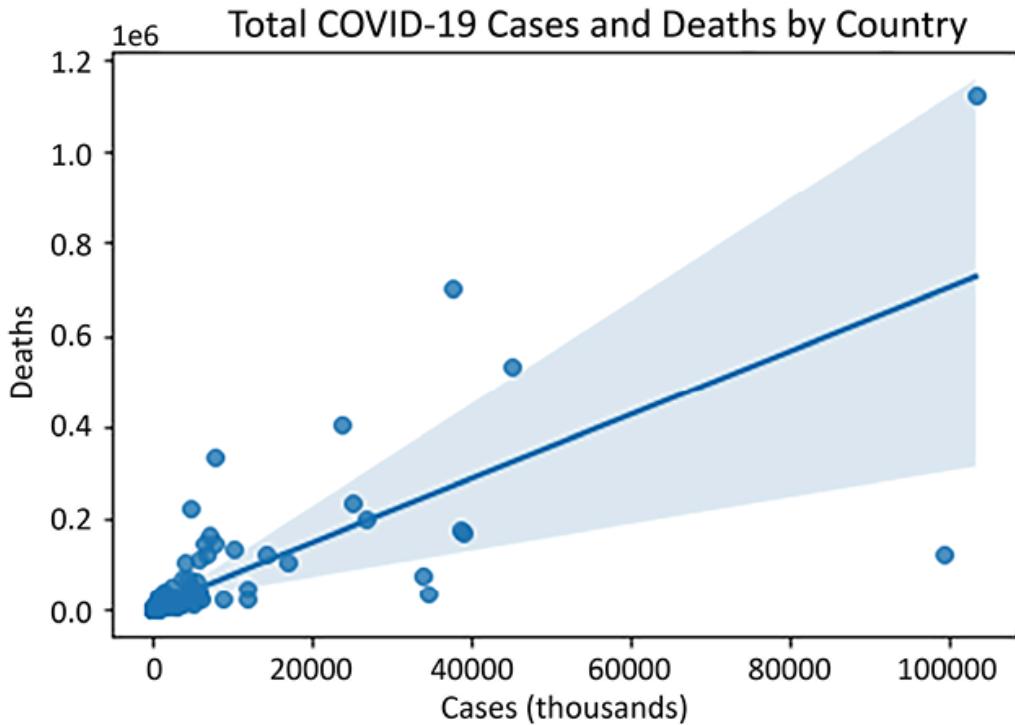


Figure 4.7: Scatterplot of total cases and deaths with a linear regression line

## 6. 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 two countries with fewer than 40 million cases and more than 400 thousand deaths:

```
covidtotals.loc[(covidtotals.total_cases<40000000) \ 
& (covidtotals.total_deaths>400000)].T
```

|                 |               |                |
|-----------------|---------------|----------------|
| iso_code        | BRA           | RUS            |
| lastdate        | 2023-10-01    | 2024-01-28     |
| location        | Brazil        | Russia         |
| total_cases     | 37,519,960.00 | 23,774,451.00  |
| total_deaths    | 702,116.00    | 401,884.00     |
| total_cases_pm  | 174,257.35    | 164,286.55     |
| total_deaths_pm | 3,260.90      | 2,777.11       |
| population      | 215313504     | 144713312      |
| pop_density     | 25.04         | 8.82           |
| median_age      | 33.50         | 39.60          |
| gdp_per_capita  | 14,103.45     | 24,765.95      |
| hosp_beds       | 2.20          | 8.05           |
| vac_per_hund    | NaN           | NaN            |
| aged_65_older   | 8.55          | 14.18          |
| life_expectancy | 75.88         | 72.58          |
| hum_dev_ind     | 0.77          | 0.82           |
| region          | South America | Eastern Europe |

## 7. Examine unexpected values below the regression line.

There are two countries with more than 30 million cases but fewer than 100 thousand deaths:

```
covidtotals.loc[(covidtotals.total_cases>30000000) \
& (covidtotals.total_deaths<100000)].T
```

|                 |               |               |
|-----------------|---------------|---------------|
| iso_code        | JPN           | KOR           |
| lastdate        | 2023-05-14    | 2023-09-10    |
| location        | Japan         | South Korea   |
| total_cases     | 33,803,572.00 | 34,571,873.00 |
| total_deaths    | 74,694.00     | 35,934.00     |
| total_cases_pm  | 272,715.69    | 667,207.06    |
| total_deaths_pm | 602.61        | 693.50        |
| population      | 123951696     | 51815808      |
| pop_density     | 347.78        | 527.97        |
| median_age      | 48.20         | 43.40         |
| gdp_per_capita  | 39,002.22     | 35,938.37     |
| hosp_beds       | 13.05         | 12.27         |

|                 |           |           |
|-----------------|-----------|-----------|
| vac_per_hund    | NaN       | NaN       |
| aged_65_older   | 27.05     | 13.91     |
| life_expectancy | 84.63     | 83.03     |
| hum_dev_ind     | 0.92      | 0.92      |
| region          | East Asia | East Asia |

8. Make a scatterplot of total cases per million by total deaths per million:

```
ax = sns.regplot(x="total_cases_pm", y="total_deaths_pm",
ax.set(xlabel="Cases Per Million", ylabel="Deaths Per Million")
plt.show()
```

This produces the following scatterplot:

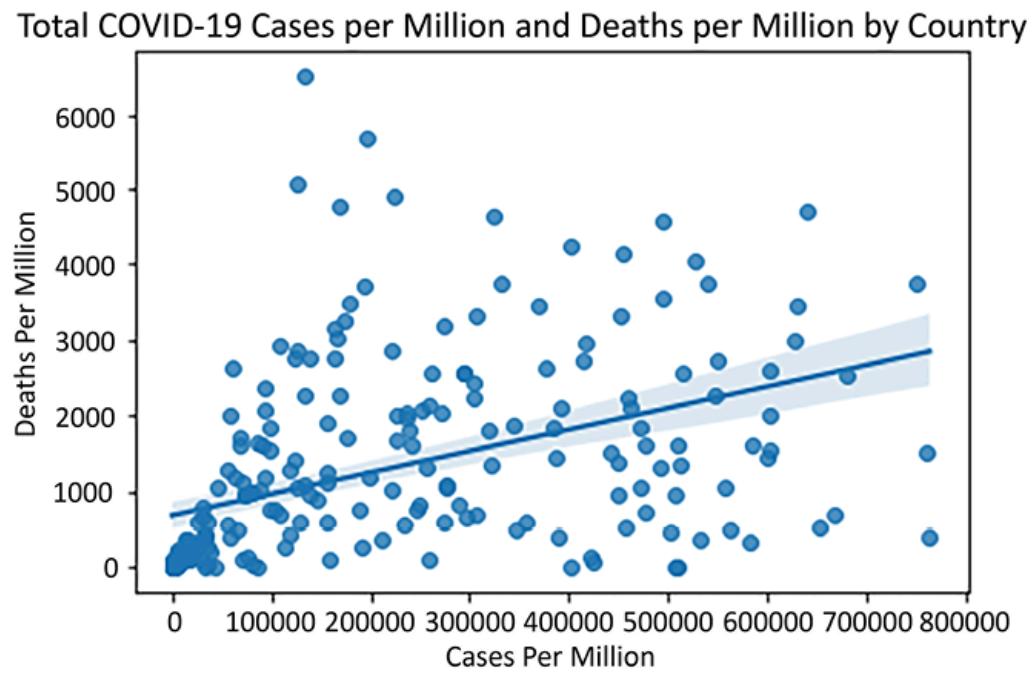


Figure 4.8: Scatterplot of cases and deaths per million with a linear regression line

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 the 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 (0.76) 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, Qatar. It is reasonable to wonder if there are potential reporting issues.

We make a scatterplot in *step 5* of the total cases and deaths. The strong upward sloping relationship between the two is confirmed, but there are a couple of countries whose deaths are above the regression line. We can see that two countries (Brazil and Russia) have higher deaths than would be predicted by the number of cases. Two countries, Japan and South Korea, have a much lower number of deaths.

Not surprisingly, there is even more scatter around the regression line in the scatterplot of cases per million and deaths per million. There is a positive relationship, but the slope of the line is not very steep.

## 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 will 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 and why certain variables take on certain values in certain situations and not others. We want to get to the point where there are no 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 or NLS**). We do that in the next recipe.

We do much more with Matplotlib and Seaborn in *Chapter 5, Using Visualizations for the Identification of Unexpected Values*.

## 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 DataFrames to allow us to easily compare values for a particular row with 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 NLS data, 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 9, Fixing Messy Data When*

*Aggregating*. However, it is not necessary to review *Chapter 9* to follow along, even if you have no experience with those tools.



### Data 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 who were born between 1980 and 1985, with annual follow-ups each year up until 2023. For this recipe, I pulled 106 variables on grades, employment, income, and attitudes toward government from the hundreds of data items on the survey. NLS data can be downloaded from [nlsinfo.org](http://nlsinfo.org).

## 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 those with 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/nls97f.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
```

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

The dataset has weeks worked each year from 2000 through 2023, and college enrollment status each month from February 1997 through October 2022. We use the ability of the `loc` accessor to choose all columns from the column indicated on the left of the colon through to the column indicated on the right; for example,

```
nls97.loc[:, "colenroct15":"colenrfeb22"]:
```

```
nls97[['wageincome20', 'highestgradecompleted',  
       'highestdegree']].head(3).T
```

```
personid          135335      999406  \\  
wageincome20        NaN      115,000  
highestgradecompleted      NaN      14  
highestdegree      4. Bachelor's 2. High School  
personid          151672  
wageincome20        NaN  
highestgradecompleted      16  
highestdegree      4. Bachelor's
```

```
nls97.loc[:, "weeksworked18":"weeksworked22"].head(3).T
```

```
personid      135335  999406  151672  
weeksworked18    NaN      52      52  
weeksworked19    NaN      52       9  
weeksworked20    NaN      52       0  
weeksworked21    NaN      46       0  
weeksworked22    NaN      NaN       3
```

```
nls97.loc[:, "colenroct15":"colenrfeb22"].head(2).T
```

```
personid          135335      999406  
colenroct15  1. Not enrolled 1. Not enrolled  
colenrfeb16        NaN  1. Not enrolled
```

|             |     |                 |
|-------------|-----|-----------------|
| colenroct16 | NaN | 1. Not enrolled |
| colenrfeb17 | NaN | 1. Not enrolled |
| colenroct17 | NaN | 1. Not enrolled |
| colenrfeb18 | NaN | 1. Not enrolled |
| colenroct18 | NaN | 1. Not enrolled |
| colenrfeb19 | NaN | 1. Not enrolled |
| colenroct19 | NaN | 1. Not enrolled |
| colenrfeb20 | NaN | 1. Not enrolled |
| colenroct20 | NaN | 1. Not enrolled |
| colenrfeb21 | NaN | 1. Not enrolled |
| colenroct21 | NaN | 1. Not enrolled |
| colenrfeb22 | NaN |                 |

Show individuals with wage income but no weeks worked:

```
nls97.loc[(nls97.weeksworked20==0) &
           (nls97.wageincome20>0),
           ['weeksworked20', 'wageincome20']]
```

|                        | weeksworked20 | wageincome20 |
|------------------------|---------------|--------------|
| personid               |               |              |
| 674877                 | 0             | 40,000       |
| 692251                 | 0             | 12,000       |
| 425230                 | 0             | 150,000      |
| 391939                 | 0             | 10,000       |
| 510545                 | 0             | 72,000       |
|                        | ...           | ...          |
| 947109                 | 0             | 1,000        |
| 706862                 | 0             | 85,000       |
| 956396                 | 0             | 130,000      |
| 907078                 | 0             | 10,000       |
| 274042                 | 0             | 130,000      |
| [132 rows x 2 columns] |               |              |

3. Check whether an individual was ever enrolled in a four-year college.

Chain several methods. First, create a DataFrame 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 four 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 four-year college between February 1997 and October 2022. 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, which are to see whether the individual was enrolled at a four-year college at some point:

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

|             |        |        |
|-------------|--------|--------|
| personid    | 135335 | 999406 |
| colenrfEB97 | False  | False  |
| colenroCT97 | False  | False  |
| colenrfEB98 | False  | False  |
| colenroCT98 | False  | False  |
| colenrfEB99 | False  | False  |
| colenroCT99 | True   | False  |
| colenrfEB00 | True   | False  |
| colenroCT00 | True   | True   |
| colenrfEB01 | True   | True   |
| colenroCT01 | True   | False  |

```
colenrfeb02    True    False
colenroct02    True     True
colenrfeb03    True     True
```

```
nls97.filter(like="colenr").\
  apply(lambda x: x.str[0:1]=='3').\
  any(axis=1).head(2)
```

```
personid
135335    True
999406    True
dtype: bool
```

4. 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` of any month, but who never had a value of 3 (bachelor enrollment). Note the ~ before the second half of the test, for negation. There are 24 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), "colenrfeb17":"colenrfeb22"]\nlen(nobach)
```

```
nobach.head(2).T
```

```
personid          793931          787976
.....abbreviated to save space
colenrfeb01      1. Not enrolled      1. Not enrolled
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       1. Not enrolled      1. Not enrolled
colenrfeb04       2. 2-year college     1. Not enrolled
colenroct04       4. Graduate program    1. Not enrolled
colenrfeb05       4. Graduate program    1. Not enrolled
.....
colenrfeb14      1. Not enrolled      1. Not enrolled
colenroct14       1. Not enrolled      2. 2-year college
colenrfeb15      1. Not enrolled      2. 2-year college
colenroct15       1. Not enrolled      2. 2-year college
colenrfeb16      1. Not enrolled      1. Not enrolled
colenroct16       1. Not enrolled      4. Graduate program
colenrfeb17      1. Not enrolled      4. Graduate program
colenroct17       1. Not enrolled      4. Graduate program
colenrfeb18      1. Not enrolled      4. Graduate program
colenroct18       1. Not enrolled      1. Not enrolled
.....
```

5. Show individuals with bachelor's degrees or more, but no four-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()
```

```
highestdegree
0. None          877
1. GED           1167
2. High School   3531
3. Associates     766
4. Bachelors      1713
5. Masters         704
6. PhD             64
7. Professional    130
Name: count, 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":"colenrfeb22"]
len(no4yearenrollment)
```

```
42
```

```
no4yearenrollment.head(2).T
```

```
personid          417244          124616
.....abbreviated to save space
colenroct04      1. Not enrolled  2. 2-year college
colenrfeb05      1. Not enrolled  2. 2-year college
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  1. Not enrolled
colenroct07      1. Not enrolled  1. Not enrolled
colenrfeb08      1. Not enrolled  1. Not enrolled
colenroct08      1. Not enrolled  1. Not enrolled
colenrfeb09      2. 2-year college 1. Not enrolled
colenroct09      2. 2-year college 1. Not enrolled
colenrfeb10      2. 2-year college 1. Not enrolled
```

```
colenroct10    2. 2-year college      1. Not enrolled
colenrfeb11    2. 2-year college      1. Not enrolled
colenroct11    2. 2-year college      1. Not enrolled
colenrfeb12    2. 2-year college      1. Not enrolled
colenroct12    1. Not enrolled       1. Not enrolled
colenrfeb13    1. Not enrolled       1. Not enrolled
```

## 6. Show individuals with a high wage income.

Define high wages as three standard deviations above the mean. It looks as though wage income values have been truncated at \$380,288:

```
highwages = \
nls97.loc[nls97.wageincome20 >
nls97.wageincome20.mean() + \
(nls97.wageincome20.std()*3),
['wageincome20']]
highwages
```

```
wageincome20
personid
989896      380,288
718416      380,288
693498      380,288
811201      380,288
553982      380,288
...
303838      380,288
366297      380,288
436132      380,288
964406      380,288
433818      380,288
[104 rows x 1 columns]
```

7. Show individuals with large changes in weeks worked for the most recent year.

Calculate the average value for weeks worked between 2016 and 2020 for each person (`nls97.loc[:, "weeksworked16": "weeksworked20"].mean(axis=1)`). We indicate `axis=1` to calculate the mean across columns for each individual, rather than over individuals. We then find rows where the mean is *not* between half and twice the number of weeks worked in 2021. (Notice our use of the `~` operator earlier.) We also indicate that we are not interested in rows that satisfy those criteria by being `null` for weeks worked in 2021. There are 1,099 individuals with sharp changes in weeks worked in 2021, compared with the 2016 to 2020 average:

```
workchanges = nls97.loc[~nls97.loc[:, "weeksworked16": "weeksworked20"].mean(axis=1).\
between(nls97.weeksworked21*0.5, \
nls97.weeksworked21*2) \
& ~nls97.weeksworked21.isnull(), \
"weeksworked16": "weeksworked21"]
len(workchanges)
```

```
1099
```

```
workchanges.head(6).T
```

| personid      | 151672 | 620126 | ... | 692251 | 483488 |
|---------------|--------|--------|-----|--------|--------|
| weeksworked16 | 53     | 45     | ... | 0      | 53     |
| weeksworked17 | 52     | 0      | ... | 0      | 52     |
| weeksworked18 | 52     | 0      | ... | 0      | 52     |
| weeksworked19 | 9      | 0      | ... | 0      | 52     |
| weeksworked20 | 0      | 0      | ... | 0      | 15     |

```
weeksworked21      0      0  ...     51      13
[6 rows x 6 columns]
```

8. 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, ['hi
pd.crosstab(ltgrade12.highestgradecompleted, ltgrade12.hi
```

| highestdegree         | 0. None        | 1. GED        | \ |
|-----------------------|----------------|---------------|---|
| highestgradecompleted |                |               |   |
| 5                     | 0              | 0             |   |
| 6                     | 11             | 4             |   |
| 7                     | 23             | 7             |   |
| 8                     | 108            | 82            |   |
| 9                     | 98             | 182           |   |
| 10                    | 105            | 207           |   |
| 11                    | 113            | 204           |   |
| highestdegree         | 2. High School | 3. Associates |   |
| highestgradecompleted |                |               |   |
| 5                     | 1              | 0             |   |
| 6                     | 0              | 1             |   |
| 7                     | 1              | 0             |   |
| 8                     | 7              | 0             |   |
| 9                     | 8              | 1             |   |
| 10                    | 14             | 1             |   |
| 11                    | 42             | 2             |   |

These steps reveal several logical inconsistencies 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 you to quickly run any query against the data that you might imagine.

Some of the inconsistencies or unexpected values suggest either respondent or entry error and warrants further investigation. It is hard to explain positive values for wage income when the `weeks worked` value 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 2021 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 *Chapter 3, Taking the Measure of Your Data*, demonstrate some of the techniques for subsetting data used here. We examine the `apply` functions in more detail in *Chapter 9, Fixing Messy Data When Aggregating*.

## 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 the model fit for all of the other observations.

## Getting ready

The code in this recipe requires the `matplotlib` and `statsmodels` libraries. You can install them 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-19 cases and deaths per country.

## How to do it...

We will use the `statsmodels` `OLS` method to fit a linear regression model of the total cases per million of the population. We will then identify those countries that have the greatest influence on that model:

1. Import `pandas`, `matplotlib`, and `statsmodels`, and load the COVID-19 case data:

```
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm
covidtotals = pd.read_csv("data/covidtotals.csv")
covidtotals.set_index("iso_code", inplace=True)
```

2. 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
covidanalysis.describe()
```

|       | total_cases_pm | pop_density | median_age | gdp_per_capita |
|-------|----------------|-------------|------------|----------------|
| count | 180            | 180         | 180        | 180            |
| mean  | 167,765        | 204         | 30         | 30             |
| std   | 190,965        | 631         | 9          | 9              |
| min   | 354            | 2           | 15         | 15             |
| 25%   | 11,931         | 36          | 22         | 22             |
| 50%   | 92,973         | 82          | 29         | 29             |
| 75%   | 263,162        | 205         | 38         | 38             |
| max   | 763,475        | 7,916       | 48         | 48             |

3. 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 those 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      |
|----------------|------------|----------|--------|
| Const          | -2.382e+05 | 3.41e+04 | -6.980 |
| pop_density    | 12.4060    | 14.664   | 0.846  |
| median_age     | 11570      | 1291.446 | 8.956  |
| gdp_per_capita | 2.9674     | 0.621    | 4.777  |

#### 4. 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      | 0.70    |
| SGP      | 3.12    |

```
covidanalysis.loc[influence.cooks_d>0.5]
```

|          | total_cases_pm | pop_density | median_age | gdp_per_ |
|----------|----------------|-------------|------------|----------|
| iso_code |                |             |            |          |

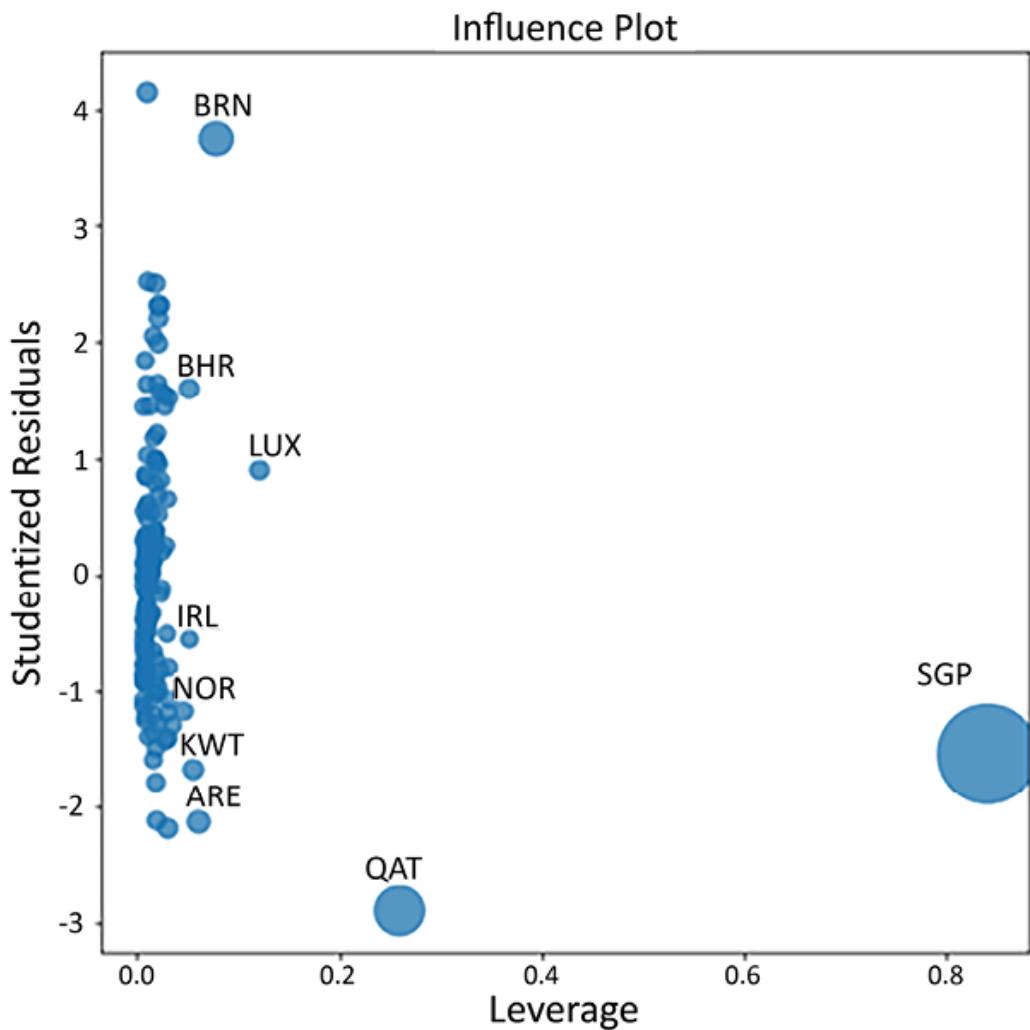
|     |          |        |    |   |
|-----|----------|--------|----|---|
| QAT | 190, 909 | 227    | 32 | 1 |
| SGP | 531, 184 | 7, 916 | 42 |   |

## 5. Create an influence plot.

Countries with higher Cook's Distance values have larger circles:

```
fig, ax = plt.subplots(figsize=(8,8))
sm.graphics.influence_plot(lm, ax = ax, alpha=5, criterion
plt.show()
```

This produces the following plot:



*Figure 4.9: Influence plot, including countries with the highest Cook's Distance*

## 6. Run the model without the two outliers.

Removing these outliers impacts each coefficient of the model, but particularly population density (which is still not significant at the 95% confidence level):

```
covidanalysisminusoutliers = covidanalysis.loc[influence.c
lm = getlm(covidanalysisminusoutliers)
lm.summary()
```

|                | coef       | std err  | t      | F |
|----------------|------------|----------|--------|---|
| <hr/>          |            |          |        |   |
| const          | -2.158e+05 | 3.43e+04 | -6.288 | 6 |
| pop_density    | 61.2396    | 34.260   | 1.788  | 6 |
| median_age     | 9968.5170  | 1346.416 | 7.404  | 6 |
| gdp_per_capita | 4.1112     | 0.704    | 5.841  | 6 |

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 11570 for median age in the first regression results indicates that every one-year increase in median age is associated with an 11570 increase in cases per million people. That number is substantially smaller in the model without outliers, 9969.

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.

## There's more...

We ran a linear regression model in this recipe, not so much because we were interested in the parameter estimates of the model, but because we wanted to determine whether there were observations with potential outsized 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 neighbors 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 neighbors (KNN)** 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 **Python Outlier Detection (PyOD)** 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 KNN to identify countries whose attributes indicate that they are most anomalous:

1. Load `pandas`, `pyod`, and `sklearn`, along with the COVID-19 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)
```

2. Create a standardized DataFrame for the analysis columns:

```
standardizer = StandardScaler()
analysisvars = ['location', 'total_cases_pm',
...    'total_deaths_pm', 'pop_density',
...    'median_age', 'gdp_per_capita']
covidanalysis = covidtotals.loc[:, analysisvars].drop
covidanalysisstand = standardizer.fit_transform(covid
```

3. 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, met  
metric='minkowski', metric_params=None, n_jobs=1, n_nei  
radius=1.0)
```

```
y_pred = clf.labels_  
y_scores = clf.decision_scores_
```

#### 4. Show the predictions from the model.

Create a DataFrame from the `y_pred` and `y_scores` NumPy arrays. Set the index to the `covidanalysis` DataFrame index so that we can easily combine it with that DataFrame 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=2)
```

|          | outlier | scores |
|----------|---------|--------|
| iso_code |         |        |
| BHR      | 1       | 2.69   |
| BRA      | 0       | 0.75   |
| ZWE      | 0       | 0.21   |
| BGR      | 1       | 1.62   |
| CHN      | 0       | 0.94   |
| BGD      | 1       | 1.52   |
| GRD      | 0       | 0.68   |
| UZB      | 0       | 0.37   |

```
MMR          0    0.37
ECU          0    0.58
```

```
pred.outlier.value_counts()
```

```
0    162
1     18
Name: outlier, dtype: int64
```

```
pred.groupby(['outlier'])[['scores']].agg(['min','median',
```

```
scores
      min   median   max
outlier
0       0.08    0.60   1.40
1       1.42    1.65  11.94
```

## 5. Show COVID-19 data for the outliers.

First, merge the `covidanalysis` and `pred` DataFrames:

```
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      531,183.84
QAT         Qatar        190,908.72
```

|          |                        |            |
|----------|------------------------|------------|
| BHR      | Bahrain                | 473,167.02 |
| LUX      | Luxembourg             | 603,439.46 |
| PER      | Peru                   | 133,239.00 |
| BRN      | Brunei                 | 763,475.44 |
| MDV      | Maldives               | 356,423.66 |
| MLT      | Malta                  | 227,422.82 |
| ARE      | United Arab Emirates   | 113,019.21 |
| BGR      | Bulgaria               | 195,767.89 |
|          | total_deaths_pm scores |            |
| iso_code |                        |            |
| SGP      | 346.64                 | 11.94      |
| QAT      | 256.02                 | 3.04       |
| BHR      | 1,043.31               | 2.69       |
| LUX      | 1,544.16               | 2.49       |
| PER      | 6,507.66               | 2.27       |
| BRN      | 396.44                 | 2.26       |
| MDV      | 603.29                 | 1.98       |
| MLT      | 1,687.63               | 1.96       |
| ARE      | 248.81                 | 1.69       |
| BGR      | 5,703.52               | 1.62       |

These steps show how we can use KNN 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 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 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. However, 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 used the standard scaler in *step 2*, which returned the  $z$ -score for each value in the DataFrame. 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...

KNN 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 testing datasets, for example. PyOD allows this to be done easily, but this is not necessary for our purposes here.

## See also

We go into detail on transforming data in *Chapter 8, Encoding, Transforming, and Scaling Features*. A good resource on using KNN is *Data Cleaning and Exploration with Machine Learning*, also by me.

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-19 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 `StandardScaler` and `IsolationForest` modules from `sklearn`:

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import IsolationForest
covidtotals = pd.read_csv("data/covidtotals.csv")
covidtotals.set_index("iso_code", inplace=True)
```

2. Create a standardized analysis DataFrame.

First, remove all rows with missing data:

```
analysisvars = ['location', 'total_cases_pm', 'total_deaths_
...   '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()
```

```
covidanalysissstand = standardizer.fit_transform(covidanal)
```

### 3. 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(covidanalysissstand)
```

```
IsolationForest(contamination=0.1)
```

```
covidanalysis['anomaly'] = \  
    clf.predict(covidanalysissstand)  
covidanalysis['scores'] = \  
    clf.decision_function(covidanalysissstand)  
covidanalysis.anomaly.value_counts()
```

```
1           156  
-1          18  
Name: anomaly, dtype: int64
```

### 4. Create outlier and inlier DataFrames.

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']] .\nsort_values(['scores']) .\nhead(10)
```

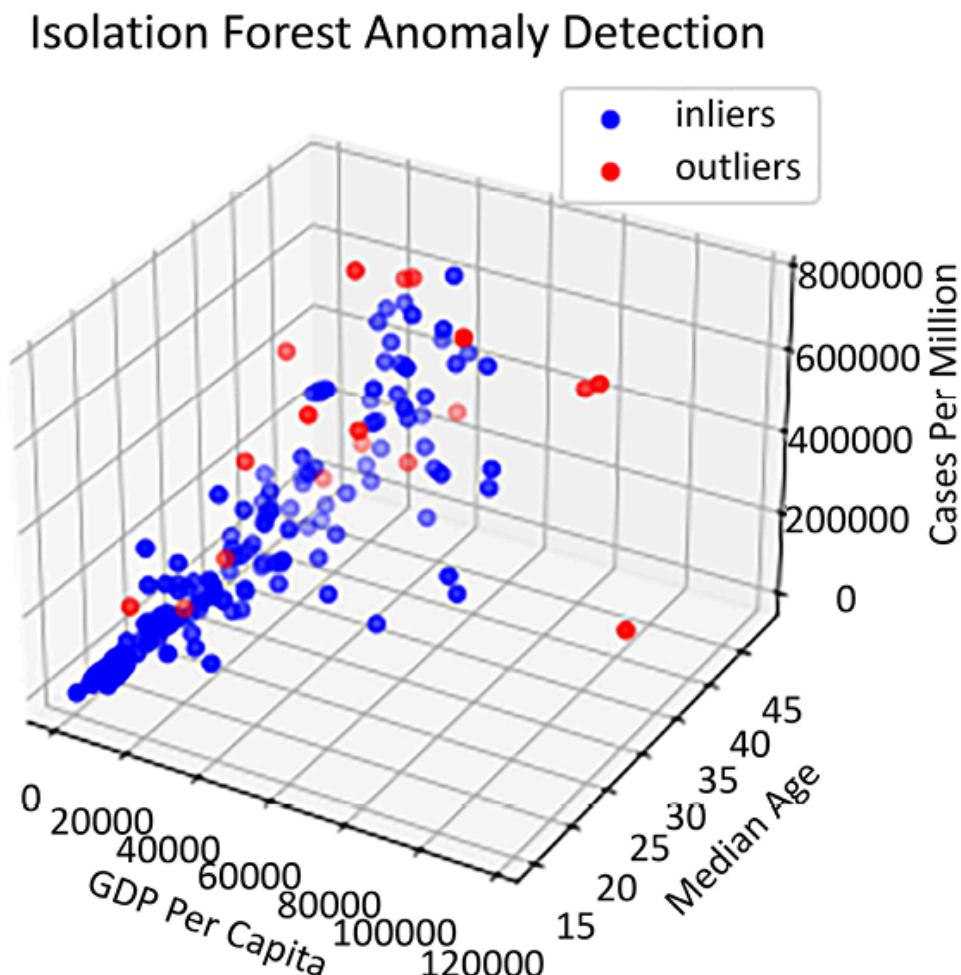
| iso_code | location   | total_cases_pm | total_deaths_pm | \ |
|----------|------------|----------------|-----------------|---|
| SGP      | Singapore  | 531,183.84     | 346.64          |   |
| BHR      | Bahrain    | 473,167.02     | 1,043.31        |   |
| BRN      | Brunei     | 763,475.44     | 396.44          |   |
| QAT      | Qatar      | 190,908.72     | 256.02          |   |
| PER      | Peru       | 133,239.00     | 6,507.66        |   |
| MLT      | Malta      | 227,422.82     | 1,687.63        |   |
| MDV      | Maldives   | 356,423.66     | 603.29          |   |
| LUX      | Luxembourg | 603,439.46     | 1,544.16        |   |
| BGR      | Bulgaria   | 195,767.89     | 5,703.52        |   |
| BGD      | Bangladesh | 11,959.46      | 172.22          |   |
| iso_code | median_age | gdp_per_capita | scores          |   |
| iso_code |            |                |                 |   |
| SGP      | 42.40      | 85,535.38      | -0.26           |   |
| BHR      | 32.40      | 43,290.71      | -0.09           |   |
| BRN      | 32.40      | 71,809.25      | -0.09           |   |
| QAT      | 31.90      | 116,935.60     | -0.08           |   |
| PER      | 29.10      | 12,236.71      | -0.08           |   |
| MLT      | 42.40      | 36,513.32      | -0.06           |   |
| MDV      | 30.60      | 15,183.62      | -0.06           |   |
| LUX      | 39.70      | 94,277.96      | -0.06           |   |
| BGR      | 44.70      | 18,563.31      | -0.04           |   |
| BGD      | 27.50      | 3,523.98       | -0.04           |   |

## 5. 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,
```

```
ax.scatter3D(outlier.gdp_per_capita, outlier.median_a  
ax.legend()  
plt.tight_layout()  
plt.show()
```

This produces the following plot:



*Figure 4.10: 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 KNN for anomaly detection.

## How it works...

We used Isolation Forest in this recipe much like we used KNN in the previous recipe. In *step 3*, we passed a standardized dataset to the Isolation Forest `fit` method, and then used its `predict` and `decision_function` methods to get the anomaly flag and score, respectively. We used 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 a 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 KNN results. Singapore, Bahrain, and Qatar have three of the four highest (most negative) anomaly scores.

## There's more...

Isolation Forest is a good alternative to KNN, particularly when working with large datasets. The efficiency of its algorithm allows it to handle large samples and a high number of 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 helped 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 KNN.) You can read more about Isolation Forest here:

<https://cs.nju.edu.cn/zhouzh/zhouzh.files/publication/icdm08b.pdf>.

## Using PandasAI to identify outliers

We can use PandasAI to support some of the work we have done in this chapter to identify outliers. We can check for extreme values based on a univariate analysis. We can look at bivariate and multivariate relationships as well. PandasAI will also help us generate visualizations easily.

## 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-19 data again, which is available in the GitHub repository, as well as 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.

The PandasAI library is improving rapidly, and some things have changed, even since I began writing this book. I have used PandasAI version 2.0.30 in this recipe. It also matters which version of pandas you use with it. I have used pandas version 2.2.1 in this recipe.

# How to do it...

We create a PandasAI instance in the following steps and use it to look for extreme and unexpected values in the COVID-19 data:

1. We import `pandas` and the `PandasAI` library:

```
import pandas as pd
from pandasai.llm.openai import OpenAI
from pandasai import SmartDataframe
llm = OpenAI(api_token="Your API key")
```

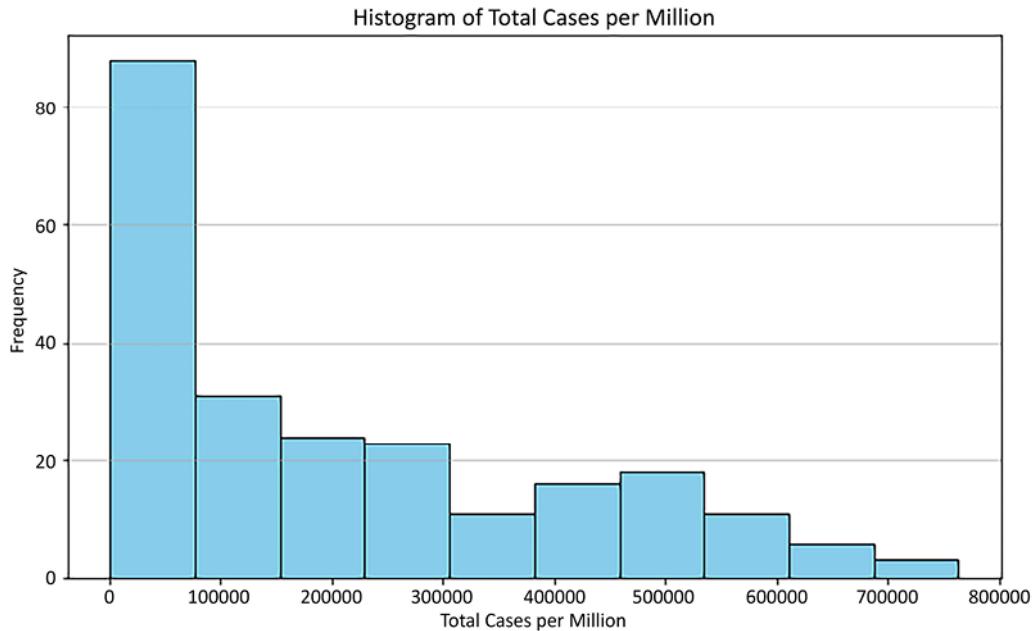
2. We load the COVID-19 data and create a `PandasAI SmartDataframe`:

```
covidtotals = pd.read_csv("data/covidtotals.csv")
covidtotalssdf = SmartDataframe(covidtotals, config={
```

3. We can pass natural language queries to the `chat` method of the `SmartDataframe`. This includes plotting:

```
covidtotalssdf.chat("Plot histogram of total cases pe
```

This produces the following plot:

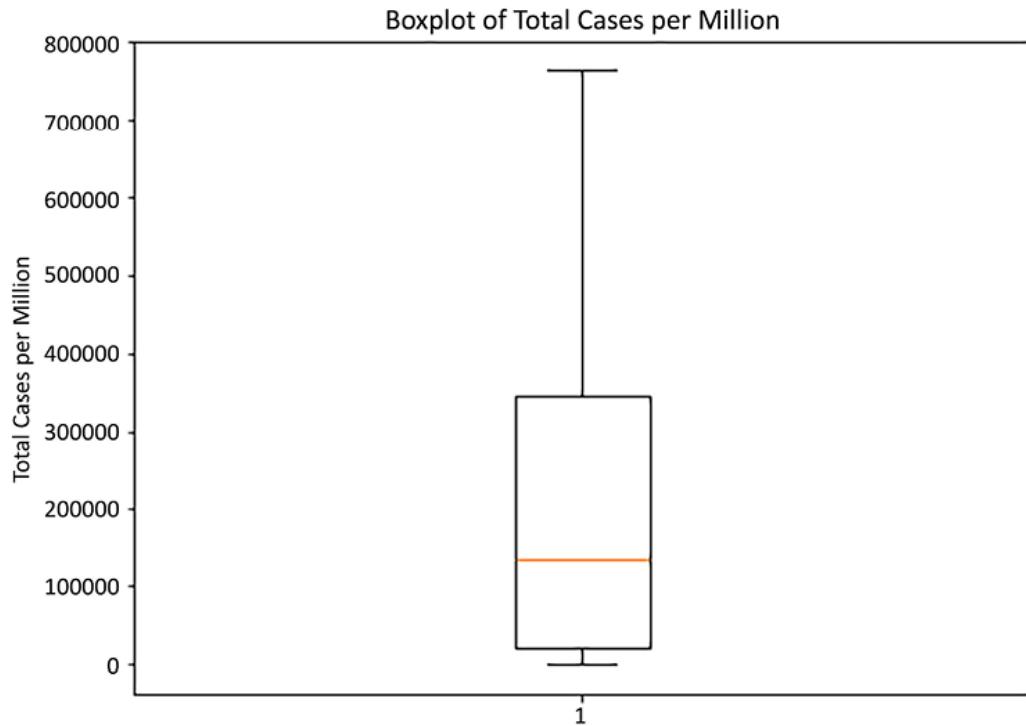


*Figure 4.11: Histogram of total cases per million*

4. We can also create a boxplot:

```
covidtotalssdf.chat("Show box plot of total cases per
```

This produces the following plot:

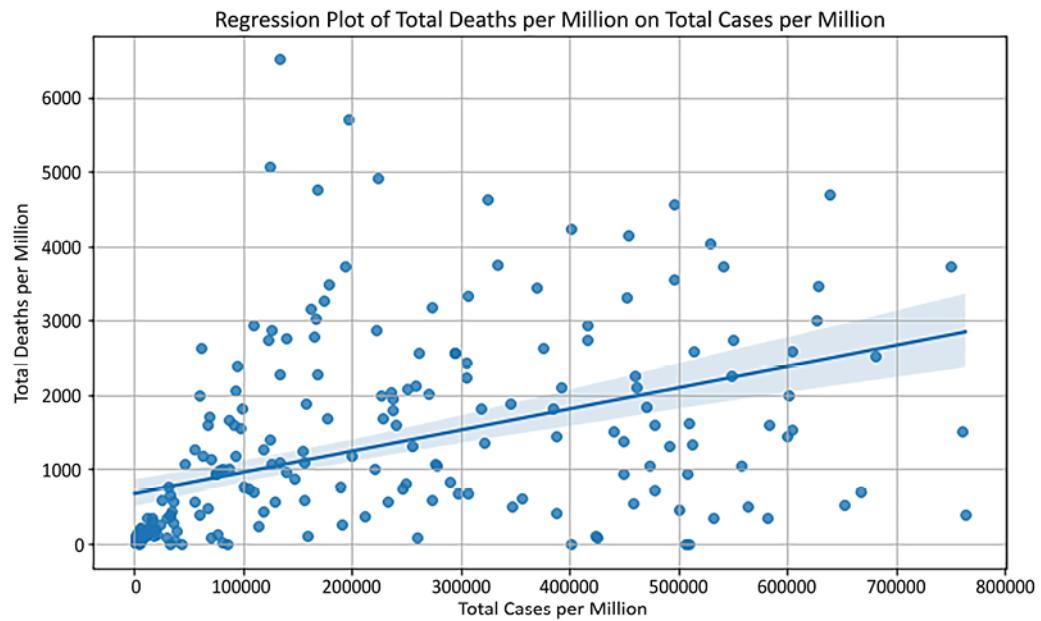


*Figure 4.12: Boxplot of total cases per million*

5. We can also show a scatterplot of the relationship between cases and deaths. We indicate that we want to use `regplot` to make sure that we get a regression line:

```
covidtotalssdf.chat("regplot total_deaths_pm on total
```

This produces the following plot:



*Figure 4.13: Scatterplot of the relationship between cases and deaths*

## 6. Show high and low values for total cases:

```
covidtotalssdf.chat("Show total cases per million for
```

| iso_code | location | tot                          |
|----------|----------|------------------------------|
| 190      | SVN      | Slovenia                     |
| 67       | FRO      | Faeroe Islands               |
| 194      | KOR      | South Korea                  |
| 12       | AUT      | Austria                      |
| 180      | SMR      | San Marino                   |
| 52       | CYP      | Cyprus                       |
| 30       | BRN      | Brunei                       |
| 228      | YEM      | Yemen                        |
| 148      | NER      | Niger                        |
| 40       | TCD      | Chad                         |
| 204      | TZA      | Tanzania                     |
| 186      | SLE      | Sierra Leone                 |
| 32       | BFA      | Burkina Faso                 |
| 54       | COD      | Democratic Republic of Congo |

This sorted the data in ascending order for the high group, and then sorted the data in ascending order for the low group.

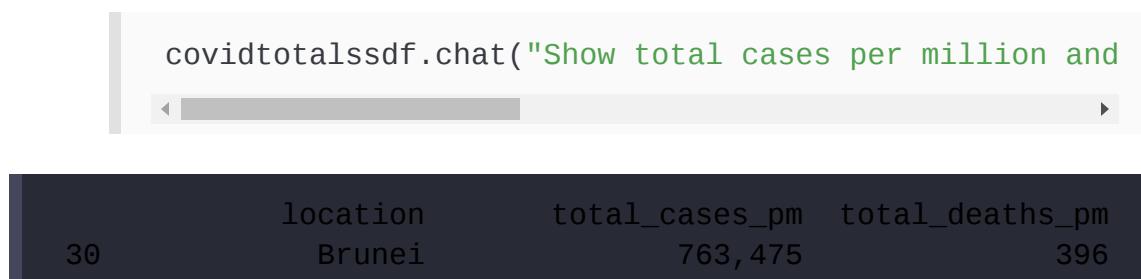
7. We can find the countries with the highest number of cases for each region:



covidtotalssdf.chat("Show total cases per million for

| region          | location                  | total_case |
|-----------------|---------------------------|------------|
| Caribbean       | Martinique                | 626        |
| Central Africa  | Sao Tome and Principe     | 29         |
| Central America | Costa Rica                | 237        |
| Central Asia    | Armenia                   | 162        |
| East Africa     | Reunion                   | 507        |
| East Asia       | Brunei                    | 763        |
| Eastern Europe  | Cyprus                    | 760        |
| North Africa    | Tunisia                   | 93         |
| North America   | Saint Pierre and Miquelon | 582        |
| Oceania / Aus   | Niue                      | 508        |
| South America   | Falkland Islands          | 505        |
| South Asia      | Bahrain                   | 473        |
| Southern Africa | Saint Helena              | 401        |
| West Africa     | Cape Verde                | 108        |
| West Asia       | Israel                    | 512        |
| Western Europe  | San Marino                | 756        |

8. We can get `chat` to show us countries where the number of cases was high but the number of deaths was relatively low:



covidtotalssdf.chat("Show total cases per million and

| 30 | location | total_cases_pm | total_deaths_pm |
|----|----------|----------------|-----------------|
|    | Brunei   | 763,475        | 396             |

|     |                           |         |     |
|-----|---------------------------|---------|-----|
| 46  | Cook Islands              | 422,910 | 117 |
| 68  | Falkland Islands          | 505,919 | 0   |
| 81  | Greenland                 | 211,899 | 372 |
| 93  | Iceland                   | 562,822 | 499 |
| 126 | Marshall Islands          | 387,998 | 409 |
| 142 | Nauru                     | 424,947 | 79  |
| 150 | Niue                      | 508,709 | 0   |
| 156 | Palau                     | 346,439 | 498 |
| 167 | Qatar                     | 190,909 | 256 |
| 172 | Saint Barthelemy          | 500,910 | 455 |
| 173 | Saint Helena              | 401,037 | 0   |
| 177 | Saint Pierre and Miquelon | 582,158 | 340 |
| 187 | Singapore                 | 531,184 | 347 |
| 209 | Tonga                     | 158,608 | 112 |
| 214 | Tuvalu                    | 259,638 | 88  |
| 217 | United Arab Emirates      | 113,019 | 249 |
| 226 | Vietnam                   | 118,387 | 440 |

These were just a few examples of how PandasAI can be used to help us find outliers or unexpected values with very little code.

## How it works...

We used PandasAI with the large language model provided by OpenAI in this recipe. You just need an API token. You can get one from [platform.openai.com](https://platform.openai.com). After you have a token, all you need to do to get started with sending natural language queries to a database is import the OpenAI and `SmartDataframe` modules and instantiate a `SmartDataframe` object.

We created a `SmartDataframe` object in *step 2* with `covidtotalssdf = SmartDataframe(covidtotals, config={"llm": "11m"})`. Once we have a `SmartDataframe`, we can pass a variety of natural language instructions to its `chat` method. In this recipe, this ranged from requesting visualizations

and finding the highest and lowest values to examining values for subsets of the data.

It is a good idea to regularly check the `pandasai.log` file, which will be in the same folder as your Python script. Here is the code PandasAI generated in response to `covidtotalssdf.chat("Show total cases per million and total deaths per million for locationss with high total_cases_pm and low total_deaths_pm"):`

```
import pandas as pd
# Filter rows with high total_cases_pm and low total_deaths_pm
filtered_df = dfs[0][(dfs[0]['total_cases_pm'] > 100000) & (dfs[0]['total_deaths_pm'] < 1000)]
# Select only the required columns
result_df = filtered_df[['location', 'total_cases_pm', 'total_deaths_pm']]
result = {"type": "dataframe", "value": result_df}
```

```
2024-04-18 09:30:01 [INFO] Executing Step 4: CachePopulation
2024-04-18 09:30:01 [INFO] Executing Step 5: CodeCleaning
2024-04-18 09:30:01 [INFO]
Code running:
```
filtered_df = dfs[0][(dfs[0]['total_cases_pm'] > 100000) & (dfs[0]['total_deaths_pm'] < 1000)]
result_df = filtered_df[['location', 'total_cases_pm', 'total_deaths_pm']]
result = {'type': 'dataframe', 'value': result_df}
```

## There's more...

We generated a boxplot in *step 4*, which is an incredibly useful tool for visualizing the distribution of a continuous variable. The box shows the interquartile range, which is the distance between the first and third

quartiles. The line in the box shows the median. We go into much more detail on boxplots in *Chapter 5, Using Visualizations for the Identification of Unexpected Values*.

## See also

The *Using generative AI to create descriptive statistics* recipe in *Chapter 3, Taking the Measure of Your Data*, provides some additional information on how PandasAI uses OpenAI, and on generating overall and by-group statistics and visualizations. We use PandasAI throughout this book whenever it is a good tool for improving our data preparation work or making it easier.

## Summary

This chapter introduced pandas tools for identifying outliers in our data. We explored a variety of univariate, bivariate, and multivariate approaches to detect observations sufficiently out of range, or otherwise unusual enough, to distort our analysis. These approaches included using the interquartile range to identify extreme values, investigating relationships with a correlated variable, and using parametric and non-parametric multivariate techniques such as linear regression and KNN respectively. We also saw how visualizations can help us get a better feel for how a variable is distributed, and how it moves with a correlated variable. We will go into much greater detail on how to create and interpret visualizations in the next chapter.

**Join our community on Discord**

Join our community's Discord space for discussions with the author and other readers:

<https://discord.gg/p8uSgEAETX>



# 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

## Technical requirements

You will need Pandas, Numpy, Matplotlib, and Seaborn to complete the recipes in this chapter. I used `pandas 2.1.4`, but the code will run on `pandas 1.5.3` or later.

The code in this chapter can be downloaded from the book's GitHub repository, <https://github.com/PacktPublishing/Python-Data-Cleaning-Cookbook-Second-Edition>.

## 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 90<sup>th</sup> 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 to 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 depending 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 covering Earth surface temperatures and COVID-19 cases in this recipe. The surface temperature DataFrame has one row per weather station. The COVID-19 DataFrame has one row per country with total cases and demographic information.



## Note

The surface temperature DataFrame has the average temperature reading (in °C) in 2023 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.ncei.noaa.gov/products/land-based-station/global-historical-climatology-network-monthly>.

*Our World in Data* provides COVID data for public use at <https://ourworldindata.org/covid-cases>. The data used in this recipe was downloaded on March 3, 2024.

## How to do it...

We take a close look at the distribution of surface temperatures by weather station in 2023 and total COVID-19 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 the data on surface temperatures and COVID-19 cases:

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

```
import statsmodels.api as sm
landtemps = pd.read_csv("data/landtemps2023avgs.csv")
covidtotals = pd.read_csv("data/covidtotals.csv", parse_dates=True)
covidtotals.set_index("iso_code", inplace=True)
```

## 2. 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       | \       |
|-------|--------------------------|---------------|---------|
| 11924 | WOLF_POINT_29_ENE        | United States |         |
| 10671 | LITTLE_GRASSY            | United States |         |
| 10278 | FLOWERY_TRAIL_WASHINGTON | United States |         |
| 8436  | ROCKSPRINGS              | United States |         |
| 1715  | PETERBOROUGH             | Canada        |         |
| 5650  | TRACY_PUMPING_PLT        | United States |         |
| 335   | NEPTUNE_ISLAND           | Australia     |         |
| 372   | EUDUNDA                  | Australia     |         |
| 2987  | KOZHIKODE                | India         |         |
| 7588  | TRYON                    | United States |         |
|       | latabs                   | elevation     | avgtemp |
| 11924 | 48                       | 636           | 6       |
| 10671 | 37                       | 1,859         | 10      |
| 10278 | 48                       | 792           | 8       |
| 8436  | 30                       | 726           | 20      |
| 1715  | 44                       | 191           | 8       |
| 5650  | 38                       | 19            | 18      |
| 335   | 35                       | 32            | 16      |
| 372   | 34                       | 415           | 16      |

```
2987      11       5     30  
7588      35      366     16
```

### 3. Show some descriptive statistics.

Also, look at skewness and kurtosis:

```
landtemps.describe()
```

```
      latabs  elevation  avgtemp  
count   12,137      12,137    12,137  
mean      40        598       12  
std       13        775        8  
min       0       -350      -57  
25%      35         78        6  
50%      41        271       11  
75%      47        824       17  
max      90       9,999      34
```

```
landtemps.avgtemp.skew()
```

```
-0.3856060165979757
```

```
landtemps.avgtemp.kurtosis()
```

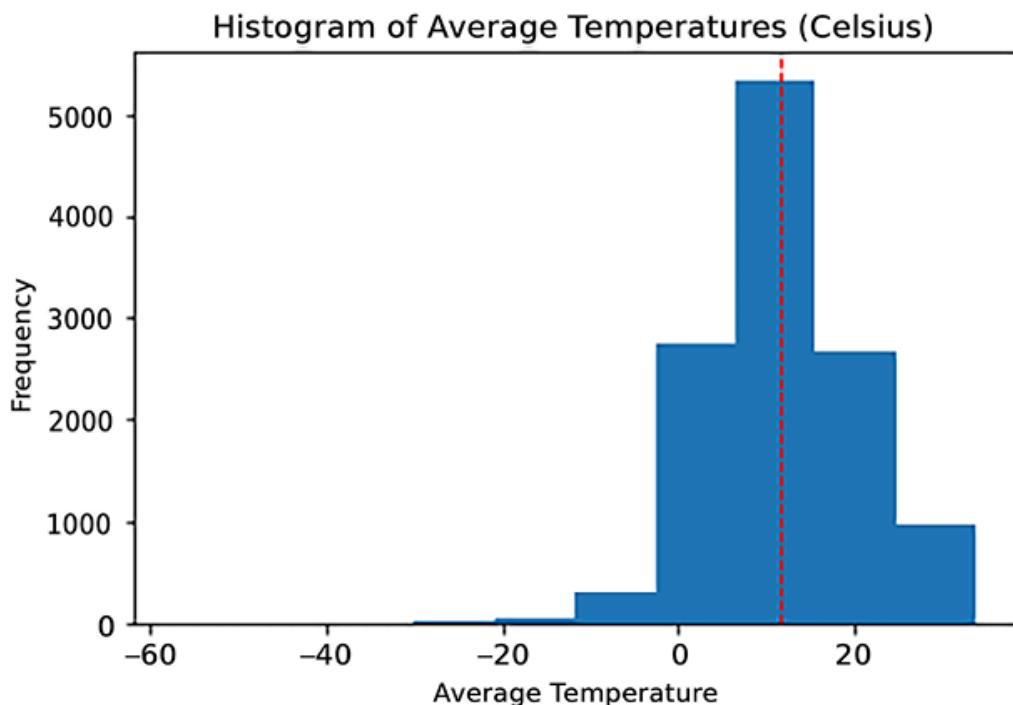
```
2.7939884544586033
```

### 4. 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='dashed')
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*

5. 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']),  
plt.title("QQ Plot of Average Temperatures")  
plt.show()
```

This results in the following QQ plot:

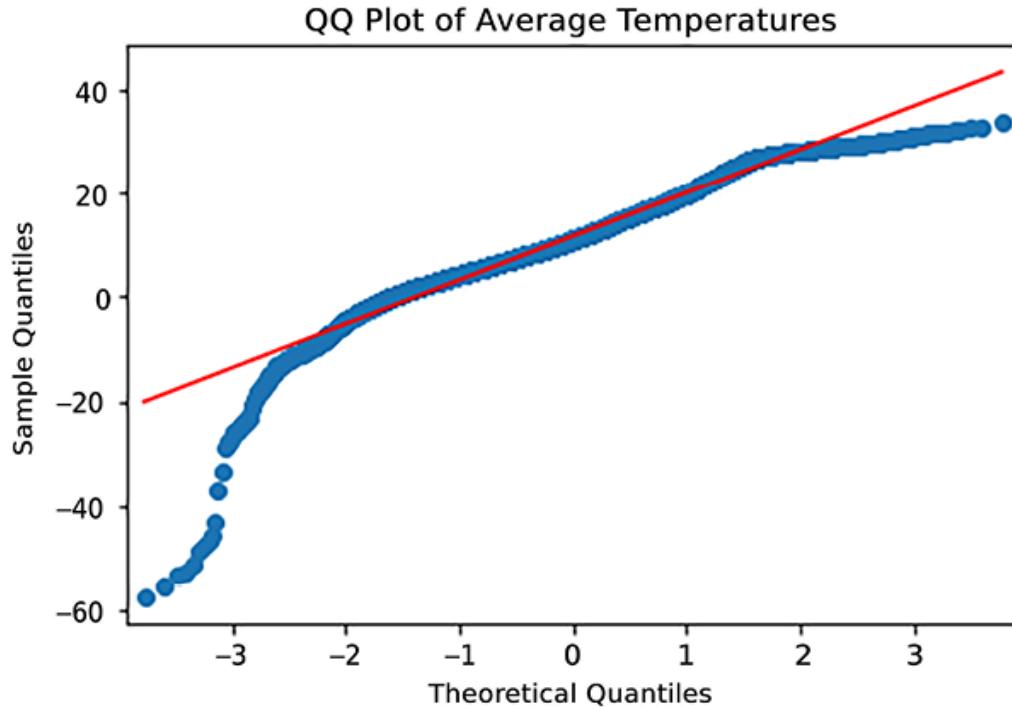


Figure 5.2: Plot of average temperature by station compared with a normal distribution

6. Show the skewness and kurtosis for total COVID-19 cases per million.

This is from the COVID-19 DataFrame, which has one row for each country:

```
covidtotals.total_cases_pm.skew()
```

```
0.8349032460009967
```

```
covidtotals.total_cases_pm.kurtosis()
```

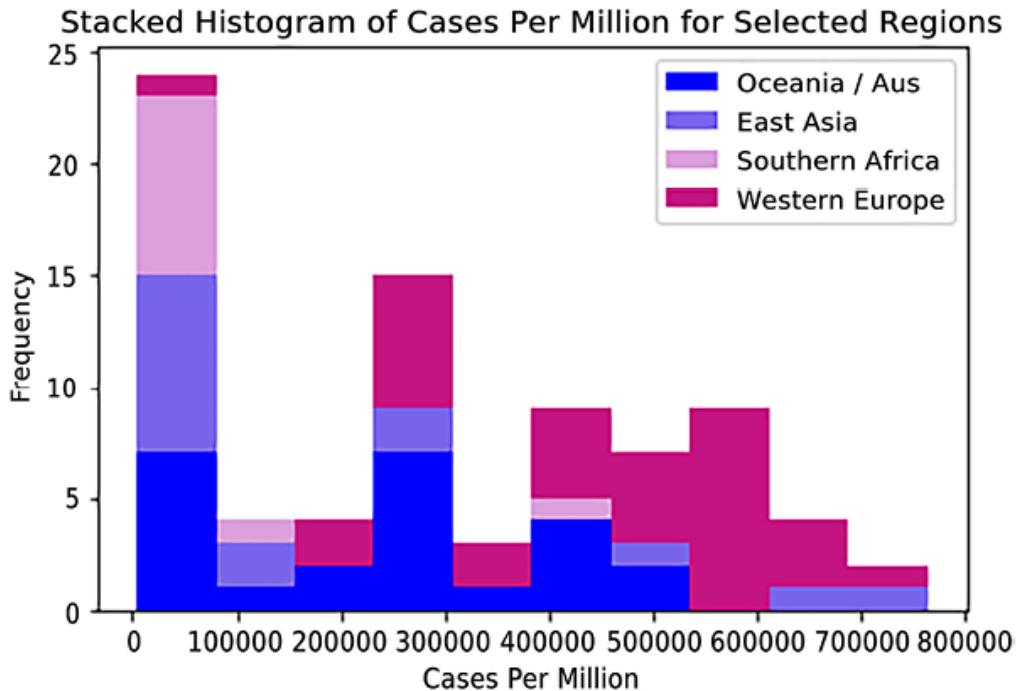
```
-0.4280595203351645
```

## 7. Do a stacked histogram of the COVID-19 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 how much unlike the other regions Western Europe is, accounting for almost all countries with cases per million above 500,000:

```
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 Regions")  
plt.xlabel("Cases Per Million")  
plt.ylabel("Frequency")  
plt.legend()  
plt.show()
```

This results in the following stacked histogram:



*Figure 5.3: Stacked histogram of the number of countries per region at different levels of cases per million*

## 8. 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.suptitle("Histograms of COVID-19 Cases Per Million by
axes = axes.ravel()
for j, ax in enumerate(axes):
...     ax.hist(covidtotals.loc[covidtotals.region==showregi
...     total_cases_pm, bins=7)
...     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:

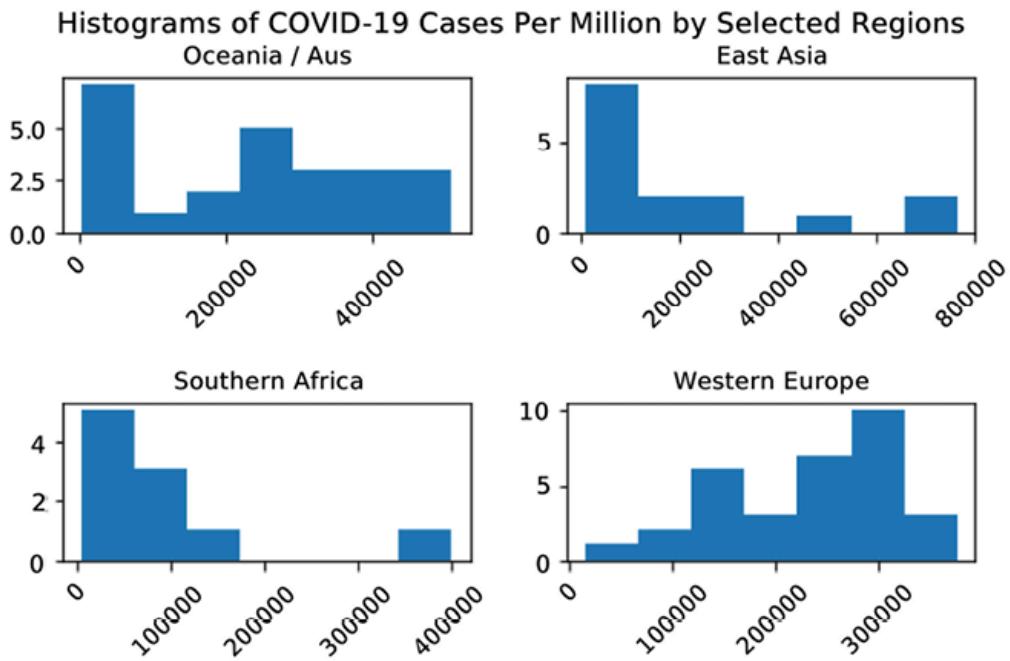


Figure 5.4: Histograms of numbers of countries by region at different levels of cases per million

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 the 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 them to be applied 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 in *Steps 3-5* show. It is skewed to the left (skew of `-0.39`) and has tails that are close to normal (kurtosis of 2.79, 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-19 case data.

The skew and kurtosis of the COVID-19 `cases per million` variable show that it is some distance from normal. The skew of 0.83 and kurtosis of -0.43 indicate a somewhat positive skew and much skinnier 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 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  $(1.5 * \text{IQR}) + \text{the third quartile value}$ , or less than the first quartile value  $- (1.5 * \text{IQR})$ , was considered an outlier. That is precisely what is revealed in a boxplot.

## Getting ready

We will work with cumulative data on COVID-19 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-19 cases and deaths:

1. Load the `pandas` and `matplotlib` libraries.

Also, load the NLS and COVID-19 data:

```
import pandas as pd
import matplotlib.pyplot as plt
nls97 = pd.read_csv("data/nls97f.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
covidtotals = pd.read_csv("data/covidtotals.csv", parse_dates=True)
covidtotals.set_index("iso_code", inplace=True)
```

2. 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

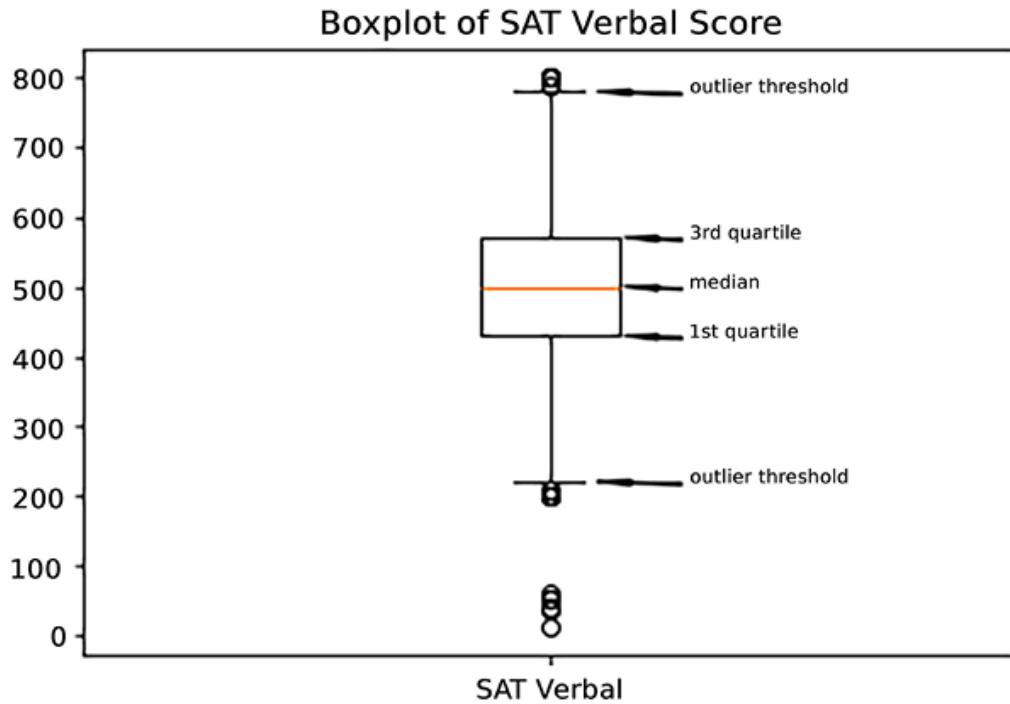
third quartile. The whiskers go from that rectangle to 1.5 times the IQR. Any values above or below the whiskers (what we have labeled the outlier threshold) are considered outliers (we use `annotate` to point to the first and third quartile points, the median, and to the outlier thresholds):

```
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, 780))
plt.annotate('3rd quartile', xy=(1.08, 570), xytext=(1.15, 570))
plt.annotate('median', xy=(1.08, 500), xytext=(1.15, 500))
plt.annotate('1st quartile', xy=(1.08, 430), xytext=(1.15, 430))
plt.annotate('outlier threshold', xy=(1.05, 220), xytext=(1.15, 220))
plt.title("Boxplot of SAT Verbal Score")
plt.show()
```

This results in the following boxplot:



*Figure 5.5: Boxplot of SAT verbal scores with labels for interquartile range and outliers*

3. Next, show some descriptives on weeks worked:

```
weeksworked = nls97.loc[:, ['highestdegree',
    'weeksworked20', 'weeksworked21']]
weeksworked.describe()
```

|       | weeksworked20 | weeksworked21 |
|-------|---------------|---------------|
| count | 6,971         | 6,627         |
| mean  | 38            | 36            |
| std   | 21            | 18            |
| min   | 0             | 0             |
| 25%   | 21            | 35            |
| 50%   | 52            | 43            |
| 75%   | 52            | 50            |
| max   | 52            | 52            |

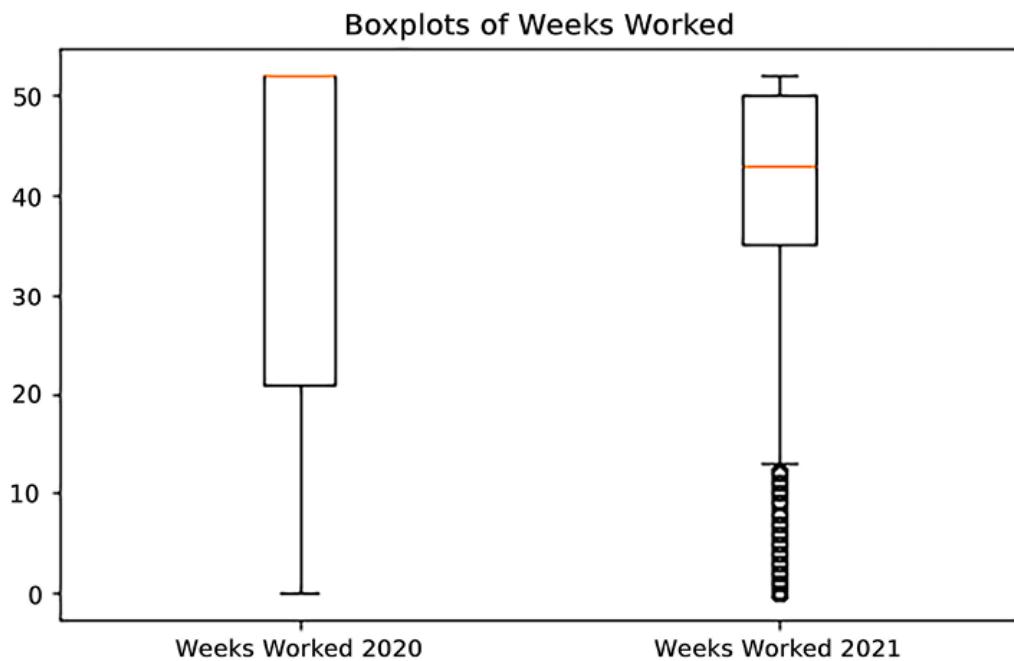
4. Do boxplots of weeks worked:

```

plt.boxplot([weeksworked.weeksworked20.dropna(),
    weeksworked.weeksworked21.dropna()],
    labels=[ 'Weeks Worked 2020', 'Weeks Worked 2021'])
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*

## 5. Show some descriptives for the COVID-19 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']

```

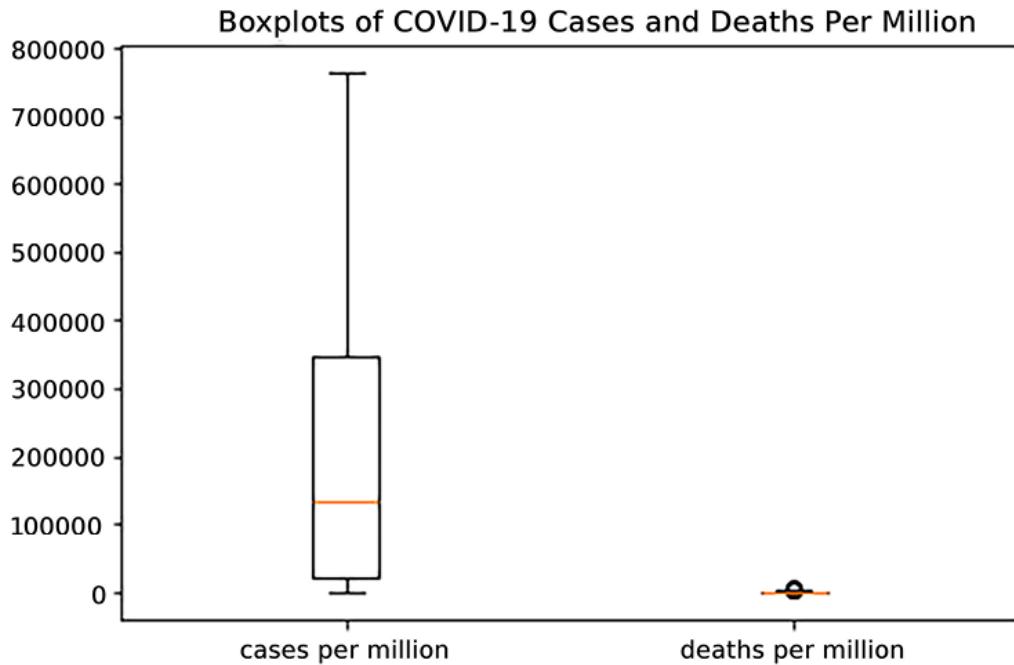
```
covidtotalsonly = covidtotals[totvars]
covidtotalsonly.describe()
```

|       | total_cases | total_deaths | total_cases_pm | total_de |
|-------|-------------|--------------|----------------|----------|
| count | 231         | 231          | 231            | 231      |
| mean  | 3,351,599   | 30,214       | 206,178        |          |
| std   | 11,483,212  | 104,779      | 203,858        |          |
| min   | 4           | 0            | 354            |          |
| 25%   | 25,672      | 178          | 21,822         |          |
| 50%   | 191,496     | 1,937        | 133,946        |          |
| 75%   | 1,294,286   | 14,150       | 345,690        |          |
| max   | 103,436,829 | 1,127,152    | 763,475        |          |

## 6. Do a boxplot of cases and deaths per million:

```
fig, ax = plt.subplots()
plt.title("Boxplots of COVID-19 Cases and Deaths Per
ax.boxplot([covidtotalsonly.total_cases_pm,covidtotal
...   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*

## 7. 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-19 cases and deaths. Fortunately, Matplotlib allows us to create multiple subplots on each figure, each of which can use different  $x$  and  $y$  axes. We can also present the data in thousands to improve readability:

```
fig, axes = plt.subplots(2, 2)
fig.suptitle("Boxplots of COVID-19 Cases and Deaths in Thor"
axes = axes.ravel()
for j, ax in enumerate(axes):
    ax.boxplot(covidtotalsonly.iloc[:, j]/1000, labels=[tot\)
plt.tight_layout()
fig.subplots_adjust(top=0.9)
plt.show()
```

This results in the following boxplots:

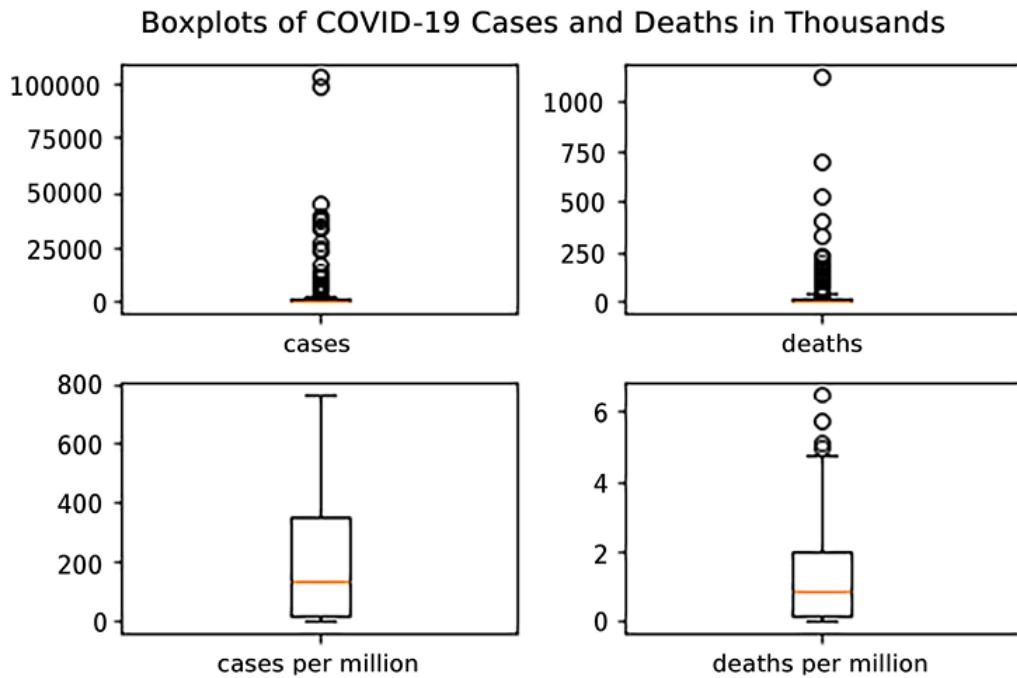


Figure 5.8: Boxplots with different y axes

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-19 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 2020 and 2021 in *Step 4* show variables that are distributed much differently than SAT scores. The medians are much higher than the mean. This suggests a negative skew. Also, notice that there are no whiskers or outliers at the upper end of the distribution for the 2020 value as the median 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, such as salaries for different age groups, number of children by marital status, or litter sizes of 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-19 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-19 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/nls97f.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
covidtotals = pd.read_csv("data/covidtotals.csv",
                           parse_dates=["lastdate"])
covidtotals.set_index("iso_code", inplace=True)

```

2. 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'])['weeksworked21'].\
    apply(gettots).unstack()

```

|                | min | qr1 | med | qr3 | max | count |
|----------------|-----|-----|-----|-----|-----|-------|
| highestdegree  |     |     |     |     |     |       |
| 0. None        | 0   | 0   | 39  | 49  | 52  | 487   |
| 1. GED         | 0   | 7   | 42  | 50  | 52  | 853   |
| 2. High School | 0   | 27  | 42  | 50  | 52  | 2,529 |
| 3. Associates  | 0   | 38  | 43  | 49  | 52  | 614   |
| 4. Bachelors   | 0   | 40  | 43  | 50  | 52  | 1,344 |
| 5. Masters     | 0   | 41  | 45  | 52  | 52  | 614   |

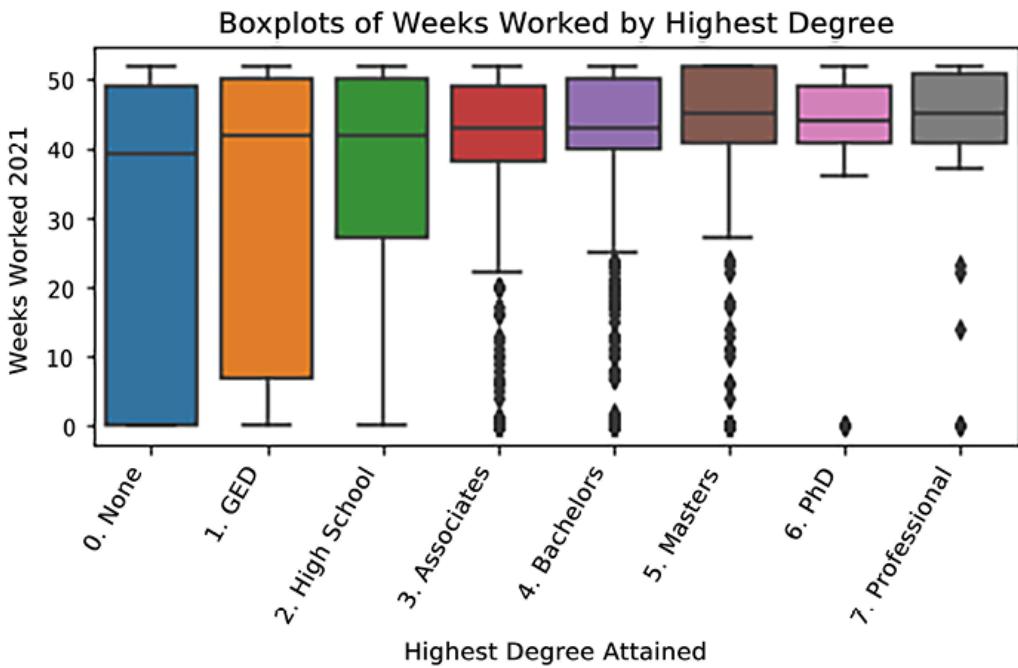
|                 |   |    |    |    |    |     |
|-----------------|---|----|----|----|----|-----|
| 6. PhD          | 0 | 41 | 44 | 49 | 52 | 59  |
| 7. Professional | 0 | 41 | 45 | 51 | 52 | 105 |

3. 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 almost all of the range of values. The value at the 25<sup>th</sup> percentile is 0:

```
myplt = \
    sns.boxplot(x='highestdegree', y='weeksworked21',
    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 2021')
myplt.set_xticklabels(myplt.get_xticklabels(), rotation=60)
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*

- 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()
```

| region      | min     | qr1     | med     | qr3     | max     | count |
|-------------|---------|---------|---------|---------|---------|-------|
| Caribbean   | 2,979   | 128,448 | 237,966 | 390,758 | 626,793 | 26    |
| Central Af  | 434     | 2,888   | 4,232   | 9,948   | 29,614  | 11    |
| Central Am  | 2,319   | 38,585  | 70,070  | 206,306 | 237,539 | 7     |
| Central As  | 1,787   | 7,146   | 45,454  | 79,795  | 162,356 | 6     |
| East Africa | 660     | 2,018   | 4,062   | 71,435  | 507,765 | 15    |
| East Asia   | 8,295   | 26,930  | 69,661  | 285,173 | 763,475 | 15    |
| East. Eu    | 104,252 | 166,930 | 223,685 | 459,646 | 760,161 | 21    |
| North Afr   | 4,649   | 6,058   | 34,141  | 74,463  | 93,343  | 5     |
| North Am    | 60,412  | 108,218 | 214,958 | 374,862 | 582,158 | 4     |
| Oceania/Aus | 4,620   | 75,769  | 259,196 | 356,829 | 508,709 | 24    |

|           |        |         |         |         |         |    |
|-----------|--------|---------|---------|---------|---------|----|
| South Am  | 19,529 | 101,490 | 133,367 | 259,942 | 505,919 | 14 |
| South As  | 5,630  | 11,959  | 31,772  | 80,128  | 473,167 | 9  |
| South. Af | 4,370  | 15,832  | 40,011  | 67,775  | 401,037 | 10 |
| West Af   | 363    | 1,407   | 2,961   | 4,783   | 108,695 | 16 |
| West Asia | 354    | 78,447  | 123,483 | 192,995 | 512,388 | 16 |
| West. Eu  | 32,178 | 289,756 | 465,940 | 587,523 | 750,727 | 32 |

## 5. Do boxplots of cases per million by region.

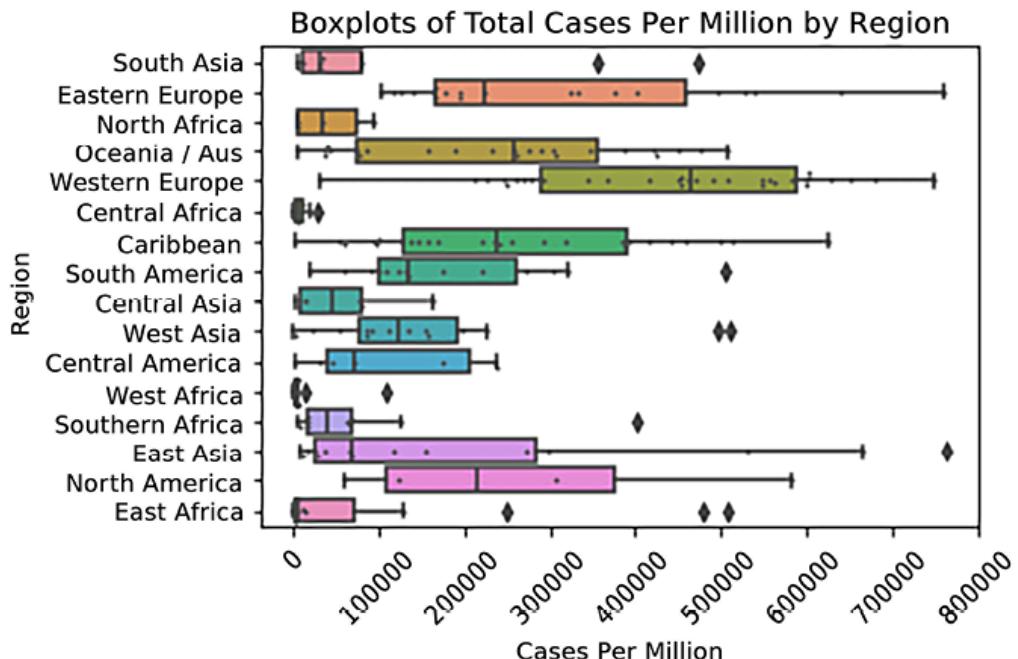
Flip the axes since there are a large number of regions. Also, do a swarm plot to give some sense of the number of countries by region. The swarm plot displays a dot for each country in each region. Some of the IQRs are hard to see because of the extreme values:

```

sns.boxplot(x='total_cases_pm', y='region', data=covidtot
sns.swarmplot(y="region", x="total_cases_pm", data=covidt
plt.title("Boxplots of Total Cases Per Million by Region")
plt.xlabel("Cases Per Million")
plt.ylabel("Region")
plt.xticks(rotation=45)
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*

6. Show the highest values for cases per million:

```
highvalue = covidtotals.total_cases_pm.quantile(0.9)
highvalue
```

```
512388.401
```

```
covidtotals.loc[covidtotals.total_cases_pm>=highvalue
['location','total_cases_pm']]
```

|          | location | total_cases_pm |
|----------|----------|----------------|
| iso_code |          |                |
| AND      | Andorra  | 601,367.7      |
| AUT      | Austria  | 680,262.6      |
| BRN      | Brunei   | 763,475.4      |
| CYP      | Cyprus   | 760,161.5      |
| DNK      | Denmark  | 583,624.9      |

|     |                           |           |
|-----|---------------------------|-----------|
| FRO | Faeroe Islands            | 652,484.1 |
| FRA | France                    | 603,427.6 |
| GIB | Gibraltar                 | 628,882.7 |
| GRC | Greece                    | 540,380.1 |
| GLP | Guadeloupe                | 513,528.3 |
| GGY | Guernsey                  | 557,817.1 |
| ISL | Iceland                   | 562,822.0 |
| ISR | Israel                    | 512,388.4 |
| JEY | Jersey                    | 599,218.4 |
| LVA | Latvia                    | 528,300.3 |
| LIE | Liechtenstein             | 548,113.3 |
| LUX | Luxembourg                | 603,439.5 |
| MTQ | Martinique                | 626,793.1 |
| PRT | Portugal                  | 549,320.5 |
| SPM | Saint Pierre and Miquelon | 582,158.0 |
| SMR | San Marino                | 750,727.2 |
| SGP | Singapore                 | 531,183.8 |
| SVN | Slovenia                  | 639,407.7 |
| KOR | South Korea               | 667,207.1 |

7. Redo the boxplots without the extreme values:

```

sns.boxplot(x='total_cases_pm', y='region', data=covi
sns.swarmplot(y="region", x="total_cases_pm", data=co
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:

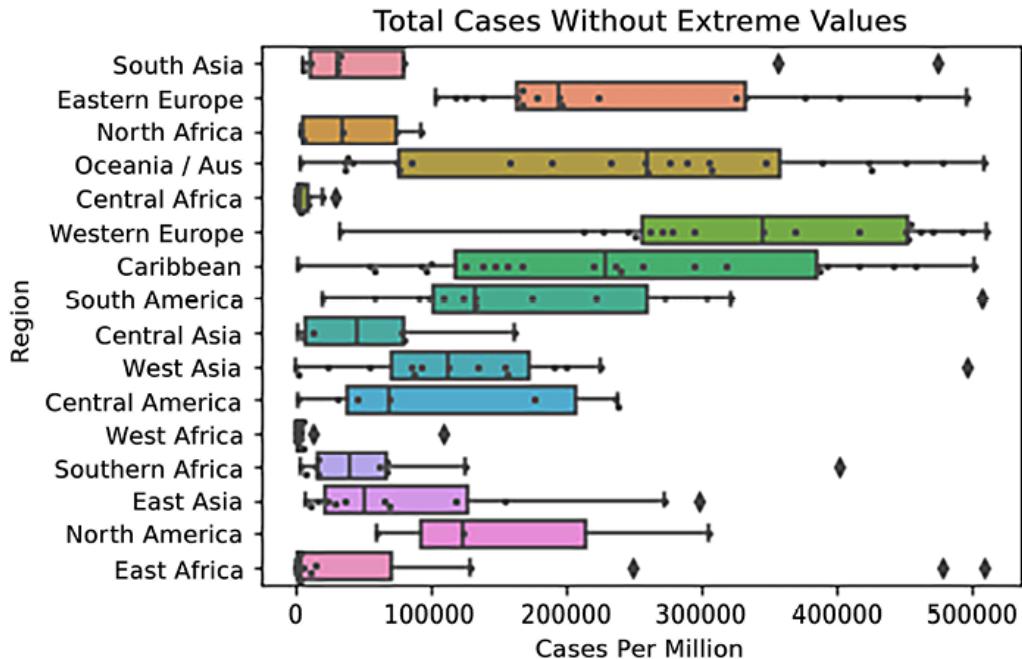


Figure 5.11: Boxplots of cases per million by region without the extreme values

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 DataFrame with the following statement:

```
nls97.groupby(['highestdegree'])['weeksworked21'].apply(gettots)
```

The `groupby` method creates a DataFrame 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 7*, we remove all values of `total_cases_pm` greater than or equal to 512,388. This improves the presentation of some details on the visualization.

## 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 2021 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 an outlier is 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 North Africa would be a high outlier for the dataset as a whole. The maximum value for North Africa is actually lower than the first quartile value for Western Europe.

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 a positive skew. Compare the boxplot for Eastern Europe 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 Eastern Europe.

## See also

We work much more with `groupby` in *Chapter 9, Fixing Messy Data When Aggregating*. We work more with `stack` and `unstack` in *Chapter 11*,

*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 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 NLS data:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
nls97 = pd.read_csv("data/nls97f.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
```

2. Do a violin plot of the SAT verbal score:

```
sns.violinplot(y=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='left')
plt.text(0.065, nls97.satverbal.quantile(0.75), '3rd quartile')
plt.text(0.05, nls97.satverbal.median(), 'Median', horizontalalignment='center')
plt.text(0.065, nls97.satverbal.quantile(0.25), '1st quartile')
plt.text(0.08, 210, 'outlier threshold', horizontalalignment='left')
plt.text(-0.4, 500, 'frequency', horizontalalignment='right')
plt.show()
```

This results in the following violin plot:

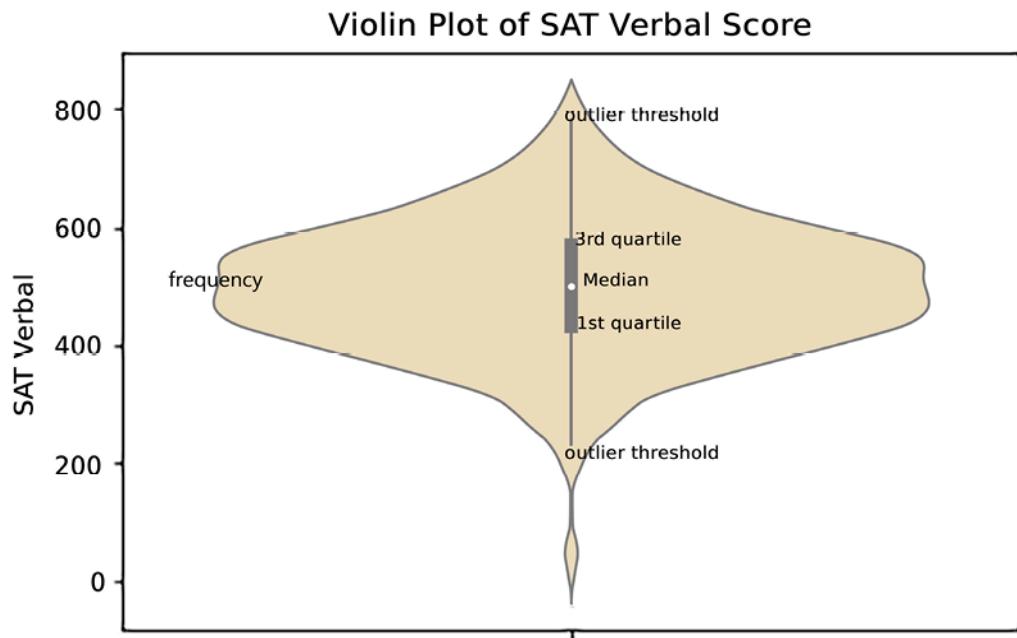


Figure 5.12: Violin plot of SAT verbal score with labels for IQR and outlier threshold

3. Get some descriptives for weeks worked:

```
nls97.loc[:, ['weeksworked20', 'weeksworked21']].descr
```

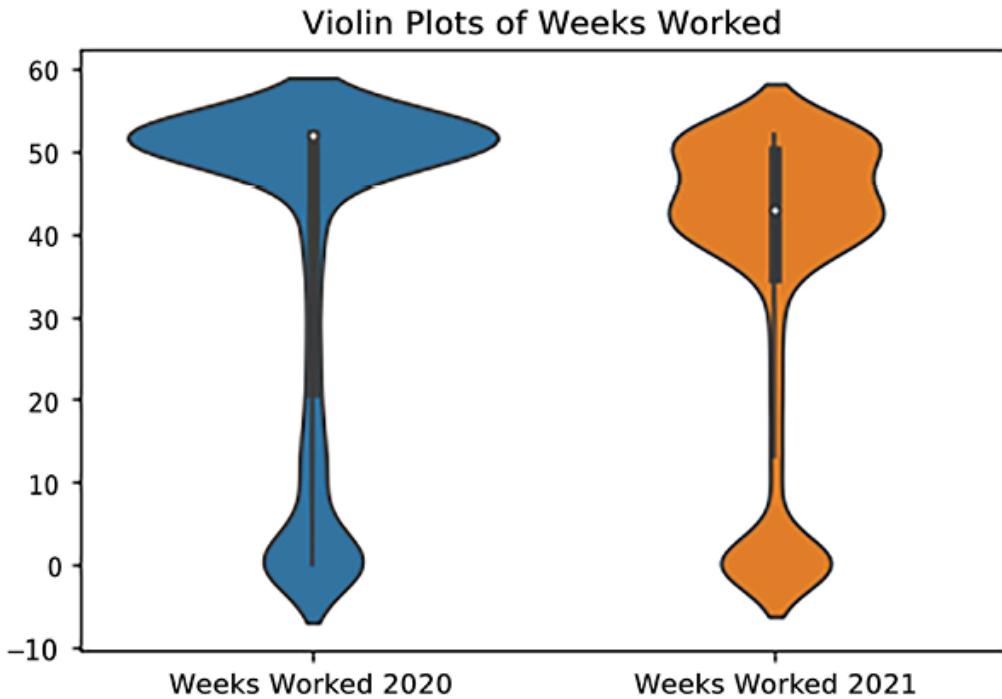
|       | weeksworked20 | weeksworked21 |
|-------|---------------|---------------|
| count | 6,971         | 6,627         |
| mean  | 38            | 36            |
| std   | 21            | 18            |
| min   | 0             | 0             |
| 25%   | 21            | 35            |
| 50%   | 52            | 43            |
| 75%   | 52            | 50            |
| max   | 52            | 52            |

4. Show weeks worked for 2020 and 2021.

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 2020 and 2021:

```
myplt = sns.violinplot(data=nls97.loc[:, ['weeksworked20',
myplt.set_title("Violin Plots of Weeks Worked")
myplt.set_xticklabels(["Weeks Worked 2020", "Weeks Worked 2021"])
plt.show()
```

This results in the following violin plots:



*Figure 5.13: Violin plots showing the spread and shape of the distribution for two variables side by side*

## 5. 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:

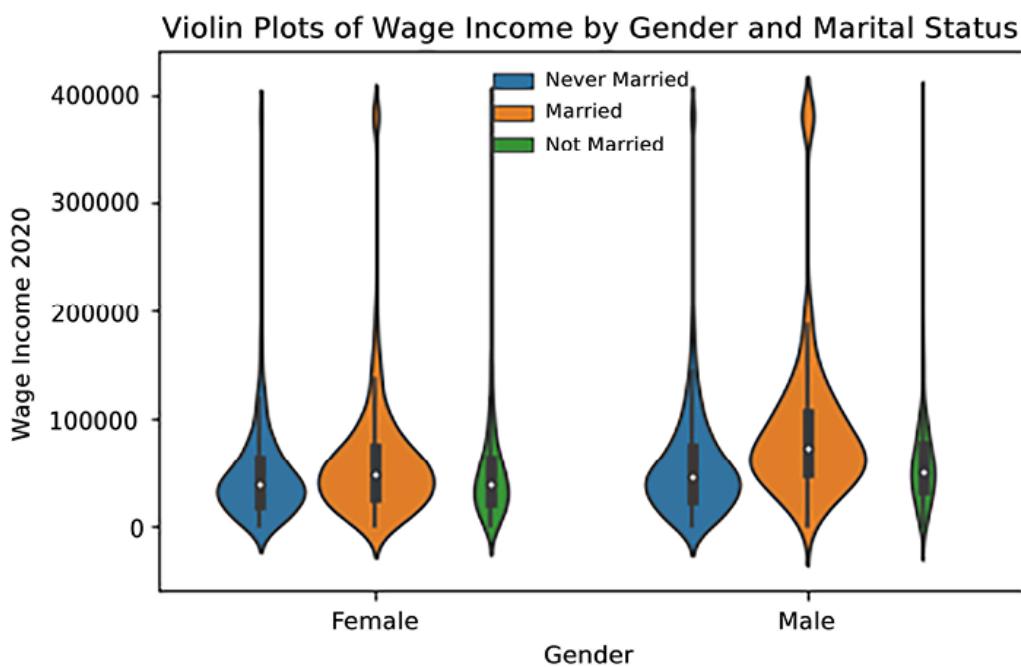
```
nls97["maritalstatuscollapsed"] = \
nls97.maritalstatus.replace(['Married',
    'Never-married', 'Divorced', 'Separated',
    'Widowed'], \
['Married', 'Never Married', 'Not Married',
    'Not Married', 'Not Married'])
sns.violinplot(x="gender", y="wageincome20", hue="maritalst
```

```

data=nls97, scale="count")
plt.title("Violin Plots of Wage Income by Gender and Marital Status")
plt.xlabel('Gender')
plt.ylabel('Wage Income 2020')
plt.legend(title="", loc="upper center", framealpha=0, fontsize=10)
plt.tight_layout()
plt.show()

```

This results in the following violin plots:



*Figure 5.14: Violin plots showing the spread and shape of the distribution across two different groups*

6. Do violin plots of weeks worked by highest degree attained:

```

nls97 = nls97.sort_values(['highestdegree'])
myplt = sns.violinplot(x='highestdegree',y='weeksworke
myplt.set_xticklabels(myplt.get_xticklabels(), rotation=45)
myplt.set_title("Violin Plots of Weeks Worked by Highest Degree Attained")
myplt.set_xlabel('Highest Degree Attained')
myplt.set_ylabel('Weeks Worked 2021')

```

```
plt.tight_layout()  
plt.show()
```

This results in the following violin plots:

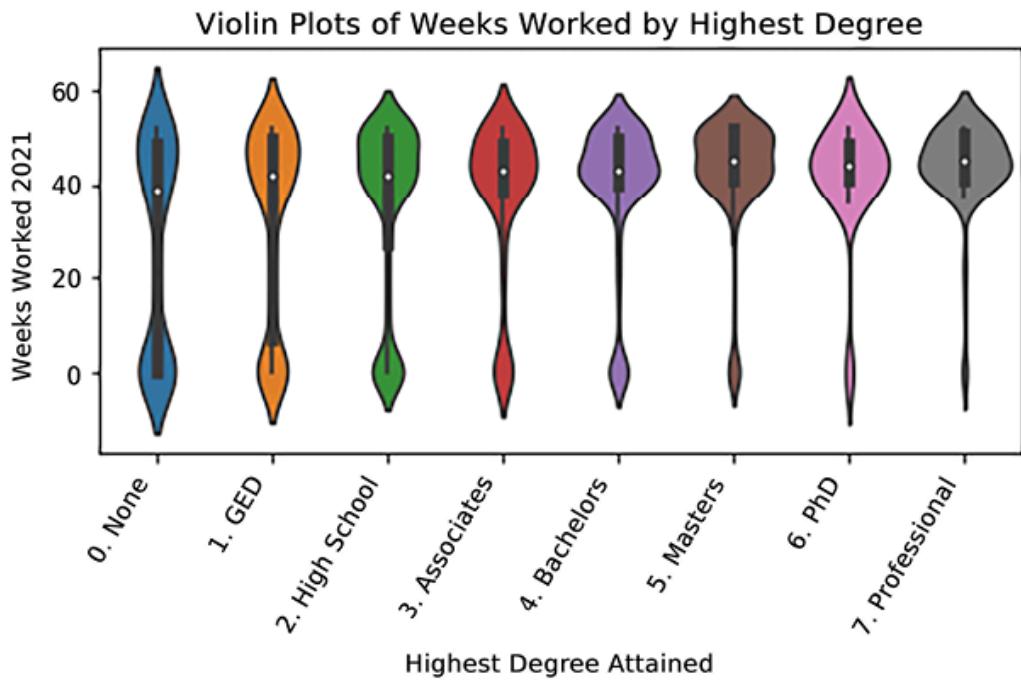


Figure 5.15: Violin plots showing the spread and shape of the distribution by group

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, the 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 DataFrame 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 the legend in the best place on the figure, and make the box transparent  
(`framealpha=0`):

```
plt.legend(title="", loc="upper center", framealpha=0, fontsize=
```

## 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 2020 is different enough from weeks worked in 2021 to give the careful analyst pause. The IQR is quite different

—31 for 2020 (21 to 52) and 15 for 2021 (35 to 50). (The weeks worked in 2020 distribution was likely impacted by the pandemic.)

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 \$380,288. 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 were nearly as likely to work 5 or fewer weeks in 2021 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 2023 for 12,137 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` and `seaborn`:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
landtemps = pd.read_csv("data/landtemps2023avgs.csv")
```

2. 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 2023")
plt.show()
```

This results in the following scatter plot:

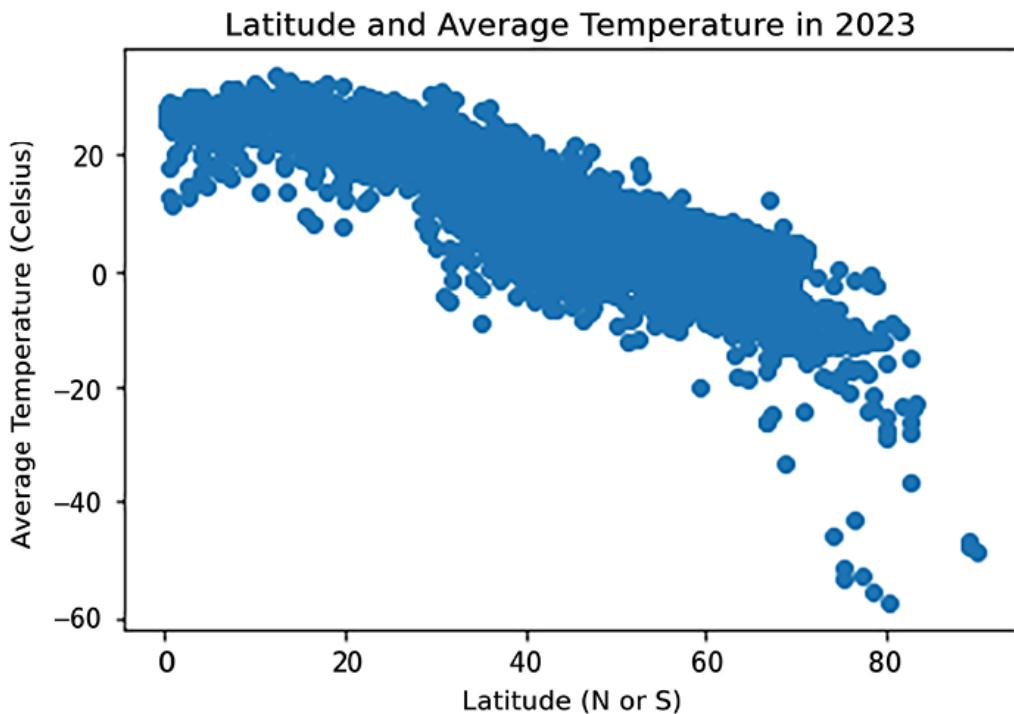


Figure 5.16: Scatter plot of latitude by average temperature

### 3. Show the high elevation points in red.

Create low and high elevation DataFrames. 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 2023")
plt.show()
```

This results in the following scatter plot:

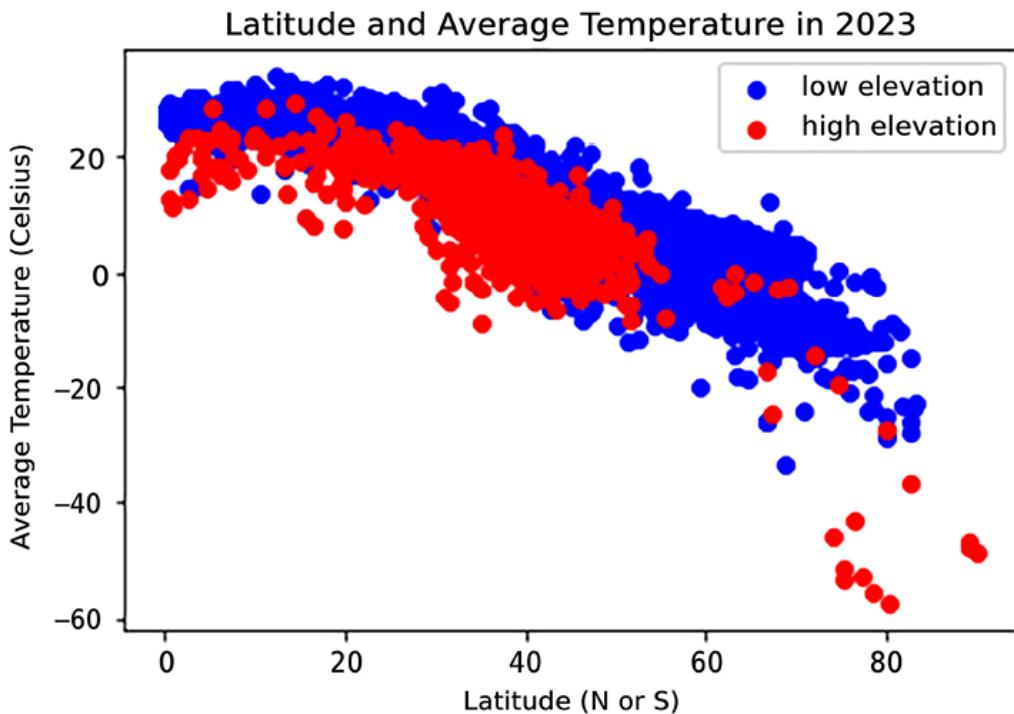


Figure 5.17: Scatter plot of latitude by average temperature and elevation

4. 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 2023")
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')
ax.scatter3D(high.elevation, high.latabs, high.avgtemp, label='High')
ax.legend()
plt.show()
```

This results in the following scatter plot:

## Latitude, Temperature, and Elevation in 2023

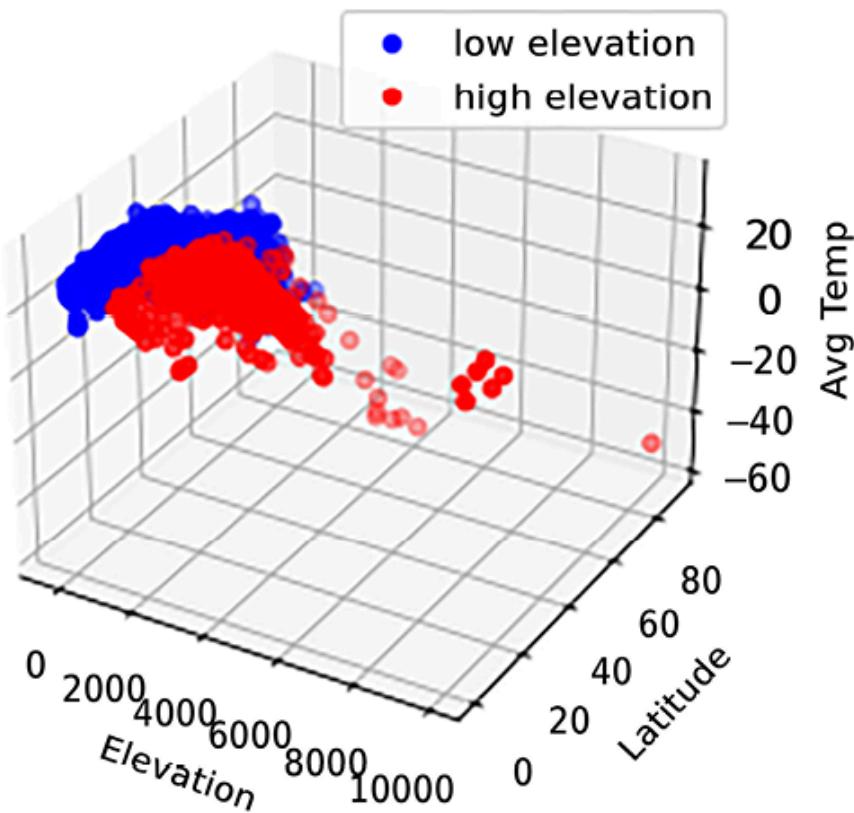


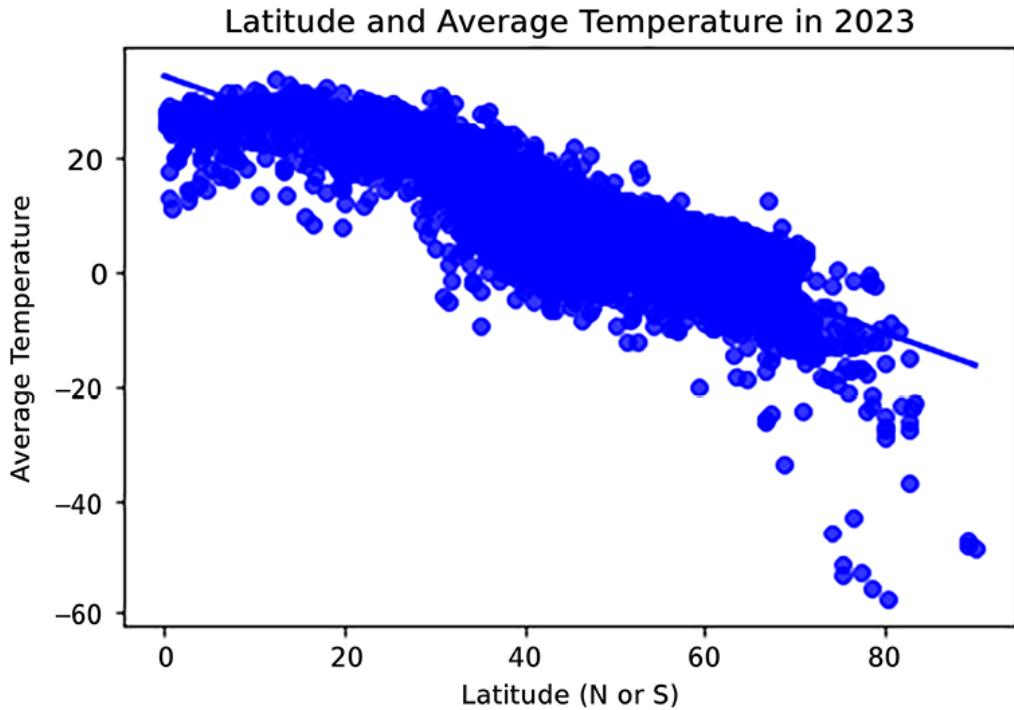
Figure 5.18: 3D scatter plot of latitude and elevation by average temperature

5. Show a regression line of latitude on temperature.

Use `regplot` to get a regression line:

```
sns.regplot(x="latabs", y="avgtemp", color="blue", data=lat)
plt.title("Latitude and Average Temperature in 2023")
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*

6. 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'] = np.where(landtemps.elevation<=100
sns.lmplot(x="latabs", y="avgtemp", hue="elevation", palette="viridis")
plt.xlabel("Latitude (N or S)")
plt.ylabel("Average Temperature")
plt.yticks(np.arange(-60, 40, step=20))
plt.title("Latitude and Average Temperature in 2023")
plt.show()
```

This results in the following scatter plot:

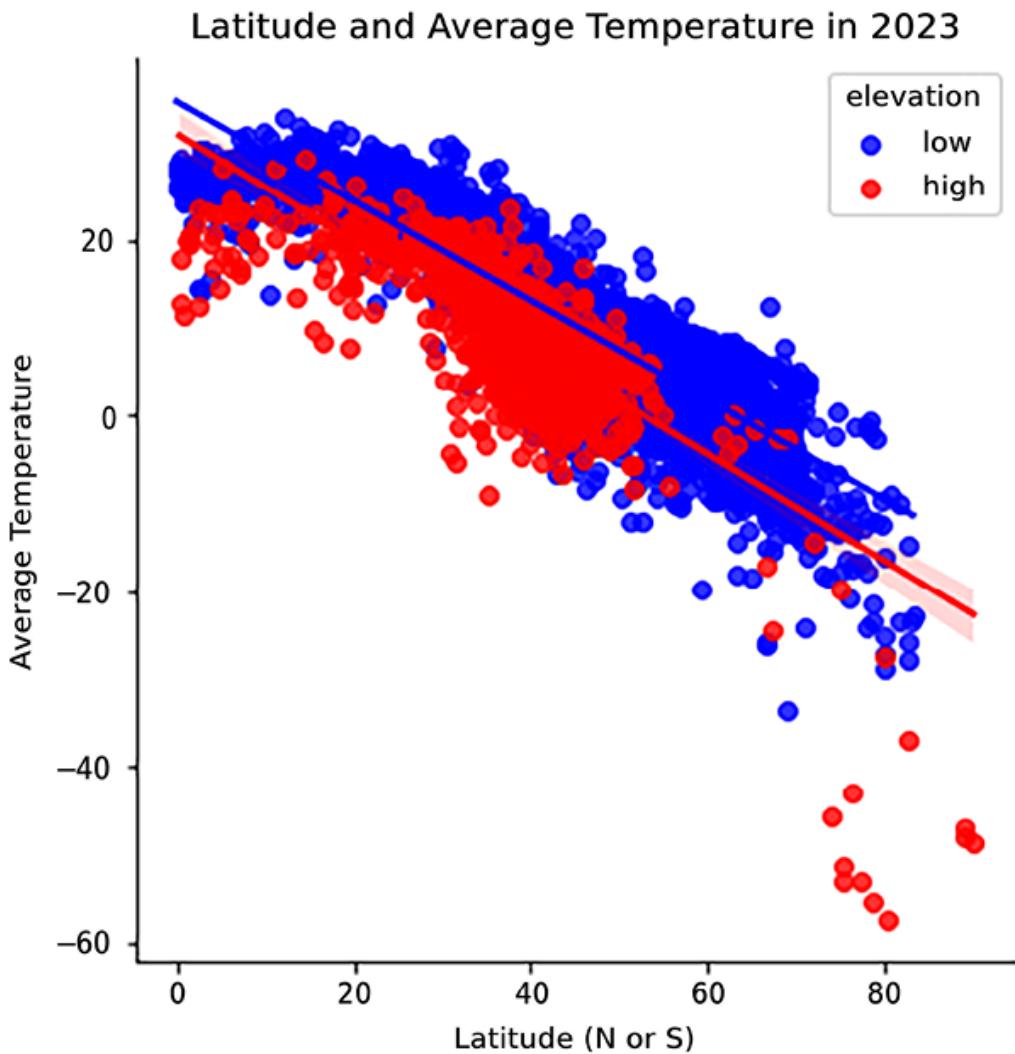


Figure 5.20: Scatter plot of latitude by temperature with separate regression lines for elevation

7. Show some stations above the low- and high-elevation regression lines. We can use the `high` and `low` DataFrames we created in Step 3:

```
high.loc[(high.latabs>38) & \
...     (high.avgtemp>=18),
...     ['station', 'country', 'latabs',
...      'elevation', 'avgtemp']]
```

|    | station | country | latabs | elevation | avgtemp |
|----|---------|---------|--------|-----------|---------|
| 82 | YEREVAN | Armenia | 40     | 1,113     | 19      |

|      |          |          |    |       |    |
|------|----------|----------|----|-------|----|
| 3968 | LAJES_AB | Portugal | 39 | 1,016 | 19 |
|------|----------|----------|----|-------|----|

```
low.loc[(low.latabs>47) & \
...      (low.avgtemp>=14),
...      ['station','country','latabs',
...       'elevation','avgtemp']]
```

|      |                       | station       | country | latabs | elevati |
|------|-----------------------|---------------|---------|--------|---------|
| 1026 | COURTENAY_PUNTLEDGE   |               | Canada  | 50     |         |
| 1055 | HOWE_SOUNDPAK_ROCKS   |               | Canada  | 49     |         |
| 1318 | FORESTBURG_PLANT_SITE |               | Canada  | 52     |         |
| 2573 | POINTE_DU_TALUT       |               | France  | 47     |         |
| 2574 | NANTES_ATLANTIQUE     |               | France  | 47     |         |
| 4449 | USTORDYNNSKIJ         |               | Russia  | 53     |         |
| 6810 | WALKER_AH_GWAH_CHING  | United States |         | 47     |         |
| 7050 | MEDICINE LAKE_3_SE    | United States |         | 48     |         |
| 8736 | QUINCY                | United States |         | 47     |         |
| 9914 | WINDIGO_MICHIGAN      | United States |         | 48     |         |

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 as 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 set the projection of our axes to `3d—plt.axes(projection='3d')`, as we did 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 *Step 7*, we view some of the outliers: those stations with temperatures much higher than the regression line for their group. We would want to investigate the data for the `LAJES_AB` station in Portugal and the `YEREVAN` station in Armenia (`((high.latabs>38) & (high.avgtemp>=18))`). The average temperatures are higher than would be predicted at the given latitude and elevation level.

## 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, via the creative use of colors (when the third dimension is categorical), or by changing the size of the dots (the *Using linear regression to identify data points with high influence* recipe in *Chapter 4, Identifying 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 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-19 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 COVID-19 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/coviddaily.csv", parse
```

2. View a couple of rows of the COVID-19 daily data:

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

|                |            |            |
|----------------|------------|------------|
| iso_code       | 628        | 26980      |
| casedate       | AND        | PRT        |
| location       | 2020-03-15 | 2022-12-04 |
| continent      | Andorra    | Portugal   |
| new_cases      | Europe     | Europe     |
| new_deaths     | 1          | 3,963      |
| population     | 0          | 69         |
| pop_density    | 79843      | 10270857   |
| median_age     | 164        | 112        |
| gdp_per_capita | NaN        | 46         |
| hosp_beds      | NaN        | 27,937     |
|                |            | 3          |

|                 |                |                |
|-----------------|----------------|----------------|
| vac_per_hund    | NaN            | NaN            |
| aged_65_older   | NaN            | 22             |
| life_expectancy | 84             | 82             |
| hum_dev_ind     | 1              | 1              |
| region          | Western Europe | Western Europe |

### 3. Calculate new cases and deaths by day.

Select dates between July 1, 2023, and March 3, 2024, and then use `groupby` to summarize cases and deaths across all countries for each day:

```
coviddailytotals = \
    coviddaily.loc[coviddaily.casedate.\
        between('2023-07-01', '2024-03-03')].\\
    groupby(['casedate'])[['new_cases', 'new_deaths']].\
    sum().\
    reset_index()
```

| casedate      | new_cases | new_deaths |
|---------------|-----------|------------|
| 27 2024-01-07 | 181,487   | 1,353      |
| 3 2023-07-23  | 254,984   | 596        |
| 22 2023-12-03 | 282,319   | 1,535      |
| 18 2023-11-05 | 158,346   | 1,162      |
| 23 2023-12-10 | 333,155   | 1,658      |
| 17 2023-10-29 | 144,325   | 905        |
| 21 2023-11-26 | 238,282   | 1,287      |

### 4. Show line plots for new cases and new deaths by day.

Show the cases and deaths on different subplots:

```
fig = plt.figure()
plt.suptitle("New COVID-19 Cases and Deaths By Day Worldwide")
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:

New COVID-19 Cases and Deaths By Day Worldwide 2023-2024

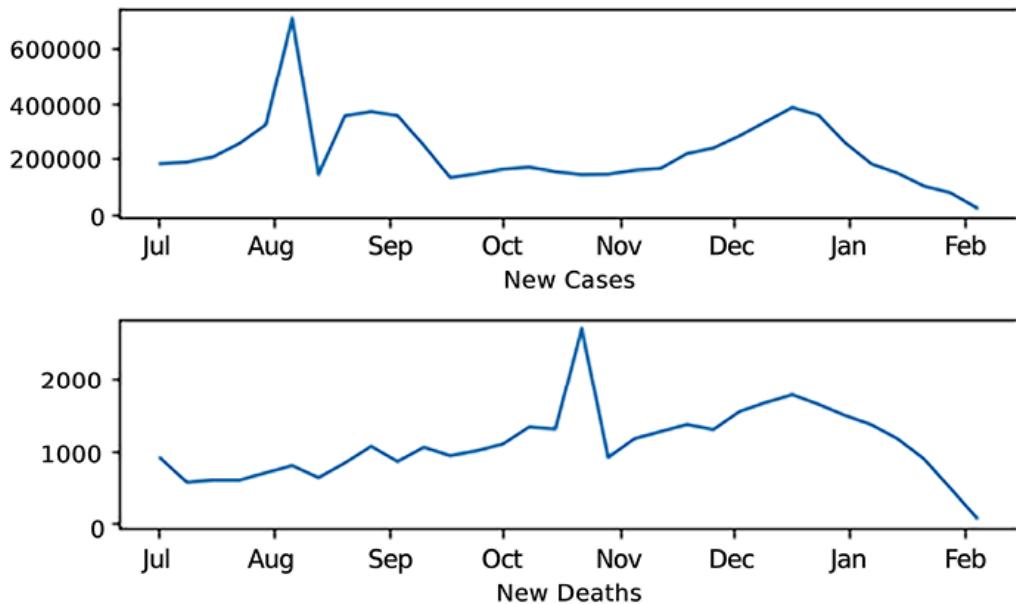


Figure 5.21: Daily trend lines of worldwide COVID-19 cases and deaths

5. Calculate new cases and deaths by day and region:

```

regiontotals = \
coviddaily.loc[coviddaily.casedate.\
between('2023-07-01','2024-03-03')].\

```

```

groupby(['casedate', 'region'])\ 
    [['new_cases', 'new_deaths']].\ 
    sum().\ 
    reset_index()
regiontotals.sample(7, random_state=1)

```

|     | casedate   | region        | new_cases | new_deaths |
|-----|------------|---------------|-----------|------------|
| 110 | 2023-08-13 | West Asia     | 2,313     | 25         |
| 147 | 2023-09-03 | Central Asia  | 600       | 7          |
| 494 | 2024-02-04 | Oceania / Aus | 12,594    | 38         |
| 325 | 2023-11-19 | East Asia     | 20,088    | 15         |
| 189 | 2023-09-17 | West Africa   | 85        | 0          |
| 218 | 2023-10-01 | South America | 4,203     | 54         |
| 469 | 2024-01-21 | Oceania / Aus | 17,503    | 129        |

## 6. 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-19 Cases By Day and Region in 2023-2024")
plt.gca().get_xaxis().set_major_formatter(DateFormatter("%Y-%m-%d"))
plt.ylabel("New Cases")
plt.legend()
plt.show()

```

This results in the following line plots:

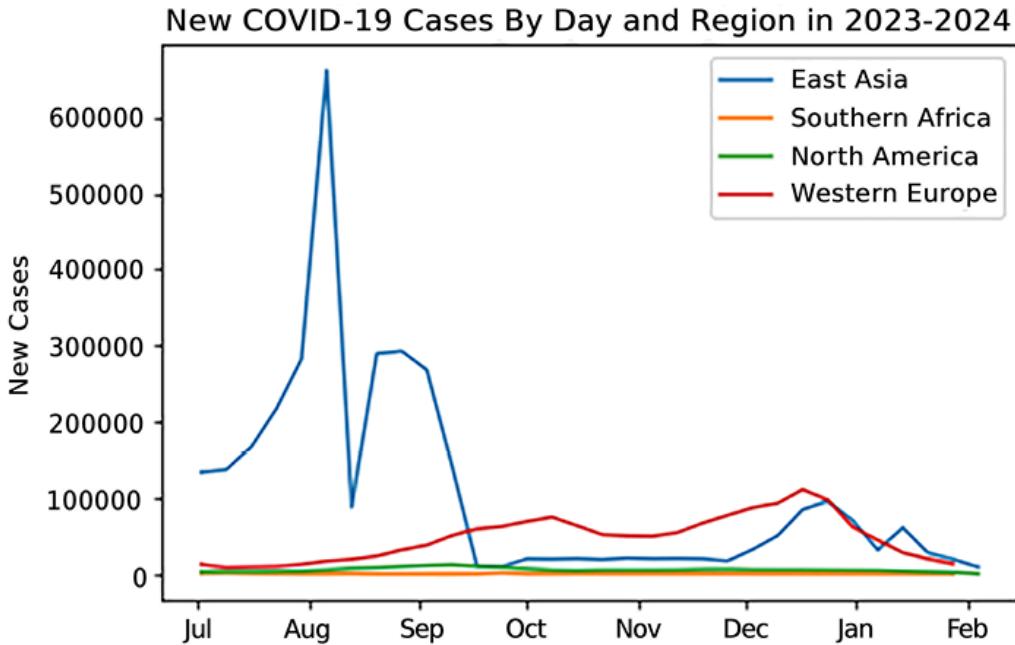


Figure 5.22: Daily trend lines of COVID-19 cases by region

7. Use a stacked plot to examine the trend in a region more closely.

See if one country (Brazil) in South America is driving the trend line.

Create a DataFrame (`sa`) for `new_cases` by day for South America.

Add a Series for `new_cases` in Brazil to the `sa` DataFrame. Then, create a new Series in the `sa` DataFrame for South America cases minus Brazil cases (`sacasesnobr`):

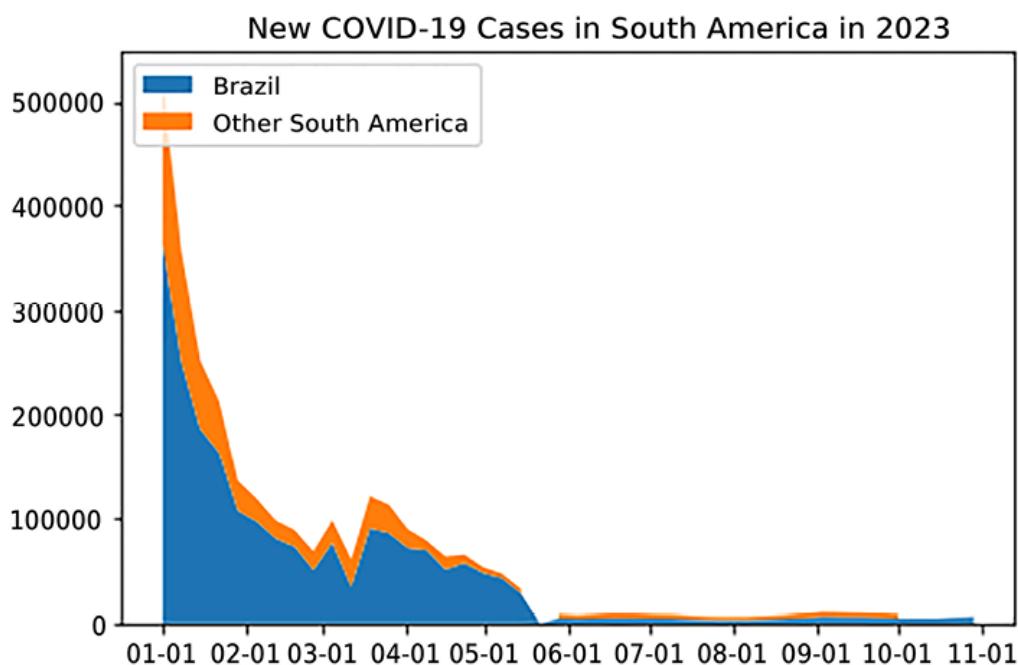
```
sa = \
coviddaily.loc[(coviddaily.casedate.\
between('2023-01-01', '2023-10-31')) & \
(coviddaily.region=='South America'),
['casedate','new_cases']] .\
groupby(['casedate']) .\
[[['new_cases']] .\
sum() .\
reset_index() .\
rename(columns={'new_cases':'sacases'})\nbr = coviddaily.loc[(coviddaily.\
```

```

location=='Brazil') & \
(coviddaily.casedate. \
between('2023-01-01','2023-10-31')), \
['casedate','new_cases']].\
rename(columns={'new_cases':'brcases'})
sa = pd.merge(sa, br, left_on=['casedate'], right_on=[ 'cas
sa.fillna({"sacases": 0},
inplace=True)
sa['sacasesnbr'] = sa.sacases-sa.brcases
fig = plt.figure()
ax = plt.subplot()
ax.stackplot(sa.casedate, sa.sacases, sa.sacasesnbr, lab
ax.xaxis.set_major_formatter(DateFormatter("%m-%d"))
plt.title("New COVID-19 Cases in South America in 2023")
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 Brazil and the rest of South America*

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-19 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 new Brazil cases out of the cases for South America. Once we do that, we can do a stacked plot with the South America cases (minus Brazil) and Brazil. This

figure suggests that the trends in new cases in South America in 2023 were largely driven by trends in Brazil.

## There's more...

The figure produced in *Step 6* reveals a couple of potential data issues. There is an unusual spike in April of 2023 in East Asia. It is important to examine these totals to check whether 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.

## See also

We cover `groupby` in more detail in *Chapter 9, Fixing Messy Data When Aggregating*. We go over merging data, as we did in *Step 7*, in *Chapter 10, 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 (such as Pearson, Spearman, and 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-19 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-19 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", par
```

2. Generate a correlation matrix.

View part of the matrix:

```

corr = covidtotals.corr(numeric_only=True)
corr[['total_cases','total_deaths',
      'total_cases_pm','total_deaths_pm']]

```

|                 | total_cases    | total_deaths    | \ |
|-----------------|----------------|-----------------|---|
| total_cases     | 1.00           | 0.76            |   |
| total_deaths    | 0.76           | 1.00            |   |
| total_cases_pm  | 0.10           | 0.01            |   |
| total_deaths_pm | 0.15           | 0.27            |   |
| population      | 0.70           | 0.47            |   |
| pop_density     | -0.03          | -0.04           |   |
| median_age      | 0.29           | 0.19            |   |
| gdp_per_capita  | 0.19           | 0.13            |   |
| hosp_beds       | 0.21           | 0.05            |   |
| vac_per_hund    | 0.02           | -0.07           |   |
| aged_65_older   | 0.29           | 0.19            |   |
| life_expectancy | 0.19           | 0.11            |   |
| hum_dev_ind     | 0.26           | 0.21            |   |
|                 | total_cases_pm | total_deaths_pm |   |
| total_cases     | 0.10           | 0.15            |   |
| total_deaths    | 0.01           | 0.27            |   |
| total_cases_pm  | 1.00           | 0.44            |   |
| total_deaths_pm | 0.44           | 1.00            |   |
| population      | -0.13          | -0.07           |   |
| pop_density     | 0.19           | 0.02            |   |
| median_age      | 0.74           | 0.69            |   |
| gdp_per_capita  | 0.66           | 0.29            |   |
| hosp_beds       | 0.48           | 0.39            |   |
| vac_per_hund    | 0.24           | -0.07           |   |
| aged_65_older   | 0.72           | 0.68            |   |
| life_expectancy | 0.69           | 0.49            |   |
| hum_dev_ind     | 0.76           | 0.60            |   |

3. 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

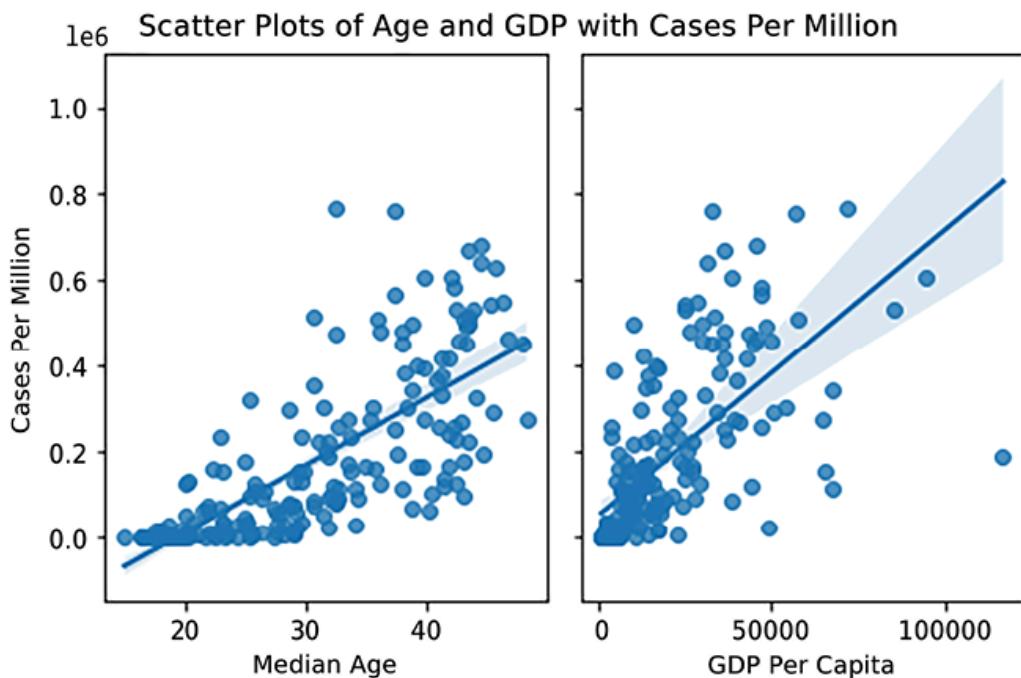
`sharey=True`:

```

fig, axes = plt.subplots(1, 2, sharey=True)
sns.regplot(x="median_age", y="total_cases_pm", data=covid)
sns.regplot(x="gdp_per_capita", y="total_cases_pm", data=covid)
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 Million")
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*

4. Generate a heat map of the correlation matrix:

```

sns.heatmap(corr, xticklabels=corr.columns, yticklabels=corr.columns)
plt.title('Heat Map of Correlation Matrix')

```

```
plt.tight_layout()  
plt.show()
```

This results in the following heat map:

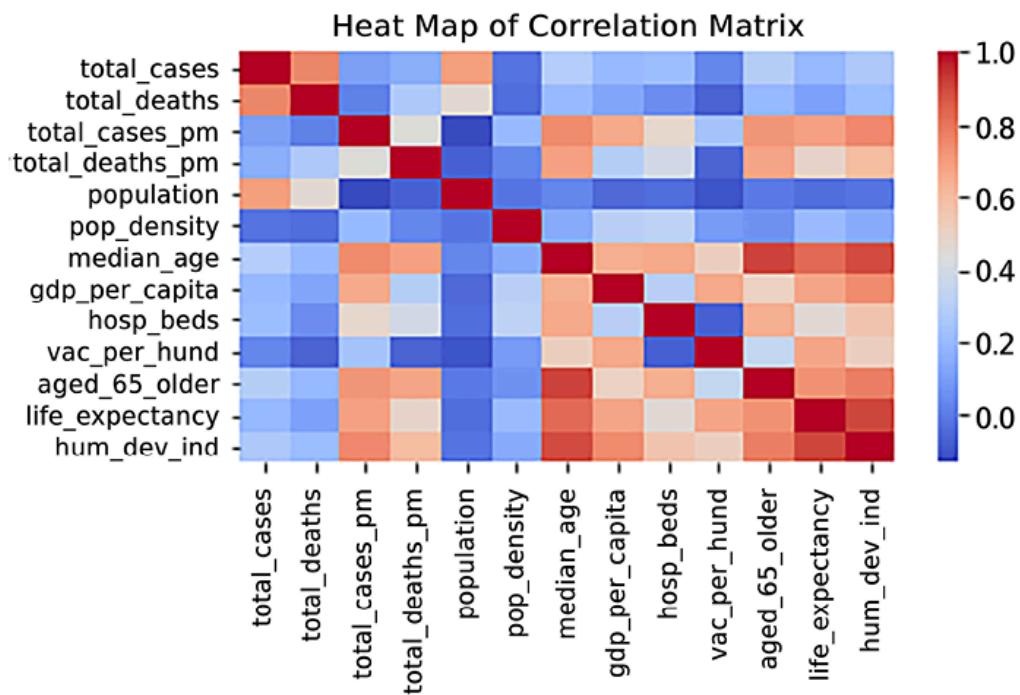


Figure 5.25: Heat map of COVID-19 data, with strongest correlations in red and peach

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 make 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 0.74 (median age and

cases per million) and when it is 0.66 (GDP per capita and cases per million). Countries with higher median ages or higher GDP per capita tend to have higher cases per million of population.

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 peach rectangles indicate the substantially positive correlations, such as those between median age, GDP per capita, and human development index and cases per million. The dark rectangles indicate that the relationships are strongly negative, such as that between vaccinations per 100,000 of the population and deaths 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 Outliers in Subsets of Data*.

## Summary

Histograms, boxplots, scatter plots, violin plots, line plots, and heat maps are all essential tools for understanding how variables are distributed.

Scatter plots, violin plots, and heat maps can also help us better understand

relationships between variables, whether they are continuous or categorical. We created visualizations with all of those tools in this chapter. In the next chapter, we will examine how to create new Series in pandas, or modify values in an existing Series.

## Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review.

Scan the QR code below to get a free eBook of your choice.



# 6

## Cleaning and Exploring Data with Series Operations

We can view the recipes in the first few chapters of this book as, essentially, diagnostic. We imported some raw data and then generated descriptive statistics about key variables. This gave us a sense of how the values for those variables were distributed and helped us identify outliers and unexpected values. We then examined the relationships between variables to look for patterns, and deviations from those patterns, including logical inconsistencies. In short, our primary goal so far has been to figure out what is going on with our data.

But, not very long into a data exploration and cleaning project, we invariably need to alter the initial values for some of our variables across some of our observations. For example, we might need to create a new column that is based on the values of one or more other columns. Or, we might want to change values that are in a certain range, say less than 0, or over some threshold amount, perhaps setting them to the mean, or to missing. Fortunately, the pandas Series object offers a large number of methods for manipulating data values.

The recipes in this chapter demonstrate how to use pandas methods to update Series values once we have figured out what needs to be done. Ideally, we need to take the time to carefully examine our data before

manipulating the values of our variables. We should have measures of central tendency, indicators of distribution shape and spread, correlations, and visualizations in front of us before we update the variable's values, or before creating new variables based on them. We should also have a good sense of outliers and missing values, understand how they affect summary statistics, and have preliminary plans for imputing new values or otherwise adjusting them.

Having done that, we will be ready to perform some data cleaning tasks. These tasks usually involve working directly with a pandas Series object, regardless of whether we are changing values for an existing Series or creating a new one. This often involves changing values conditionally, altering only those values that meet specific criteria, or assigning multiple possible values based on existing values for that Series, or values for another Series.

How we assign such values varies significantly by the Series' data type, either for the Series to be changed or a criterion Series. Querying and cleaning string data bears little resemblance to those tasks with date or numeric data. With strings, we often need to evaluate whether some string fragment does or does not have a certain value, strip the string of some meaningless characters, or convert the value into a numeric or date value. With dates, we might need to look for invalid or out-of-range dates, or even calculate date intervals.

Fortunately, pandas Series have an enormous number of tools for manipulating strings, numeric, and date values. We will explore many of the most useful tools in this chapter. Specifically, we will cover the following recipes:

- Getting values from a pandas Series

- Showing summary statistics for a pandas Series
- Changing Series values
- Changing Series values conditionally
- Evaluating and cleaning string Series data
- Working with dates
- Using OpenAI for Series operations

Let's get started!

## Technical requirements

You will need pandas, NumPy, and Matplotlib to complete the recipes in this chapter. I used pandas 2.1.4, but the code will run on pandas 1.5.3 or later.

The code in this chapter can be downloaded from the book's GitHub repository, <https://github.com/PacktPublishing/Python-Data-Cleaning-Cookbook-Second-Edition>.

## Getting values from a pandas Series

A pandas Series is a one-dimensional array-like structure that takes a NumPy data type. Each Series also has an index, an array of data labels. If an index is not specified when the Series is created, it will be the default index of 0 through N-1.

There are several ways to create a pandas Series, including from a list, dictionary, NumPy array, or a scalar. In our data cleaning work, we will most frequently be accessing data Series by selecting columns of

DataFrames, using either attribute access (`dataframename.columnname`) or bracket notation (`dataframename['columnname']`). Attribute access cannot be used to set values for Series, but bracket notation will work for all Series operations.

In this recipe, we'll explore several ways we can get values from a pandas Series. These techniques are very similar to the methods we used to get rows from a pandas DataFrame, which we covered in the *Selecting rows* recipe of *Chapter 3, Taking the Measure of Your Data*.

## Getting ready

We will be working with data from the **National Longitudinal Survey (NLS)** in this recipe—primarily with data about each respondent's overall high school **Grade Point Average (GPA)**.



### Data 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 who were born between 1980 and 1985, with annual follow-ups each year until 2023. Survey data is available for public use at [nlsinfo.org](http://nlsinfo.org).

## How to do it...

For this recipe, we select Series values using the bracket operator and the `loc` and `iloc` accessors. Let's get started:

## 1. Import the required `pandas` and NLS data:

```
import pandas as pd  
nls97 = pd.read_csv("data/nls97f.csv", low_memory=False)  
nls97.set_index("personid", inplace=True)
```

### Note



Whether you use the bracket operator, the `loc` accessor, or the `iloc` accessor is largely a matter of preference. It is usually easier to use the `loc` accessor when you know the index label for the rows you want. When it is easier to refer to rows by their absolute position, the bracket operator or `iloc` accessor will probably be a better choice. The examples in this recipe illustrate this.

## 2. Create a Series from the GPA overall column.

Show the first few values and associated index labels using `head`. The default number of values shown for `head` is 5. The index for the Series is the same as the DataFrame's index, which is `personid`:

```
gpaoverall = nls97.gpaoverall  
type(gpaoverall)
```

`pandas.core.series.Series`

```
gpaoverall.head()
```

```
personid
135335    3.09
999406    2.17
151672     NaN
750699    2.53
781297    2.43
Name: gpaoverall, dtype: float64
```

```
gpaoverall.index
```

```
Index([135335, 999406, 151672, 750699, 781297, 613800,
       403743, 474817, 530234, 351406,
       ...
       290800, 209909, 756325, 543646, 411195, 505861,
       368078, 215605, 643085, 713757],
      dtype='int64', name='personid', length=8984)
```

### 3. Select GPA values using the bracket operator.

Use slicing to create a Series with every value from the first value to the fifth. Notice that we get the same values that we got with the `head` method in *step 2*. Not including a value to the left of the colon in `gpaoverall[:5]` means that it will start from the beginning. `gpaoverall[0:5]` will give the same results. Similarly, `gpaoverall[-5:]` shows the values from the fifth to the last position. This produces the same results as `gpaoverall.tail()`:

```
gpaoverall[:5]
```

```
135335    3.09
999406    2.17
151672     NaN
750699    2.53
```

```
781297    2.43  
Name: gpaoverall, dtype: float64
```

```
gpaoverall.tail()
```

```
personid  
505861      NaN  
368078      NaN  
215605    3.22  
643085    2.30  
713757      NaN  
Name: gpaoverall, dtype: float64
```

```
gpaoverall[-5:]
```

```
personid  
505861      NaN  
368078      NaN  
215605    3.22  
643085    2.30  
713757      NaN  
Name: gpaoverall, dtype: float64
```

#### 4. Select values using the `loc` accessor.

We pass an index label (a value for `personid` in this case) to the `loc` accessor to return a scalar. We get a Series if we pass a list of index labels, regardless of whether there's one or more. We can even pass a range, separated by a colon. We'll do this here with

```
gpaoverall.loc[135335:151672]:
```

```
gpaoverall.loc[135335]  
3.09
```

```
gpaoverall.loc[[135335]]
```

```
personid  
135335    3.09  
Name: gpaoverall, dtype: float64
```

```
gpaoverall.loc[[135335, 999406, 151672]]
```

```
personid  
135335    3.09  
999406    2.17  
151672    NaN  
Name: gpaoverall, dtype: float64
```

```
gpaoverall.loc[135335:151672]
```

```
personid  
135335    3.09  
999406    2.17  
151672    NaN  
Name: gpaoverall, dtype: float64
```

## 5. Select values using the `iloc` accessor.

`iloc` differs from `loc` in that it takes a list of row numbers rather than labels. It works similarly to bracket operator slicing. In this step, we pass a one-item list with a value of 0. We then pass a five-item list,

`[0,1,2,3,4]`, to return a Series containing the first five values. We get the same result if we pass `[:5]` to the accessor:

```
gpaoverall.iloc[[0]]
```

```
personid
135335    3.09
Name: gpaoverall, dtype: float64
```

```
gpaoverall.iloc[[0,1,2,3,4]]
```

```
personid
135335    3.09
999406    2.17
151672     NaN
750699    2.53
781297    2.43
Name: gpaoverall, dtype: float64
```

```
gpaoverall.iloc[:5]
```

```
personid
135335    3.09
999406    2.17
151672     NaN
750699    2.53
781297    2.43
Name: gpaoverall, dtype: float64
```

```
gpaoverall.iloc[-5:]
```

```
personid
505861    NaN
368078    NaN
215605    3.22
643085    2.30
713757    NaN
Name: gpaoverall, dtype: float64
```

Each of these ways of accessing pandas Series values—the bracket operator, the `loc` accessor, and the `iloc` accessor—has many use cases, particularly the `loc` accessor.

## How it works...

We used the `[]` bracket operator in *step 3* to perform standard Python-like slicing to create a Series. This operator allows us to easily select data based on position using a list, or a range of values indicated with slice notation. This notation takes the form of `[start:end:step]`, where `1` is assumed for `step` if no value is provided. When a negative number is used for `start`, it represents the number of rows from the end of the original Series.

The `loc` accessor, used in *step 4*, selects data by index labels. Since `personid` is the index for the Series, we can pass a list of one or more `personid` values to the `loc` accessor to get a Series with those labels and associated GPA values. We can also pass a range of labels to the accessor, which will return a Series with GPA values from the index label to the left of the colon and the index label to the right inclusive. So,

`gpaoverall.loc[135335:151672]` returns a Series with GPA values for `personid` between `135335` and `151672`, including those two values.

As shown in *step 5*, the `iloc` accessor takes row positions rather than index labels. We can pass either a list of integers or a range using slicing notation.

## Showing summary statistics for a pandas Series

There are a large number of pandas Series methods for generating summary statistics. We can easily get the mean, median, maximum, or minimum values for a Series with the `mean`, `median`, `max`, and `min` methods, respectively. The incredibly handy `describe` method will return all of these statistics, as well as several others. We can also get the Series value at any percentile using `quantile`. These methods can be used across all values for a Series, or just for selected values. This will be demonstrated in this recipe.

## Getting ready

We will continue working with the overall GPA column from the NLS.

## How to do it...

Let's take a good look at the distribution of the overall GPA for the DataFrame and for the selected rows. To do this, follow these steps:

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

```
import pandas as pd
import numpy as np
nls97 = pd.read_csv("data/nls97f.csv",
low_memory=False)
nls97.set_index("personid", inplace=True)
```

2. Gather some descriptive statistics:

```
gpaoverall = nls97.gpaoverall  
gpaoverall.mean()
```

```
2.8184077281812145
```

```
gpaoverall.describe()
```

```
count    6,004.00  
mean      2.82  
std       0.62  
min       0.10  
25%      2.43  
50%      2.86  
75%      3.26  
max      4.17  
Name: gpaoverall, dtype: float64
```

```
gpaoverall.quantile(np.arange(0.1,1.1,0.1))
```

```
0.10    2.02  
0.20    2.31  
0.30    2.52  
0.40    2.70  
0.50    2.86  
0.60    3.01  
0.70    3.17  
0.80    3.36  
0.90    3.60  
1.00    4.17  
Name: gpaoverall, dtype: float64
```

3. Show descriptives for a subset of the Series:

```
gpaoverall.loc[gpaoverall.between(3,3.5)].head(5)
```

```
personid
135335    3.09
370417    3.41
684388    3.00
984178    3.15
730045    3.44
Name: gpaoverall, dtype: float64
```

```
gpaoverall.loc[gpaoverall.between(3,3.5)].count()
```

```
1679
```

```
gpaoverall.loc[(gpaoverall<2) | (gpaoverall>4)].sample(5)
```

```
personid
382527    1.66
436086    1.86
556245    4.02
563504    1.94
397487    1.84
Name: gpaoverall, dtype: float64
```

```
gpaoverall.loc[gpaoverall>gpaoverall.quantile(0.99)].  
...     agg(['count', 'min', 'max'])
```

```
count      60.00
min       3.98
```

```
max      4.17  
Name: gpaoverall, dtype: float64
```

#### 4. Test for a condition across all values.

Check for GPA values above 4 and if all the values are above or equal to 0. (We generally expect GPA to be between 0 and 4.) Also, count how many values are missing:

```
(gpaoverall>4).any() # any person has GPA greater than 4
```

```
True
```

```
(gpaoverall>=0).all() # all people have GPA greater than 0
```

```
False
```

```
(gpaoverall>=0).sum() # of people with GPA greater than or equal to 0
```

```
6004
```

```
(gpaoverall==0).sum() # of people with GPA equal to 0
```

```
0
```

```
gpaoverall.isnull().sum() # of people with missing value 1
```

```
2980
```

5. Show descriptives for a subset of the Series based on values in a different column.

Show the mean high school GPA for individuals with a wage income in 2020 that's above the 75<sup>th</sup> percentile, as well as for those with a wage income that's below the 25<sup>th</sup> percentile:

```
nls97.loc[nls97.wageincome20 > nls97.wageincome20.quantile(0.75)]
```

```
3.0672837022132797
```

```
nls97.loc[nls97.wageincome20 < nls97.wageincome20.quantile(0.25)]
```

```
2.6852676399026763
```

6. Show descriptives and frequencies for a Series containing categorical data:

```
nls97.maritalstatus.describe()
```

```
count      6675
unique       5
top    Married
freq      3068
Name: maritalstatus, dtype: object
```

```
nls97.maritalstatus.value_counts()
```

```
Married           3068
Never-married    2767
Divorced          669
Separated         148
Widowed           23
Name: maritalstatus, dtype: int64
```

Once we have a Series, we can use a wide variety of pandas tools to calculate descriptive statistics for all or part of that Series.

## How it works...

The Series `describe` method is quite useful as it gives us a good sense of the central tendency and spread of continuous variables. It is also often helpful to see the value at each decile. We obtained this in *step 2* by passing a list of values ranging from 0.1 to 1.0 to the `quantile` method of the Series.

We can use these methods on subsets of a Series. In *step 3*, we obtained the count of GPA values between 3 and 3.5. We can also select values based on their relationship to a summary statistic; for example,

`gpaoverall>gpaoverall.quantile(0.99)` selects GPA values that are greater than the 99<sup>th</sup> percentile value. We then passed the resulting Series to the `agg` method using method chaining, which returned multiple summary statistics (`agg(['count', 'min', 'max'])`).

Sometimes, we need to test whether some condition is true across all the values in a Series. The `any` and `all` methods are useful for this. `any` returns `True` when at least one value in the Series satisfies the condition (such as

`(gpaoverall>4).any()`). `all` returns `True` when all the values in the Series satisfy the condition. When we chain the test condition with `sum` (`((gpaoverall>=0).sum())`), we get a count of all the `True` values since pandas interprets `True` values as 1 when performing numerical operations.

`(gpaoverall>4)` is a shorthand for creating a Boolean Series with the same index as `gpaoverall`. It has a value of `True` when `gpaoverall` is greater than 4, and `False` otherwise:

```
(gpaoverall>4)
```

```
personid
135335    False
999406    False
151672    False
750699    False
781297    False
505861    False
368078    False
215605    False
643085    False
713757    False
Name: gpaoverall, Length: 8984, dtype: bool
```

We sometimes need to generate summary statistics for a Series that has been filtered by another Series. We did this in *step 5* by calculating the mean high school GPA for individuals with a wage income that's above the third quartile, as well as for individuals with a wage income that's below the first quartile.

The `describe` method is most useful with continuous variables, such as `gpaoverall`, but it also provides useful information when used with categorical variables, such as `maritalstatus` (see *step 6*). This returns the

count of non-missing values, the number of different values, the category that occurs most frequently, and the frequency of that category.

However, when working with categorical data, the `value_counts` method is more frequently used. It provides the frequency of each category in the Series.

## There's more...

Working with Series is so fundamental to pandas data cleaning tasks that data analysts quickly find that the tools that were used in this recipe are part of their daily data cleaning workflow. Typically, not much time elapses between the initial data import stage and using Series methods such as `describe`, `mean`, `sum`, `isnull`, `all`, and `any`.

## See also

We are just scratching the surface of aggregating data in this chapter. We'll go through this more thoroughly in *Chapter 9, Fixing Messy Data When Aggregating*.

## Changing Series values

During the data cleaning process, we often need to change the values in a data Series or create a new one. We can change all the values in a Series, or just the values in a subset of our data. Most of the techniques we have been using to get values from a Series can be used to update Series values, though some minor modifications are necessary.

# Getting ready

We will work with the overall high school GPA column from the NLS in this recipe.

## How to do it...

We can change the values in a pandas Series for all rows, as well as for selected rows. We can update a Series with scalars by performing arithmetic operations on other Series, and by using summary statistics. Let's take a look at this:

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

```
import pandas as pd
nls97 = pd.read_csv("data/nls97f.csv",
low_memory=False)
nls97.set_index("personid", inplace=True)
```

2. Edit all the values based on a scalar.

Multiply `gpaoverall` by 100:

```
nls97.gpaoverall.head()
```

```
personid
135335    3.09
999406    2.17
151672      NaN
750699    2.53
781297    2.82
Name: gpaoverall, dtype: float64
```

```
gpaoverall100 = nls97['gpaoverall'] * 100
gpaoverall100.head()
```

```
personid
135335    309.00
999406    217.00
151672      NaN
750699    253.00
781297    243.00
Name: gpaoverall, dtype: float64
```

### 3. Set values using index labels.

Use the `loc` accessor to specify which values to change by index label:

```
nls97.loc[[135335], 'gpaoverall'] = 3
nls97.loc[[999406, 151672, 750699], 'gpaoverall'] = 0
nls97.gpaoverall.head()
```

```
personid
135335    3.00
999406    0.00
151672    0.00
750699    0.00
781297    2.43
Name: gpaoverall, dtype: float64
```

### 4. Set values using an operator on more than one Series.

Use the `+` operator to calculate the number of children, which is the sum of children who live at home and children who do not live at home:

```
nl97['childnum'] = nl97.childathome + nl97.childnotathome  
nl97.childnum.value_counts().sort_index()
```

```
0.00      23  
1.00    1364  
2.00    1729  
3.00    1020  
4.00     420  
5.00     149  
6.00      55  
7.00      21  
8.00       7  
9.00       1  
12.00      2  
Name: childnum, dtype: int64
```

## 5. Use index labels to set values to a summary statistic.

Use the `loc` accessor to select `personid` from `100061` to `100292`:

```
nl97.loc[135335:781297, 'gpaoverall'] = nl97.gpaoverall.mean()  
nl97.gpaoverall.head()
```

```
personid  
135335    2.82  
999406    2.82  
151672    2.82  
750699    2.82  
781297    2.82  
Name: gpaoverall, dtype: float64
```

## 6. Set values using position.

Use the `iloc` accessor to select by position. An integer, or slice notation (`start:end:step`), can be used to the left of the comma to indicate the rows where the values should be changed. An integer is used to the right of the comma to select the column. The `gpaoverall` column is in the 16<sup>th</sup> position (which is 15 since the column index is zero-based):

```
nls97.iloc[0, 15] = 2  
nls97.iloc[1:4, 15] = 1  
nls97.gpaoverall.head()
```

```
personid  
135335    2.00  
999406    1.00  
151672    1.00  
750699    1.00  
781297    2.43  
Name: gpaoverall, dtype: float64
```

## 7. Set the GPA values after filtering.

Change all GPA values over 4 to 4:

```
nls97.gpaoverall.nlargest()
```

```
personid  
312410      4.17  
639701      4.11  
850001      4.10  
279096      4.08  
620216      4.07  
Name: gpaoverall, dtype: float64
```

```
nls97.loc[nls97.gpaoverall>4, 'gpaoverall'] = 4  
nls97.gpaoverall.nlargest()
```

```
personid  
588585    4.00  
864742    4.00  
566248    4.00  
990608    4.00  
919755    4.00  
Name: gpaoverall, dtype: float64
```

The preceding steps showed us how to update Series values with scalars, arithmetic operations, and summary statistics values.

## How it works...

The first thing to observe is that, in *step 2*, pandas vectorizes the multiplication by a scalar. It knows that we want to apply the scalar to all rows. Essentially, `nls97['gpaoverall'] * 100`, creates a temporary Series with all values set to 100, and with the same index as the `gpaoverall` Series. It then multiplies `gpaoverall` by that Series of 100 values. This is known as broadcasting.

We can use a lot of what we learned in the first recipe of this chapter, about how to get values from a Series, to select particular values to update. The main difference here is that we use the `loc` and `iloc` accessors of the DataFrame (`nls97.loc`) rather than the Series (`nls97.gpaoverall.loc`). This is to prevent the dreaded `SettingWithCopyWarning`, which warns us about setting values on a copy of a DataFrame.

`nls97.gpaoverall.loc[[135335]] = 3` triggers that warning, while `nls97.loc[[135335], 'gpaoverall'] = 3` does not.

In *step 4*, we saw how pandas handles numerical operations with two or more Series. Operations such as addition, subtraction, multiplication, and division are very much like the operations we perform on scalars in standard Python, only with vectorization. (This is made possible by pandas index alignment. Remember that Series in the same DataFrame will have the same index.) If you are familiar with NumPy, then you already have a good idea of how this works.

## There's more...

It is useful to notice that `nls97.loc[[135335], 'gpaoverall']` returns a Series, while `nls97.loc[[135335], ['gpaoverall']]` returns a DataFrame:

```
type(nls97.loc[[135335], 'gpaoverall'])
```

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

```
type(nls97.loc[[135335], ['gpaoverall']])
```

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

If the second argument of the `loc` accessor is a string, it will return a Series. If it is a list, even if the list contains only 1 item, it will return a DataFrame.

For any of the operations we discussed in this recipe, it is good to be mindful of how pandas treats missing values. For example, in *step 4*, if either `childathome` or `childnotathome` is missing, then the operation will return `missing`. We'll discuss how to handle situations like this in, *Identifying and Fixing Missing Values* recipe in the next chapter.

## See also

*Chapter 3, Taking the Measure of Your Data*, goes into greater detail on the use of the `loc` and `iloc` accessors, particularly in the *Selecting rows* and *Selecting and organizing columns* recipes.

# Changing Series values conditionally

Changing Series values is often more complicated than the previous recipe suggests. We often need to set Series values based on the values of one or more other Series for that row of data. This is complicated further when we need to set Series values based on values from *other* rows; say, a previous value for an individual, or the mean for a subset. We will deal with these complications in this and the next recipe.

## Getting ready

We will work with land temperature data and the NLS data in this recipe.



### Data note

The land temperature dataset contains the average temperature readings (in Celsius) in 2023 from over 12,000 stations across the world, though the majority of the stations are in the United States. The raw dataset was retrieved from the Global Historical Climatology Network integrated database. It has been made available for public use by the United States National Oceanic and Atmospheric

Administration at

<https://www.ncei.noaa.gov/products/land-based-station/global-historical-climatology-network-monthly>.

## How to do it...

We will use NumPy's `where` and `select` methods to assign Series values based on the values of that Series, the values of other Series, and summary statistics. We'll then use the `lambda` and `apply` functions to construct more complicated criteria for assignment. Let's get started:

1. Import `pandas` and `numpy`, and then load the NLS and land temperature data:

```
import pandas as pd
import numpy as np
nls97 = pd.read_csv("data/nls97f.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
landtemps = pd.read_csv("data/landtemps2023avgs.csv")
```

2. Use NumPy's `where` function to create a categorical Series containing two values.

Let's do a quick check of the distribution of `elevation` values:

```
landtemps.elevation.quantile(np.arange(0.2, 1.1, 0.2))
```

|     |       |
|-----|-------|
| 0.2 | 47.9  |
| 0.4 | 190.5 |
| 0.6 | 395.0 |

```
0.8    1,080.0  
1.0    9,999.0  
Name: elevation, dtype: float64
```

```
landtemps['elevation_group'] = np.where(landtemps.elevation <= 1.0,  
landtemps.elevation_group = landtemps.elevation_group.astype('category'),  
landtemps.groupby(['elevation_group'],  
observed=False)['elevation'].\\  
agg(['count', 'min', 'max']))
```

```
   count      min      max  
elevation_group  
High          2428  1,080  9,999  
Low          9709  -350  1,080
```

### Note

You may have noticed that we passed a value of `False` to the `groupby observed` attribute. This is the default value of `observed` for all pandas versions prior to 2.1.0. `observed=True` is the default for `groupby` in subsequent pandas versions. When `observed` is `True` and there is a column in `groupby` that is categorical, only observed values are shown. This does not affect the summary statistics in the previous step. I show it only to alert you to the upcoming change in the default value. I omit it in the rest of this chapter.



3. Use NumPy's `where` method to create a categorical Series containing three values.

Set values above the 80<sup>th</sup> percentile to 'High', values above the median and up to the 80<sup>th</sup> percentile to 'Medium', and the remaining values to 'Low':

```
landtemps['elevation_group'] = \
    np.where(landtemps.elevation>
        landtemps.elevation.quantile(0.8), 'High',
        np.where(landtemps.elevation>landtemps.elevation.\n            median(), 'Medium', 'Low'))
landtemps.elevation_group = landtemps.elevation_group.astype(str)
landtemps.groupby(['elevation_group'])['elevation'].\\
    agg(['count', 'min', 'max'])
```

| elevation_group | count | min   | max   |
|-----------------|-------|-------|-------|
| High            | 2428  | 1,080 | 9,999 |
| Low             | 6072  | -350  | 271   |
| Medium          | 3637  | 271   | 1,080 |

#### 4. Use NumPy's `select` method to evaluate a list of conditions.

Set up a list of test conditions and another list for the result. We want individuals with a GPA less than 2 and no degree earned to be in one category, individuals with no degree but with a higher GPA to be in a second category, individuals with a degree but a low GPA in a third category, and the remaining individuals in a fourth category:

```
test = [(nls97.gpaoverall<2) &
    (nls97.highestdegree=='0. None'),
    nls97.highestdegree=='0. None',
    nls97.gpaoverall<2]
result = ['1. Low GPA/No Dip', '2. No Diploma',
    '3. Low GPA']
nls97['hsachieve'] = np.select(test, result, '4. Did Okay')
```

```
nls97[['hsachieve', 'gpaoverall', 'highestdegree']].\n    sample(7, random_state=6)
```

| personid | hsachieve         | gpaoverall | highestdegr |
|----------|-------------------|------------|-------------|
| 102951   | 1. Low GPA/No Dip | 1.4        | 0. No       |
| 583984   | 4. Did Okay       | 3.3        | 2. High Sch |
| 116430   | 4. Did Okay       | NaN        | 3. Associat |
| 859586   | 4. Did Okay       | 2.3        | 2. High Sch |
| 288527   | 4. Did Okay       | 2.7        | 4. Bachelor |
| 161698   | 4. Did Okay       | 3.4        | 4. Bachelor |
| 943703   | 2. No Diploma     | NaN        | 0. No       |

```
nls97.hsachieve.value_counts().sort_index()
```

```
hsachieve\n1. Low GPA/No Dip      90\n2. No Diploma           787\n3. Low GPA              464\n4. Did Okay             7643\nName: count, dtype: int64
```

While NumPy's `select` method is very handy for relatively straightforward assignment of values conditionally, it can be difficult to use when the assignment is more complicated. We can use a user-defined function when `select` would be unwieldy.

5. Let's use `apply` and a user-defined function to do the same Series value assignment that we did in the previous step. We create a function, `gethsachieve`, with the logic for assigning values to a new variable, `hsachieve2`. We pass this function to `apply` and indicate `axis=1` to apply the function to all rows.

We will use this same technique in the next step to handle a more complicated assignment, one based on more columns and conditions.

```
def gethsachieve(row):
    if (row.gpaoverall<2 and row.highestdegree=="0. None"):
        hsachieve2 = "1. Low GPA/No Dip"
    elif (row.highestdegree=="0. None"):
        hsachieve2 = "2. No Diploma"
    elif (row.gpaoverall<2):
        hsachieve2 = "3. Low GPA"
    else:
        hsachieve2 = '4. Did Okay'
    return hsachieve2
nls97['hsachieve2'] = nls97.apply(gethsachieve, axis=1)
nls97.groupby(['hsachieve', 'hsachieve2']).size()
```

```
hsachieve      hsachieve2
1. Low GPA/No Dip 1. Low GPA/No Dip      90
2. No Diploma      2. No Diploma      787
3. Low GPA          3. Low GPA       464
4. Did Okay         4. Did Okay     7643
dtype: int64
```

Notice that we get the same values for `hsachieve2` in this step that we got for `hsachieve` in the previous step.

6. Now, let's use `apply` and a user-defined function for a more complicated calculation, one that is based on the values of several variables.

The `getsleepdeprivedreason` function below creates a variable that categorizes survey respondents by the possible reasons why they might get fewer than 6 hours of sleep a night. We base this on NLS survey responses about a respondent's employment status, the

number of children who live with the respondent, wage income, and highest grade completed:

```
def getsleepdeprivedreason(row):
    sleepdeprivedreason = "Unknown"
    if (row.nightlyhrssleep>=6):
        sleepdeprivedreason = "Not Sleep Deprived"
    elif (row.nightlyhrssleep>0):
        if (row.weeksworked20+row.weeksworked21 < 80):
            if (row.childathome>2):
                sleepdeprivedreason = "Child Rearing"
            else:
                sleepdeprivedreason = "Other Reasons"
        else:
            if (row.wageincome20>=62000 or row.highestgradecomp]:
                sleepdeprivedreason = "Work Pressure"
            else:
                sleepdeprivedreason = "Income Pressure"
    else:
        sleepdeprivedreason = "Unknown"
    return sleepdeprivedreason
```

7. Use `apply` to run the function for all rows:

```
nls97['sleepdeprivedreason'] = nls97.apply(getsleepdeprivedreason)
nls97.sleepdeprivedreason = nls97.sleepdeprivedreason
nls97.sleepdeprivedreason.value_counts()
```

| sleepdeprivedreason |      |
|---------------------|------|
| Not Sleep Deprived  | 5595 |
| Unknown             | 2286 |
| Income Pressure     | 453  |
| Work Pressure       | 324  |
| Other Reasons       | 254  |

```
Child Rearing          72  
Name: count, dtype: int64
```

8. We can use a lambda function with `transform` if we want to work with particular columns but we do not need to pass them to a user-defined function. Let's try that by using `lambda` to test several columns in one statement.

The `colenr` columns have enrollment status in February and October of each year for each person. We want to test whether any of the college enrollment columns has a value of `3. 4-year college`. Use `filter` to create a DataFrame of the `colenr` columns. Then, use `transform` to call a lambda function that tests the first character of each `colenr` column. (We can just look at the first character and see whether it has a value of 3.) That is then passed to `any` to evaluate whether any (one or more) of the columns has a 3 in the first character. (We only show values for college enrollment between 2000 and 2004 due to space considerations, but we check all the values for the college enrollment columns between 1997 and 2022.) This can be seen in the following code:

```
nls97.loc[[999406, 750699],  
'colenrfeb00':'colenroct04'].T
```

| personid    | 999406            | 750699            |
|-------------|-------------------|-------------------|
| colenrfeb00 | 1. Not enrolled   | 1. Not enrolled   |
| colenroct00 | 3. 4-year college | 1. Not enrolled   |
| colenrfeb01 | 3. 4-year college | 1. Not enrolled   |
| colenroct01 | 2. 2-year college | 1. Not enrolled   |
| colenrfeb02 | 1. Not enrolled   | 2. 2-year college |
| colenroct02 | 3. 4-year college | 1. Not enrolled   |
| colenrfeb03 | 3. 4-year college | 1. Not enrolled   |

```
colenroct03    3. 4-year college      1. Not enrolled
colenrfeb04    3. 4-year college      1. Not enrolled
colenroct04    3. 4-year college      1. Not enrolled
```

```
nls97['baenrollment'] = nls97.filter(like="colenr").\
...     transform(lambda x: x.str[0:1]=='3').\
...     any(axis=1)
nls97.loc[[999406, 750699], ['baenrollment']].T
```

```
personid      999406  750699
baenrollment   True    False
```

```
nls97.baenrollment.value_counts()
```

```
baenrollment
False      4987
True       3997
Name: count, dtype: int64
```

The preceding steps demonstrate several techniques we can use to set the values for a Series conditionally.

## How it works...

If you have used `if-then-else` statements in SQL or Microsoft Excel, then NumPy's `where` should be familiar to you. It follows the form of `where` (test condition, clause if `True`, clause if `False`). In *step 2*, we tested whether the value of elevation for each row is greater than the value at the 80<sup>th</sup> percentile. If `True`, we returned `'High'`. We returned `'Low'` otherwise. This is a basic `if-then-else` construction.

Sometimes, we need to nest a test within a test. We did this in *step 3* to create three elevation groups; high, medium, and low. Instead of a simple statement in the `False` section (after the second comma), we used another `where` statement. This changes it from an `else` clause to an `else if` clause. It takes the form of `where(test condition, statement if True, where(test condition, statement if True, statement if False))`.

It is possible to add many more nested `where` statements, though that is not advisable. When we need to evaluate a slightly more complicated test, NumPy's `select` method comes in handy. In *step 4*, we passed a list of tests, as well as a list of results of that test, to `select`. We also provided a default value of `4. Did Okay` for any case where none of the tests was `True`. When multiple tests are `True`, the first one that is `True` is used.

Once the logic becomes even more complicated, we can use `apply`. The DataFrame `apply` method can be used to send each row of a DataFrame to a function by specifying `axis=1`. *step 5* demonstrates how to reproduce the same logic as with *step 4* using `apply` and a user-defined function.

In *steps 6* and *7*, we created a Series that categorizes reasons for being sleep deprived based on weeks worked, the number of children living with the respondent, wage income, and highest grade completed. If the respondent did not work most of 2020 and 2021, and if more than two children lived with them, `sleepdeprivedreason` is set to “Child Rearing.” If the respondent did not work most of 2020 and 2021 and two or fewer children lived with them, `sleepdeprivedreason` is set to “Other Reasons.” If they worked most of 2020 and 2021, then `sleepdeprivedreason` is “Work Pressure” if they had a high salary or completed 4 years of college, and is “Income Pressure” otherwise. Of course, these categories are somewhat

contrived, but they do illustrate how to use a function to create a Series based on complicated relationships among other Series.

In *step 8*, we used `transform` to call a lambda function that tests whether the first character of each college enrollment value is 3. But first, we used the `filter` DataFrame method to select all the college enrollment columns. We could have paired the `lambda` function with `apply` to achieve the same result, but `transform` is typically more efficient.

You may have noticed that we changed the data type of the new Series we created in *steps 2* and *3* to `category`. The new Series was an `object` data type initially. We reduced memory usage by changing the type to `category`.

We used another incredibly useful method in *step 2*, somewhat incidentally.

`landtemps.groupby(['elevation_group'])` creates a DataFrame `groupby` object that we pass to an aggregate (`agg`) function. This gives us a count, min, and max for each `elevation_group`, allowing us to confirm that our group classification works as expected.

## There's more...

It has been a long time since I have had a data cleaning project that did not involve a NumPy `where` or `select` statement, nor a `lambda` or `apply` statement. At some point, we need to create or update a Series based on values from one or more other Series. It is a good idea to get comfortable with these techniques.

Whenever there is a built-in pandas function that does what we need, it is better to use that than `apply`. The great advantage of `apply` is that it is quite generic and flexible, but that is also why it is more resource-intensive than

the optimized functions. However, it is a great tool when we want to create a Series based on complicated relationships between existing Series.

Another way to perform *steps 6* and *7* is to add a lambda function to `apply`. This produces the same results:

```
def getsleepdeprivedreason(childathome, nightlyhrssleep, wagein
...     sleepdeprivedreason = "Unknown"
...     if (nightlyhrssleep>=6):
...         sleepdeprivedreason = "Not Sleep Deprived"
...     elif (nightlyhrssleep>0):
...         if (weeksworked16+weeksworked17 < 80):
...             if (childathome>2):
...                 sleepdeprivedreason = "Child Rearing"
...             else:
...                 sleepdeprivedreason = "Other Reasons"
...         else:
...             if (wageincome>=62000 or highestgradecompleted>=16):
...                 sleepdeprivedreason = "Work Pressure"
...             else:
...                 sleepdeprivedreason = "Income Pressure"
...     else:
...         sleepdeprivedreason = "Unknown"
...     return sleepdeprivedreason
...
nls97['sleepdeprivedreason'] = nls97.apply(lambda x: getsleepde
```

One advantage of this approach is that it makes it more clear which Series contribute to the calculation.

## See also

We'll go over DataFrame `groupby` objects in detail in *Chapter 9, Fixing Messy Data When Aggregating*. We examined various techniques we can

use to select columns from a DataFrame, including `filter`, in *Chapter 3, Taking the Measure of Your Data*.

## Evaluating and cleaning string Series data

There are many string cleaning methods in Python and pandas. This is a good thing. Given the great variety of data stored in strings, it is important to have a wide range of tools to call upon when performing string evaluation and manipulation: when selecting fragments of a string by position, when checking whether a string contains a pattern, when splitting a string, when testing a string's length, when joining two or more strings, when changing the case of a string, and so on. We'll explore some of the methods that are used most frequently for string evaluation and cleaning in this recipe.

### Getting ready

We will work with the NLS data in this recipe. (The NLS data was actually a little too clean for this recipe. To illustrate working with strings with trailing spaces, I added trailing spaces to the `maritalstatus` column values.)

### How to do it...

In this recipe, we will perform some common string evaluation and cleaning tasks. We'll use `contains`, `endswith`, and `findall` to search for patterns, trailing blanks, and more complicated patterns, respectively.

We will also create a function for processing string values before assigning values to a new Series and then use `replace` for simpler processing. Let's get started:

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

```
import pandas as pd
import numpy as np
nls97 = pd.read_csv("data/nls97ca.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
```

2. Test whether a pattern exists in a string.

Use `contains` to examine the `govprovidejobs` (government should provide jobs) responses for the “Definitely not” and “Probably not” values. In the `where` call, handle missing values first to make sure that they do not end up in the first `else` clause (the section after the second comma):

```
nls97.govprovidejobs.value_counts()
```

```
2. Probably           617
3. Probably not      462
1. Definitely         454
4. Definitely not     300
Name: govprovidejobs, dtype: int64
```

```
nls97['govprovidejobsdefprob'] = \
    np.where(nls97.govprovidejobs.isnull(),
             np.nan,
             np.where(nls97.govprovidejobs.str.\
```

```
    contains("not"), "No", "Yes"))
pd.crosstab(nls97.govprovidejobs, nls97.govprovidejobsdefpr
```

| govprovidejobsdefprob | No  | Yes |
|-----------------------|-----|-----|
| govprovidejobs        |     |     |
| 1. Definitely         | 0   | 454 |
| 2. Probably           | 0   | 617 |
| 3. Probably not       | 462 | 0   |
| 4. Definitely not     | 300 | 0   |

### 3. Handle leading or trailing spaces in a string.

Create an ever-married Series. First, examine the values of `maritalstatus`. Notice that there are two stray values indicating never-married. They are “Never-married” with an extra space at the end, unlike the other values of “Never-married” with no trailing spaces. Use `startswith` and `endswith` to test for a leading or trailing space, respectively. Use `strip` to remove the trailing space before testing for ever-married. `strip` removes leading and trailing spaces (`lstrip` removes leading spaces, while `rstrip` removes trailing spaces, so `rstrip` would have also worked in this example):

```
nls97.maritalstatus.value_counts()
```

|                           |      |
|---------------------------|------|
| Married                   | 3066 |
| Never-married             | 2764 |
| Divorced                  | 663  |
| Separated                 | 154  |
| Widowed                   | 23   |
| Never-married             | 2    |
| Name: count, dtype: int64 |      |

```
nl97.maritalstatus.str.startswith(' ') .any()
```

```
False
```

```
nl97.maritalstatus.str.endswith(' ') .any()
```

```
True
```

```
nl97['evermarried'] = \
    np.where(nl97.maritalstatus.isnull(), np.nan,
            np.where(nl97.maritalstatus.str.\
                strip() == "Never-married", "No", "Yes"))
pd.crosstab(nl97.maritalstatus, nl97.evermarried)
```

| evermarried   | No   | Yes  |
|---------------|------|------|
| maritalstatus |      |      |
| Divorced      | 0    | 663  |
| Married       | 0    | 3066 |
| Never-married | 2764 | 0    |
| Never-married | 2    | 0    |
| Separated     | 0    | 154  |
| Widowed       | 0    | 23   |

#### 4. Use `isin` to compare a string value to a list of values:

```
nl97['receivedba'] = \
    np.where(nl97.highestdegree.isnull(), np.nan,
            np.where(nl97.highestdegree.str[0:1].\
                isin(['4', '5', '6', '7']), "Yes", "No"))
pd.crosstab(nl97.highestdegree, nl97.receivedba)
```

| receivedba    | No | Yes |
|---------------|----|-----|
| highestdegree |    |     |

|                 |      |      |
|-----------------|------|------|
| 0. None         | 953  | 0    |
| 1. GED          | 1146 | 0    |
| 2. High School  | 3667 | 0    |
| 3. Associates   | 737  | 0    |
| 4. Bachelors    | 0    | 1673 |
| 5. Masters      | 0    | 603  |
| 6. PhD          | 0    | 54   |
| 7. Professional | 0    | 120  |

We occasionally need to identify the location of a particular character in a string. This is sometimes because we need to get text before or after that point, or treat that text differently. Let's try this with the highest degree attained column that we have already worked with. We will create a new column that does not have the number prefix. For example, 2. *High School* will become *High School*.

5. Use `find` to get the location of the period in `highestdegree` values and retrieve the text after that.

Before we do that, we assign 99. *Unknown* to missing values. This is not necessary but it helps us be clear about how we are handling all values, including the missing ones. It also adds a useful complication. After we do that, the leading numbers can be 1 or 2 digits.

Next, we create a lambda function, `onlytext`, that we will use to identify the location of the text we want and then use it to pull that text. We then use the `transform` method of the `highestdegree` Series to call the `onlytext` function:

```
nls97.fillna({"highestdegree": "99. Unknown"},  
    inplace=True)  
onlytext = lambda x: x[x.find(".") + 2:]  
highestdegreenonum = nls97.highestdegree.\  
    astype(str).transform(onlytext)
```

```
highestdegreenonum.value_counts(dropna=False).\\
sort_index()
```

```
highestdegree
Associates      737
Bachelors       1673
GED              1146
High School     3667
Masters          603
None             953
PhD              54
Professional    120
Unknown          31
Name: count, dtype: int64
```

You probably noticed that there is a space between the period and the start of the text we want. To account for this, the `onlytext` function pulls text starting two spaces over from the period.

### Note

We did not need to name a lambda function to achieve the results we wanted. We could have just entered a lambda function in the `transform` method. However, since we have a number of columns in the NLS data that have a similar prefix, it is good to have a function that we can reuse with another column.

We sometimes need to find all occurrences in a string of a certain value or a certain type of value, say a number. The pandas Series function, `findall`, can be used to return one or more occurrences of a

value in a string. A list is returned of the string fragments that satisfy the given criteria. Let's do a straightforward example before moving on to a more complicated one.

Use `findall` to count the number of times `r` appears for each `maritalstatus` value for the first few rows of data. First, show the values for `maritalstatus`, then show the list that is returned from `findall` for each value:

```
nls97.maritalstatus.head()
```

```
personid
100061      Married
100139      Married
100284      Never-married
100292      NaN
100583      Married
Name: maritalstatus, dtype: object
```

```
nls97.maritalstatus.head().str.findall("r")
```

```
personid
100061      [r, r]
100139      [r, r]
100284      [r, r, r]
100292      NaN
100583      [r, r]
Name: maritalstatus, dtype: object
```

6. Let's also show a count of the number of times that `r` appears.

Use `concat` to show the `maritalstatus` value, the list from `findall`, and the length of the list all on the same line:

```
pd.concat([nls97.maritalstatus.head(),
           nls97.maritalstatus.head().str.findall("r"),
           nls97.maritalstatus.head().str.findall("r").\
               str.len()],
          axis=1)
```

|          | maritalstatus | maritalstatus | maritalstatus |
|----------|---------------|---------------|---------------|
| personid |               |               |               |
| 100061   | Married       | [r, r]        |               |
| 100139   | Married       | [r, r]        |               |
| 100284   | Never-married | [r, r, r]     |               |
| 100292   | NaN           |               | NaN           |
| 100583   | Married       | [r, r]        |               |

We can also use `findall` to return types of values. For example, we can use a regular expression to return a list with all numbers in a string. We do that in the next couple of steps.

7. Use `findall` to create a list of all numbers in the `weeklyhrstv` (hours spent each week watching television) string. The "`\d+`" regular expression that's passed to `findall` indicates that we just want numbers:

```
pd.concat([nls97.weeklyhrstv.head(), \
...     nls97.weeklyhrstv.str.findall("\d+").head()], a
```

|          | weeklyhrstv           | weeklyhrstv |
|----------|-----------------------|-------------|
| personid |                       |             |
| 100061   | 11 to 20 hours a week | [11, 20]    |
| 100139   | 3 to 10 hours a week  | [3, 10]     |
| 100284   | 11 to 20 hours a week | [11, 20]    |

|        |                      |         |
|--------|----------------------|---------|
| 100292 | NaN                  | NaN     |
| 100583 | 3 to 10 hours a week | [3, 10] |

8. Use the list created by `findall` to create a numerical Series from the `weeklyhrstv` text.

Let's define a function that retrieves the last element in the list created by `findall` for each value of `weeklyhrstv`. The `getnum` function also adjusts that number so that it's closer to the midpoint of the two numbers, where there is more than one number. We then use `apply` to call this function, passing it the list created by `findall` for each value. `crosstab` shows that the new `weeklyhrstvnum` column does what we want it to do:

```
def getnum(numlist):
    ...     highval = 0
    ...     if (type(numlist) is list):
    ...         lastval = int(numlist[-1])
    ...         if (numlist[0]=='40'):
    ...             highval = 45
    ...         elif (lastval==2):
    ...             highval = 1
    ...         else:
    ...             highval = lastval - 5
    ...     else:
    ...         highval = np.nan
    ...     return highval
    ...
nls97['weeklyhrstvnum'] = nls97.weeklyhrstv.str.\
    ... findall("\d+").apply(getnum)
nls97[['weeklyhrstvnum', 'weeklyhrstv']].head(7)
```

|          | weeklyhrstvnum | weeklyhrstv           |
|----------|----------------|-----------------------|
| personid |                |                       |
| 100061   | 15             | 11 to 20 hours a week |
| 100139   | 5              | 3 to 10 hours a week  |

|        |     |                            |
|--------|-----|----------------------------|
| 100284 | 15  | 11 to 20 hours a week      |
| 100292 | NaN | NaN                        |
| 100583 | 5   | 3 to 10 hours a week       |
| 100833 | 5   | 3 to 10 hours a week       |
| 100931 | 1   | Less than 2 hours per week |

```
pd.crosstab(nls97.weeklyhrstv, nls97.weeklyhrstvnum)
```

| weeklyhrstvnum        | 1    | 5    | 15   | 25  |
|-----------------------|------|------|------|-----|
| weeklyhrstv           |      |      |      |     |
| 11 to 20 hours a week | 0    | 0    | 1145 | 0   |
| 21 to 30 hours a week | 0    | 0    | 0    | 299 |
| 3 to 10 hours a week  | 0    | 3625 | 0    | 0   |
| 31 to 40 hours a week | 0    | 0    | 0    | 0   |
| Less than 2 hrs.      | 1350 | 0    | 0    | 0   |
| More than 40 hrs.     | 0    | 0    | 0    | 0   |

## 9. Replace the values in a Series with alternative values.

The `weeklyhrscomputer` (hours spent each week on a computer) Series does not sort nicely with its current values. We can fix this by replacing the values with letters that indicate order. We'll start by creating a list containing the old values and another list containing the new values that we want. We then use the Series `replace` method to replace the old values with the new values. Whenever `replace` finds a value from the old values list, it replaces it with a value from the same list position in the new list:

```
comphrsold = ['Less than 1 hour a week',
              '1 to 3 hours a week', '4 to 6 hours a week',
              '7 to 9 hours a week', '10 hours or more a week']
comphrsnew = ['A. Less than 1 hour a week',
```

```
'B. 1 to 3 hours a week', 'C. 4 to 6 hours a week',
'D. 7 to 9 hours a week', 'E. 10 hours or more a week']
nls97.weeklyhrscomputer.value_counts().sort_index()
```

```
1 to 3 hours a week      733
10 hours or more a week 3669
4 to 6 hours a week     726
7 to 9 hours a week     368
Less than 1 hour a week 296
Name: weeklyhrscomputer, dtype: int64
```

```
nls97.weeklyhrscomputer.replace(comphrsold, comphrsnew, inplace=True)
nls97.weeklyhrscomputer.value_counts().sort_index()
```

```
A. Less than 1 hour a week      296
B. 1 to 3 hours a week        733
C. 4 to 6 hours a week        726
D. 7 to 9 hours a week        368
E. 10 hours or more a week    3669
Name: weeklyhrscomputer, dtype: int64
```

The steps in this recipe demonstrate some of the common string evaluation and manipulation tasks we can perform in pandas.

## How it works...

We frequently need to examine a string to see whether there is a pattern. We can use the string `contains` method to do this. If we know exactly where the expected pattern will be, we can use standard slice notation, `[start:stop:step]`, to select text from start through stop-1. (The default value for `step` is 1.) For example, in *step 4*, we got the first character from

`highestdegree` with `nls97.highestdegree.str[0:1]`. We then used `isin` to test whether the first string appears in a list of values. (`isin` works for both character and numeric data.)

Sometimes, we need to pull multiple values from a string that satisfy a condition. `findall` is helpful in those situations as it returns a list of all values satisfying the condition. It can be paired with a regular expression when we are looking for something more general than a literal. In *steps 8* and *9*, we were looking for any number.

## There's more...

It is important to be deliberate while handling missing values when creating a Series based on values for another Series. Missing values may satisfy the `else` condition in a `where` call when that is not our intention. In *steps 2, 3, and 4*, we made sure that we handled the missing values appropriately by testing for them at the beginning of the `where` call.

We also need to be careful about the casing of the letters when making string comparisons. For example, `Probably` and `probably` are not equal. One way to get around this is to use the `upper` or `lower` methods when doing comparisons when a potential difference in case is not meaningful.

`upper("Probably") == upper("PROBABLY")` is actually `True`.

## Working with dates

Working with dates is rarely straightforward. Data analysts need to successfully parse date values, identify invalid or out-of-range dates, impute dates when they're missing, and calculate time intervals. There are surprising hurdles at each of these steps, but we are halfway there once

we've parsed the date value and have a datetime value in pandas. We will start by parsing date values in this recipe before working our way through the other challenges.

## Getting ready

We will work with the NLS and COVID-19 case daily data in this recipe. The COVID-19 daily data contains one row for each reporting day for each country. (The NLS data was actually a little too clean for this purpose. To illustrate working with missing date values, I set one of the values for birth month to missing.)



### Data note

Our World in Data provides COVID-19 public use data at <https://ourworldindata.org/covid-cases>. The data used in this recipe was downloaded on March 3, 2024.

## How to do it...

In this recipe, we will convert numeric data into datetime data, first by confirming that the data has valid date values and then by using `fillna` to replace missing dates. We will then calculate some date intervals; that is, the age of respondents for the NLS data and the days since the first COVID-19 case for the COVID-19 daily data. Let's get started:

1. Import `pandas` and the `relativedelta` module from `dateutils`, and then load the NLS and COVID-19 case daily data:

```
import pandas as pd
from dateutil.relativedelta import relativedelta
covidcases = pd.read_csv("data/covidcases.csv")
nls97 = pd.read_csv("data/nls97c.csv")
nls97.set_index("personid", inplace=True)
```

## 2. Show the birth month and year values.

Notice that there is one missing value for birth month. Other than that, the data that we will use to create the `birthdate` Series looks pretty clean:

```
nls97[['birthmonth', 'birthyear']].isnull().sum()
```

```
birthmonth      1
birthyear       0
dtype: int64
```

```
nls97.birthmonth.value_counts(dropna=False).\\
    sort_index()
```

```
birthmonth
1      815
2      693
3      760
4      659
5      689
6      720
7      762
8      782
9      839
10     765
11     763
12     736
```

```
NaN      1  
Name: count, dtype: int64
```

```
nls97.birthyear.value_counts().sort_index()
```

```
1980      1691  
1981      1874  
1982      1841  
1983      1807  
1984      1771  
Name: birthyear, dtype: int64
```

3. Use the `fillna` method to set a value for the missing birth month.

Pass the average of `birthmonth`, rounded to the nearest integer, to `fillna`. This will replace the missing value for `birthmonth` with the mean of `birthmonth`. Notice that one more person now has a value of 6 for `birthmonth`:

```
nls97.fillna({"birthmonth":\n    int(nls97.birthmonth.mean())}, inplace=True)\n    nls97.birthmonth.value_counts(dropna=False).\\n        sort_index()
```

```
birthmonth\n1      815\n2      693\n3      760\n4      659\n5      689\n6      721\n7      762\n8      782\n9      839
```

```
10    765  
11    763  
12    736  
Name: count, dtype: int64
```

#### 4. Use `month` and year `integers` to create a datetime column.

We can pass a dictionary to the pandas `to_datetime` function. The dictionary needs to contain a key for year, month, and day. Notice that there are no missing values for `birthmonth`, `birthyear`, and `birthdate`:

```
nls97['birthdate'] = pd.to_datetime(dict(year=nls97.birthyear,  
nls97[['birthmonth','birthyear','birthdate']].head()
```

|          | birthmonth | birthyear | birthdate  |
|----------|------------|-----------|------------|
| personid |            |           |            |
| 100061   | 5          | 1980      | 1980-05-15 |
| 100139   | 9          | 1983      | 1983-09-15 |
| 100284   | 11         | 1984      | 1984-11-15 |
| 100292   | 4          | 1982      | 1982-04-15 |
| 100583   | 6          | 1980      | 1980-06-15 |

```
nls97[['birthmonth','birthyear','birthdate']].isnull().sum()
```

```
birthmonth      0  
birthyear      0  
birthdate      0  
dtype: int64
```

## 5. Calculate age using a datetime column.

First, define a function that will calculate age when given a start date and an end date. Notice that we create a **Timestamp** object, `rundate`, and assign it a value of `2024-03-01` to use for the end date of our age calculation:

```
def calcage(startdate, enddate):
    ...     age = enddate.year - startdate.year
    ...     if (enddate.month<startdate.month or (enddate.month-
    ...         age = age -1
    ...     return age
    ...
rundate = pd.to_datetime('2024-03-01')
nls97["age"] = nls97.apply(lambda x: calcage(x.birthdate,
nls97.loc[100061:100583, ['age', 'birthdate']]
```

| personid | age | birthdate  |
|----------|-----|------------|
| 100061   | 43  | 1980-05-15 |
| 100139   | 40  | 1983-09-15 |
| 100284   | 39  | 1984-11-15 |
| 100292   | 41  | 1982-04-15 |
| 100583   | 43  | 1980-06-15 |

## 6. We can use the `relativedelta` module instead for the age calculation.

We just need to do the following:

```
nls97["age2"] = nls97.\
    apply(lambda x: relativedelta(rundate,
        x.birthdate).years,
    axis=1)
```

## 7. We should confirm that we get the same values as in *step 5*:

```
(nls97['age']!=nls97['age2']).sum()
```

```
0
```

```
nls97.groupby(['age', 'age2']).size()
```

```
age   age2
39    39      1463
40    40      1795
41    41      1868
42    42      1874
43    43      1690
44    44       294
dtype: int64
```

## 8. Convert a string column into a datetime column.

The `casedate` column is an `object` data type, not a `datetime` data type:

```
covidcases.iloc[:, 0:6].dtypes
```

```
iso_code          object
continent        object
location         object
casedate         object
total_cases      float64
new_cases        float64
dtype: object
```

```
covidcases.iloc[:, 0:6].sample(2, random_state=1).T
```

```
      628          26980
iso_code        AND          PRT
casedate  2020-03-15  2022-12-04
continent       Europe       Europe
location        Andorra     Portugal
total_cases        2    5,541,211
new_cases         1      3,963
```

```
covidcases['casedate'] = pd.to_datetime(covidcases.casedat
covidcases.iloc[:, 0:6].dtypes
```

```
iso_code          object
continent         object
location          object
casedate   datetime64[ns]
total_cases       float64
new_cases         float64
dtype: object
```

## 9. Show descriptive statistics for the datetime column:

```
covidcases.casedate.nunique()
```

```
214
```

```
covidcases.casedate.describe()
```

```
count          36501
mean  2021-12-16 05:41:07.954302720
min   2020-01-05 00:00:00
25%   2021-01-31 00:00:00
50%   2021-12-12 00:00:00
75%   2022-10-09 00:00:00
```

```
max           2024-02-04 00:00:00  
Name: casedate, dtype: object
```

## 10. Create a `timedelta` object to capture a date interval.

For each day, calculate the number of days since the first case was reported for each country. First, create a DataFrame that shows the first day of new cases for each country and then merge it with the full COVID-19 cases data. Then, for each day, calculate the number of days from `firstcasedate` to `casedate`:

```
firstcase = covidcases.loc[covidcases.new_cases>0, ['location',  
...     sort_values(['location', 'casedate']).\  
...     drop_duplicates(['location'], keep='first').\  
...     rename(columns={'casedate':'firstcasedate'})  
covidcases = pd.merge(covidcases, firstcase, left_on=['location'], right_on='location')  
covidcases['dayssincefirstcase'] = covidcases.casedate - covidcases.firstcasedate  
covidcases.dayssincefirstcase.describe()
```

```
count                36501  
mean    637 days 01:36:55.862579112  
std      378 days 15:34:06.667833980  
min            0 days 00:00:00  
25%        315 days 00:00:00  
50%        623 days 00:00:00  
75%        931 days 00:00:00  
max       1491 days 00:00:00  
Name: dayssincefirstcase, dtype: object
```

This recipe showed how it's possible to parse date values and create a datetime Series, as well as how to calculate time intervals.

## How it works...

The first task when working with dates in pandas is converting them properly into a pandas datetime Series. We tackled a couple of the most common issues in *steps 3, 4, and 8*: missing values, date conversion from integer parts, and date conversion from strings. `birthmonth` and `birthyear` are integers in the NLS data. We confirmed that those values are valid values for date months and date years. If, for example, there were month values of 0 or 20, the conversion to pandas datetime would fail.

Missing values for `birthmonth` or `birthyear` will result in a missing `birthdate`. We used `fillna` for the missing value for `birthmonth`, assigning it to the mean value of `birthmonth`. In *step 5*, we calculated an age for each person as of March 1, 2024, using the new `birthdate` column. The `calcage` function that we created adjusts for individuals whose birth dates come later in the year than March 1.

Data analysts often receive data files containing date values as strings. The `to_datetime` function is the analyst's key ally when this happens. It is often smart enough to figure out the format of the string date data without us having to specify a format explicitly. However, in *step 8*, we told `to_datetime` to use the `%Y-%m-%d` format with our data.

*Step 9* told us that there were 214 unique days where COVID-19 cases were reported. The first reported day was January 5, 2020, and the last was February 4, 2024.

The first two statements in *step 10* involved techniques (sorting and dropping duplicates) that we will not explore in detail until *Chapter 9, Fixing Messy Data When Aggregating*, and *Chapter 10, Addressing Data Issues When Combining DataFrames*. All you need to understand here is the objective: creating a DataFrame with one row per `location` (country), and with the date of the first reported COVID-19 case. We did this by only

selecting rows from the full data where `new_cases` is greater than 0, before sorting that by `location` and `casedate` and keeping the first row for each `location`. We then changed the name of `casedate` to `firstcasedate` before merging the new `firstcase` DataFrame with the COVID-19 daily cases data.

Since both `casedate` and `firstcasedate` are datetime columns, subtracting the latter from the former will result in a `timedelta` value. This gives us a Series that is the number of days after the first day of `new_cases` for each country for each reporting day. The greatest duration (`dayssincefirstcase`) between a reported case date (`casedate`) and date of first case (`firstcasedate`) is 1491 days, or just over 4 years. This interval calculation is useful if we want to track trends by how long the virus has been obviously present in a country, rather than by date.

## See also

Instead of using `sort_values` and `drop_duplicates` in *step 10*, we could have used `groupby` to achieve similar results. We'll explore `groupby` a fair bit in *Chapter 9, Fixing Messy Data When Aggregating*. We also did a merge in *step 10*. *Chapter 10, Addressing Data Issues When Combining DataFrames*, will be devoted to this topic.

## Using OpenAI for Series operations

Many of the Series operations demonstrated in the previous recipes in this chapter can be assisted by AI tools, including by PandasAI, with the large language model from OpenAI. In this recipe, we examine how to use

PandasAI to query Series values, create new Series, set Series values conditionally, and do some rudimentary reshaping of DataFrames.

## Getting ready

We will work with the NLS and COVID-19 case daily data again in this recipe. We will also work with PandasAI, which can be installed with `pip install pandasai`. You also need to get a token from [openai.com](https://openai.com) to send a request to the OpenAI API.

## How to do it...

The following steps create a PandasAI `SmartDataframe` object, and then use the `chat` method of that object to submit natural language instructions for a range of Series operations:

1. We first need to import the `OpenAI` and `SmartDataframe` modules from PandasAI. We also have to instantiate an `llm` object:

```
import pandas as pd
from pandasai.llm.openai import OpenAI
from pandasai import SmartDataframe
llm = OpenAI(api_token="Your API Token")
```

2. We load the NLS and COVID-19 data and create a `SmartDataFrame` object. We pass the `llm` object as well as a pandas DataFrame:

```
covidcases = pd.read_csv("data/covidcases.csv")
nls97 = pd.read_csv("data/nls97f.csv")
nls97.set_index("personid", inplace=True)
nls97sdf = SmartDataframe(nls97, config={"llm": llm})
```

3. Now we are ready to generate summary statistics on Series from our `SmartDataframe`. We can ask for the average for a single Series, or for multiple Series:

```
nls97sdf.chat("Show average of gpaoverall")
```

```
2.8184077281812128
```

```
nls97sdf.chat("Show average for each weeks worked col")
```

|               | Average Weeks Worked |
|---------------|----------------------|
| weeksworked00 | 26.42                |
| weeksworked01 | 29.78                |
| weeksworked02 | 31.83                |
| weeksworked03 | 33.51                |
| weeksworked04 | 35.10                |
| weeksworked05 | 37.34                |
| weeksworked06 | 38.44                |
| weeksworked07 | 39.29                |
| weeksworked08 | 39.33                |
| weeksworked09 | 37.51                |
| weeksworked10 | 37.12                |
| weeksworked11 | 38.06                |
| weeksworked12 | 38.15                |
| weeksworked13 | 38.79                |
| weeksworked14 | 38.73                |
| weeksworked15 | 39.67                |
| weeksworked16 | 40.19                |
| weeksworked17 | 40.37                |
| weeksworked18 | 40.01                |
| weeksworked19 | 41.22                |
| weeksworked20 | 38.35                |

```
weeksworked21 36.17  
weeksworked22 11.43
```

4. We can also summarize Series values by another Series, usually one that is categorical:

```
nls97sdf.chat("Show satmath average by gender")
```

```
Female    Male  
0   486.65  516.88
```

5. We can also create a new Series with the `chat` method of `SmartDataframe`. We do not need to use the actual column names. For example, PandasAI will figure out that we want the `childathome` Series when we write *child at home*:

```
nls97sdf = nls97sdf.chat("Set childnum to child at ho  
nls97sdf[['childnum', 'childathome', 'childnotathome']]  
sample(5, random_state=1)
```

```
      childnum childathome childnotathome  
personid  
211230      2.00      2.00      0.00  
990746      3.00      3.00      0.00  
308169      3.00      1.00      2.00  
798458      NaN       NaN       NaN  
312009      NaN       NaN       NaN
```

6. We can use the `chat` method to create Series values conditionally:

```
nls97sdf = nls97sdf.chat("evermarried is 'No' when ma
```

```
nls97sdf.groupby(['evermarried','maritalstatus']).size
```

```
evermarried  maritalstatus
No           Never-married    2767
Yes          Divorced        669
              Married         3068
              Separated       148
              Widowed         23
dtype: int64
```

7. PandasAI is quite flexible regarding the language you might use here. For example, the following provides the same results as in *step 6*:

```
nls97sdf = nls97sdf.chat("if maritalstatus is 'Never-
nls97sdf.groupby(['evermarried2','maritalstatus']).size
```

```
evermarried2  maritalstatus
No           Never-married    2767
Yes          Divorced        669
              Married         3068
              Separated       148
              Widowed         23
dtype: int64
```

8. We can do calculations across a number of similarly named columns:

```
nls97sdf = nls97sdf.chat("set weeksworkedavg to the average of all weeksworked00-weeksworked22 columns and assign that to a new column called weeksworkedavavg.")
```

This will calculate the average of all `weeksworked00-weeksworked22` columns and assign that to a new column called `weeksworkedavavg`.

9. We can easily impute values where they are missing based on summary statistics:

```
nls97sdf.gpaenglish.describe()
```

```
count    5,798
mean      273
std       74
min       0
25%     227
50%     284
75%     323
max     418
Name: gpaenglish, dtype: float64
```

```
nls97sdf = nls97sdf.chat("set missing gpaenglish to t
nls97sdf.gpaenglish.describe()
```

```
count    8,984
mean      273
std       59
min       0
25%     264
50%     273
75%     298
max     418
Name: gpaenglish, dtype: float64
```

10. We can also use PandasAI to do some reshaping, similar to what we did in the previous recipe. Recall that we worked with the COVID-19 cases data and wanted the first row of data for each country. Let's do a simplified version of that the traditional way first:

```
firstcase = covidcases.\
    sort_values(['location', 'casedate']).\
    drop_duplicates(['location'], keep='first')
```

```
firstcase.set_index('location', inplace=True)
firstcase.shape
```

```
(231, 67)
```

```
firstcase[['iso_code', 'continent', 'casedate',
           'total_cases', 'new_cases']].head(2).T
```

|             |             |            |
|-------------|-------------|------------|
| location    | Afghanistan | Albania    |
| iso_code    | AFG         | ALB        |
| continent   | Asia        | Europe     |
| casedate    | 2020-03-01  | 2020-03-15 |
| total_cases | 1.00        | 33.00      |
| new_cases   | 1.00        | 33.00      |

11. We can get the same results by creating a `SmartDataframe` and using the `chat` method. The natural language I use here is remarkably straightforward, *Show first casedate and location and other values for each country*:

```
covidcasessdf = SmartDataframe(covidcases, config={"l
firstcasesdf = covidcasessdf.chat("Show first casedat
firstcasesdf.shape
```

```
(231, 7)
```

```
firstcasesdf[['location', 'continent', 'casedate',
              'total_cases', 'new_cases']].head(2).T
```

|          |       |             |
|----------|-------|-------------|
| iso_code | ABW   | AFG         |
| location | Aruba | Afghanistan |

|             | North America | Asia       |
|-------------|---------------|------------|
| casedate    | 2020-03-22    | 2020-03-01 |
| total_cases | 5.00          | 1.00       |
| new_cases   | 5.00          | 1.00       |

Notice that PandasAI makes smart choices about the columns to get. We get the columns we need rather than all of them. We could have also just passed the names of the columns we wanted to the `chat`.

That's a little PandasAI and OpenAI magic for you! One pretty ordinary sentence passed to the `chat` method did all of the work for us.

## How it works...

Much of the work when using PandasAI is really just importing the relevant libraries and instantiating large language model and `SmartDataframe` objects. Once that's done, simple sentences sent to the `chat` method of the `SmartDataframe` are sufficient to summarize Series values and create new Series.

PandasAI excels at generating simple statistics from Series. We don't even need to remember the Series name exactly, as we saw in *step 3*. Often the natural language we might use can be more intuitive than traditional pandas methods like `groupby`. The *Show satmath average by gender* value passed to `chat` in *step 4* is a good example of that.

Operations on Series, including the creation of a new Series, is also quite straightforward. In *step 5*, we create a total number of children Series (`childnum`) by instructing the `SmartDataframe` to add the number of children living at home to the number of children not living at home. We don't even

provide the literal Series names, *childathome* and *childnotathome* respectively. PandasAI figures out what we mean.

*Steps 6 and 7* demonstrate the flexibility that being able to use natural language for our Series operations provides. We get the same result if we pass *evermarried* is ‘No’ when *maritalstatus* is ‘Never-married’, else ‘Yes’ to `chat` in *step 6* or if *maritalstatus* is ‘Never-married’ set *evermarried2* to ‘No’, otherwise ‘Yes’ in *step 7*.

We can also do fairly extensive DataFrame reshaping with simple natural language instructions, as in *step 11*. We add *and other values* to the instructions to get columns other than *casedate*. PandasAI also figures out that *location* makes sense as the index.

## There’s more...

Given that PandasAI tools are still so new, data scientists are just figuring out now how to best integrate these tools into our data cleaning and analysis workflow. There are two obvious use cases for PandasAI: 1) checking the accuracy of Series operations we do in a more traditional way, and 2) doing Series operations in a more intuitive way when pandas or NumPy tools are somewhat less straightforward, such as with pandas `groupby` or the NumPy `where` function.

PandasAI can also be used to build interactive interfaces for querying a data store, such as a data dashboard. We can use AI tools to help end users interrogate organizational data more effectively. As we saw in *Chapter 3, Taking the Measure of Your Data*, PandasAI is also great for quickly creating visualizations.

## See also

We do much more aggregating of data in *Chapter 9, Fixing Messy Data When Aggregating*, including aggregating data across rows and resampling of date and time data.

## Summary

This chapter explored a wide range of pandas Series methods for exploring and handling data of different types: numeric, strings, and dates. We learned how to get values from and generate summary statistics from a Series. We also learned how to update Series values, and how to do that for subsets of data or conditionally. We also explored specific challenges of working with string or date Series, and how to use Series methods to address those challenges. Finally, we saw how PandasAI could be used to explore and make changes to Series. In the next chapter, we will explore how to identify and fix missing values.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.gg/p8uSgEAETX>



# 7

## Identifying and Fixing Missing Values

I think I speak for many data analysts and scientists when I write, rarely is there something so seemingly small and trivial that is of as much consequence as a missing value. We spend a good deal of our time worrying about missing values because they can have a dramatic, and surprising, effect on our analysis. This is most likely to happen when missing values are not random, but are correlated with a dependent variable. For example, if we are doing a longitudinal study of earnings, but individuals with lower education are more likely to skip the earnings question each year, there is a decent chance that this will bias our parameter estimate for education.

Of course, identifying missing values is not even half of the battle. We then need to decide how to handle them. Do we remove any observation with a missing value for one or more variables? Do we impute a value based on a sample-wide statistic like the mean? Or assign a value based on a more targeted statistic, like the mean for those in a certain class? Do we think of this differently for time series or longitudinal data where the nearest temporal value might make the most sense? Or should we use a more complex multivariate technique for imputing values, perhaps based on regression or  $k$ -nearest neighbors?

The answer to all of the preceding questions is, “yes.” At some point we will want to use each of these techniques. We will want to be able to answer why or why not to all of these possibilities when making a final choice about missing value imputation. Each will make sense depending on the situation.

We will go over techniques in this chapter for identifying the missing values for each variable, and for observations where values for a large number of the variables are absent. We will then explore strategies for imputing values, such as setting values to the overall mean, to the mean for a given category, and forward filling. We also examine multivariate techniques for imputing values and discuss when they are appropriate.

Specifically, we will explore the following recipes in this chapter:

- Identifying missing values
- Cleaning missing values
- Imputing values with regression
- Using  $k$ -nearest neighbors for imputation
- Using random forest for imputation
- Using PandasAI for imputation

## Technical requirements

You will need pandas, NumPy, and Matplotlib to complete the recipes in this chapter. I used pandas 2.1.4, but the code will run on pandas 1.5.3 or later.

The code in this chapter can be downloaded from the book’s GitHub repository, <https://github.com/PacktPublishing/Python-Data-Cleaning-Cookbook-Second-Edition>.

# Identifying missing values

Since identifying missing values is such an important part of the workflow of analysts, any tool we use needs to make it easy to regularly check for such values. Fortunately, pandas makes it quite simple to identify missing values.

## Getting ready

We will work with the **National Longitudinal Survey (NLS)** data in this chapter. The NLS data has one observation per survey respondent. Data for employment, earnings, and college enrollment for each year are stored in columns with suffixes representing the year, such as `weeksworked21` and `weeksworked22` for weeks worked in `2021` and `2022` respectively.

We will also work with the COVID-19 data again. This dataset has one observation for each country with total COVID-19 cases and deaths, as well as some demographic data for each country.

### Data 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 who were born between 1980 and 1985, with annual follow-ups each year through 2023. For this recipe, I pulled 104 variables on grades, employment, income, and attitudes toward the government from the hundreds of data items on the survey.

NLS data can be downloaded from [nlsinfo.org/](https://nlsinfo.org/).



*Our World in Data* provides COVID-19 data for public use at <https://ourworldindata.org/covid-cases>.

The dataset includes total cases and deaths, tests administered, hospital beds, and demographic data such as median age, gross domestic product, and life expectancy. The dataset used in this recipe was downloaded on March 3, 2024.

## How to do it...

We will use pandas functions to identify both missing values and logical missing values (non-missing values that nonetheless connote missing values).

1. Let's start by loading the NLS and COVID-19 data:

```
import pandas as pd
import numpy as np
nls97 = pd.read_csv("data/nls97g.csv",
    low_memory=False)
nls97.set_index("personid", inplace=True)
covidtotals = pd.read_csv("data/covidtotalswithmissin
    low_memory=False)
covidtotals.set_index("iso_code", inplace=True)
```

2. Next, we count the number of missing values for each variable. We can use the `isnull` method to test if each value is missing. It will return True if the value is missing and False if not. We can then use `sum` to count the number of True values, since `sum` will treat each True value as 1 and False value as 0. We indicate `axis=0` to sum over columns rather than across rows:

```
covidtotals.shape
```

```
(231, 16)
```

```
demovars = ['pop_density', 'aged_65_older',
            'gdp_per_capita', 'life_expectancy', 'hum_dev_ind']
covidtotals[demovars].isnull().sum(axis=0)
```

```
pop_density      22
aged_65_older    43
gdp_per_capita   40
life_expectancy   4
hum_dev_ind       44
dtype: int64
```

43 of the 231 countries have null values for `aged_65_older`. We have `life_expectancy` for almost all countries.

3. If we want the number of missing values for each row, we can specify `axis=1` when summing. The following code creates a Series, `demovarsmisscnt`, with the number of missing values for the demographic variables for each country. 178 countries have values for all of the variables, but 16 are missing values for 4 of the 5 variables, and 4 are missing values for all of the variables:

```
demovarsmisscnt = covidtotals[demovars].isnull().sum(
demovarsmisscnt.value_counts().sort_index()
```

```
0      178
1       8
2      14
3      11
```

```
4      16  
5      4  
Name: count, dtype: int64
```

4. Let's take a look at a few of the countries with 4 or more missing values. There is very little demographic data available for these countries:

```
covidtotals.loc[demovarsmisscnt>=4, ['location']] + de  
sample(5, random_state=1).T
```

|                 |                  |                          |
|-----------------|------------------|--------------------------|
| iso_code        | FLK              | SPN                      |
| location        | Falkland Islands | Saint Pierre and Miquelc |
| pop_density     | NaN              | NaN                      |
| aged_65_older   | NaN              | NaN                      |
| gdp_per_capita  | NaN              | NaN                      |
| life_expectancy | 81               | 81                       |
| hum_dev_ind     | NaN              | NaN                      |
| iso_code        | GGY              | COK                      |
| location        | Guernsey         | Cook Islands             |
| pop_density     | NaN              | NaN                      |
| aged_65_older   | NaN              | NaN                      |
| gdp_per_capita  | NaN              | NaN                      |
| life_expectancy | NaN              | 76                       |
| hum_dev_ind     | NaN              | NaN                      |

5. Let's also check the missing values for total cases and deaths. There is one missing value for cases per million of the population and one missing value for deaths per million:

```
totvars = ['location', 'total_cases_pm', 'total_deaths_  
covidtotals[totvars].isnull().sum(axis=0)
```

```
location          0  
total_cases_pm    1  
total_deaths_pm   1  
dtype: int64
```

6. We can easily check if one country is missing both cases per million and deaths per million. We see that 230 countries are not missing either, and just one country is missing both:

```
totvarsmisscnt = covidtotals[totvars].isnull().sum()  
totvarsmisscnt.value_counts().sort_index()
```

```
0    230  
2     1  
Name: count, dtype: int64
```

Sometimes we have logical missing values that we need to transform into actual missing values. This happens when the dataset designers use valid values as codes for missing values. These are often values like 9, 99, or 999, based on the allowable number of digits for the variable. Or it might be a more complicated coding scheme where there are codes for different reasons for there being missing values. For example, on the NLS dataset the codes reveal why the respondent did not provide an answer for a question: -3 is an invalid skip, -4 is a valid skip, and -5 is a non-interview.

7. The last 4 columns on the NLS DataFrame have data on the highest grade completed for the respondent's mother and father, parental income, and the mother's age when the respondent was born. Let's examine logical missing values for those columns, starting with `motherhighgrade`.

```
nlsparents = nls97.iloc[:, -4:]
nlsparents.loc[nlsparents.motherhighgrade.between(-5,
    'motherhighgrade'].value_counts()
```

```
motherhighgrade
-3      523
-4      165
Name: count, dtype: int64
```

8. There are 523 invalid skips and 165 valid skips. Let's look at a few individuals that have at least one of these non-response values for these four variables:

```
nlsparents.loc[nlsparents.transform(lambda x: x.betwe
```

```
          motherage  parentincome  fatherhighgrade  mc
personid
135335            26           -3                16
999406            19           -4                17
151672            26          63000               -3
781297            34           -3                12
613800            25           -3               -3
...                 ...
209909            22          6100               -3
505861            21           -3               -4
368078            19           -3                13
643085            21          23000               -3
713757            22          23000               -3
[3831 rows x 4 columns]
```

9. For our analysis, the reason why there is a non-response is not important. Let's just count the number of non-responses for each of the columns, regardless of the reason for the non-response:

```
nlsparents.transform(lambda x: x.between(-5, -1)).sum()
```

```
motherage          608  
parentincome      2396  
fatherhighgrade   1856  
motherhighgrade    688  
dtype: int64
```

10. We should set these values to missing before using these columns in our analysis. We can use `replace` to set all values between -5 and -1 to missing. When we check for actual missing values we get the expected counts:

```
nlsparents.replace(list(range(-5, 0)), np.nan, inplace=True)  
nlsparents.isnull().sum()
```

```
motherage          608  
parentincome      2396  
fatherhighgrade   1856  
motherhighgrade    688  
dtype: int64
```

## How it works...

We made good use of lambda functions and `transform` in *step 8* and *step 9* to search for values in a specified range across multiple columns.

`transform` works in much the same way as `apply`. Both are methods of `DataFrames` or of `Series`, allowing us to pass one or more columns of data to a function. In this case, we use a lambda function, but we could have also used a named function, as we did in the *Changing Series values*

*conditionally* recipe in *Chapter 6, Cleaning and Exploring Data with Series Operations*.

This recipe demonstrated some very handy pandas techniques to identify the number of missing values for each variable, and observations with a large number of missing values. We also examined how to find logical missing values and convert them to actual missing values. Next, we will take our first look at cleaning missing values.

## Cleaning missing values

We go over some of the most straightforward approaches for handling missing values in this recipe. This includes dropping observations where there are missing values; assigning a sample-wide summary statistic, such as the mean, to the missing values; and assigning values based on the mean value for an appropriate subset of the data.

## How to do it...

We will find and then remove observations from the NLS data that have mainly missing data for key variables. We will also use pandas methods to assign alternative values to missing values, such as the variable mean:

1. Let's load the NLS data and select some of the educational data.

```
import pandas as pd
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
schoolrecordlist = ['satverbal','satmath','gpaoverall'
    'gpaenglish', 'gpamath','gpascience','highestdegree'
    'highestgradecompleted']
```

```
schoolrecord = nls97[schoolrecordlist]
schoolrecord.shape
```

```
(8984, 8)
```

2. We can use the techniques we explored in the previous recipe to identify missing values. `schoolrecord.isnull().sum(axis=0)` gives us the number of missing values for each column. The overwhelming majority of observations have missing values for `satverbal`, 7,578 out of 8,984. Only 31 observations have missing values for `highestdegree`:

```
schoolrecord.isnull().sum(axis=0)
```

```
satverbal          7578
satmath            7577
gpaoverall        2980
gpaenglish         3186
gpamath           3218
gpascience        3300
highestdegree      31
highestgradecompleted  2321
dtype: int64
```

3. We can create a Series, `misscnt`, with the number of missing variables for each observation with `misscnt = schoolrecord.isnull().sum(axis=1)`. 949 observations have 7 missing values for the educational data, and 10 are missing values for all 8 columns. In the following code we also take a look at a few observations with 7 or more missing values. It looks like `highestdegree` is often the one variable that is present, which is not

surprising given that we have already discovered that `highestdegree` is rarely missing:

```
misscnt = schoolrecord.isnull().sum(axis=1)
misscnt.value_counts().sort_index()
```

```
0      1087
1      312
2     3210
3     1102
4      176
5      101
6    2037
7      949
8      10
dtype: int64
```

```
schoolrecord.loc[misscnt>=7].head(4).T
```

|                       | 403743 | 101705 | 943703  | 406679  |
|-----------------------|--------|--------|---------|---------|
| personid              |        |        |         |         |
| satverbal             | NaN    | NaN    | NaN     | NaN     |
| satmath               | NaN    | NaN    | NaN     | NaN     |
| gpaoverall            | NaN    | NaN    | NaN     | NaN     |
| gpaenglish            | NaN    | NaN    | NaN     | NaN     |
| gpamath               | NaN    | NaN    | NaN     | NaN     |
| gpascience            | NaN    | NaN    | NaN     | NaN     |
| highestdegree         | 1. GED | 1. GED | 0. None | 0. None |
| highestgradecompleted | NaN    | NaN    | NaN     | NaN     |

- Let's drop observations that have missing values for 7 or more variables, out of 8. We can accomplish this by setting the `thresh` parameter of `dropna` to `2`. This will drop observations that have fewer than 2 non-missing values. We get the expected number of observations after the `dropna`;  $8984 - 949 - 10 = 8025$ :

```
schoolrecord = schoolrecord.dropna(thresh=2)
schoolrecord.shape
```

```
(8025, 8)
```

```
schoolrecord.isnull().sum(axis=1).value_counts().sort
```

```
0      1087
1      312
2     3210
3     1102
4      176
5      101
6    2037
dtype: int64
```

There are a fair number of missing values for `gpaoverall`, 2,980, though we have valid values for two-thirds of the observations

$((8984 - 2980) / 8984)$ . We might be able to salvage this as a variable if we do a good job of imputing missing values. This is likely more desirable than just removing these observations. We do not want to lose that data if we can avoid it, particularly if individuals with a missing `gpaoverall` are different from others in ways that will matter for our predictions.

5. The most straightforward approach is to assign the overall mean for `gpaoverall` to the missing values. The following code uses the pandas Series `fillna` method to assign all missing values of `gpaoverall` to the Series mean value. The first argument to `fillna` is the value you want for all missing values, in this case, `schoolrecord.gpaoverall.mean()`. Note that we need to remember to

set the `inplace` parameter to True to actually overwrite the existing values:

```
schoolrecord = nls97[schoolrecordlist]
schoolrecord.gpaoverall.agg(['mean', 'std', 'count'])
```

```
mean      282
std       62
count    6,004
Name: gpaoverall, dtype: float64
```

```
schoolrecord.fillna({"gpaoverall":\
    schoolrecord.gpaoverall.mean()},
    inplace=True)
schoolrecord.gpaoverall.isnull().sum()
```

```
0
```

```
schoolrecord.gpaoverall.agg(['mean', 'std', 'count'])
```

```
mean      282
std       50
count    8,984
Name: gpaoverall, dtype: float64
```

The mean is unchanged, of course, but there is a substantial reduction in the standard deviation, from 62 to 50. This is a disadvantage of using the dataset mean for all missing values.

6. The NLS data also has a fair number of missing values for `wageincome20`. The following code shows that 3,783 observations have missing values. We make a deep copy with the `copy` method, setting

`deep` to `True`. We would not normally do this, but in this case we don't want to change the values of `wageincome20` in the underlying DataFrame. We don't want to do that here because we will try a different method of imputing values in the next couple of code blocks:

```
wageincome20 = nls97.wageincome20.copy(deep=True)  
wageincome20.isnull().sum()
```

```
3783
```

```
wageincome20.head().T
```

```
personid  
135335      NaN  
999406    115,000  
151672      NaN  
750699     45,000  
781297    150,000  
Name: wageincome20, dtype: float64
```

7. Rather than assigning the mean value of `wageincome` to the missing values, we could use another common technique for imputing values. We could assign the nearest non-missing value from a preceding observation. We can use the `ffill` method of the Series object to do this (note that this does not impute a value for the first observation as there is no preceding value to use):

```
wageincome20.ffill(inplace=True)  
wageincome20.head().T
```

```
personid  
135335      NaN
```

```
999406    115,000
151672    115,000
750699     45,000
781297    150,000
Name: wageincome20, dtype: float64
```

```
wageincome20.isnull().sum()
```

```
1
```

### Note

If you have used `ffill` in pandas versions prior to 2.2.0, you might remember the following syntax:



```
wageincome.fillna(method="ffill", inplace=True)
```

This syntax was deprecated, starting with pandas 2.2.0. That is also true for the backward fill syntax, which we will use next.

8. We could have done a backward fill instead by using the `bfill` method. This sets missing values to the nearest following value. This produces the following:

```
wageincome20 = nls97.wageincome20.copy(deep=True)
wageincome20.head().T
```

```
personid
135335      NaN
999406    115,000
```

```
151672      NaN
750699    45,000
781297  150,000
Name: wageincome20, dtype: float64
```

```
wageincome20.std()
```

```
59616.290306039584
```

```
wageincome20.bfill(inplace=True)
wageincome20.head().T
```

```
personid
135335    115,000
999406    115,000
151672     45,000
750699     45,000
781297   150,000
Name: wageincome20, dtype: float64
```

```
wageincome20.std()
```

```
58199.4895818016
```

If missing values are randomly distributed then forward or backward filling has one advantage over using the mean. It is more likely to approximate the distribution of the non-missing values for the variable. Notice that the standard deviation did not drop much after backward filling.

There are times when it makes sense to base our imputation of values on the mean or median value for similar observations; say those that have the same value for a related variable. Let's try that in the next step.

9. In the NLS DataFrame, weeks worked in 2020 is correlated with highest degree earned. The following code shows how the mean value of weeks worked changes with degree attainment. The mean for weeks worked is 38, but it is much lower for those without a degree (28) and much higher for those with a professional degree (48). In this case, it may be a better choice to assign 28 to missing values for weeks worked for individuals who have not attained a degree, rather than 38:

```
nls97.weeksworked20.mean()
```

```
38.35403815808349
```

```
nls97.groupby(['highestdegree'])['weeksworked20'].mea
```

```
highestdegree
0. None          28
1. GED           34
2. High School   37
3. Associates    41
4. Bachelors     42
5. Masters        45
6. PhD            47
7. Professional   48
Name: weeksworked20, dtype: float64
```

10. The following code assigns the mean value of weeks worked across observations with the same degree attainment level for those observations missing weeks worked. We do this by using `groupby` to create a groupby DataFrame, `groupby(['highestdegree'])` `['weeksworked20']`. We then use `fillna` within `transform` to fill missing values with the mean for the highest degree group. Notice that we make sure to only do this imputation for observations where the highest degree information is not missing, `nls97.highestdegree.notnull()`. We will still have missing values for observations missing both highest degree and weeks worked:

```
nls97.loc[nls97.highestdegree.notnull(), 'weeksworked20'] = nls97.loc[nls97.highestdegree.notnull()].groupby(['highestdegree'])['weeksworked20'].transform(lambda x: x.fillna(x.mean()))
nls97[['weeksworked20imp', 'weeksworked20', 'highestdegree']].head(10)
```

| personid | weeksworked20imp | weeksworked20 | highestdegree |
|----------|------------------|---------------|---------------|
| 135335   | 42               | NaN           | 4. Bachelor   |
| 999406   | 52               | 52            | 2. High Schoc |
| 151672   | 0                | 0             | 4. Bachelor   |
| 750699   | 52               | 52            | 2. High Schoc |
| 781297   | 52               | 52            | 2. High Schoc |
| 613800   | 52               | 52            | 2. High Schoc |
| 403743   | 34               | NaN           | 1. GE         |
| 474817   | 51               | 51            | 5. Master     |
| 530234   | 52               | 52            | 5. Master     |
| 351406   | 52               | 52            | 4. Bachelor   |

## How it works...

When there is very little data available it can make sense to remove an observation from our analysis. We did that in *step 4*. Another common approach is the one we used in *step 5*, assigning the overall dataset mean for the variable to missing values. We saw in that example one of the disadvantages of that approach. We can end up with a significantly reduced variance in our variable.

In *step 9* we assigned values based on the mean value of that variable for a subset of our data. If we are imputing values for variable  $X_1$ , and  $X_1$  is correlated with  $X_2$ , we can use the relationship between  $X_1$  and  $X_2$  to impute a value for  $X_1$  that might make more sense than the dataset mean. This is pretty straightforward when  $X_2$  is categorical. In this case we can impute the mean value of  $X_1$  for the associated value of  $X_2$ .

These imputation strategies—removing observations with missing values, assigning a dataset mean or median, using forward or backward filling, or using a group mean for a correlated variable—are fine for many predictive analytics projects. They work best when the missing values are not correlated with a target or dependent variable. When that is true, imputing values allows us to retain the other information from those observations without biasing our estimates.

Sometimes, however, that is not the case and more complicated imputation strategies are required. The next few recipes explore multivariate techniques for cleaning missing data.

## See also

Don't worry if your understanding of what we did in *step 10*, using `groupby` and `transform`, is still a little shaky. We do much more with `groupby`,

`transform`, and `apply` in *Chapter 9, Fixing Messy Data When Aggregating*.

## Imputing values with regression

We ended the previous recipe by assigning a group mean to missing values rather than the overall sample mean. As we discussed, this is useful when the variable that determines the groups is correlated with the variable that has the missing values. Using regression to impute values is conceptually similar to this, but we typically use it when the imputation will be based on two or more variables.

Regression imputation replaces a variable's missing values with values predicted by a regression model of correlated variables. This particular kind of imputation is known as deterministic regression imputation, since the imputed values all lie on the regression line, and no error or randomness is introduced.

One potential drawback of this approach is that it can substantially reduce the variance of the variable with missing values. We can use stochastic regression imputation to address this drawback. We explore both approaches in this recipe.

## Getting ready

We will work with the `statsmodels` module to run a linear regression model in this recipe. `statsmodels` is typically included with scientific distributions of Python, but if you do not already have it, you can install it with `pip install statsmodels`.

# How to do it...

The `wageincome20` column on the NLS dataset has a number of missing values. We can use linear regression to impute values. The wage income value is the reported earnings for 2020.

1. We start by loading the NLS data again and checking for missing values for `wageincome20` and columns that might be correlated with `wageincome20`. We also load the `statsmodels` library:

```
import pandas as pd
import numpy as np
import statsmodels.api as sm
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
nls97[['wageincome20', 'highestdegree', 'weeksworked20']]
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 8984 entries, 135335 to 713757
Data columns (total 4 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   wageincome20    5201 non-null    float64
 1   highestdegree   8952 non-null    object  
 2   weeksworked20   6971 non-null    float64
 3   parentincome    6588 non-null    float64
dtypes: float64(3), object(1)
memory usage: 350.9+ KB
```

2. We are missing values for `wageincome20` for more than 3,000 observations. There are fewer missing values for the other variables. Let's convert the `highestdegree` column to numeric so that we can use it in a regression model:

```
nls97['hdegnm'] = nls97.highestdegree.str[0:1].astype(str)
nls97.groupby(['highestdegree', 'hdegnm']).size()
```

```
highestdegree    hdegnm
0. None          0           877
1. GED           1          1167
2. High School   2          3531
3. Associates    3           766
4. Bachelors     4          1713
5. Masters        5           704
6. PhD            6            64
7. Professional   7           130
dtype: int64
```

3. As we have already discovered, we need to replace logical missing values for `parentincome` with actual missing values. After that, we can run some correlations. Each of the variables has some positive correlation with `wageincome20`, particularly `hdegnm`:

```
nls97.parentincome.replace(list(range(-5,0)), np.nan,
nls97[['wageincome20', 'hdegnm', 'weeksworked20', 'pare
```

```
wageincome20    hdegnm    weeksworked20  parentincome
wageincome20      1.00      0.38          0.22       0.2
hdegnm            0.38      1.00          0.22       0.3
weeksworked20     0.22      0.22          1.00       0.6
parentincome      0.27      0.32          0.09       1.0
```

4. We should check to see if observations with missing values for wage income are different in some important way from those with non-missing values. The following code shows that these observations have significantly lower degree attainment levels, parental income, and

weeks worked. This is a clear case where assigning the overall mean would not be the best choice:

```
nls97weeksworked = nls97.loc[nls97.weeksworked20>0]
nls97weeksworked.shape
```

```
(5889, 111)
```

```
nls97weeksworked['missingwageincome'] = \
    np.where(nls97weeksworked.wageincome20.isnull(), 1, 0
nls97weeksworked.groupby(['missingwageincome'])[['hde
    'parentincome', 'weeksworked20']].\
    agg(['mean', 'count'])
```

|                   | hdegnm |       | parentincome |       | weeksworked2 |       |
|-------------------|--------|-------|--------------|-------|--------------|-------|
|                   | mean   | count | mean         | count | mean         | count |
| missingwageincome |        |       |              |       |              |       |
| 0                 | 2.81   | 4997  | 48,270.85    | 3731  | 47.97        | 501   |
| 1                 | 2.31   | 875   | 40,436.23    | 611   | 30.70        | 87    |

Notice that we just work here with rows that have positive values for weeks worked. It does not make sense for someone who did not work in 2020 to have a wage income in 2020.

5. Let's try regression imputation instead. We start by replacing missing `parentincome` values with the mean. We collapse `hdegnm` into those attaining less than a college degree, those with a college degree, and those with a post-graduate degree. We set those up as dummy variables, with `0` or `1` values when `False` or `True`. This is a tried and true method for treating categorical data in regression analysis. It

allows us to estimate different y-intercepts based on group membership.

(*Scikit-learn* has preprocessing features that can help us with tasks like these. We go over some of them in the next chapter.)

```
nls97weeksworked.parentincome. \
    fillna(nls97weeksworked.parentincome.mean(), inplace=True)
nls97weeksworked['degltcol'] = \
    np.where(nls97weeksworked.hdegnorm<=2, 1, 0)
nls97weeksworked['degcol'] = \
    np.where(nls97weeksworked.hdegnorm.between(3, 4), 1, 0)
nls97weeksworked['degadv'] = \
    np.where(nls97weeksworked.hdegnorm>4, 1, 0)
```

6. Next, we define a function, `getlm`, to run a linear model using the `statsmodels` module. The function has parameters for the name of the target or dependent variable, `ycolname`, and for the names of the features or independent variables, `xcolnames`. Much of the work is done by the `statsmodels` `fit` method, `OLS(y, X).fit()`:

```
def getlm(df, ycolname, xcolnames):
    df = df[[ycolname] + xcolnames].dropna()
    y = df[ycolname]
    X = df[xcolnames]
    X = sm.add_constant(X)
    lm = sm.OLS(y, X).fit()
    coefficients = pd.DataFrame(zip(['constant'] + xcolnames,
        lm.params, lm.pvalues), columns=['features', 'parameters',
        'pvalues'])
    return coefficients, lm
```

7. Now we can use the `getlm` function to get the parameter estimates and the model summary. All of the coefficients are positive and significant

at the 95% level, having  $p$ -values less than 0.05. As expected, wage income increases with number of weeks worked and with parental income. Having a college degree gives a \$18.5K boost to earnings, compared with not having a college degree. A post-graduate degree bumps up the earnings prediction even more, almost \$45.6K more than for those with less than a college degree. (The coefficients on degcol and degadv are interpreted as relative to those without a college degree since that is the omitted dummy variable.)

```
xvars = ['weeksworked20', 'parentincome', 'degcol', 'degadv']  
coefficients, lm = getlm(nls97weeksworked20, 'wageincome')
```

|   | features                  | params     | pvalues |
|---|---------------------------|------------|---------|
| 0 | constant                  | -22,868.00 | 0.00    |
| 1 | weeksworke <del>d20</del> | 1,281.84   | 0.00    |
| 2 | parentincome              | 0.26       | 0.00    |
| 3 | degcol                    | 18,549.57  | 0.00    |
| 4 | degadv                    | 45,595.94  | 0.00    |

8. We use this model to impute values for wage income where they are missing. We need to add a constant for the predictions since our model included a constant. We can convert the predictions to a DataFrame and then join it with the rest of the NLS data. Let's also take a look at some of the predictions to see if they make sense.

```
pred = lm.predict(sm.add_constant(nls97weeksworked20)  
                  .to_frame().rename(columns= {0: 'pred'})  
nls97weeksworked20 = nls97weeksworked20.join(pred)  
nls97weeksworked20['wageincomeimp'] = \  
    np.where(nls97weeksworked20.wageincome20.isnull(), \  
             nls97weeksworked20.pred, nls97weeksworked20.wageincome20)
```

```
nls97weeksworked[['wageincomeimp', 'wageincome20'] + x  
sample(10, random_state=7)
```

| personid | wageincomeimp | wageincome20 | weeksworked20 | parentincon |
|----------|---------------|--------------|---------------|-------------|
| 696721   | 380,288       | 380,288      | 52            | 81,300      |
| 928568   | 38,000        | 38,000       | 41            | 47,168      |
| 738731   | 38,000        | 38,000       | 51            | 17,000      |
| 274325   | 40,698        | NaN          | 7             | 34,800      |
| 644266   | 63,954        | NaN          | 52            | 78,000      |
| 438934   | 70,000        | 70,000       | 52            | 31,000      |
| 194288   | 1,500         | 1,500        | 13            | 39,000      |
| 882066   | 52,061        | NaN          | 52            | 32,000      |
| 169452   | 110,000       | 110,000      | 52            | 48,600      |
| 284731   | 25,000        | 25,000       | 52            | 47,168      |
| personid | degcol        | degadv       |               |             |
| 696721   | 1             | 0            |               |             |
| 928568   | 0             | 0            |               |             |
| 738731   | 1             | 0            |               |             |
| 274325   | 0             | 1            |               |             |
| 644266   | 0             | 0            |               |             |
| 438934   | 1             | 0            |               |             |
| 194288   | 0             | 0            |               |             |
| 882066   | 0             | 0            |               |             |
| 169452   | 1             | 0            |               |             |
| 284731   | 0             | 0            |               |             |

9. We should look at some summary statistics for our wage income imputation and compare that with the actual wage income values. (Remember that the `wageincomeimp` column has the actual value for `wageincome20` when it was not missing, and imputed values otherwise.) The mean for `wageincomeimp` is somewhat less than that for `wageincome20`, which we anticipated given that folks with missing wage income had lower values for correlated variables. But the

standard deviation is also lower. This can happen with deterministic regression imputation:

```
nls97weeksworked[['wageincomeimp', 'wageincome20']].\n    agg(['count', 'mean', 'std'])
```

|       | wageincomeimp | wageincome20 |
|-------|---------------|--------------|
| count | 5,889         | 5,012        |
| mean  | 59,290        | 63,424       |
| std   | 57,529        | 60,011       |

10. Stochastic regression imputation adds a normally distributed error to the predictions based on the residuals from our model. We want this error to have a mean of zero with the same standard deviation as our residuals. We can use NumPy's normal function for that with

`np.random.normal(0, lm.resid.std(), nls97.shape[0])`. The `lm.resid.std()` gets us the standard deviation of the residuals from our model. The final parameter value, `nls97.shape[0]`, indicates how many values to create; in this case we want a value for every row in our data.

We can join those values with our data and then add the error, `randomadd`, to our prediction. We set a seed so that we can reproduce the results:

```
np.random.seed(0)\nrandomadd = np.random.normal(0, lm.resid.std(),\n    nls97weeksworked.shape[0])\nrandomadddf = pd.DataFrame(randomadd, columns=[ 'randomadd' \n        index=nls97weeksworked.index)\nnls97weeksworked = nls97weeksworked.join(randomadddf)
```

```
nls97weeksworked['stochasticpred'] = \
nls97weeksworked.pred + nls97weeksworked.randomadd
```

11. This should increase the variance but not have much of an effect on the mean. Let's confirm that. We first need to replace missing wage income values with the stochastic prediction:

```
nls97weeksworked['wageincomeimpstoc'] = \
np.where(nls97weeksworked.wageincome20.isnull(),
nls97weeksworked.stochasticpred, nls97weeksworked.wa
nls97weeksworked[['wageincomeimpstoc', 'wageincome20']
agg(['count', 'mean', 'std'])
```

|       | wageincomeimpstoc | wageincome20 |
|-------|-------------------|--------------|
| count | 5,889             | 5,012        |
| mean  | 59,485            | 63,424       |
| std   | 60,773            | 60,011       |

That seems to have worked. The imputed variable based on our stochastic prediction has pretty much the same standard deviation as the wage income variable.

## How it works...

Regression imputation is a good way to take advantage of all the data we have to impute values for a column. It is often superior to the imputation methods we examined in the previous recipe, particularly when missing values are not random. Deterministic regression imputation does, however, have two important limitations: it assumes a linear relationship between the regressors (our predictor variables) and the variable to be imputed, and it

can substantially reduce the variance of the imputed variable, as we saw in *steps 8 and 9*.

If we use stochastic regression imputation we will not artificially reduce our variance. We did this in *step 10*. This gave us better results, though it did not address the possible issue of a non-linear relationship between regressors and the imputed variable.

Before we started using machine learning widely for this work, regression imputation was our go to multivariate approach for imputation. We now have the option of using algorithms like  $k$ -nearest neighbors and random forest for this task, which have advantages over regression imputation in some cases. KNN imputation, unlike regression imputation, does not assume a linear relationship between variables, or that those variables are normally distributed. We explore KNN imputation in the next recipe.

## Using $k$ -nearest neighbors for imputation

**k-Nearest Neighbors (KNN)** is a popular machine learning technique because it is intuitive and easy to run and yields good results when there is not a large number of variables and observations. For the same reasons, it is often used to impute missing values. As its name suggests, KNN identifies the  $k$  observations whose variables are most similar to each observation. When used to impute missing values, KNN uses the nearest neighbors to determine what fill values to use.

## Getting ready

We will work with the KNN imputer from scikit-learn version 1.3.0. If you do not already have scikit-learn, you can install it with `pip install scikit-learn`.

## How to do it...

We can use KNN imputation to do the same imputation we did in the previous recipe on regression imputation.

1. We start by importing the `KNNImputer` from `scikit-learn` and loading the NLS data again:

```
import pandas as pd
import numpy as np
from sklearn.impute import KNNImputer
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
```

2. Next, we prepare the variables. We collapse degree attainment into three categories—less than college, college, and post-college degree—each category represented by a different dummy variable. We also convert logical missing values for parent income to actual missing values:

```
nls97['hdegnm'] = \
    nls97.highestdegree.str[0:1].astype('float')
nls97['parentincome'] = \
    nls97.parentincome.\
        replace(list(range(-5, 0)),\
            np.nan)
```

3. Let's create a DataFrame with just wage income and a few correlated variables. We also select only those rows with positive values for weeks worked:

```
wagedatalist = ['wageincome20', 'weeksworked20',
                 'parentincome', 'hdegnm']
wagedata = \
    nls97.loc[nls97.weeksworked20>0, wagedatalist]
wagedata.shape
```

```
(5889, 6)
```

4. We are now ready to use the `fit_transform` method of the KNN imputer to get values for all missing values in the passed DataFrame, `wagedata`. `fit_transform` returns a NumPy array with all the non-missing values from `wagedata`, plus the imputed ones. We convert this array into a DataFrame using the same index as `wagedata`. This will make it easy to join the data in the next step. (This will be a familiar step to folks who have some experience using scikit-learn. We will go over it in more detail in the next chapter.)

We need to specify the value to use for number of nearest neighbors, for  $k$ . We use a general rule of thumb for determining  $k$ , the square root of the number of observations divided by 2 ( $\sqrt{N}/2$ ). That gives us 38 for  $k$  in this case.

```
impKNN = KNNImputer(n_neighbors=38)
newvalues = impKNN.fit_transform(wagedata)
wagedatalistimp = ['wageincomeimp', 'weeksworked20imp',
```

```
'parentincomeimp', 'hdegnumpimp']  
wagedataimp = pd.DataFrame(newvalues, columns=wagedatalist)
```

5. We join the imputed data with the original NLS wage data and take a look at a few observations. Notice that with KNN imputation that we did not need to do any pre-imputation for missing values of correlated variables. (With regression imputation, we set parent income to the dataset mean.)

```
wagedata = wagedata.\  
join(wagedataimp[['wageincomeimp', 'weeksworked20imp']]  
wagedata[['wageincome20', 'wageincomeimp', 'weeksworked'  
'weeksworked20imp']]).sample(10, random_state=7)
```

| personid | wageincome20 | wageincomeimp | weeksworked20 | weekswor |
|----------|--------------|---------------|---------------|----------|
| 696721   | 380,288      | 380,288       | 52            |          |
| 928568   | 38,000       | 38,000        | 41            |          |
| 738731   | 38,000       | 38,000        | 51            |          |
| 274325   | NaN          | 11,771        | 7             |          |
| 644266   | NaN          | 59,250        | 52            |          |
| 438934   | 70,000       | 70,000        | 52            |          |
| 194288   | 1,500        | 1,500         | 13            |          |
| 882066   | NaN          | 61,234        | 52            |          |
| 169452   | 110,000      | 110,000       | 52            |          |
| 284731   | 25,000       | 25,000        | 52            |          |

6. Let's take a look at summary statistics for the original and imputed variables. Not surprisingly, the imputed wage income mean is lower than the original mean. As we discovered in the previous recipe, observations with missing wage income have lower degree attainment,

weeks worked, and parental income. We also lose some of the variance in wage income.

```
wagedata[['wageincome20', 'wageincomeimp']].\nagg(['count', 'mean', 'std'])
```

|       | wageincome20 | wageincomeimp |
|-------|--------------|---------------|
| count | 5,012        | 5,889         |
| mean  | 63,424       | 59,467        |
| std   | 60,011       | 57,218        |

That was easy! The preceding steps gave us reasonable imputations for wage income, and also for other variables with missing values, with minimal data preparation on our part.

## How it works...

Most of the work in this recipe was done in *step 4*, when we passed our DataFrame to the `fit_transform` method of the KNN imputer. The KNN imputer returned a NumPy array with imputations for missing values for all columns in our data, including wage income. It did this imputation based on values for the  $k$  most similar observations. We converted the NumPy array into a DataFrame that we joined with the initial DataFrame in *step 5*.

KNN does imputations without making any assumptions about the distribution of the underlying data. With regression imputation, the standard assumptions for linear regression apply, that there is a linear relationship between variables and that they are distributed normally. If this is not the case, KNN is likely a better approach to imputation.

We did need to make an initial assumption about an appropriate value for  $k$ , what is known as a hyperparameter. Model builders generally do

hyperparameter tuning to find the best value of  $k$ . Hyperparameter tuning for KNN is beyond the scope of this book, but I step the reader through it in my book *Data Cleaning and Exploration with Machine Learning*. We made a reasonable assumption about a good value for  $k$  in *step 4*.

## There's more...

Despite these advantages, KNN imputation does have limitations. As we just discussed, we had to tune the model with an initial assumption about a good value for  $k$ , based only on our knowledge of the size of the dataset. There is some risk of overfitting—fitting the data with non-missing values for the target variable so well that our estimates for the missing values are unreliable—as we increase the value of  $k$ . Hyperparameter tuning can help us identify the best value for  $k$ .

KNN is also computationally expensive and may be impractical for very large datasets. Finally, KNN imputation may not perform well when the correlation is weak between the variable to be imputed and the predictor variables, or when those variables are very highly correlated. An alternative to KNN for imputation, random forest imputation, can help us avoid the disadvantages of both KNN and regression imputation. We explore random forest imputation next.

## See also

There is a fuller discussion of KNN, and examples with real-world data, in my book *Data Cleaning and Exploration with Machine Learning*. That discussion will give you a better understanding of how the algorithm works, and will contrast it with other non-parametric machine learning algorithms,

such as random forest. We look at random forest for imputing values in the next recipe.

## Using random forest for imputation

Random forest is an ensemble learning method, using bootstrap aggregating, also known as bagging, to improve model accuracy. It makes predictions by repeatedly taking the mean of multiple trees, yielding progressively better estimates. We will use the MissForest algorithm in this recipe, which is an application of the random forest algorithm to missing value imputation.

MissForest starts by filling in the median or mode (for continuous or categorical variables respectively) for missing values, then uses random forest to predict values. Using this transformed dataset, with missing values replaced by initial predictions, MissForest generates new predictions, perhaps replacing the initial prediction with a better one. MissForest will typically go through at least 4 iterations of this process.

## Getting ready

You will need to install the `MissForest` and `MiceForest` modules to run the code in this recipe. You can install both with `pip`.

## How to do it...

Running MissForest is even easier than using the KNN imputer, which we used in the previous recipe. We will impute values for the same wage income data that we worked with previously.

1. Let's start by importing the `MissForest` module and loading the NLS data. We import `missforest`, and also `miceforest`, which we discuss in later steps:

```
import pandas as pd
import numpy as np
from missforest.missforest import MissForest
import miceforest as mf
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
```

2. We should do the same data cleaning that we did in the previous recipe:

```
nls97['hdegnm'] = \
nls97.highestdegree.str[0:1].astype('float')
nls97['parentincome'] = \
nls97.parentincome.\
    replace(list(range(-5,0)),
    np.nan)
wagedatalist = ['wageincome20', 'weeksworked20', 'paren
    'hdegnm']
wagedata = \
nls97.loc[nls97.weeksworked20>0, wagedatalist]
```

3. Now we are ready to run `MissForest`. Notice that the process is remarkably similar to our process for using the KNN imputer:

```
imputer = MissForest()
wagedataimp = imputer.fit_transform(wagedata)
wagedatalistimp = \
['wageincomeimp', 'weeksworked20imp', 'parentincomeimp']
wagedataimp.rename(columns=\
{'wageincome20': 'wageincome20imp',
'weeksworked20': 'weeksworked20imp',
```

```
'parentincome' : 'parentincomeimp'}, inplace=True)
wagedata = \
wagedata.join(wagedataimp[['wageincome20imp',
'weeksworked20imp']])
```

4. Let's take a look at a few of our imputed values and some summary statistics. The imputed values have a lower mean. This is not surprising given that we have already learned that the missing values are not distributed randomly, as individuals with lower degree attainment and weeks worked are more likely to have missing values for wage income:

```
wagedata[['wageincome20', 'wageincome20imp',
'weeksworked20', 'weeksworked20imp']].\
sample(10, random_state=7)
```

| personid | wageincome20 | wageincome20imp | weeksworked20 | weekswor |
|----------|--------------|-----------------|---------------|----------|
| 696721   | 380,288      | 380,288         | 52            |          |
| 928568   | 38,000       | 38,000          | 41            |          |
| 738731   | 38,000       | 38,000          | 51            |          |
| 274325   | NaN          | 6,143           | 7             |          |
| 644266   | NaN          | 85,050          | 52            |          |
| 438934   | 70,000       | 70,000          | 52            |          |
| 194288   | 1,500        | 1,500           | 13            |          |
| 882066   | NaN          | 74,498          | 52            |          |
| 169452   | 110,000      | 110,000         | 52            |          |
| 284731   | 25,000       | 25,000          | 52            |          |

```
wagedata[['wageincome20', 'wageincome20imp',
'weeksworked20', 'weeksworked20imp']].\
agg(['count', 'mean', 'std'])
```

|       | wageincome20 | wageincome20imp | weeksworked20 | weekswor |
|-------|--------------|-----------------|---------------|----------|
| count | 5,012        | 5,889           | 5,889         |          |
| mean  | 63,424       | 59,681          | 45            |          |
| std   | 60,011       | 57,424          | 14            |          |

MissForest uses the random forest algorithm to generate highly accurate predictions. Unlike KNN, it does not require tuning with an initial value for  $k$ . It also is computationally less expensive than KNN. Perhaps most importantly, random forest imputation is less sensitive to low or very high correlation among variables, though that was not an issue in this example.

## How it works...

We largely follow the same process here as we did with KNN imputation in the previous recipe. We start by cleaning the data a bit, extracting a numeric variable from the highest degree text, and replacing logical missing values for parental income with actual missing values.

We then pass our data to the `fit_transform` method of a `MissForest` imputer. This method returns a DataFrame with imputed values for all columns.

## There's more...

We could have used Multiple Imputation by Chained Equations (MICE), which can be implemented using random forests, for our imputation instead. One advantage of this approach is that MICE adds a random component to imputations, likely further reducing the possibility of overfitting even over `missforest`.

`miceforest` can be run in much the same way as `missforest`.

1. We create a `kernel` with the `miceforest` instance we created in *step 1*:

```
kernel = mf.ImputationKernel(  
    data=wagedata[wagedatalist],  
    save_all_iterations=True,  
    random_state=1  
)  
kernel.mice(3,verbose=True)
```

```
Initialized logger with name mice 1-3  
Dataset 0  
1 | degltcol | degcol | degadv | weeksworked20 | parentincc  
2 | degltcol | degcol | degadv | weeksworked20 | parentincc  
3 | degltcol | degcol | degadv | weeksworked20 | parentincc
```

```
wagedataimpmice = kernel.complete_data()
```

2. Then we can view the results of our imputation:

```
wagedataimpmice.rename(columns=\  
    {'wageincome20':'wageincome20impmice',  
     'weeksworked20':'weeksworked20impmice',  
     'parentincome':'parentincomeimpmice'},  
    inplace=True)  
wagedata = wagedata[wagedatalist].\  
    join(wagedataimpmice[['wageincome20impmice',  
                         'weeksworked20impmice']])  
wagedata[['wageincome20','wageincome20impmice',  
          'weeksworked20','weeksworked20impmice']].\  
agg(['count','mean','std'])
```

|       | wageincome20 | wageincome20impmice | weeksworked20 |
|-------|--------------|---------------------|---------------|
| count | 5,012        | 5,889               | 5,889         |
| mean  | 63,424       | 59,191              | 45            |
| std   | 60,011       | 58,632              | 14            |

```
    weeksworked20imp mice
count           5,889
mean            45
std             14
```

This produces very similar results as `missforest`. Both approaches are excellent choices for missing value imputation.

## Using PandasAI for imputation

Many of the missing value imputation tasks we have explored in this chapter can also be completed using PandasAI. As we have discussed in previous chapters, AI tools can help us check the work we have done with traditional tools and can suggest alternative approaches that did not occur to us. It always makes sense, though, to look under the hood and be sure we understand what PandasAI, or other AI tools, are doing.

We will use PandasAI in this recipe to identify missing values, impute missing values based on summary statistics, and assign missing values based on machine learning algorithms.

## Getting ready

We will work with PandasAI in this recipe. It can be installed with `pip install pandasai`. You also need to get a token from [openai.com](https://openai.com) to send a request to the OpenAI API.

## How to do it...

In this recipe, we will carry out many of the tasks we have done earlier in this chapter using AI tools instead.

1. We start by importing the `pandas` and `numpy` libraries and `openAI` and `pandasai`. We will work a fair bit with the PandasAI `SmartDataFrame` module in this recipe. We will also load the NLS data:

```
import pandas as pd
import numpy as np
from pandasai.llm.openai import OpenAI
from pandasai import SmartDataframe
llm = OpenAI(api_token="Your API Key")
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
```

2. We do the same data cleaning on the parent income and highest degree variables that we did in previous recipes:

```
nls97['hdegnun'] = nls97.highestdegree.str[0:1].astype(str)
nls97['parentincome'] = \
    nls97.parentincome.\n        replace(list(range(-5,0)),\n            np.nan)
```

3. We create a DataFrame with just the wage and degree data, and then a `SmartDataFrame` from PandasAI:

```
wagedatalist = ['wageincome20', 'weeksworked20',
                 'parentincome', 'hdegnun']
wagedata = nls97[wagedatalist]
wagedatasdf = SmartDataFrame(wagedata, config={"llm":
```

4. Show non-missing counts, averages, and standard deviations for all the variables. We send a natural language command to the `chat` method of the `SmartDataFrame` object to do that. Since `hdegnm` (highest degree) is a categorical variable, `chat` does not show means or standard deviations:

```
wagedatasdf.chat("Show the counts, means, and standar
```

|               | count | mean   | std    |
|---------------|-------|--------|--------|
| wageincome20  | 5,201 | 62,888 | 59,616 |
| weeksworked20 | 6,971 | 38     | 21     |
| parentincome  | 6,588 | 46,362 | 42,144 |

5. Let's impute values for the missing values based on the average for each variable. The `chat` method will return a pandas DataFrame in this case. There are no longer missing values for the income and weeks worked variables, but PandasAI figured out that the degree categorical variable should not be imputed based on average:

```
wagedatasdf = \
wagedatasdf.chat("Impute missing values based on ave
wagedatasdf.chat("Show the counts, means, and standar
```

|               | count | mean   | std    |
|---------------|-------|--------|--------|
| wageincome20  | 8,984 | 62,888 | 45,358 |
| weeksworked20 | 8,984 | 38     | 18     |
| parentincome  | 8,984 | 46,362 | 36,088 |

6. Let's look again at the values for highest degree. Notice that the most frequent value is `2`, which you may recall from earlier recipes

represents high school completion.

```
wagedatasdf.hdegnm.value_counts(dropna=False).sort_i
```

```
hdegnm
0    877
1   1167
2   3531
3    766
4   1713
5    704
6     64
7    130
NaN   32
Name: count, dtype: int64
```

7. We can set missing values for the degree variables to their most frequent non-missing value, which is not an uncommon way to handle missing values for categorical variables. All of the missing values now have a value of 2:

```
wagedatasdf = \
    wagedatasdf.chat("Impute missings based on most freq"
    wagedatasdf.hdegnm.value_counts(dropna=False).sort_i
```

```
hdegnm
0    877
1   1167
2   3563
3    766
4   1713
5    704
6     64
```

```
7    130
Name: count, dtype: int64
```

8. We could have used the built-in `SmartDataframe` function, `impute_missing_values`, instead. This will use forward fill to impute missing values. No values are imputed for the highest degree variable, `hdegnom`.

```
wagedatasdf = SmartDataframe(wagedata, config={"llm":  
wagedatasdf = \  
    wagedatasdf.impute_missing_values()  
wagedatasdf.chat("Show the counts, means, and standar
```

|               | count | mean   | std    |
|---------------|-------|--------|--------|
| wageincome20  | 8,983 | 62,247 | 59,559 |
| weeksworked20 | 8,983 | 39     | 21     |
| parentincome  | 8,982 | 46,096 | 42,632 |

9. We can use KNN for missing value imputation for the income and weeks worked variables. We start over with an unchanged DataFrame. After the imputation, the `wageincome20` mean is lower than it was originally, as shown in *step 4*. This is not surprising, since we have seen in other recipes that individuals with missing `wageincome20` have lower values for other values correlated with `wageincome20`. The reduction in the standard deviation for `wageincome20` and `parentincome` is not great. The mean and standard deviation for `weeksworked20` are largely unchanged, which is good.

```
wagedatasdf = SmartDataframe(wagedata, config={"llm":  
wagedatasdf = wagedatasdf.chat("Impute missings for f
```

```
wagedatasdf.chat("Show the counts, means, and standar
```

|               | Counts | Means  | Std Devs |
|---------------|--------|--------|----------|
| hdegnm        | 8952   | NaN    | NaN      |
| parentincome  | 8984   | 44,805 | 36,344   |
| wageincome20  | 8984   | 58,356 | 47,378   |
| weeksworked20 | 8984   | 38     | 18       |

## How it works...

Whenever we pass a natural language command to the `chat` method of a `SmartDataframe`, pandas code is generated to run that command. Some of that is very familiar code to generate summary statistics. However, it also can generate code to run machine learning algorithms such as KNN or random forest. As discussed in previous chapters, it is always a good idea to review the `pandasai.log` file after running `chat` to understand the code that was created.

This recipe demonstrated how to use PandasAI to identify and impute values where they are missing. AI tools, particularly large language models, make it easy to pass natural language commands to generate code like the code we created earlier in this chapter.

## Summary

We have explored the most popular approaches for missing value imputation in this chapter, and have discussed the advantages and disadvantages of each approach. Assigning an overall sample mean is not usually a good approach, particularly when observations with missing

values are different from other observations in important ways. We also can substantially reduce our variance. Forward or backward filling allows us to maintain the variance in our data, but works best when the proximity of observations is meaningful, such as with time series or longitudinal data. In most non-trivial cases we will want to use a multivariate technique, such as regression, KNN, or random forest imputation. We examined all these approaches in this chapter, and for the next chapter, we will learn about encoding, transforming, and scaling features.

## Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review.  
Scan the QR code below to get a free eBook of your choice.



# 8

## Encoding, Transforming, and Scaling Features

Our data cleaning efforts are often intended to prepare that data for use with a machine learning algorithm. Machine learning algorithms typically require some form of encoding of variables. Our models also often perform better with some form of scaling so that features with higher variability do not overwhelm the optimization. We show examples of that in this chapter and of how standardizing addresses the issue.

Machine learning algorithms typically require some form of encoding of variables. We almost always need to encode our features for algorithms to understand them correctly. For example, most algorithms cannot make sense of the values *female* or *male*, or know not to treat zip codes as ordinal. Although not typically necessary, scaling is often a very good idea when we have features with vastly different ranges. When we are using algorithms that assume a Gaussian distribution of our features, some form of transformation may be required for our features to be consistent with that assumption.

Specifically, we explore the following in this chapter:

- Creating training datasets and avoiding data leakage
- Identifying irrelevant or redundant observations to be removed

- Encoding categorical features: one-hot encoding
- Encoding categorical features: ordinal encoding
- Encoding features with medium or high cardinality
- Transforming features
- Binning features
- $k$ -means binning
- Scaling features

## Technical requirements

You will need pandas, NumPy, and Matplotlib to complete the recipes in this chapter. I used pandas 2.1.4, but the code will run on pandas 1.5.3 or later.

The code in this chapter can be downloaded from the book's GitHub repository, <https://github.com/PacktPublishing/Python-Data-Cleaning-Cookbook-Second-Edition>.

## Creating training datasets and avoiding data leakage

One of the biggest threats to the performance of our models is data leakage. **Data leakage** occurs whenever our models are informed by data that is not in the training dataset. We sometimes inadvertently assist our model training with information that cannot be gleaned from the training data alone, and we end up with a too-rosy assessment of our model's accuracy.

Data scientists do not really intend for this to happen, hence the term “leakage.” This is not a “*don't do it*” kind of discussion. We all know not to

do it. This is more of a “*which steps should I take to avoid the problem?*” discussion. It is actually quite easy to have some data leakage unless we develop routines to prevent it.

For example, if we have missing values for a feature, we might impute the mean across the whole dataset for those values. However, in order to validate our model, we subsequently split our data into training and testing datasets. We would then have accidentally introduced data leakage into our training dataset, since information from the full dataset (the global mean) would have been used.

Data leakage can significantly compromise our model evaluation, making it look like our predictions are much more reliable than they actually are. Among the practices that data scientists have adopted to avoid this is to establish separate training and testing datasets as close to the beginning of the analysis as possible.

### Note



I have mainly used the term *variable* in this book when referring to some statistical property of something that can be counted or measured, such as age or duration of time. I have used *column* when referring to some specific operation or attribute of a column of data in a dataset. In this chapter, I will frequently use the word *feature* to refer to variables used for predictive analysis. In machine learning, we typically refer to features (also known as independent or predictor variables) and targets (also known as dependent or response variables).

# Getting ready

We will work extensively with the scikit-learn library throughout this chapter. You can use `pip` to install scikit-learn with `pip install scikit-learn`. The code in this chapter uses `sklearn` version 0.24.2.

We can use scikit-learn to create training and testing DataFrames for the **National Longitudinal Survey of Youth (NLS)** data.



## Data 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 who were born between 1980 and 1985, with annual follow-ups each year through 2023. For this recipe, I pulled 104 variables on grades, employment, income, and attitudes toward government from the hundreds of data items on the survey. The NLS data can be downloaded for public use from [nlsinfo.org](http://nlsinfo.org).

# How to do it...

We will use scikit-learn to create training and testing data:

1. First, we import the `train_test_split` module from `sklearn` and load the NLS data:

```
import pandas as pd
from sklearn.model_selection import train_test_split
```

```
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
```

2. Then, we can create training and testing DataFrames for the features (`x_train` and `x_test`) and the target (`y_train` and `y_test`). `wageincome20` is the target variable in this example. We set the `test_size` parameter to 0.3 to leave 30% of the observations for testing. We will only work with the **Scholastic Assessment Test (SAT)** and **Grade Point Average (GPA)** data from the NLS. We need to remember to set a value for `random_state` to make sure we will get the same DataFrames should we need to rerun `train_test_split` later:

```
feature_cols = ['satverbal', 'satmath', 'gpascience',
                'gpaenglish', 'gpamath', 'gpaoverall']
x_train, X_test, y_train, y_test = \
    train_test_split(nls97[feature_cols], \
                    nls97[['wageincome20']], test_size=0.3, random_stat
```

3. Let's take a look at the training DataFrames created with `train_test_split`. We get the expected number of observations, 6,288, 70% of the total number of observations in the NLS DataFrame of 8,984:

```
nls97.shape[0]
```

```
8984
```

```
x_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 6288 entries, 639330 to 166002
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   satverbal    1010 non-null   float64
 1   satmath      1010 non-null   float64
 2   gpascience   4022 non-null   float64
 3   gpaenglish   4086 non-null   float64
 4   gpamath      4076 non-null   float64
 5   gpaoverall   4237 non-null   float64
 dtypes: float64(6)
 memory usage: 343.9 KB
```

```
y_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 6288 entries, 639330 to 166002
Data columns (total 1 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   wageincome20 3652 non-null   float64
 dtypes: float64(1)
 memory usage: 98.2 KB
```

4. Let's also look at the testing DataFrames. We get 30% of the observations as we expected:

```
x_test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 2696 entries, 624557 to 201518
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   satverbal    396 non-null   float64
 1   satmath      397 non-null   float64
```

```
2    gpascience    1662 non-null    float64
3    gpaenglish    1712 non-null    float64
4    gpamath       1690 non-null    float64
5    gpaoverall    1767 non-null    float64
dtypes: float64(6)
memory usage: 147.4 KB
```

```
y_test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 2696 entries, 624557 to 201518
Data columns (total 1 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   wageincome20  1549 non-null   float64
dtypes: float64(1)
memory usage: 42.1 KB
```

These steps demonstrated how to create training and testing DataFrames.

## How it works...

For data scientists who use Python, and regularly use machine learning algorithms, `train_test_split` is a very popular option to avoid data leakage while preparing data for analysis.

`train_test_split` will return four DataFrames: a DataFrame with training data consisting of the features or independent variables we intend to use in our analysis, a DataFrame with testing data for those same variables, a DataFrame with training data for our target variable (also known as a response or dependent variable), and a testing DataFrame with that target variable.

The first arguments of `test_train_split` can take DataFrames, NumPy arrays, or some other two-dimensional array-like structure. Here, we pass a pandas DataFrame with our features to the first argument, and then another pandas DataFrame with just our target variable. We also specify that we want the testing data to be 30% of our dataset's rows.

Rows are selected randomly by `test_train_split`. We need to provide a value for `random_state` if we want to reproduce the split.

## See also

When our projects involve predictive modeling and evaluation of these models, our data preparation needs to be part of a machine learning pipeline, which often starts with splitting the data between training and testing data. Scikit-learn provides great tools for constructing machine learning pipelines that go from data preparation all the way through to model evaluation. A good resource for mastering those techniques is my book *Data Cleaning and Exploration with Machine Learning*.

We will use `sklearn`'s `train_test_split` to create separate training and testing DataFrames in the rest of this chapter. Next, we begin our feature engineering work by removing features that are obviously unhelpful because they have the same data as another feature, or there is no variation in the responses.

## Removing redundant or unhelpful features

During the process of data cleaning and manipulation, we often end up with data that is no longer meaningful. Perhaps we subsetted data based on a single feature value and we have retained that feature, even though it now has the same value for all observations. Alternatively, for the subset of the data that we are using, two features have the same value. Ideally, we catch those redundancies during our data cleaning. However, if we do not catch them during that process, we can use the open source `feature-engine` package to help us with that.

There also may be features that are so highly correlated that it is very unlikely that we could build a model that could use all of them effectively. `feature-engine` has a method, `DropCorrelatedFeatures`, that makes it easy to remove a feature when it is highly correlated with another feature.

## Getting ready

We will work extensively with the `feature-engine` and `category_encoders` packages in this chapter. You can use pip to install these packages with `pip install feature-engine` and `pip install category_encoders`. The code in this chapter uses version 1.7.0 of `feature-engine` and version 2.6.3 of `category_encoders`. Note that either `pip install feature-engine` or `pip install feature_engine` will work.

We will work with land temperature data, in addition to the NLS data, in this section. We will only load temperature data for Poland here.

### Data note

The land temperature DataFrame has the average temperature reading (in °C) in 2023 from over 12,000



stations across the world, although 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.ncei.noaa.gov/products/land-based-station/global-historical-climatology-network-monthly>.

## How to do it...

1. Let's import the modules we need from `feature_engine` and `sklearn`, and then load the NLS data and temperature data for Poland. The data from Poland was pulled from a larger dataset of 12,000 weather stations across the world. We use `dropna` to drop observations with any missing data:

```
import pandas as pd
import feature_engine.selection as fesel
from sklearn.model_selection import train_test_split
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
ltpoland = pd.read_csv("data/ltpoland.csv")
ltpoland.set_index("station", inplace=True)
ltpoland.dropna(inplace=True)
```

2. Next, we create training and testing DataFrames, as we did in the previous section:

```
feature_cols = ['satverbal', 'satmath', 'gpascience',
                'gpaenglish', 'gpamath', 'gpaoverall']
```

```
x_train, x_test, y_train, y_test = \
train_test_split(nls97[feature_cols], \
nls97[['wageincome20']], test_size=0.3, random_stat
```

3. We can use the pandas `corr` method to see how these features are correlated:

```
x_train.corr()
```

|            | satverbal  | satmath | gpascience | \ |
|------------|------------|---------|------------|---|
| satverbal  | 1.00       | 0.74    | 0.42       |   |
| satmath    | 0.74       | 1.00    | 0.47       |   |
| gpascience | 0.42       | 0.47    | 1.00       |   |
| gpaenglish | 0.44       | 0.44    | 0.67       |   |
| gpamath    | 0.36       | 0.50    | 0.62       |   |
| gpaoverall | 0.40       | 0.48    | 0.79       |   |
|            | gpaenglish | gpamath | gpaoverall |   |
| satverbal  | 0.44       | 0.36    | 0.40       |   |
| satmath    | 0.44       | 0.50    | 0.48       |   |
| gpascience | 0.67       | 0.62    | 0.79       |   |
| gpaenglish | 1.00       | 0.61    | 0.84       |   |
| gpamath    | 0.61       | 1.00    | 0.76       |   |
| gpaoverall | 0.84       | 0.76    | 1.00       |   |

`gpaoverall` is highly correlated with `gpascience`, `gpaenglish`, and `gpamath`. The `corr` method returns the Pearson coefficients by default. This is fine when we can assume a linear relationship between the features. When this assumption does not make sense, we should consider requesting Spearman coefficients instead. We can do that by passing `spearman` to the method parameter of `corr`.

4. Let's drop features that have a correlation higher than 0.75 with another feature. We pass 0.75 to the `threshold` parameter of `DropCorrelatedFeatures`, and we indicate that we want to use Pearson

coefficients and evaluate all features by setting variables to `None`. We use the `fit` method on the training data and then transform both the training and testing data. The `info` method shows that the resulting training DataFrame (`x_train_tr`) has all of the features except `gpaoverall`, which has a `0.79` and `0.84` correlation with `gpascience` and `gpaenglish`, respectively (`DropCorrelatedFeatures` will evaluate from left to right, so if `gpamath` and `gpaoverall` are highly correlated, it will drop `gpaoverall`).

If `gpaoverall` had been to the left of `gpamath`, it would have dropped `gpamath`):

```
tr = fesel.DropCorrelatedFeatures(variables=None, method='l1')
tr.fit(X_train)
X_train_tr = tr.transform(X_train)
X_test_tr = tr.transform(X_test)
X_train_tr.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 6288 entries, 639330 to 166002
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype  
---  -- 
 0   satverbal   1010 non-null   float64
 1   satmath     1010 non-null   float64
 2   gpascience  4022 non-null   float64
 3   gpaenglish  4086 non-null   float64
 4   gpamath     4076 non-null   float64
dtypes: float64(5)
memory usage: 294.8 KB
```

We would typically evaluate a feature more carefully before deciding to drop it. However, there are times when feature selection is part of a

pipeline and we need to automate the process. This can be done with `DropCorrelatedFeatures`, since all `feature_engine` methods can be brought into a scikit-learn pipeline.

5. Let's now create training and testing DataFrames from the land temperatures data for Poland. The value of `year` is the same for all observations, as is the value for `country`. The value for `latabs` is also the same as for `latitude` for each observation:

```
feature_cols = ['year', 'month', 'latabs', 'latitude', 'e  
    'longitude', 'country']  
X_train, X_test, y_train, y_test = \  
    train_test_split(ltpoland[feature_cols], \  
        ltpoland[['temperature']], test_size=0.3, random_st  
    X_train.sample(5, random_state=99)
```

| station   | year | month | latabs | latitude | elevation | longit |
|-----------|------|-------|--------|----------|-----------|--------|
| SIEDLCE   | 2023 | 11    | 52     | 52       | 152       |        |
| OKECIE    | 2023 | 6     | 52     | 52       | 110       |        |
| BALICE    | 2023 | 1     | 50     | 50       | 241       |        |
| BALICE    | 2023 | 7     | 50     | 50       | 241       |        |
| BIALYSTOK | 2023 | 11    | 53     | 53       | 151       |        |

```
X_train.year.value_counts()
```

```
year  
2023    84  
Name: count, dtype: int64
```

```
X_train.country.value_counts()
```

```
country
Poland    84
Name: count, dtype: int64
```

```
(X_train.latitude!=X_train.latabs).sum()
```

```
0
```

6. Let's drop features with the same value throughout the training dataset.

Notice that `year` and `country` are removed after the transform:

```
tr = fesel.DropConstantFeatures()
tr.fit(X_train)
X_train_tr = tr.transform(X_train)
X_test_tr = tr.transform(X_test)
X_train_tr.head()
```

|           | month | latabs | latitude | elevation | longitude |
|-----------|-------|--------|----------|-----------|-----------|
| station   |       |        |          |           |           |
| OKECIE    | 1     | 52     | 52       | 110       | 21        |
| LAWICA    | 8     | 52     | 52       | 94        | 17        |
| LEBA      | 11    | 55     | 55       | 2         | 18        |
| SIEDLCE   | 10    | 52     | 52       | 152       | 22        |
| BIALYSTOK | 11    | 53     | 53       | 151       | 23        |

7. Let's drop features that have the same values as other features. In this case, the transform drops `latitude`, which has the same values as

`latabs`:

```
tr = fesel.DropDuplicateFeatures()
tr.fit(X_train_tr)
X_train_tr = tr.transform(X_train_tr)
X_train_tr.head()
```

| station   | month | latabs | elevation | longitude |
|-----------|-------|--------|-----------|-----------|
| OKECIE    | 1     | 52     | 110       | 21        |
| LAWICA    | 8     | 52     | 94        | 17        |
| LEBA      | 11    | 55     | 2         | 18        |
| SIEDLCE   | 10    | 52     | 152       | 22        |
| BIALYSTOK | 11    | 53     | 151       | 23        |

## How it works...

This fixes some obvious problems with our features in the NLS data and the temperature data for Poland. We dropped `gpaoverall` from a DataFrame that has the other GPA features because it is highly correlated with them. We also removed redundant data, dropping features with the same value throughout the DataFrame and features that duplicate the values of another feature.

In step 6, we used the `fit` method of the *feature engine selection* object. This gathers the information needed to do the transformation we request after that. The transformation in this case is to drop features with constant values. We typically perform the fitting on training data only. We can combine the fit and transform on the training data by using `fit_transform`, which we will do in most of this chapter.

The rest of this chapter explores somewhat messier feature engineering challenges: encoding, transforming, binning, and scaling.

## Encoding categorical features: one-hot encoding

There are several reasons why we may need to encode features before using them in most machine learning algorithms. First, these algorithms typically require numeric data. Second, when a categorical feature is represented with numbers, for example, 1 for female and 2 for male, we need to encode the values so that they are recognized as categorical. Third, the feature might actually be ordinal, with a discrete number of values that represent some meaningful ranking. Our models need to capture that ranking. Finally, a categorical feature might have a large number of values (known as high cardinality), and we might want our encoding to collapse categories.

We can handle the encoding of features with a limited number of values, say 15 or fewer, with one-hot encoding. We go over one-hot encoding in this recipe and then discuss ordinal encoding in the next recipe. We will look at strategies for handling categorical features with high cardinality after that.

One-hot encoding takes a feature and creates a binary vector for each value of that feature. So, if a feature, called *letter*, has three unique values, *A*, *B*, and *C*, one-hot encoding creates three binary vectors to represent those values. The first binary vector, which we can call *letter\_A*, has 1 whenever *letter* has a value of *A*, and 0 when it is *B* or *C*. *letter\_B* and *letter\_C* would be coded similarly. The transformed features, *letter\_A*, *letter\_B*, and *letter\_C*, are often referred to as **dummy variables**. *Figure 8.1* illustrates one-hot encoding.

| letter | letter_A | letter_B | letter_C |
|--------|----------|----------|----------|
| A      | 1        | 0        | 0        |
| B      | 0        | 1        | 0        |

|   |   |   |   |
|---|---|---|---|
| C | 0 | 0 | 1 |
|---|---|---|---|

Figure 8.1: One-hot encoding of a categorical feature

## Getting ready

We will use the `OneHotEncoder` and `OrdinalEncoder` modules in the next two recipes from `feature_engine` and `scikit_learn`, respectively. We will continue working with the NLS data.

## How to do it...

A number of features from the NLS data are appropriate for one-hot encoding. We encode some of those features in the following code blocks:

1. Let's start by importing the `OneHotEncoder` module from `feature_engine` and loading the data. We also import the `OrdinalEncoder` module from `scikit-learn`, since we will use it later.

```
import pandas as pd
from feature_engine.encoding import OneHotEncoder
from sklearn.preprocessing import OrdinalEncoder
from sklearn.model_selection import train_test_split
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
```

2. Next, we create training and testing DataFrames for the NLS data.

For our purposes in this recipe, we drop rows with missing data:

```
feature_cols = ['gender', 'maritalstatus', 'colenroct99']
nls97demo = nls97[['wageincome20'] + feature_cols].dropna()
X_demo_train, X_demo_test, y_demo_train, y_demo_test = \
```

```
train_test_split(nls97demo[feature_cols], \
nls97demo[['wageincome20']], test_size=0.3, random_state
```

3. One option we have for the encoding is the pandas `get_dummies` method. We can use it to indicate that we want to convert the `gender` and `maritalstatus` features. `get_dummies` gives us a dummy variable for each value of `gender` and `maritalstatus`. For example, `gender` has the values Female and Male. `get_dummies` creates a feature, `gender_Female`, which is 1 when `gender` is Female and 0 when `gender` is Male. When `gender` is Male, `gender_Male` is 1 and `gender_Female` is 0. This is a tried-and-true method of doing this encoding that has served statisticians well for many years:

```
pd.get_dummies(X_demo_train,
columns=['gender', 'maritalstatus'], dtype=float).\
head(2).T
```

|                             | 606986            | 764231          |
|-----------------------------|-------------------|-----------------|
| colenroct99                 | 3. 4-year college | 1. Not enrolled |
| gender_Female               | 1                 | 0               |
| gender_Male                 | 0                 | 1               |
| maritalstatus_Divorced      | 0                 | 0               |
| maritalstatus_Married       | 0                 | 1               |
| maritalstatus_Never-married | 1                 | 0               |
| maritalstatus_Separated     | 0                 | 0               |
| maritalstatus_Widowed       | 0                 | 0               |

We are not creating a new DataFrame with `get_dummies` because we will be using a different technique to do the encoding later in this recipe.

We typically create  $k-1$  dummy variables for  $k$  unique values for a feature. So, if `gender` has two values in our data, we only need to create one dummy variable. If we know the value for `gender_Female`, we also know the value of `gender_Male`, so the latter variable is redundant. Similarly, we know the value of `maritalstatus_Divorced` if we know the values of the other `maritalstatus` dummies. Creating a redundancy in this way is inelegantly referred to as the **dummy variable trap**. To prevent this problem, we drop one dummy from each group.

### Note



For some machine learning algorithms, such as linear regression, dropping one dummy variable is actually required. In estimating the parameters of a linear model, the matrix is inverted. If our model has an intercept, and all dummy variables are included, the matrix cannot be inverted.

4. We can set the `get_dummies` `drop_first` parameter to `True` to drop the first dummy from each group:

```
pd.get_dummies(X_demo_train,  
                columns=['gender', 'maritalstatus'], dtype=float,  
                drop_first=True).head(2).T
```

|                             | 606986            | 764231          |
|-----------------------------|-------------------|-----------------|
| colenroct99                 | 3. 4-year college | 1. Not enrolled |
| gender_Male                 | 0                 | 1               |
| maritalstatus_Married       | 0                 | 1               |
| maritalstatus_Never-married | 1                 | 0               |

|                         |   |   |
|-------------------------|---|---|
| maritalstatus_Separated | 0 | 0 |
| maritalstatus_Widowed   | 0 | 0 |

An alternative to `get_dummies` is the one-hot encoder in either `sklearn` or `feature_engine`. These one-hot encoders have the advantage that they can be easily brought into a machine learning pipeline, and they can persist information gathered from the training dataset to the testing dataset.

5. Let's use the `OneHotEncoder` from `feature_engine` to do the encoding. We set `drop_last` to `True` to drop one of the dummies from each group. We fit the encoding to the training data and then transform both the training and testing data.

```
ohe = OneHotEncoder(drop_last=True,
                     variables=['gender', 'maritalstatus'])
ohe.fit(X_demo_train)
X_demo_train_ohe = ohe.transform(X_demo_train)
X_demo_test_ohe = ohe.transform(X_demo_test)
X_demo_train_ohe.filter(regex='gen|mar', axis="column")
```

|                             | 606986 | 764231 |
|-----------------------------|--------|--------|
| gender_Female               | 1      | 0      |
| maritalstatus_Never-married | 1      | 0      |
| maritalstatus_Married       | 0      | 1      |
| maritalstatus_Divorced      | 0      | 0      |
| maritalstatus_Separated     | 0      | 0      |

This demonstrates that one-hot encoding is a fairly straightforward way to prepare nominal data for a machine learning algorithm.

# How it works...

The pandas `get_dummies` method is a handy way to create dummy variables or one-hot encoding. We saw this in *step 3* where we simply passed the training DataFrame, and the columns where we want dummy variables, to `get_dummies`. Notice that we used `float` for `dtype`. Depending on your version of pandas, this is necessary to return 0 and 1 values rather than true and false values.

We typically need to remove one of the values in a dummy variable group to avoid the *dummy variable trap*. We can set `drop_first` to `True` to drop the first dummy variable from each dummy variable group. We did that in *step 4*.

We looked at another tool for one-hot encoding, `feature_engine`, in *step 5*. We are able to accomplish the same task as `get_dummies` using `feature_engine`'s `OneHotEncoder`. The advantage of using `feature_engine` is its variety of tools for working within scikit-learn data pipelines, including being able to handle categories in either the training or testing DataFrame, but not in both.

# There's more

I did not discuss scikit-learn's own one-hot encoder in this recipe. It works very much like the one-hot encoder for `feature_engine`. There is not much advantage of using one rather than the other, although I find it handy that `feature_engine`'s `transform` and `fit_transform` methods return DataFrames, whereas those methods for scikit-learn return a NumPy array.

# Encoding categorical features: ordinal encoding

Categorical features can be either nominal or ordinal. Gender and marital status are nominal. Their values do not imply order. For example, never married is not a higher value than divorced.

When a categorical feature is ordinal, however, we want the encoding to capture the ranking of the values. For example, if we have a feature that has the values low, medium, and high, one-hot encoding would lose this ordering. Instead, a transformed feature with values of 1, 2, and 3 for low, medium, and high, respectively, would be better. We can accomplish this with ordinal encoding.

The college enrollment feature on the NLS dataset can be considered an ordinal feature. The values range from *1. Not enrolled* to *3. 4-year college*. We should use ordinal encoding to prepare it for modeling. We do that next.

## Getting ready

We will use the `OrdinalEncoder` module in this recipe from `scikit-learn`.

## How to do it...

1. College enrollment for 1999 might be a good candidate for ordinal encoding. Let's first take a look at the values of `colenroct99` prior to encoding. The values are strings, but there is an implied order:

```
x_demo_train.colenroct99.\n    sort_values().unique()
```

```
array(['1. Not enrolled', '2. 2-year college ', '3. 4-year
       dtype=object)
```

```
x_demo_train.head()
```

```
      gender  maritalstatus      colenroct99
606986  Female  Never-married  3. 4-year college
764231    Male        Married  1. Not enrolled
673419    Male  Never-married  3. 4-year college
185535    Male        Married  1. Not enrolled
903011    Male  Never-married  1. Not enrolled
```

We need to be careful about assumptions of linearity here. For example, if we are trying to model the impact of the college enrollment feature on some target variable, we cannot assume that movement from 1 to 2 (from not enrolled in college to enrolled for 2 years in college) has the same impact as movement from 2 to 3 (from 2-year college to 4-year college enrollment).

2. We can tell the `OrdinalEncoder` to rank the values in the implied order by passing the above array to the `categories` parameter. We can then use `fit_transform` to transform the college enrollment field `colenroct99`. (The `fit_transform` method of sklearn's `OrdinalEncoder` returns a NumPy array, so we need to use the pandas DataFrame method to create a DataFrame.) Finally, we join the encoded features with the other features from the training data:

```
oe = OrdinalEncoder(categories=\n    [X_demo_train.colenroct99.sort_values().\\n        unique()])\ncolenr_enc = \\n    pd.DataFrame(oe.fit_transform(X_demo_train[['colenr
```

```
    columns=['colenroct99'], index=X_demo_train.index  
X_demo_train_enc = \  
    X_demo_train[['gender','maritalstatus']].\  
    join(colenr_enc)
```

3. Let's view a few observations from the resulting DataFrame. We should also compare the counts of the original college enrollment feature to the transformed feature:

```
X_demo_train_enc.head()
```

```
      gender  maritalstatus  colenroct99  
606986  Female  Never-married          2  
764231     Male       Married          0  
673419     Male  Never-married          2  
185535     Male       Married          0  
903011     Male  Never-married          0
```

```
X_demo_train.colenroct99.value_counts().\  
sort_index()
```

```
      colenroct99  
1. Not enrolled      2843  
2. 2-year college     137  
3. 4-year college     324  
Name: count, dtype: int64
```

```
X_demo_train_enc.colenroct99.value_counts().\  
sort_index()
```

```
      colenroct99  
0              2843  
1              137
```

```
2           324  
Name: count, dtype: int64
```

The ordinal encoding replaces the initial values for `colernoct99` with numbers from 0 to 2. It is now in a form that is consumable by many machine learning models, and we have retained the meaningful ranking information.

## How it works...

*Scikit-learn's* `OrdinalEncoder` is fairly straightforward to use. We passed an array of values to use for the categories that is sorted in a meaningful order. We did this at the start of *step 2* when we instantiated an `OrdinalEncoder` object. We then passed training data with just the `colenroct99` column to the `fit_transform` method of the `OrdinalEncoder`. We then converted the NumPy array returned by `fit_transform` to a DataFrame, using the training data index, and we used `join` to append the rest of the training data.

## There's more

Ordinal encoding is appropriate for non-linear models such as decision trees. It might not make sense in a linear regression model because that would assume that the distance between values was equally meaningful along the whole distribution. In this example, that would assume that the increase from 0 to 1 (from no enrollment to 2-year enrollment) is the same thing as the increase from 1 to 2 (from 2-year enrollment to 4-year enrollment).

One-hot and ordinal encoding are relatively straightforward approaches to engineering categorical features. It can be more complicated to deal with categorical features when there are many more unique values. We go over a couple of techniques for handling those features in the next section.

## Encoding categorical features with medium or high cardinality

When we are working with a categorical feature that has many unique values, say 15 or more, it can be impractical to create a dummy variable for each value. When there is high cardinality, a very large number of unique values, there may be too few observations with certain values to provide much information for our models. At the extreme, with an ID variable, there is just one observation for each value.

There are a couple of ways to handle medium or high cardinality. One is to create dummies for the top k categories and group the remaining values into an *other* category. Another is to use feature hashing, also known as the hashing trick. We will explore both strategies in this recipe.

## Getting ready

We continue to use the `OneHotEncoder` from `feature_engine` in this recipe. We will also use the `HashingEncoder` from `category_encoders`. We will be working with COVID-19 data in this recipe, which has total cases and deaths by country, as well as demographic data.



### Data note



Our World in Data provides COVID-19 public use data at  
<https://ourworldindata.org/covid-cases>.

The data used in this recipe was downloaded on March 3, 2024.

## How to do it...

1. Let's create training and testing DataFrames from the COVID-19 data, and then import the `feature_engine` and `category_encoders` libraries:

```
import pandas as pd
from feature_engine.encoding import OneHotEncoder
from category_encoders.hashing import HashingEncoder
from sklearn.model_selection import train_test_split
covidtotals = pd.read_csv("data/covidtotals.csv")
feature_cols = ['location', 'population',
                 'aged_65_older', 'life_expectancy', 'region']
covidtotals = covidtotals[['total_cases']] + feature_c
X_train, X_test, y_train, y_test = \
    train_test_split(covidtotals[feature_cols],\
        covidtotals[['total_cases']], test_size=0.3, random
```

The feature region has 16 unique values, the first 5 of which have counts of 10 or more:

```
X_train.region.value_counts()
```

| region         |    |
|----------------|----|
| Eastern Europe | 15 |
| Western Europe | 15 |
| West Asia      | 12 |
| South America  | 11 |
| Central Africa | 10 |

```
East Asia          9
Caribbean         9
Oceania / Aus     9
West Africa        7
Southern Africa    7
Central Asia       6
South Asia         6
East Africa        5
Central America    5
North Africa        4
North America      1
Name: count, dtype: int64
```

2. We can use the `OneHotEncoder` from `feature_engine` again to encode the `region` feature. This time, we use the `top_categories` parameter to indicate that we only want to create dummies for the top 6 category values. Any values for `region` that do not fall into the top 6 will have a 0 value for all of the dummies:

```
ohe = OneHotEncoder(top_categories=6, variables=['reg
covidtotals_ohe = ohe.fit_transform(covidtotals)
covidtotals_ohe.filter(regex='location|region',
axis="columns").sample(5, random_state=2).T
```

|                       | 31       | 157       | 2       | 17    |
|-----------------------|----------|-----------|---------|-------|
| location              | Bulgaria | Palestine | Algeria | Russi |
| region_Eastern Europe | 1        | 0         | 0       |       |
| region_Western Europe | 0        | 0         | 0       |       |
| region_West Africa    | 0        | 0         | 0       |       |
| region_West Asia      | 0        | 1         | 0       |       |
| region_East Asia      | 0        | 0         | 0       |       |
| region_Caribbean      | 0        | 0         | 0       |       |

An alternative approach to one-hot encoding, when a categorical feature has many unique values, is to use **feature hashing**.

Feature hashing maps a large number of unique feature values to a smaller number of dummy variables. We can specify the number of dummy variables to create. Each feature value maps to one and only one dummy variable combination. However, collisions are possible—that is, some feature values might map to the same dummy variable combination. The number of collisions increases as we decrease the number of requested dummy variables.

3. We can use the `HashingEncoder` from `category_encoders` to do feature hashing. We use `n_components` to indicate that we want 6 dummy variables (we copy the `region` feature before we do the transform so that we can compare the original values to the new dummies):

```
x_train['region2'] = X_train.region
he = HashingEncoder(cols=['region'], n_components=6)
X_train_enc = he.fit_transform(X_train)
X_train_enc.\n    groupby(['col_0','col_1','col_2','col_3','col_4',
        'col_5','region2']).\
    size().reset_index(name="count")
```

|   | col_0 | col_1 | col_2 | col_3 | col_4 | col_5 | region         |
|---|-------|-------|-------|-------|-------|-------|----------------|
| 0 | 0     | 0     | 0     | 0     | 0     | 1     | Caribbea       |
| 1 | 0     | 0     | 0     | 0     | 0     | 1     | Central Afric  |
| 2 | 0     | 0     | 0     | 0     | 0     | 1     | East Afric     |
| 3 | 0     | 0     | 0     | 0     | 0     | 1     | North Afric    |
| 4 | 0     | 0     | 0     | 0     | 1     | 0     | Central Americ |
| 5 | 0     | 0     | 0     | 0     | 1     | 0     | Eastern Europ  |
| 6 | 0     | 0     | 0     | 0     | 1     | 0     | North Americ   |
| 7 | 0     | 0     | 0     | 0     | 1     | 0     | Oceania / Au   |
| 8 | 0     | 0     | 0     | 0     | 1     | 0     | Southern Afric |
| 9 | 0     | 0     | 0     | 0     | 1     | 0     | West Asi       |

|    |   |   |   |   |   |   |                |
|----|---|---|---|---|---|---|----------------|
| 10 | 0 | 0 | 0 | 0 | 1 | 0 | Western Europe |
| 11 | 0 | 0 | 0 | 1 | 0 | 0 | Central Asia   |
| 12 | 0 | 0 | 0 | 1 | 0 | 0 | East Asia      |
| 13 | 0 | 0 | 0 | 1 | 0 | 0 | South Asia     |
| 14 | 0 | 0 | 1 | 0 | 0 | 0 | West Africa    |
| 15 | 1 | 0 | 0 | 0 | 0 | 0 | South America  |

This gives us a large number of collisions, unfortunately. For example, Caribbean, Central Africa, East Africa, and North Africa all get the same dummy variable values. In this case at least, using one-hot encoding and specifying the number of categories, or increasing the number of components for the hashing encoder, would give us better results.

## How it works...

We used the `OneHotEncoder` from `feature_engine` in the same way we did in the *Encoding categorical features: one-hot encoding* recipe. The difference here is that we limited the dummies to the six regions with the most number of rows (countries in this case). All countries not in one of the top six regions got zeroes for all dummies, such as Algeria in *step 2*.

In *step 3*, we used the `HashingEncoder` from `category_encoders`. We indicated the column to use, `region`, and that we wanted six dummies. We used the `fit_transform` method of the `HashingEncoder` to fit and transform our data, just as we did with the `OneHotEncoder` of `feature_engine` and the `OrdinalEncoder` of scikit-learn.

We have covered common encoding strategies in the last three recipes: one-hot encoding, ordinal encoding, and feature hashing. Almost all of our categorical features will require some kind of encoding before we can use them in a model. But we sometimes need to alter our features in other ways,

including with transformations, binning, and scaling. We consider the reasons why we might need to alter our features in these ways, and explore tools to do that, in the next three recipes.

# Using mathematical transformations

We sometimes want to use features that do not have a Gaussian distribution with a machine learning algorithm that assumes our features are distributed in that way. When that happens, we either need to change our minds about which algorithm to use (choose KNN or random forest rather than linear regression, for example) or transform our features so that they approximate a Gaussian distribution. We go over a couple of strategies for doing the latter in this recipe.

## Getting ready

We will use the transformation module from feature engine in this recipe. We continue to work with the COVID-19 data, which has one row for each country with the total cases and deaths and some demographic data.

## How to do it...

1. We start by importing the `transformation` module from `feature_engine`, `train_test_split` from `sklearn`, and `stats` from `scipy`. We also create a training and testing DataFrame with the COVID-19 data:

```
import pandas as pd
from feature_engine import transformation as vt
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from scipy import stats
covidtotals = pd.read_csv("data/covidtotals.csv")
feature_cols = ['location', 'population',
                 'aged_65_older', 'life_expectancy', 'region']
covidtotals = covidtotals[['total_cases']] + feature_c
X_train, X_test, y_train, y_test = \
    train_test_split(covidtotals[feature_cols],\
        covidtotals[['total_cases']], test_size=0.3, random
```

2. Let's take a look at how total cases by country are distributed. We should also calculate the skew:

```
y_train.total_cases.skew()
```

```
6.092053479609332
```

```
plt.hist(y_train.total_cases)
plt.title("Total COVID-19 Cases (in millions)")
plt.xlabel('Cases')
plt.ylabel("Number of Countries")
plt.show()
```

This produces the following histogram:

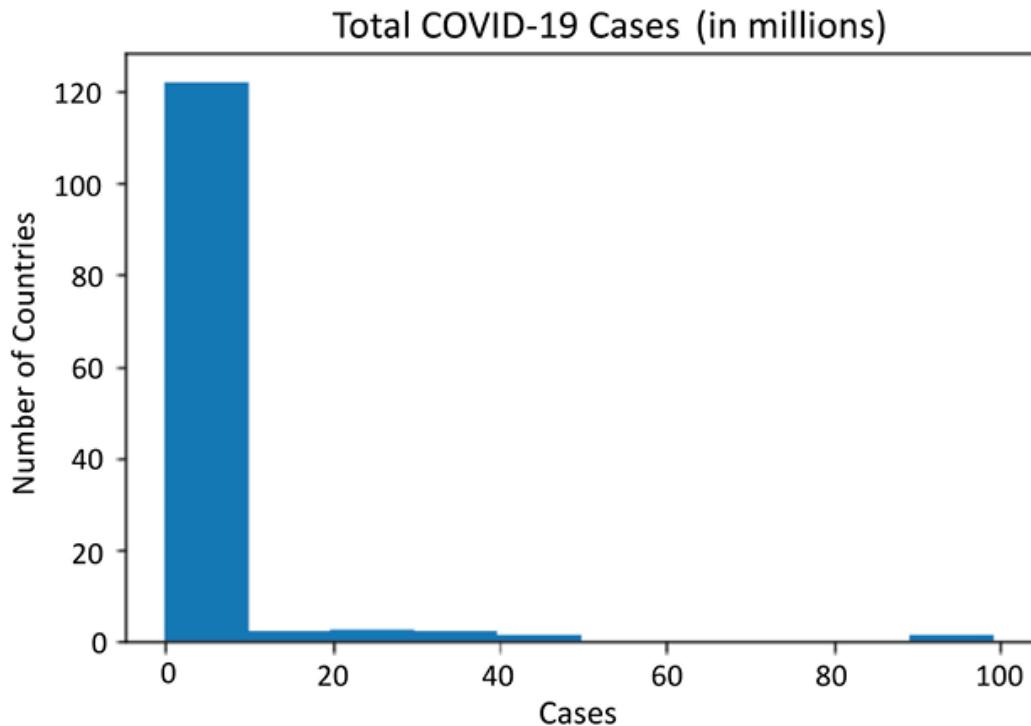


Figure 8.1: Histogram of total COVID-19 cases

This illustrates the very high skew for total cases. It actually looks log-normal, which is not surprising given the large number of very low values and several very high values.

3. Let's try a log transformation. All we need to do to get `feature_engine` to do the transformation is to call `LogTransformer` and pass the feature or features we would like to transform:

```
tf = vt.LogTransformer(variables = ['total_cases'])
y_train_tf = tf.fit_transform(y_train)
y_train_tf.total_cases.skew()
```

```
0.09944093918837159
```

```
plt.hist(y_train_tf.total_cases)
plt.title("Total COVID-19 Cases (log transformation)")
```

```
plt.xlabel('Cases')
plt.ylabel("Number of Countries")
plt.show()
```

This produces the following histogram:

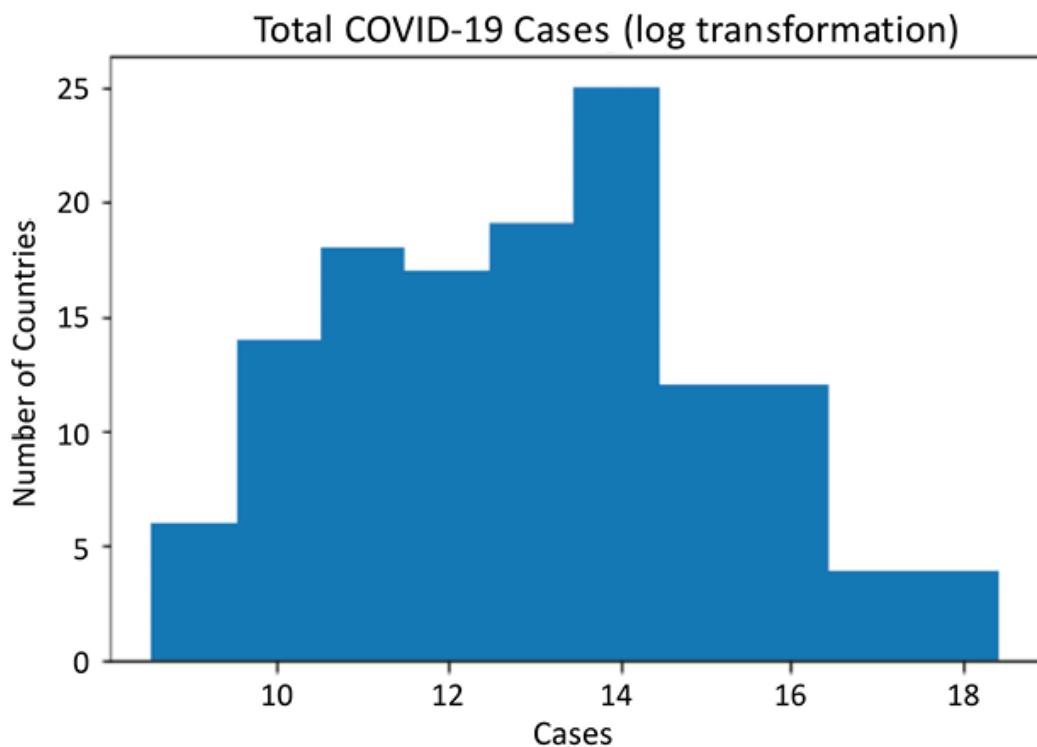


Figure 8.2: Histogram of total COVID-19 cases with log transformation

Effectively, log transformations increase variability at the lower end of the distribution and decrease variability at the upper end. This produces a more symmetrical distribution. This is because the slope of the logarithmic function is steeper for smaller values than for larger ones.

4. This is definitely a big improvement, but let's also try a Box-Cox transformation to see what results we get:

```
tf = vt.BoxCoxTransformer(variables = ['total_cases'])
y_train_tf = tf.fit_transform(y_train)
y_train_tf.total_cases.skew()
```

```
0.010531307863482307
```

```
plt.hist(y_train_tf.total_cases)
plt.title("Total COVID-19 Cases (Box Cox transformation")
plt.xlabel('Cases')
plt.ylabel("Number of Countries")
plt.show()
```

This produces the following histogram:

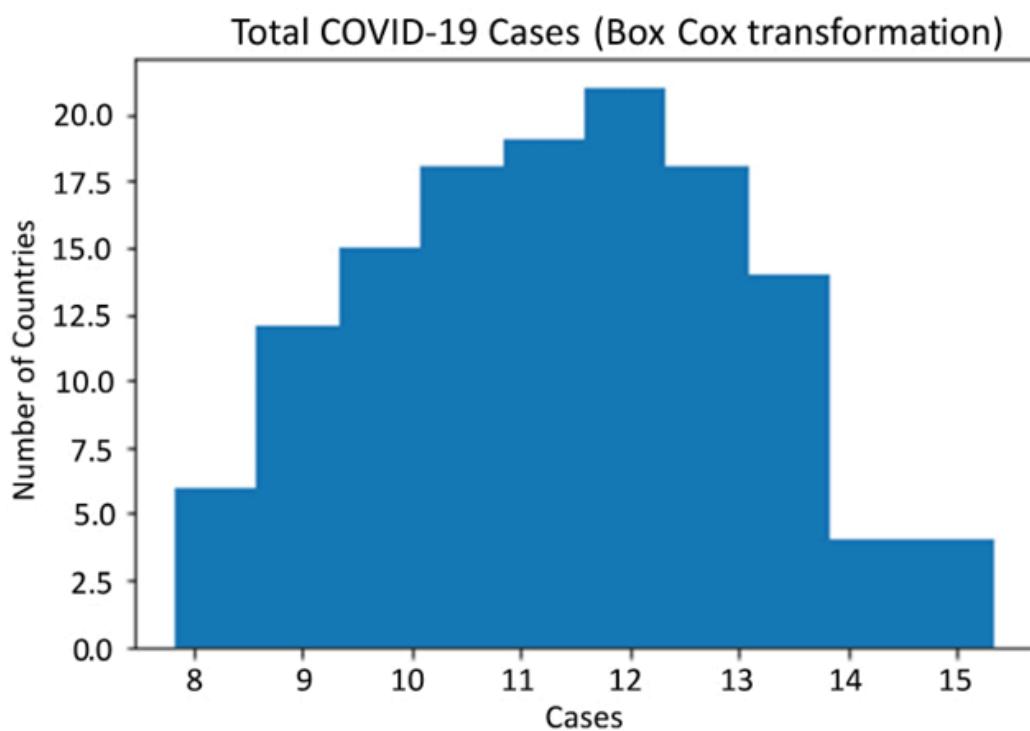


Figure 8.3: Histogram of total COVID-19 cases with a Box-Cox transformation

Box-Cox transformations identify a value for lambda between -5 and 5 that generates a distribution that is closest to normal. It uses the following equation for the transformation:

$$y(\lambda) = (y\lambda - 1) / \lambda, \text{if } \lambda \neq 0$$

or

$$y(\lambda) = \log y, \text{if } \lambda = 0$$

Where  $y(\lambda)$  is our transformed feature. Just for fun, let's see the value of lambda that was used to transform `total_cases`:

```
stats.boxcox(y_train.total_cases)[1]
```

```
-0.020442184436288167
```

The lambda for the Box-Cox transformation is -0.02. For comparison, the lambda for a feature with a Gaussian distribution would be 1.000, meaning that no transformation would be necessary.

## How it works...

Many of our research or modeling projects require some transformation of features or target variables for us to produce good results. Tools like *feature engine* make it easy for us to incorporate such transformations in our data preparation process. We imported the `transformation` module in *step 1*, and then we used it to do a log transformation in *step 3* and a Box-Cox transformation in *step 4*.

The transformed total cases feature looked good after the log and Box-Cox transformations. This will likely be an easier target to model. It is also easy to integrate this transformation with a pipeline with other preprocessing steps. `Feature_engine` has a number of other transformations that are implemented similarly to the log and Box-Cox transformations.

## See also

You may be wondering how we make predictions or evaluate a model with a transformed target. It is actually fairly straightforward to set up our pipeline to restore values to their original values when we make predictions. I go over this in detail in my book *Data Cleaning and Exploration with Machine Learning*.

# Feature binning: equal width and equal frequency

We sometimes want to convert a feature from continuous to categorical. The process of creating  $k$  equally spaced intervals from the minimum to the maximum value of a distribution is called **binning**, or the somewhat less friendly **discretization**. Binning can address several important issues with a feature: skew, excessive kurtosis, and the presence of outliers.

## Getting ready

Binning might be a good choice with the COVID-19 total cases data. It might also be useful with other variables in the dataset, including total deaths and population, but we will only work with total cases for now.

`total_cases` is the target variable in the following code, so it is a column—the only column—on the `y_train` DataFrame.

Let's try equal width and equal frequency binning with the COVID-19 data.

## How to do it...

1. We first need to import the `EqualFrequencyDiscretiser` and `EqualWidthDiscretiser` from `feature_engine`. We also need to create training and testing DataFrames from the COVID-19 data:

```
import pandas as pd
from feature_engine.discretisation import EqualFrequencyDiscretiser
from feature_engine.discretisation import EqualWidthDiscretiser
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.model_selection import train_test_split
covidtotals = pd.read_csv("data/covidtotals.csv")
feature_cols = ['location', 'population',
                 'aged_65_older', 'life_expectancy', 'region']
covidtotals = covidtotals[['total_cases']] + feature_cols
X_train, X_test, y_train, y_test = \
    train_test_split(covidtotals[feature_cols], \
                     covidtotals[['total_cases']], test_size=0.3, random_state=42)
```

2. We can use the pandas `qcut` method, and its `q` parameter, to create 10 bins of relatively equal frequency:

```
y_train['total_cases_group'] = \
    pd.qcut(y_train.total_cases, q=10,
            labels=[0,1,2,3,4,5,6,7,8,9])
y_train.total_cases_group.value_counts().\
    sort_index()
```

```
total_cases_group
0    14
1    13
2    13
3    13
4    13
5    13
6    13
7    13
8    13
9    13
Name: count, dtype: int64
```

3. We can accomplish the same thing with the `EqualFrequencyDiscretiser`. First, we define a function to run the transformation. The function takes a `feature_engine` transformation and the training and testing DataFrames. It returns the transformed DataFrames (it is not necessary to define a function, but it makes sense here, since we will repeat these steps later in this recipe):

```
def runtransform(bt, dftrain, dftest):
    bt.fit(dftrain)
    train_bins = bt.transform(dftrain)
    test_bins = bt.transform(dftest)
    return train_bins, test_bins
```

4. Next, we create an `EqualFrequencyDiscretiser` transformer and call the `runtransform` function we just created:

```
y_train.drop(['total_cases_group'], axis=1, inplace=True)
bintransformer = efd(q=10, variables=['total_cases'])
y_train_bins, y_test_bins = runtransform(bintransformer, y_train, y_test)
y_train_bins.total_cases.value_counts().sort_index()
```

```
total_cases
0    14
1    13
2    13
3    13
4    13
5    13
6    13
7    13
8    13
9    13
Name: count, dtype: int64
```

This gives us the same results as `qcut`, but it has the advantage that it is easier to bring into a machine learning pipeline, since we are using `feature_engine` to produce it. The equal frequency binning addresses both the skew and outlier problems.

5. The `EqualWidthDiscretiser` works similarly:

```
bintransformer = ewd(bins=10, variables=['total_cases'])
y_train_bins, y_test_bins = runtransform(bintransformer)
y_train_bins.total_cases.value_counts().sort_index()
```

```
total_cases
0    122
1      2
2      3
3      2
4      1
9      1
Name: count, dtype: int64
```

This is a far less successful transformation. Almost all of the values are at the bottom of the distribution in the data prior to the binning, so

it is not surprising that equal width binning would have the same problem. It results in only 6 bins, despite the fact that we have requested 10.

6. Let's examine the range of each bin. We can see here that the equal width binner is not even able to construct equal width bins because of the small number of observations at the top of the distribution:

```
y_train_bins = y_train_bins.\  
    rename(columns={'total_cases':'total_cases_group'})  
    join(y_train)  
y_train_bins.groupby("total_cases_group")["total_case  
    agg(['min', 'max'])]
```

|                   | min        | max        |
|-------------------|------------|------------|
| total_cases_group |            |            |
| 0                 | 5,085      | 8,633,769  |
| 1                 | 11,624,000 | 13,980,340 |
| 2                 | 23,774,451 | 26,699,442 |
| 3                 | 37,519,960 | 38,437,756 |
| 4                 | 45,026,139 | 45,026,139 |
| 9                 | 99,329,249 | 99,329,249 |

Although equal width binning was a bad choice in this case, there are many times when it makes sense. It can be useful when data is more uniformly distributed, or when the equal widths make sense substantively.

## k-means binning

Another option is to use  $k$ -means clustering to determine the bins. The  $k$ -means algorithm randomly selects  $k$  data points as centers of clusters, and then it assigns the other data points to the closest cluster. The mean of each

cluster is computed, and the data points are reassigned to the nearest new cluster. This process is repeated until the optimal centers are found.

When  $k$ -means is used for binning, all data points in the same cluster will have the same ordinal value.

## Getting ready

We will use scikit-learn this time for our binning. *Scikit-learn* has a great tool for creating bins based on  $k$ -means, `KBinsDiscretizer`.

## How to do it...

1. We start by instantiating a `KBinsDiscretizer` object. We will use it to create bins with the COVID-19 cases data:

```
kbins = KBinsDiscretizer(n_bins=10, encode='ordinal',
                         strategy='kmeans', subsample=None)
y_train_bins = \
    pd.DataFrame(kbins.fit_transform(y_train),
                 columns=['total_cases'], index=y_train.index)
y_train_bins.total_cases.value_counts().sort_index()
```

|   | total_cases |
|---|-------------|
| 0 | 57          |
| 1 | 19          |
| 2 | 25          |
| 3 | 10          |
| 4 | 11          |
| 5 | 2           |
| 6 | 3           |
| 7 | 2           |
| 8 | 1           |

```
9      1  
Name: count, dtype: int64
```

2. Let's compare the skew and kurtosis of the original total cases variable to that of the binned variable. Recall that we would expect a skew of 0 and a kurtosis near 3 for a variable with a Gaussian distribution. The distribution of the binned variable is much closer to Gaussian:

```
y_train.total_cases.agg(['skew', 'kurtosis'])
```

```
skew      6.092  
kurtosis   45.407  
Name: total_cases, dtype: float64
```

```
y_train_bins.total_cases.agg(['skew', 'kurtosis'])
```

```
skew      1.504  
kurtosis   2.281  
Name: total_cases, dtype: float64
```

3. Let's take a closer look at the range of total cases values in each bin. The first bin goes up to 272,010 total cases, and the next goes up to 834,470. There is a fair bit of drop-off in terms of the number of countries after about 8.6 million total cases. We might consider reducing the number of bins to 5 or 6:

```
y_train_bins.rename(columns={'total_cases': 'total_cas  
y_train.join(y_train_bins).\\
```

```
groupby(['total_cases_bin'])['total_cases'].\\
agg(['min', 'max', 'size'])
```

|                 | min        | max        | size |
|-----------------|------------|------------|------|
| total_cases_bin |            |            |      |
| 0               | 5,085      | 272,010    | 57   |
| 1               | 330,417    | 834,470    | 19   |
| 2               | 994,037    | 2,229,538  | 25   |
| 3               | 2,465,545  | 4,536,733  | 10   |
| 4               | 5,269,967  | 8,633,769  | 11   |
| 5               | 11,624,000 | 13,980,340 | 2    |
| 6               | 23,774,451 | 26,699,442 | 3    |
| 7               | 37,519,960 | 38,437,756 | 2    |
| 8               | 45,026,139 | 45,026,139 | 1    |
| 9               | 99,329,249 | 99,329,249 | 1    |

These steps demonstrate how to use  $k$ -means for binning.

## How it works...

All we need to run  $k$ -means binning is to instantiate a `KBinsDiscretizer` object. We indicated the number of bins we wanted, `10`, and that we wanted the bins to be `ordinal`. We specified `ordinal` because we want higher bin values to reflect higher total cases values. We converted the NumPy array returned from the scikit-learn `fit_transform` into a DataFrame. This is often not necessary in a data pipeline, but we did it here because we will use the DataFrame in subsequent steps.

Binning can help us address skew, kurtosis, and outliers in our data. It does, however, mask much of the variation in the feature and reduces its explanatory potential. Often, some form of scaling, such as min-max or z-score, is a better option. Let's examine feature scaling in the next recipe.

# Feature scaling

Often, the features we want to use in our model are on very different scales. Put another way, the distance between the min and max values, or the range, varies substantially across possible features. In the COVID-19 data for example, the `total cases` feature goes from 5,000 to almost 100 million, while `aged 65 or older` goes from 9 to 27 (the number represents the percent of population).

Having features on very different scales impacts many machine learning algorithms. For example, KNN models often use Euclidean distance, and features with greater ranges will have greater influence on the model. Scaling can address this problem.

We will go over two popular approaches to scaling in this section: **min-max scaling** and **standard** (or **z-score**) scaling. Min-max scaling replaces each value with its location in the range. More precisely:

$$z_{ij} = (x_{ij} - \min_j) / (\max_j - \min_j)$$

Here,  $z_{ij}$  is the min-max score,  $x_{ij}$  is the value for the  $i^{\text{th}}$  observation of the  $j^{\text{th}}$  feature, and  $\min_j$  and  $\max_j$  are the min and max values of the  $j^{\text{th}}$  feature.

Standard scaling normalizes the feature values around a mean of 0. Those who studied undergraduate statistics will recognize it as the z-score. Specifically:

$$z_{ij} = (x_{ij} - u_j) / s_j$$

Here,  $x_{ij}$  is the value for the  $i^{\text{th}}$  observation of the  $j^{\text{th}}$  feature,  $u_j$  is the mean for feature  $j$ , and  $s_j$  is the standard deviation for that feature.

# Getting ready

We will use scikit-learn's preprocessing module for all of the transformations in this recipe. We will work with the COVID-19 data again.

## How to do it...

We can use scikit-learn's preprocessing module to get the min-max and standard scalers:

1. We start by importing the `preprocessing` module and creating training and testing DataFrames from the COVID-19 data:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, StandardScaler
covidtotals = pd.read_csv("data/covidtotals.csv")
feature_cols = ['population', 'total_deaths',
                 'aged_65_older', 'life_expectancy']
covidtotals = covidtotals[['total_cases']] + feature_cols
X_train, X_test, y_train, y_test = \
    train_test_split(covidtotals[feature_cols], \
                     covidtotals[['total_cases']], test_size=0.3, random_state=42)
```

2. We can now run the min-max scaler. As we have done in previous recipes when working with the scikit-learn `fit_transform`, we convert the NumPy array so that it returns to a DataFrame using the columns and index from the training DataFrame. Notice that all features now have values between 0 and 1:

```
scaler = MinMaxScaler()
X_train_mms = pd.DataFrame(scaler.fit_transform(X_train), columns=feature_cols, index=y_train.index)
```

```
    columns=X_train.columns, index=X_train.index)
X_train_mms.describe()
```

|       | population | total_deaths | aged_65_older | life_exp |
|-------|------------|--------------|---------------|----------|
| count | 131.00     | 131.00       | 131.00        | 131.00   |
| mean  | 0.03       | 0.05         | 0.34          |          |
| std   | 0.13       | 0.14         | 0.28          |          |
| min   | 0.00       | 0.00         | 0.00          |          |
| 25%   | 0.00       | 0.00         | 0.11          |          |
| 50%   | 0.01       | 0.01         | 0.24          |          |
| 75%   | 0.02       | 0.03         | 0.60          |          |
| max   | 1.00       | 1.00         | 1.00          |          |

3. We run the standard scaler in the same manner:

```
scaler = StandardScaler()
X_train_ss = pd.DataFrame(scaler.fit_transform(X_train),
                           columns=X_train.columns, index=X_train.index)
X_train_ss.describe()
```

|       | population | total_deaths | aged_65_older | life_exp |
|-------|------------|--------------|---------------|----------|
| count | 131.00     | 131.00       | 131.00        | 131.00   |
| mean  | -0.00      | -0.00        | -0.00         | -0.00    |
| std   | 1.00       | 1.00         | 1.00          | 1.00     |
| min   | -0.28      | -0.39        | -1.24         |          |
| 25%   | -0.26      | -0.38        | -0.84         |          |
| 50%   | -0.22      | -0.34        | -0.39         |          |
| 75%   | -0.09      | -0.15        | 0.93          |          |
| max   | 7.74       | 6.95         | 2.37          |          |

If we have outliers in our data, robust scaling might be a good option. Robust scaling subtracts the median from each value of a variable and divides that value by the interquartile range. So, each value is:

$$z_{ij} = (x_{ij} - \text{median}_j) / (\text{3rd quantile}_j - \text{1st quantile}_j)$$

Where  $x_{ij}$  is the value for the  $j^{th}$  feature, and  $\text{median}_j$ ,  $\text{3}^{rd} \text{ quantile}_j$ , and  $\text{1}^{st} \text{ quantile}_j$ , are the median, third, and first quantile of the  $j^{th}$  feature, respectively. Robust scaling is less sensitive to extreme values, since it does not use the mean or variance.

4. We can use scikit-learn's `RobustScaler` module to do robust scaling:

```
scaler = RobustScaler()
X_train_rs = pd.DataFrame(scaler.fit_transform(X_train),
                           columns=X_train.columns, index=X_train.index)
X_train_rs.describe()
```

|       | population | total_deaths | aged_65_older | li |
|-------|------------|--------------|---------------|----|
| count | 131.00     | 131.00       | 131.00        |    |
| mean  | 1.29       | 1.51         | 0.22          |    |
| std   | 5.81       | 4.44         | 0.57          |    |
| min   | -0.30      | -0.20        | -0.48         |    |
| 25%   | -0.22      | -0.16        | -0.26         |    |
| 50%   | 0.00       | 0.00         | 0.00          |    |
| 75%   | 0.78       | 0.84         | 0.74          |    |
| max   | 46.09      | 32.28        | 1.56          |    |

The previous steps demonstrate three popular scaling transformations, standard scaling, min-max scaling, and robust scaling.

## How it works...

We use feature scaling with most machine learning algorithms. Although it is not often required, it yields noticeably better results. Min-max scaling

and standard scaling are popular scaling techniques, but there are times when robust scaling might be the better option.

*Scikit-learn*'s `preprocessing` module makes it easy to use a variety of scaling transformations. We just need to instantiate the scaler and then run the `fit`, `transform`, or `fit_transform` method.

## Summary

We have covered a wide range of feature engineering techniques in this chapter. We used tools to drop redundant or highly correlated features. We explored the most common kinds of encoding—one-hot, ordinal, and hashing encoding. We then used transformations to improve the distribution of our features. Finally, we used common binning and scaling approaches to address skew, kurtosis, and outliers, and to adjust for features with widely different ranges. In the next chapter, we'll learn how to fix messy data when aggregating.

## Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review.  
Scan the QR code below to get a free eBook of your choice.



# 9

## Fixing Messy Data When Aggregating

Earlier chapters of this book introduced techniques to generate summary statistics on a whole DataFrame. We used methods such as `describe`, `mean`, and `quantile` to do that. This chapter covers more complicated aggregation tasks: aggregating by categorical variables and using aggregation to change the structure of DataFrames.

After the initial stages of data cleaning, analysts spend a substantial amount of their time doing what Hadley Wickham has called *splitting-applying-combining*—that is, we subset data by groups, apply some operation to those subsets, and then draw conclusions about a dataset as a whole. In slightly more specific terms, this involves generating descriptive statistics by key categorical variables. For the `nls97` dataset, this might be gender, marital status, and the highest degree received. For the COVID-19 data, we might segment the data by country or date.

Often, we need to aggregate data to prepare it for subsequent analysis. Sometimes, the rows of a DataFrame are disaggregated beyond the desired unit of analysis, and some aggregation has to be done before analysis can begin. For example, our DataFrame might have bird sightings by species per day over the course of many years. Since those values jump around, we might decide to smooth that out by working only with the total sightings by

species per month, or even per year. Another example is household and car repair expenditures. We might need to summarize those expenditures over a year.

There are several ways to aggregate data using NumPy and pandas, each with particular strengths. We explore the most useful approaches in this chapter: from looping with `itertuples`, to navigating over NumPy arrays, to several techniques using the DataFrame `groupby` method, and pivot tables. It is helpful to have a good understanding of the full range of tools available in pandas and NumPy, since almost all data analysis projects require some aggregation, aggregation is among the most consequential steps we take in the data cleaning process, and the best tool for the job is determined more by the attributes of the data than by our personal preferences.

Specifically, the recipes in this chapter examine the following:

- Looping through data with `itertuples` (an anti-pattern)
- Calculating summaries by group with NumPy arrays
- Using `groupby` to organize data by groups
- Using more complicated aggregation functions with `groupby`
- Using user-defined functions and `apply` with `groupby`
- Using `groupby` to change the unit of analysis of a DataFrame
- Using the pandas `pivot_table` function to change the unit of analysis

## Technical requirements

You will need pandas, NumPy, and Matplotlib to complete the recipes in this chapter. I used pandas 2.1.4, but the code will run on pandas 1.5.3 or later.

The code in this chapter can be downloaded from the book's GitHub repository, <https://github.com/PacktPublishing/Python-Data-Cleaning-Cookbook-Second-Edition>.

## Looping through data with `itertuples` (an anti-pattern)

In this recipe, we will iterate over the rows of a DataFrame and generate our own totals for a variable. In subsequent recipes in this chapter, we will use NumPy arrays, and then some pandas-specific techniques, to accomplish the same tasks.

It may seem odd to begin this chapter with a technique that we are often cautioned against using. But I used to do the equivalent of looping every day 35 years ago in SAS, and on select occasions as recently as 10 years ago in R. That is why I still find myself thinking conceptually about iterating over rows of data, sometimes sorted by groups, even though I rarely implement my code in this manner. I think it is good to hold onto that conceptualization, even when using other pandas methods that work for us more efficiently.

I do not want to leave the impression that pandas-specific techniques are always markedly more efficient either. pandas users probably find themselves using `apply` more than they would like, an approach that is only somewhat faster than looping.

## Getting ready

We will work with the COVID-19 case daily data in this recipe. It has one row per day per country, each row having the number of new cases and new deaths for that day. It reflects totals as of March 2024.

We will also be working with land temperature data from 87 weather stations in Brazil in 2023. Most weather stations had one temperature reading for each month.

### Data note

Our World in Data provides COVID-19 public use data at <https://ourworldindata.org/covid-cases>.

The dataset includes total cases and deaths, tests administered, hospital beds, and demographic data such as median age, gross domestic product, and diabetes prevalence. The dataset used in this recipe was downloaded on March 3, 2024.



The land temperature DataFrame has the average temperature reading (in °C) in 2023 from over 12,000 stations across the world, although 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.ncei.noaa.gov/products/land-based-station/global-historical-climatology-network-monthly>.

# How to do it...

We will use the `itertuples` DataFrame method to loop over the rows of the COVID-19 daily data and the monthly land temperature data for Brazil. We add logic to handle missing data and unexpected changes in key variable values from one period to the next:

1. Import `pandas` and `numpy`, and load the COVID-19 and land temperature data:

```
import pandas as pd
coviddaily = pd.read_csv("data/coviddaily.csv", parse
ltbrazil = pd.read_csv("data/ltbrazil.csv")
```

2. Sort data by location and date:

```
coviddaily = coviddaily.sort_values(['location', 'case
```

3. Iterate over rows with `itertuples`.

Use `itertuples`, which allows us to iterate over all rows as named tuples. Sum new cases over all the dates for each country. With each change of country (`location`), append the running total to `rowlist`, and then set the count to `0` (note that `rowlist` is a list and we are appending a dictionary to `rowlist` with each change of country. A list of dictionaries is a good place to temporarily store data you might eventually want to convert to a DataFrame):

```
prevloc = 'ZZZ'
rowlist = []
casecnt = 0
```

```
for row in coviddaily.itertuples():
...     if (prevloc!=row.location):
...         if (prevloc!='ZZZ'):
...             rowlist.append({'location':prevloc, 'casecnt':casecnt})
...             casecnt = 0
...             prevloc = row.location
...             casecnt += row.new_cases
...
rowlist.append({'location':prevloc, 'casecnt':casecnt})
len(rowlist)
```

231

```
rowlist[0:4]
```

```
[{'location': 'Afghanistan', 'casecnt': 231539.0},
 {'location': 'Albania', 'casecnt': 334863.0},
 {'location': 'Algeria', 'casecnt': 272010.0},
 {'location': 'American Samoa', 'casecnt': 8359.0}]
```

#### 4. Create a DataFrame from the list of summary values, `rowlist`.

Pass the list we created in the previous step to the pandas `DataFrame` method:

```
covidtotals = pd.DataFrame(rowlist)
covidtotals.head()
```

|   | location    | casecnt |
|---|-------------|---------|
| 0 | Afghanistan | 231,539 |
| 1 | Albania     | 334,863 |
| 2 | Algeria     | 272,010 |

|   |                |        |
|---|----------------|--------|
| 3 | American Samoa | 8,359  |
| 4 | Andorra        | 48,015 |

5. Now, let's do the same for the land temperature data. We start by sorting it by `station` and `month`.

Also, drop rows with missing values for temperature:

```
ltbrazil = ltbrazil.sort_values(['station', 'month'])
ltbrazil = ltbrazil.dropna(subset=['temperature'])
```

6. Exclude rows where there is a large change from one period to the next.

Calculate the average temperature for the year, excluding values for temperature more than 3°C greater than or less than the temperature for the previous month:

```
prevstation = 'ZZZ'
prevtemp = 0
rowlist = []
tempcnt = 0
stationcnt = 0
for row in ltbrazil.itertuples():
    if (prevstation!=row.station):
        if (prevstation!='ZZZ'):
            rowlist.append({'station':prevstation, 'avgtemp':
            tempcnt = 0
            stationcnt = 0
            prevstation = row.station
            # choose only rows that are within 3 degrees of the
            if ((0 <= abs(row.temperature-prevtemp) <= 3) or (st
            tempcnt += row.temperature
            stationcnt += 1
            prevtemp = row.temperature
```

```
...  
rowlist.append({'station':prevstation, 'avgtemp':tempcnt/s  
rowlist[0:5]
```

```
[{'station': 'ALTAMIRA', 'avgtemp': 27.729166666666668, 'stationcnt': 12},  
 {'station': 'ALTA_FLORESTA_AERO', 'avgtemp': 32.49333333333333, 'stationcnt': 9},  
 {'station': 'ARAXA', 'avgtemp': 21.52142857142857, 'stationcnt': 7},  
 {'station': 'BACABAL', 'avgtemp': 28.591666666666667, 'stationcnt': 6},  
 {'station': 'BAGE', 'avgtemp': 19.615000000000002, 'stationcnt': 10}]
```

## 7. Create a DataFrame from the summary values.

Pass the list we created in the previous step to the pandas `DataFrame` method:

```
ltbrazilavgs = pd.DataFrame(rowlist)  
ltbrazilavgs.head()
```

```
   station      avgtemp  stationcnt  
0    ALTAMIRA        28            12  
1  ALTA_FLORESTA_AERO     32             9  
2       ARAXA        22              7  
3      BACABAL        29              6  
4        BAGE        20            10
```

This gives us a DataFrame with average temperatures for 2023 and the number of observations for each station.

## How it works...

After sorting the COVID-19 daily data by `location` and `casedate` in *step 2*, we loop through our data one row at a time and do a running tally of new cases in *step 3*. We set that tally back to `0` when we get to a new country, and then resume counting. Notice that we do not actually append our summary of new cases until we get to the next country. This is because there is no way to tell that we are on the last row for any country until we get to the next country. That is not a problem because we append the summary to `rowlist` right before we reset the value to `0`. That also means that we need to do something special to output the totals for the last country, since there is no next country. We do this with a final append after the loop is complete. This is a fairly standard approach to looping through data and outputting totals by group.

The summary DataFrame we create in *steps 3* and *4* can be created more efficiently, both in terms of the analyst's time and our computer's workload, with other pandas techniques that we cover in this chapter. But that becomes a more difficult call when we need to do more complicated calculations, particularly those that involve comparing values across rows.

*Steps 6* and *7* provide an example of this. We want to calculate the average temperature for each station for the year. Most stations have one reading per month. However, we are concerned that there might be some outlier values for temperature, defined here by a change of more than 3°C from one month to the next. We want to exclude those readings from the calculation of the mean for each station. It is fairly straightforward to do that while iterating over the data, by storing the previous value of temperature (`prevtemp`) and comparing it to the current value.

## There's more...

We could have used `iterrows` in *step 3* rather than `itertuples`, with almost exactly the same syntax. Since we do not need the functionality of `iterrows` here, we use `itertuples`. The `itertuples` method is easier on system resources than `iterrows`. This is because you iterate over tuples with `itertuples`, but over Series, with the associated type checking, with `iterrows`.

The hardest tasks to complete when working with tabular data involve calculations across rows: summing data across rows, basing a calculation on values in a different row, and generating running totals. Such calculations are complicated to implement and resource-intensive, regardless of language. However, it is hard to avoid having to do them, particularly when working with panel data. Some values for variables in a given period might be determined by values in a previous period. This is often more complicated than the running totals we have done in this recipe.

For decades, data analysts have tried to address these data-cleaning challenges by looping through rows, carefully inspecting categorical and summary variables for data problems, and then handling the summation accordingly. Although this continues to be the approach that provides the most flexibility, pandas provides a number of data aggregation tools that run more efficiently and are easier to code. The challenge is to match the ability of looping solutions to adjust for invalid, incomplete, or atypical data. We explore these tools later in this chapter.

## Calculating summaries by group with NumPy arrays

We can accomplish much of what we did in the previous recipe with `itertuples` using NumPy arrays. We can also use NumPy arrays to get summary values for subsets of our data.

## Getting ready

We will work again with the COVID-19 daily data and the Brazil land temperature data.

## How to do it...

We copy DataFrame values to a NumPy array. We then navigate over the array, calculating totals by group and checking for unexpected changes in values:

1. Import `pandas` and `numpy`, and load the COVID-19 and land temperature data:

```
import pandas as pd
coviddaily = pd.read_csv("data/coviddaily.csv", parse
ltbrazil = pd.read_csv("data/ltbrazil.csv")
```

2. Create a list of locations:

```
loclist = coviddaily.location.unique().tolist()
```

3. Use a NumPy array to calculate sums by location.

Create a NumPy array of the location and new cases data. We then can iterate over the location list we created in the previous step and select all new case values (`casevalues[j][1]`) for each location

(`casevalues[j][0]`). We then sum the new case values for that location:

```
rowlist = []
casevalues = coviddaily[['location', 'new_cases']].to_numpy
for locitem in loclist:
...     cases = [casevalues[j][1] for j in range(len(casevalues))
...             if casevalues[j][0]==locitem]
...     rowlist.append(sum(cases))
...
len(rowlist)
```

```
231
```

```
len(loclist)
```

```
231
```

```
rowlist[0:5]
```

```
[231539.0, 334863.0, 272010.0, 8359.0, 48015.0]
```

```
casetotals = pd.DataFrame(zip(loclist, rowlist), columns=[| casetotals.head()
```

|   | location    | casetotals |
|---|-------------|------------|
| 0 | Afghanistan | 231,539    |
| 1 | Albania     | 334,863    |
| 2 | Algeria     | 272,010    |

|   |                |        |
|---|----------------|--------|
| 3 | American Samoa | 8,359  |
| 4 | Andorra        | 48,015 |

4. Sort the land temperature data and drop rows with missing values for temperature:

```
ltbrazil = ltbrazil.sort_values(['station','month'])
ltbrazil = ltbrazil.dropna(subset=['temperature'])
```

5. Use a NumPy array to calculate the average temperature for the year.

Exclude rows where there is a large change from one period to the next:

```
prevstation = 'ZZZ'
prevtemp = 0
rowlist = []
tempvalues = ltbrazil[['station', 'temperature']].to_numpy()
tempcnt = 0
stationcnt = 0
for j in range(len(tempvalues)):
...     station = tempvalues[j][0]
...     temperature = tempvalues[j][1]
...     if (prevstation!=station):
...         if (prevstation!='ZZZ'):
...             rowlist.append({'station':prevstation, 'avgtemp':
...             tempcnt = 0
...             stationcnt = 0
...             prevstation = station
...             if ((0 <= abs(temperature-prevtemp) <= 3) or (station
...             tempcnt += temperature
...             stationcnt += 1
...             prevtemp = temperature
...             ...
```

```
    rowlist.append({'station':prevstation, 'avgtemp':tempcnt/s
rowlist[0:5]
```

```
[{'station': 'ALTAMIRA', 'avgtemp': 27.729166666666668, 'stationcnt': 12}, {'station': 'ALTA_FLORESTA_AERO', 'avgtemp': 32.49333333333333, 'stationcnt': 9}, {'station': 'ARAXA', 'avgtemp': 21.52142857142857, 'stationcnt': 7}, {'station': 'BACABAL', 'avgtemp': 28.591666666666667, 'stationcnt': 6}, {'station': 'BAGE', 'avgtemp': 19.615000000000002, 'stationcnt': 10}]
```

## 6. Create a DataFrame of the land temperature averages:

```
ltbrazilavgs = pd.DataFrame(rowlist)
ltbrazilavgs.head()
```

|   | station            | avgtemp | stationcnt |
|---|--------------------|---------|------------|
| 0 | ALTAMIRA           | 28      | 12         |
| 1 | ALTA_FLORESTA_AERO | 32      | 9          |
| 2 | ARAXA              | 22      | 7          |
| 3 | BACABAL            | 29      | 6          |
| 4 | BAGE               | 20      | 10         |

This gives us a DataFrame with the average temperature and number of observations per station. Notice that we get the same results as in the final step of the previous recipe.

## How it works...

NumPy arrays can be quite useful when we are working with tabular data but need to do some calculations across rows. This is because accessing items over the equivalent of rows is not really that different from accessing

items over the equivalent of columns in an array. For example, `casevalues[5][0]` (the sixth “row” and first “column” of the array) is accessed in the same way as `casevalues[20][1]`. Navigating over a NumPy array is also faster than iterating over a pandas DataFrame.

We take advantage of this in *step 3*. We get all of the array rows for a given location (`if casevalues[j][0]==locitem`) with a list comprehension. Since we also need the `location` list in the DataFrame that we will create of summary values, we use `zip` to combine the two lists.

We start working with the land temperature data in *step 4*, first sorting it by `station` and `month`, and then dropping rows with missing values for temperature. The logic in *step 5* is almost identical to the logic in *step 6* in the previous recipe. The main difference is that we need to refer to the locations of station (`tempvalues[j][0]`) and temperature (`tempvalues[j][1]`) in the array.

## There's more...

When you need to iterate over data, NumPy arrays will generally be faster than iterating over a pandas DataFrame with `itertuples` or `iterrows`. Also, if you tried to run the list comprehension in *step 3* using `itertuples`, which is possible, you would be waiting some time for it to finish. In general, if you want to do a quick summary of values for some segment of your data, using NumPy arrays is a reasonable choice.

## See also

The remaining recipes in this chapter rely on the powerful `groupby` method of pandas DataFrames to generate group totals.

# Using groupby to organize data by groups

At a certain point in most data analysis projects, we have to generate summary statistics by groups. While this can be done using the approaches in the previous recipe, in most cases the pandas DataFrame `groupby` method is a better choice. If `groupby` can handle an aggregation task—and it usually can—it is likely the most efficient way to accomplish that task. We make good use of `groupby` in the next few recipes. We go over the basics in this recipe.

## Getting ready

We will work with the COVID-19 daily data in this recipe.

## How to do it...

We will create a pandas `groupby` DataFrame and use it to generate summary statistics by group:

1. Import `pandas` and `numpy`, and load the COVID-19 daily data:

```
import pandas as pd
coviddaily = pd.read_csv("data/coviddaily.csv", parse
```

2. Create a pandas `groupby` DataFrame:

```
countrytots = coviddaily.groupby(['location'])
type(countrytots)
```

```
<class 'pandas.core.groupby.generic.DataFrameGroupBy'>
```

### 3. Create DataFrames for the first rows of each country.

To save space, we just show the first five rows and the first five columns:

```
countrytots.first().iloc[0:5, 0:5]
```

|                | iso_code | casedate   | continent | new_deaths |
|----------------|----------|------------|-----------|------------|
| location       |          |            |           |            |
| Afghanistan    | AFG      | 2020-03-01 | Asia      |            |
| Albania        | ALB      | 2020-03-15 | Europe    |            |
| Algeria        | DZA      | 2020-03-01 | Africa    |            |
| American Samoa | ASM      | 2021-09-19 | Oceania   |            |
| Andorra        | AND      | 2020-03-08 | Europe    |            |
|                |          |            |           | new_deaths |
| location       |          |            |           |            |
| Afghanistan    |          | 0          |           |            |
| Albania        |          | 1          |           |            |
| Algeria        |          | 0          |           |            |
| American Samoa |          | 0          |           |            |
| Andorra        |          | 0          |           |            |

### 4. Create DataFrames for the last rows of each country:

```
countrytots.last().iloc[0:5, 0:5]
```

|                | iso_code | casedate   | continent | new_deaths |
|----------------|----------|------------|-----------|------------|
| location       |          |            |           |            |
| Afghanistan    | AFG      | 2024-02-04 | Asia      |            |
| Albania        | ALB      | 2024-01-28 | Europe    |            |
| Algeria        | DZA      | 2023-12-03 | Africa    |            |
| American Samoa | ASM      | 2023-09-17 | Oceania   |            |
| Andorra        | AND      | 2023-05-07 | Europe    |            |
|                |          |            |           | new_deaths |

```
location
Afghanistan          0
Albania              0
Algeria               0
American Samoa       0
Andorra              0
```

```
type(countrytots.last())
```

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

## 5. Get all the rows for a country:

```
countrytots.get_group('Zimbabwe').iloc[0:5, 0:5]
```

```
iso_code      casedate    location   continent
36305 ZWE    2020-03-22 Zimbabwe    Africa
36306 ZWE    2020-03-29 Zimbabwe    Africa
36307 ZWE    2020-04-05 Zimbabwe    Africa
36308 ZWE    2020-04-12 Zimbabwe    Africa
36309 ZWE    2020-04-19 Zimbabwe    Africa
```

## 6. Loop through the groups.

Only display rows for Malta and Kuwait:

```
for name, group in countrytots:
...     if (name[0] in ['Malta', 'Kuwait']):
...         print(group.iloc[0:5, 0:5])
...
```

```
iso_code      casedate    location   continent
17818 KWT    2020-03-01 Kuwait     Asia
```

|       |          |            |          |           |
|-------|----------|------------|----------|-----------|
| 17819 | KWT      | 2020-03-08 | Kuwait   | Asia      |
| 17820 | KWT      | 2020-03-15 | Kuwait   | Asia      |
| 17821 | KWT      | 2020-03-22 | Kuwait   | Asia      |
| 17822 | KWT      | 2020-03-29 | Kuwait   | Asia      |
|       | iso_code | casedate   | location | continent |
| 20621 | MLT      | 2020-03-08 | Malta    | Europe    |
| 20622 | MLT      | 2020-03-15 | Malta    | Europe    |
| 20623 | MLT      | 2020-03-22 | Malta    | Europe    |
| 20624 | MLT      | 2020-03-29 | Malta    | Europe    |
| 20625 | MLT      | 2020-04-05 | Malta    | Europe    |

7. Show the number of rows for each country:

```
countrytots.size()
```

| location          |     |
|-------------------|-----|
| Afghanistan       | 205 |
| Albania           | 175 |
| Algeria           | 189 |
| American Samoa    | 58  |
| Andorra           | 158 |
| Vietnam           | 192 |
| Wallis and Futuna | 23  |
| Yemen             | 122 |
| Zambia            | 173 |
| Zimbabwe          | 196 |

Length: 231, dtype: int64

8. Show summary statistics by country:

```
countrytots.new_cases.describe().head(3).T
```

| location | Afghanistan | Albania | Algeria |
|----------|-------------|---------|---------|
| count    | 205         | 175     | 189     |
| mean     | 1,129       | 1,914   | 1,439   |

|     | new_cases | new_deaths | new_recovered |
|-----|-----------|------------|---------------|
| std | 1,957     | 2,637      | 2,205         |
| min | 1         | 20         | 1             |
| 25% | 242       | 113        | 30            |
| 50% | 432       | 522        | 723           |
| 75% | 1,106     | 3,280      | 1,754         |
| max | 12,314    | 15,405     | 14,774        |

```
countrytots.new_cases.sum().head()
```

```
location
Afghanistan      231,539
Albania          334,863
Algeria           272,010
American Samoa     8,359
Andorra            48,015
Name: new_cases, dtype: float64
```

These steps demonstrate how remarkably useful the `groupby` DataFrame object is when we want to generate summary statistics by categorical variables.

## How it works...

In *step 2*, we create a pandas DataFrame `groupby` object using the pandas DataFrame `groupby` method, passing it a column or list of columns for the grouping. Once we have a `groupby` DataFrame, we can generate statistics by group with the same tools that we use to generate summary statistics for the whole DataFrame. `describe`, `mean`, `sum`, and similar methods work on the `groupby` DataFrame—or series created from it—as expected, except the summary is run for each group.

In *steps 3 and 4*, we use `first` and `last` to create DataFrames with the first and last occurrence of each group. We use `get_group` to get all the rows for a particular group in *step 5*. We can also loop over the groups and use `size` to count the number of rows for each group.

In *step 8*, we create a Series `groupby` object from the DataFrame `groupby` object. Using the resulting object's aggregation methods gives us summary statistics for a Series by group. One thing is clear about the distribution of `new_cases` from this output: it varies quite a bit by country. For example, we can see right away that the interquartile range is quite different, even for the first three countries.

## There's more...

The output from *step 8* is quite useful. It is worth saving output such as that for each important continuous variable where the distribution is meaningfully different by group.

pandas `groupby` DataFrames are extraordinarily powerful and easy to use. *step 8* shows just how easy it is to create the summaries by group that we created in the first two recipes in this chapter. Unless the DataFrame we are working with is small, or the task involves very complicated calculations across rows, the `groupby` method is a superior choice to looping.

## Using more complicated aggregation functions with `groupby`

In the previous recipe, we created a `groupby` DataFrame object and used it to run summary statistics by groups. We use chaining in this recipe to create

the groups, choose the aggregation variable(s), and select the aggregation function(s), all in one line. We also take advantage of the flexibility of the `groupby` object, which allows us to choose the aggregation columns and functions in a variety of ways.

## Getting ready

We will work with the **National Longitudinal Survey of Youth (NLS)** data in this recipe.



### Data note

The **National Longitudinal Surveys**, administered by the United States Bureau of Labor Statistics, are longitudinal surveys of individuals who were in high school in 1997 when the surveys started. Participants were surveyed each year through 2023. The surveys are available for public use at [nlsinfo.org](https://nlsinfo.org).

## How to do it...

We do more complicated aggregations with `groupby` than we did in the previous recipe, taking advantage of its flexibility:

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

```
import pandas as pd
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
```

## 2. Review the structure of the data:

```
nls97.iloc[:, 0:7].info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 8984 entries, 135335 to 713757
Data columns (total 7 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   sampletype     8984 non-null    object  
 4   ethnicity       8984 non-null    object  
 5   highestgradecompleted  6663 non-null  float64 
 6   maritalstatus   6675 non-null    object  
dtypes: float64(1), int64(2), object(4)
memory usage: 561.5+ KB
```

## 3. Review some of the categorical data:

```
catvars = ['gender', 'maritalstatus', 'highestdegree']
for col in catvars:
...     print(col, nls97[col].value_counts().\
...         sort_index(), sep="\n\n", end="\n\n\n")
...
```

```
gender
Female    4385
Male      4599
Name: count, dtype: int64
maritalstatus
Divorced      669
Married       3068
Never-married 2767
Separated      148
Widowed        23
```

```
Name: count, dtype: int64
highestdegree
0. None           877
1. GED            1167
2. High School    3531
3. Associates     766
4. Bachelors      1713
5. Masters         704
6. PhD             64
7. Professional    130
Name: count, dtype: int64
```

#### 4. Review some descriptive statistics:

```
contvars = ['satmath', 'satverbal',
...     'weeksworked06', 'gpaoverall', 'childathome']
nls97[contvars].describe()
```

|       | satmath | satverbal | weeksworked06 | gpaoverall |
|-------|---------|-----------|---------------|------------|
| count | 1,407   | 1,406     | 8,419         | 6,004      |
| mean  | 501     | 500       | 38            | 282        |
| std   | 115     | 112       | 19            | 62         |
| min   | 7       | 14        | 0             | 10         |
| 25%   | 430     | 430       | 27            | 243        |
| 50%   | 500     | 500       | 51            | 286        |
| 75%   | 580     | 570       | 52            | 326        |
| max   | 800     | 800       | 52            | 417        |

#### 5. Look at **Scholastic Assessment Test (SAT)** math scores by gender.

We pass the column name to `groupby` to group by that column:

```
nls97.groupby('gender')[['satmath']].mean()
```

```
gender
Female    487
Male      517
Name: satmath, dtype: float64
```

6. Look at SAT math scores by gender and the highest degree earned.

We can pass a list of column names to `groupby` to group by more than one column:

```
nls97.groupby(['gender','highestdegree'])['satmath'].\
mean()
```

```
gender  highestdegree
Female   0. None          414
           1. GED           405
           2. High School    426
           3. Associates     448
           4. Bachelors      503
           5. Masters         504
           6. PhD             569
           7. Professional    593
Male     0. None          545
           1. GED           320
           2. High School    465
           3. Associates     490
           4. Bachelors      536
           5. Masters         568
           6. PhD             624
           7. Professional    594
Name: satmath, dtype: float64
```

7. Look at SAT math and verbal scores by gender and the highest degree earned.

We can use a list to summarize values for more than one variable, in this case `satmath` and `satverbal`:

```
nls97.groupby(['gender','highestdegree'])[['satmath','satv
```

|        |                 |  | satmath | satverbal |
|--------|-----------------|--|---------|-----------|
| gender | highestdegree   |  |         |           |
| Female | 0. None         |  | 414     | 408       |
|        | 1. GED          |  | 405     | 390       |
|        | 2. High School  |  | 426     | 440       |
|        | 3. Associates   |  | 448     | 453       |
|        | 4. Bachelors    |  | 503     | 508       |
|        | 5. Masters      |  | 504     | 529       |
|        | 6. PhD          |  | 569     | 561       |
|        | 7. Professional |  | 593     | 584       |
| Male   | 0. None         |  | 545     | 515       |
|        | 1. GED          |  | 320     | 360       |
|        | 2. High School  |  | 465     | 455       |
|        | 3. Associates   |  | 490     | 469       |
|        | 4. Bachelors    |  | 536     | 521       |
|        | 5. Masters      |  | 568     | 540       |
|        | 6. PhD          |  | 624     | 627       |
|        | 7. Professional |  | 594     | 599       |

## 8. Do multiple aggregation functions for one variable.

Use the `agg` function to return several summary statistics:

```
nls97.groupby(['gender','highestdegree'])\n['gpaoverall'].agg(['count','mean','max','std'])
```

|        |                | count | mean | max | std |
|--------|----------------|-------|------|-----|-----|
| gender | highestdegree  |       |      |     |     |
| Female | 0. None        | 134   | 243  | 400 | 66  |
|        | 1. GED         | 231   | 230  | 391 | 66  |
|        | 2. High School | 1152  | 277  | 402 | 53  |

|      |                 | 294  | 291 | 400 | 50 |
|------|-----------------|------|-----|-----|----|
|      | 3. Associates   | 294  | 291 | 400 | 50 |
|      | 4. Bachelors    | 742  | 322 | 407 | 48 |
|      | 5. Masters      | 364  | 329 | 417 | 43 |
|      | 6. PhD          | 26   | 345 | 400 | 44 |
|      | 7. Professional | 55   | 353 | 411 | 41 |
| Male | 0. None         | 180  | 222 | 400 | 65 |
|      | 1. GED          | 346  | 223 | 380 | 63 |
|      | 2. High School  | 1391 | 263 | 396 | 49 |
|      | 3. Associates   | 243  | 272 | 383 | 49 |
|      | 4. Bachelors    | 575  | 309 | 405 | 49 |
|      | 5. Masters      | 199  | 324 | 404 | 50 |
|      | 6. PhD          | 23   | 342 | 401 | 55 |
|      | 7. Professional | 41   | 345 | 410 | 35 |

## 9. Use a dictionary for more complicated aggregations:

```
pd.options.display.float_format = '{:.1f}'.format
aggdict = {'weeksworked06': ['count', 'mean',
... 'max', 'std'], 'childathome': ['count', 'mean',
... 'max', 'std']}
nls97.groupby(['highestdegree']).agg(aggdict)
```

| highestdegree   | weeksworked06 |      |      |      | \ |
|-----------------|---------------|------|------|------|---|
|                 | count         | mean | max  | std  |   |
| 0. None         | 666           | 29.7 | 52.0 | 21.6 |   |
| 1. GED          | 1129          | 32.9 | 52.0 | 20.7 |   |
| 2. High School  | 3262          | 39.4 | 52.0 | 18.6 |   |
| 3. Associates   | 755           | 40.2 | 52.0 | 18.0 |   |
| 4. Bachelors    | 1683          | 42.3 | 52.0 | 16.2 |   |
| 5. Masters      | 703           | 41.8 | 52.0 | 16.6 |   |
| 6. PhD          | 63            | 38.5 | 52.0 | 18.4 |   |
| 7. Professional | 127           | 27.8 | 52.0 | 20.4 |   |
| childathome     |               |      |      |      |   |
| highestdegree   | count         | mean | max  | std  |   |
|                 | 408           | 1.8  | 8.0  | 1.6  |   |
| 0. None         | 702           | 1.7  | 9.0  | 1.5  |   |

|    |              | count | mean | max | std |
|----|--------------|-------|------|-----|-----|
| 2. | High School  | 1881  | 1.9  | 7.0 | 1.3 |
| 3. | Associates   | 448   | 1.9  | 6.0 | 1.1 |
| 4. | Bachelors    | 859   | 1.9  | 8.0 | 1.1 |
| 5. | Masters      | 379   | 1.9  | 6.0 | 0.9 |
| 6. | PhD          | 33    | 1.9  | 3.0 | 0.8 |
| 7. | Professional | 60    | 1.8  | 4.0 | 0.8 |

```
nls97.groupby(['maritalstatus']).agg(aggdict)
```

|  | maritalstatus | weeksworked06 |      |      |      |
|--|---------------|---------------|------|------|------|
|  |               | count         | mean | max  | std  |
|  | Divorced      | 666           | 37.5 | 52.0 | 19.0 |
|  | Married       | 3035          | 40.3 | 52.0 | 17.9 |
|  | Never-married | 2735          | 37.2 | 52.0 | 19.1 |
|  | Separated     | 147           | 33.6 | 52.0 | 20.3 |
|  | Widowed       | 23            | 37.1 | 52.0 | 19.3 |
|  | maritalstatus | childathome   |      |      |      |
|  |               | count         | mean | max  | std  |
|  | Divorced      | 530           | 1.5  | 5.0  | 1.2  |
|  | Married       | 2565          | 2.1  | 8.0  | 1.1  |
|  | Never-married | 1501          | 1.6  | 9.0  | 1.3  |
|  | Separated     | 132           | 1.5  | 8.0  | 1.4  |
|  | Widowed       | 18            | 1.8  | 5.0  | 1.4  |

We display the same summary statistics for `weeksworked06` and `childathome`, but we could have specified different aggregation functions for each using the same syntax that we used in *step 9*.

## How it works...

We first take a look at some summary statistics for key columns in the DataFrame. We get frequencies for the categorical variables in *step 3*, and

some descriptives for the continuous variables in *step 4*. It is a good idea to have summary values for the DataFrame as a whole in front of us before generating statistics by group.

We are then ready to create summary statistics using `groupby`. This involves three steps:

1. Creating a `groupby` DataFrame based on one or more categorical variables.
2. Selecting the column(s) to be used for the summary statistics.
3. Choosing the aggregation function(s).

We use chaining in this recipe to do all three in one line. So,

`nls97.groupby('gender')['satmath'].mean()` in *step 5* does three things: `nls97.groupby('gender')` creates the `groupby` DataFrame object, `['satmath']` chooses the aggregation column, and `mean()` is the aggregation function.

We can pass a column name (as in *step 5*) or a list of column names (as in *step 6*) to `groupby` to create groupings by one or more columns. We can select multiple variables for aggregation with a list of those variables, as we do in *step 7* with `[['satmath', 'satverbal']]`.

We can chain a specific summary function such as `mean`, `count`, or `max`. Alternatively, we could pass a list to `agg` to choose multiple aggregation functions, such as with `agg(['count', 'mean', 'max', 'std'])` in *step 8*. We can use the familiar pandas and NumPy aggregation functions or a user-defined function, which we explore in the next recipe.

Another important takeaway from *step 8* is that `agg` sends the aggregation columns to each function a group at a time. The calculations in each aggregation function are run for each group in the `groupby` DataFrame.

Another way to conceptualize this is that it allows us to run the same functions we are used to running across a whole DataFrame for one group at a time, accomplishing this by automating the process of sending the data for each group to the aggregation functions.

## There's more...

We first get a sense of how the categorical and continuous variables in the DataFrame are distributed. Often, we group data to see how a distribution of a continuous variable, such as weeks worked, differs by a categorical variable, such as marital status. Before doing that, it is helpful to have a good idea of how those variables are distributed across the whole dataset.

The `nls97` dataset only has SAT scores for about 1,400 of 8,984 respondents, so we need to be careful when examining SAT scores by different groups. This means that some of the counts by gender and highest degree, especially for PhD recipients, are a little too small to be reliable. There are outliers for SAT math and verbal (if we define outliers as 1.5 times the interquartile range above the third quartile or below the first quartile).

We have acceptable counts for weeks worked and the number of children living at home for all values of the highest degree, and values of marital status except for widowed. The average weeks worked for folks who received a professional degree is unexpected. It is lower than for any other group. A good next step would be to see how persistent this is over the years. (We are just looking at 2006 weeks worked here, but there are 20 years of data on weeks worked.)

## See also

The `nls97` file is panel data masquerading as individual-level data. The panel data structure can be recovered, facilitating analysis over time of areas such as employment and school enrollment. We do this in the recipes in *Chapter 11, Tidying and Reshaping Data*.

# Using user-defined functions and apply with groupby

Despite the numerous aggregation functions available in pandas and NumPy, we sometimes have to write our own to get the results we need. In some cases, this requires the use of `apply`.

## Getting ready

We will work with the NLS data in this recipe.

## How to do it...

We will create our own functions to define the summary statistics we want by group:

1. Import `pandas` and the NLS data:

```
import pandas as pd
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
```

2. Create a function to define the interquartile range:

```
def iqr(x):
...     return x.quantile(0.75) - x.quantile(0.25)
```

### 3. Run the interquartile range function.

Create a dictionary that specifies which aggregation functions to run on each analysis variable:

```
aggdict = {'weeksworked06': ['count', 'mean', iqr], 'childdat
nls97.groupby(['highestdegree']).agg(aggdict)
```

| highestdegree   | weeksworked06 |      |      | childdat |    |
|-----------------|---------------|------|------|----------|----|
|                 | count         | mean | iqr  | count    | me |
| 0. None         | 666           | 29.7 | 47.0 | 408      | 1  |
| 1. GED          | 1129          | 32.9 | 40.0 | 702      | 1  |
| 2. High School  | 3262          | 39.4 | 21.0 | 1881     | 1  |
| 3. Associates   | 755           | 40.2 | 19.0 | 448      | 1  |
| 4. Bachelors    | 1683          | 42.3 | 13.5 | 859      | 1  |
| 5. Masters      | 703           | 41.8 | 13.5 | 379      | 1  |
| 6. PhD          | 63            | 38.5 | 22.0 | 33       | 1  |
| 7. Professional | 127           | 27.8 | 43.0 | 60       | 1  |

### 4. Define a function to return selected summary statistics:

```
def gettots(x):
...     out = {}
...     out['qr1'] = x.quantile(0.25)
...     out['med'] = x.median()
...     out['qr3'] = x.quantile(0.75)
...     out['count'] = x.count()
...     return out
```

### 5. Use `apply` to run the function.

This will create a series with a multi-index, based on the `highestdegree` values and the desired summary statistics:

```
nls97.groupby(['highestdegree'])['weeksworked06'].\  
apply(gettots)
```

|    | highestdegree |                                           |
|----|---------------|-------------------------------------------|
| 0. | None          | qr1 5<br>med 35<br>qr3 52<br>count 666    |
| 1. | GED           | qr1 12<br>med 42<br>qr3 52<br>count 1,129 |
| 2. | High School   | qr1 31<br>med 52<br>qr3 52<br>count 3,262 |
| 3. | Associates    | qr1 33<br>med 52<br>qr3 52<br>count 755   |
| 4. | Bachelors     | qr1 38<br>med 52<br>qr3 52<br>count 1,683 |
| 5. | Masters       | qr1 38<br>med 52<br>qr3 52<br>count 703   |
| 6. | PhD           | qr1 30<br>med 50<br>qr3 52<br>count 63    |
| 7. | Professional  | qr1 6<br>med 30<br>qr3 49                 |

```
      count    127  
Name: weeksworked06, dtype: float64
```

6. Use `reset_index` to use the default index instead of the index created from the `groupby` DataFrame:

```
nls97.groupby(['highestdegree'])['weeksworked06'].\  
apply(gettots).reset_index()
```

|    | highestdegree  | level_1 | weeksworked06 |
|----|----------------|---------|---------------|
| 0  | 0. None        | qr1     | 5             |
| 1  | 0. None        | med     | 35            |
| 2  | 0. None        | qr3     | 52            |
| 3  | 0. None        | count   | 666           |
| 4  | 1. GED         | qr1     | 12            |
| 5  | 1. GED         | med     | 42            |
| 6  | 1. GED         | qr3     | 52            |
| 7  | 1. GED         | count   | 1,129         |
| 8  | 2. High School | qr1     | 31            |
| 9  | 2. High School | med     | 52            |
| 10 | 2. High School | qr3     | 52            |
| 11 | 2. High School | count   | 3,262         |
| 12 | 3. Associates  | qr1     | 33            |
| 13 | 3. Associates  | med     | 52            |
| 14 | 3. Associates  | qr3     | 52            |
| 15 | 3. Associates  | count   | 755           |
| 16 | 4. Bachelors   | qr1     | 38            |
| 17 | 4. Bachelors   | med     | 52            |
| 18 | 4. Bachelors   | qr3     | 52            |
| 19 | 4. Bachelors   | count   | 1,683         |
| 20 | 5. Masters     | qr1     | 38            |
| 21 | 5. Masters     | med     | 52            |
| 22 | 5. Masters     | qr3     | 52            |
| 23 | 5. Masters     | count   | 703           |
| 24 | 6. PhD         | qr1     | 30            |
| 25 | 6. PhD         | med     | 50            |
| 26 | 6. PhD         | qr3     | 52            |

```
27          6. PhD      count        63
28  7. Professional      qr1         6
29  7. Professional      med        30
30  7. Professional      qr3        49
31  7. Professional      count       127
```

7. Chain with `unstack` instead to create columns based on the summary variables.

This will create a DataFrame, with the `highestdegree` values as the index and aggregation values in the columns:

```
nlssums = nlss97.groupby(['highestdegree'])\n    ['weeksworked06'].apply(gettots).unstack()\nnlssums
```

|                 | qr1 | med | qr3 | count |
|-----------------|-----|-----|-----|-------|
| highestdegree   |     |     |     |       |
| 0. None         | 5   | 35  | 52  | 666   |
| 1. GED          | 12  | 42  | 52  | 1,129 |
| 2. High School  | 31  | 52  | 52  | 3,262 |
| 3. Associates   | 33  | 52  | 52  | 755   |
| 4. Bachelors    | 38  | 52  | 52  | 1,683 |
| 5. Masters      | 38  | 52  | 52  | 703   |
| 6. PhD          | 30  | 50  | 52  | 63    |
| 7. Professional | 6   | 30  | 49  | 127   |

```
nlssums.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 8 entries, 0. None to 7. Professional
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype  
--- 
 0   qr1     8 non-null      float64
```

```
1   med      8 non-null      float64
2   qr3      8 non-null      float64
3   count    8 non-null      float64
dtypes: float64(4)
memory usage: 320.0+ bytes
```

`unstack` is useful when we want to rotate parts of the index to the columns' axis.

## How it works...

We defined a very simple function to calculate interquartile ranges by group in *step 2*. We then included calls to that function in our list of aggregation functions in *step 3*.

*Steps 4* and *5* are a little more complicated. We define a function that calculates the first and third quartiles and median and counts the number of rows. It returns a Series with these values. By combining a `groupby` DataFrame with `apply` in *step 5*, we get the `gettots` function to return that Series for each group.

*Step 5* gives us the numbers we want, but maybe not in the best format. If, for example, we want to use the data for another operation—say, a visualization—we need to chain some additional methods. One possibility is to use `reset_index`. This will replace the multi-index with the default index. Another option is to use `unstack`. This will create columns from the second level of the index (having `qr1`, `med`, `qr3`, and `count` values).

## There's more...

Interestingly, the interquartile ranges for weeks worked and the number of children at home drop substantially as education increases. There seems to be a higher variation in those variables among groups with less education. This should be examined more closely and has implications for statistical testing, which assumes common variances across groups.

## See also

We do much more with `stack` and `unstack` in *Chapter 11, Tidying and Reshaping Data*.

# Using groupby to change the unit of analysis of a DataFrame

The DataFrame that we created in the last step of the previous recipe was something of a fortunate by-product of our efforts to generate multiple summary statistics by groups. There are times when we really do need to aggregate data to change the unit of analysis—say, from monthly utility expenses per family to annual utility expenses per family, or from students’ grades per course to students’ overall **Grade Point Average (GPA)**.

`groupby` is a good tool for collapsing the unit of analysis, particularly when summary operations are required. When we only need to select unduplicated rows—perhaps the first or last row for each individual over a given interval—then the combination of `sort_values` and `drop_duplicates` will do the trick. But we often need to do some calculation across the rows for each group before collapsing. That is when `groupby` comes in very handy.

# Getting ready

We will work with the COVID-19 case daily data, which has one row per country per day. We will also work with the Brazil land temperature data, which has one row per month per weather station.

## How to do it...

We will use `groupby` to create a DataFrame of summary values by group:

1. Import `pandas` and load the COVID-19 and land temperature data:

```
import pandas as pd
coviddaily = pd.read_csv("data/coviddaily.csv", parse
ltbrazil = pd.read_csv("data/ltbrazil.csv")
```

2. Let's view a sample of the data to remind ourselves of its structure.

There is one row per country (`location`) per date, with the number of new cases and deaths for that day (we provide a seed to random state to generate the same values each time):

```
coviddaily[['location', 'casedate',
    'new_cases', 'new_deaths']]. \
set_index(['location', 'casedate']). \
sample(10, random_state=1)
```

| location               | casedate   | new_cases |
|------------------------|------------|-----------|
| Andorra                | 2020-03-15 | 1         |
| Portugal               | 2022-12-04 | 3,963     |
| Eswatini               | 2022-08-07 | 22        |
| Singapore              | 2020-08-30 | 451       |
| Georgia                | 2020-08-02 | 46        |
| British Virgin Islands | 2020-08-30 | 14        |

|                        |            |            |
|------------------------|------------|------------|
| Thailand               | 2023-01-29 | 472        |
| Bolivia                | 2023-12-17 | 280        |
| Montenegro             | 2021-08-15 | 2,560      |
| Eswatini               | 2022-04-17 | 132        |
|                        |            | new_deaths |
| location               | casedate   |            |
| Andorra                | 2020-03-15 | 0          |
| Portugal               | 2022-12-04 | 69         |
| Eswatini               | 2022-08-07 | 2          |
| Singapore              | 2020-08-30 | 0          |
| Georgia                | 2020-08-02 | 1          |
| British Virgin Islands | 2020-08-30 | 0          |
| Thailand               | 2023-01-29 | 29         |
| Bolivia                | 2023-12-17 | 0          |
| Montenegro             | 2021-08-15 | 9          |
| Eswatini               | 2022-04-17 | 0          |

3. We can now convert the COVID-19 data from one country per day to summaries across all countries by day. To limit the amount of data to process, we only include dates between February 2023 and January 2024:

```
coviddailytotals = coviddaily.loc[coviddaily.\n    casedate.between('2023-02-01', '2024-01-31')].\n    groupby(['casedate'], as_index=False).\n    [['new_cases', 'new_deaths']].\n    sum()\ncoviddailytotals.head(10)
```

|   | casedate   | new_cases | new_deaths |
|---|------------|-----------|------------|
| 0 | 2023-02-05 | 1,385,583 | 69,679     |
| 1 | 2023-02-12 | 1,247,389 | 10,105     |
| 2 | 2023-02-19 | 1,145,666 | 8,539      |
| 3 | 2023-02-26 | 1,072,712 | 7,771      |
| 4 | 2023-03-05 | 1,028,278 | 7,001      |
| 5 | 2023-03-12 | 894,678   | 6,340      |
| 6 | 2023-03-19 | 879,074   | 6,623      |

|   |            |         |       |
|---|------------|---------|-------|
| 7 | 2023-03-26 | 833,043 | 6,711 |
| 8 | 2023-04-02 | 799,453 | 5,969 |
| 9 | 2023-04-09 | 701,000 | 5,538 |

4. Let's take a look at a couple of rows of the average temperature data for Brazil:

```
ltbrazil.head(2).T
```

|             | 0                   | 1               |
|-------------|---------------------|-----------------|
| locationid  | BR000082400         | BR000082704     |
| year        | 2023                | 2023            |
| month       | 1                   | 1               |
| temperature | 27                  | 27              |
| latitude    | -4                  | -8              |
| longitude   | -32                 | -73             |
| elevation   | 59                  | 194             |
| station     | FERNANDO_DE_NORONHA | CRUZEIRO_DO_SUL |
| countryid   | BR                  | BR              |
| country     | Brazil              | Brazil          |
| latabs      | 4                   | 8               |

5. Create a DataFrame with average temperatures for each station in Brazil.

Remove rows with missing temperature values first:

```
ltbrazil = ltbrazil.dropna(subset=['temperature'])
ltbrazilavgs = ltbrazil.groupby(['station'],
...     as_index=False).\
...     agg({'latabs':'first','elevation':'first',
...     'temperature':'mean'})
ltbrazilavgs.head(10)
```

|   | station             | latabs | elevation | temperature |
|---|---------------------|--------|-----------|-------------|
| 0 | ALTAMIRA            | 3      | 112       | 28          |
| 1 | ALTA_FLORESTA_AERO  | 10     | 289       | 32          |
| 2 | ARAXA               | 20     | 1,004     | 22          |
| 3 | BACABAL             | 4      | 25        | 29          |
| 4 | BAGE                | 31     | 242       | 20          |
| 5 | BARRA_DO_CORDA      | 6      | 153       | 28          |
| 6 | BARREIRAS           | 12     | 439       | 27          |
| 7 | BARTOLOMEU_LISANDRO | 22     | 17        | 26          |
| 8 | BAURU               | 22     | 617       | 25          |
| 9 | BELEM               | 1      | 10        | 28          |

Let's take a closer look at how the aggregation functions in these examples work.

## How it works...

In *step 3*, we first select the dates that we want. We create a DataFrame `groupby` object based on `casedate`, choose `new_cases` and `new_deaths` as the aggregation variables, and select `sum` for the aggregation function. This produces a sum for both `new_cases` and `new_deaths` for each group (`casedate`). Depending on your purposes, you may not want `casedate` to be the index, which would happen if we did not set `as_index` to `False`.

We often need to use a different aggregation function with different aggregation variables. We might want to take the first (or last) value for one variable and get the mean of the values of another variable by group. This is what we do in *step 5*. We do this by passing a dictionary to the `agg` function, with our aggregation variables as keys and the aggregation function to use as values.

# Using pivot\_table to change the unit of analysis of a DataFrame

We could have used the pandas `pivot_table` function instead of `groupby` in the previous recipe. `pivot_table` can be used to generate summary statistics by the values of a categorical variable, just as we did with `groupby`. The `pivot_table` function can also return a DataFrame, as we will see in this recipe.

## Getting ready

We will work with the COVID-19 case daily data and the Brazil land temperature data again. The temperature data has one row per month per weather station.

## How to do it...

Let's create a DataFrame from the COVID-19 data that has the total number of cases and deaths for each day across all countries:

1. We start by loading the COVID-19 and temperature data again:

```
import pandas as pd
coviddaily = pd.read_csv("data/coviddaily.csv", parse
ltbrazil = pd.read_csv("data/ltbrazil.csv")
```

2. Now, we are ready to call the `pivot_table` function. We pass a list to `values` to indicate the variables for the summary calculations. We use the `index` parameter to indicate that we want totals by `casedate`, and we indicate that we only want sums by passing that to `aggfunc`. Notice

that we get the same totals as in the previous recipe when we used `groupby`:

```
coviddailytotals = \  
    pd.pivot_table(coviddaily.loc[coviddaily.casedate.\  
        between('2023-02-01', '2024-01-31')],  
        values=['new_cases', 'new_deaths'], index='casedate'  
        aggfunc='sum')  
coviddailytotals.head(10)
```

| casedate   | new_cases | new_deaths |
|------------|-----------|------------|
| 2023-02-05 | 1,385,583 | 69,679     |
| 2023-02-12 | 1,247,389 | 10,105     |
| 2023-02-19 | 1,145,666 | 8,539      |
| 2023-02-26 | 1,072,712 | 7,771      |
| 2023-03-05 | 1,028,278 | 7,001      |
| 2023-03-12 | 894,678   | 6,340      |
| 2023-03-19 | 879,074   | 6,623      |
| 2023-03-26 | 833,043   | 6,711      |
| 2023-04-02 | 799,453   | 5,969      |
| 2023-04-09 | 701,000   | 5,538      |

3. Let's try `pivot_table` with the land temperature data and do a more complicated aggregation. We want the first value for latitude (`latabs`) and elevation for each station and the mean temperature. Recall that latitude and elevation values do not change for a station. We pass the aggregations we want as a dictionary to `aggfunc`. Again, we get the same results as in the previous recipe:

```
ltbrazil = ltbrazil.dropna(subset=['temperature'])  
ltbrazilavgs = \  
    pd.pivot_table(ltbrazil, index=['station'],  
        aggfunc={'latabs':'first', 'elevation':'first',
```

```
'temperature': 'mean'})  
ltbrazilavgs.head(10)
```

| station             | elevation | latabs | temperature |
|---------------------|-----------|--------|-------------|
| ALTAMIRA            | 112       | 3      | 28          |
| ALTA_FLORESTA_AERO  | 289       | 10     | 32          |
| ARAXA               | 1,004     | 20     | 22          |
| BACABAL             | 25        | 4      | 29          |
| BAGE                | 242       | 31     | 20          |
| BARRA_DO_CORDA      | 153       | 6      | 28          |
| BARREIRAS           | 439       | 12     | 27          |
| BARTOLOMEU_LISANDRO | 17        | 22     | 26          |
| BAURU               | 617       | 22     | 25          |
| BELEM               | 10        | 1      | 28          |

## How it works...

As we have seen, we get the same results whether we use `groupby` or `pivot_table`. Analysts should probably choose the approach that they, and members of their team, find most intuitive. Since my workflow more frequently has me using `groupby`, I am much more likely to use that approach when aggregating data to create a new DataFrame.

## Summary

We stepped through a wide range of strategies for aggregating data using NumPy and pandas in this chapter. We also discussed advantages and disadvantages of each technique, including how to select the most efficient and intuitive approach given your data and the aggregation task. Since most data cleaning and manipulation projects will involve some splitting-

applying-combining, it is a good idea to become comfortable with each of these approaches. In the next chapter, we will learn how to combine DataFrames and deal with subsequent data issues.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.gg/p8uSgEAETX>



# 10

## Addressing Data Issues When Combining DataFrames

At some point during most data cleaning projects, the analyst will have to combine data from different data tables. This involves either appending data with the same structure to existing data rows or doing a merge to retrieve columns from a different data table. The former is sometimes referred to as combining data vertically, or concatenating, while the latter is referred to as combining data horizontally, or merging.

Merges can be categorized by the amount of duplication of merge-by column values. With one-to-one merges, merge-by column values appear once on each data table. One-to-many merges have unduplicated merge-by column values on one side of the merge and duplicated merge-by column values on the other side. Many-to-many merges have duplicated merge-by column values on both sides. Merging is further complicated by the fact that there is often no perfect correspondence between merge-by values on the data tables; each data table may have values in the merge-by column that are not present in the other data table.

New data issues can be introduced when data is combined. When data is appended, it may have different logical values than the original data, even when the columns have the same names and data types. For merges, whenever merge-by values are missing on one side of a merge, the other

columns from that side will also have missing values. For one-to-one or one-to-many merges, there may be unexpected duplicates in merge-by values, resulting in values for other columns being duplicated unintentionally.

In this chapter, we will combine DataFrames vertically and horizontally and consider strategies for dealing with the data problems that often arise. Specifically, in this chapter, the recipes will cover the following topics:

- Combining DataFrames vertically
- Doing one-to-one merges
- Doing one-to-one merges by multiple columns
- Doing one-to-many merges
- Doing many-to-many merges
- Developing a merge routine

## Technical requirements

You will need pandas, NumPy, and Matplotlib to complete the recipes in this chapter. I used pandas 2.1.4, but the code will run on pandas 1.5.3 or later.

The code in this chapter can be downloaded from the book's GitHub repository, <https://github.com/PacktPublishing/Python-Data-Cleaning-Cookbook-Second-Edition>.

## Combining DataFrames vertically

There are times when we need to append rows from one data table to another. This will almost always be rows from data tables that have nearly

the same columns and data types. For example, we might get a new CSV file containing hospital patient outcomes each month and need to add that to our existing data. Alternatively, we might end up working at a school district central office and receive data from many different schools. We might want to combine this data before conducting analyses.

Even when the data structure across months and across schools (in these examples) is theoretically the same, it may not be in practice. Business practices can change from one period to another. This can be intentional or happen inadvertently due to staff turnover or some external factor. One institution or department might implement practices somewhat differently than another, and some data values might be different for some institutions or missing altogether.

We are likely to come across a change in what seems like similar data when we let our guards down, typically when we start to assume that the new data will look like the old data. I try to remember this whenever I combine data vertically. I will be referring to combining data vertically as concatenating or appending for the rest of this chapter.

In this recipe, we'll use the pandas `concat` function to append rows from a pandas DataFrame to another DataFrame. We will also do a few common checks on the `concat` operation to confirm that the resulting DataFrame is what we expected.

## Getting ready

We will work with land temperature data from several countries in this recipe. This data includes the monthly average temperature, latitude, longitude, and elevation at many weather stations in each country during 2023. The data for each country is contained in a CSV file.



## Data note

The land temperature DataFrame has the average temperature reading (in °C) in 2023 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.ncei.noaa.gov/products/land-based-station/global-historical-climatology-network-monthly>.

## How to do it...

In this recipe, we will combine similarly structured DataFrames vertically, check the values in the concatenated data, and fix missing values. Let's get started:

1. Import `pandas` and `numpy`, as well as the `os` module:

```
import pandas as pd
import numpy as np
import os
```

2. Load the data from Cameroon and Oman and check the number of rows and columns:

```
ltcameroon = pd.read_csv("data/ltcountry/ltcameroon.csv")
ltoman = pd.read_csv("data/ltcountry/ltoman.csv")
```

```
ltcameroon.shape
```

```
(48, 11)
```

```
ltoman.shape
```

```
(288, 10)
```

Compare the columns in the Cameroon and Oman DataFrames.

Glancing at the columns, we can see that the Cameroon DataFrame has the `latabs` column and the Oman DataFrame does not. We can confirm this, and that there are no other columns in one DataFrame but not the other, using `symmetric_difference`. It shows that `latabs` is the only column in just one DataFrame:

```
ltcameroon.columns
```

```
Index(['locationid', 'year', 'month', 'temperature',
       'latitude', 'longitude', 'elevation', 'station',
       'countryid', 'country', 'latabs'],
      dtype='object')
```

```
ltoman.columns
```

```
Index(['locationid', 'year', 'month', 'temperature',
       'latitude', 'longitude', 'elevation', 'station',
       'countryid', 'country'],
      dtype='object')
```

```
ltcameroon.columns.\  
symmetric_difference(ltoman.columns)
```

```
Index(['latabs'], dtype='object')
```

3. We can still concatenate the two DataFrames. The only problem is that we now have one column, `latabs`, that has non-missing values for all rows for Cameroon and all missing values for Oman. We address this problem in the last step of this recipe:

```
ltall = pd.concat([ltcameroon, ltoman])  
ltall.country.value_counts()
```

```
country  
Oman      288  
Cameroon    48  
Name: count, dtype: int64
```

```
ltall[['country', 'station', 'temperature',  
       'latitude', 'latabs']].\n       sample(5, random_state=3)
```

|     | country  | station     | temperature | latitude |
|-----|----------|-------------|-------------|----------|
| 276 | Oman     | BAHLA       | 21.44       | 23.006   |
| 26  | Oman     | DIBA        | 21.85       | 25.617   |
| 281 | Oman     | RAS_AL_HADD | 23.74       | 22.306   |
| 15  | Cameroon | GAROUA      | 33.91       | 9.336    |
| 220 | Oman     | SOHAR_MAJIS | 30.85       | 24.467   |

```
ltall.groupby(['country'])['latabs'].count()
```

```
country
Cameroon    48
Oman        0
Name: latabs, dtype: int64
```

4. Create a function to do the concatenation that incorporates some of the data checks we have done. The function takes a list of filenames, loops through the list, reads the CSV file associated with each filename into a DataFrame, and then concatenates the DataFrame. We get the expected counts. We did not check the column names. We will do that in the next step.

```
def concatfiles(filelist):
    directory = "data/ltcountry/"
    ltall = pd.DataFrame()
    for filename in filelist:
        ltnew = pd.read_csv(directory + filename + ".csv")
        print(filename + " has " +
              str(ltnew.shape[0]) + " rows.")
        ltall = pd.concat([ltall, ltnew])
    return ltall
ltall = concatfiles(['ltcameroon', 'lтоман'])
```

```
ltcameroon has 48 rows.
lтоман has 288 rows.
```

```
ltall.country.value_counts()
```

```
country
Oman      288
Cameroon   48
Name: count, dtype: int64
```

If we have many files to concatenate, it might be burdensome to create a list of the filenames. We can get Python's `os` module to help us with that by loading all files with a CSV file extension in a folder. Let's do that next, and also add some code to check columns. We will build on the code from the previous step.

## 5. Concatenate all the country data files in a folder.

Loop through all the filenames in the folder that contains the CSV files for each country. Use the `endswith` method to check that the filenames have a CSV file extension. Use `read_csv` to create a new DataFrame and print out the number of rows. Use `concat` to append the rows of the new DataFrame to the rows that have already been appended. Finally, display any columns that are missing in the most recent DataFrame, or that are in the most recent DataFrame but not the previous ones:

```
def concatallfiles():
    directory = "data/ltcountry"
    ltall = pd.DataFrame()
    for filename in os.listdir(directory):
        if filename.endswith(".csv"):
            fileloc = os.path.join(directory, filename)

            # open the next file
            with open(fileloc):
                ltnew = pd.read_csv(fileloc)
                print(filename + " has " +
                      str(ltnew.shape[0]) + " rows.")
                ltall = pd.concat([ltall, ltnew])

            # check for differences in columns
            columndiff = ltall.columns.\n                symmetric_difference(ltnew.columns)
            if (not columndiff.empty):
                print("", "Different column names for:",
```

```
    filename, columndiff, "", sep="\n")  
  
return ltall
```

6. Use the function we just created to read all of the country CSV files in a subfolder, show the number of rows, and check column names. We see again that the `ltoman` DataFrame is missing the `latabs` column:

```
ltall = concatallfiles()
```

```
ltpoland.csv has 120 rows.  
ltcameroon.csv has 48 rows.  
ltmexico.csv has 852 rows.  
ltjapan.csv has 1800 rows.  
ltindia.csv has 1116 rows.  
ltoman.csv has 288 rows.  
Different column names for:  
ltoman.csv  
Index(['latabs'], dtype='object')  
ltbrazil.csv has 1008 rows.
```

7. Show some of the combined data:

```
ltall[['country', 'station', 'month',  
       'temperature', 'latitude']].\n       sample(5, random_state=1)
```

|     | country | station             | month | temperature | latabs |
|-----|---------|---------------------|-------|-------------|--------|
| 583 | Japan   | TOKUSHIMA           | 4     | 16          |        |
| 635 | India   | NEW_DELHI_SAFDARJUN | 7     | 31          |        |
| 627 | Mexico  | COATZACOALCOSVER    | 9     | 30          |        |

|     |        |             |    |    |
|-----|--------|-------------|----|----|
| 28  | Poland | WLODAWA     | 3  | 5  |
| 775 | Mexico | ARRIAGACHIS | 11 | 28 |

## 8. Check the values in the concatenated data.

Notice that the values for `latabs` for Oman are all missing. This is because `latabs` is missing in the DataFrame for Oman (`latabs` is the absolute value of the latitude for each station):

```
ltall.country.value_counts().sort_index()
```

```
country
Brazil      1008
Cameroon     48
India        1116
Japan         1800
Mexico        852
Oman          288
Poland        120
Name: count, dtype: int64
```

```
ltall.groupby(['country']).\n    agg({'temperature': ['mean', 'max', 'count'],
        'latabs': ['mean', 'max', 'count']})
```

| country  | temperature |     |       | latabs |     |       |
|----------|-------------|-----|-------|--------|-----|-------|
|          | mean        | max | count | mean   | max | count |
| Brazil   | 25          | 34  | 900   | 14     | 34  | 1008  |
| Cameroon | 27          | 35  | 39    | 8      | 10  | 48    |
| India    | 26          | 35  | 1096  | 21     | 34  | 1116  |
| Japan    | 14          | 31  | 1345  | 36     | 45  | 1800  |
| Mexico   | 23          | 36  | 685   | 22     | 32  | 852   |

|        |    |    |     |     |     |     |
|--------|----|----|-----|-----|-----|-----|
| Oman   | 28 | 38 | 264 | NaN | NaN | 0   |
| Poland | 10 | 21 | 120 | 52  | 55  | 120 |

## 9. Fix the missing values.

Set the value of `latabs` to the value of `latitude` for Oman. (All of the `latitude` values for stations in Oman are above the equator and positive. In the Global Historical Climatology Network integrated database, latitude values above the equator are positive, while all the latitude values below the equator are negative). Do this as follows:

```
ltall['latabs'] = np.where(ltall.country=="Oman", ltall.latitude,
ltall.groupby(['country']).\n...
    agg({'temperature':['mean', 'max', 'count'],
...     'latabs':['mean', 'max', 'count']})
```

| country  | temperature |     |       | latabs |     |       |
|----------|-------------|-----|-------|--------|-----|-------|
|          | mean        | max | count | mean   | max | count |
| Brazil   | 25          | 34  | 900   | 14     | 34  | 1008  |
| Cameroon | 27          | 35  | 39    | 8      | 10  | 48    |
| India    | 26          | 35  | 1096  | 21     | 34  | 1116  |
| Japan    | 14          | 31  | 1345  | 36     | 45  | 1800  |
| Mexico   | 23          | 36  | 685   | 22     | 32  | 852   |
| Oman     | 28          | 38  | 264   | 22     | 26  | 288   |
| Poland   | 10          | 21  | 120   | 52     | 55  | 120   |

With that, we have combined the data for the seven CSV files we found in the selected folder. We have also confirmed that we have appended the correct number of rows, identified columns that are missing in some files, and fixed missing values.

# How it works...

We passed a list of pandas DataFrames to the pandas `concat` function in *step 3*. The rows from the second DataFrame were appended to the bottom of the first DataFrame. If we had listed a third DataFrame, those rows would have been appended to the combined rows of the first two DataFrames. Before concatenating, we used the `shape` attribute to check the number of rows in *step 2* and checked the column names. After concatenation in *step 3*, we confirmed that the resulting DataFrame contained the number of expected rows for each country.

We sometimes have to concatenate more than two or three files. *Steps 4* through *6* walked us through handling many files by defining a function to repeat the code. In *step 4*, we passed a list of filenames to that function.

In *steps 5* and *6*, we looked for all the CSV files in a specified folder, loaded each file that was found into memory, and then appended the rows of each file to a DataFrame. We printed the number of rows for each data file we loaded so that we could check those numbers against the totals in the concatenated data later. We also identified any DataFrames with different columns compared to the others. We used `value_counts` in *step 8* to confirm that there were the right number of rows for each country.

The pandas `groupby` method can be used to check column values from each of the original DataFrames. We group by country since that identifies the rows from each of the original DataFrames—all the rows for each DataFrame have the same value for country. (It is helpful to always have a column that identifies the original DataFrames in the concatenated DataFrame, even if that information is not needed for subsequent analysis.) In *step 8*, this helped us notice that there were no values for the `latabs`

column for Oman. We replaced the missing values for `latabs` for Oman in *step 9*.

## There's more...

Depending on the size of the DataFrames you are appending, and the available memory of your workstation, combining DataFrames may tax your machine's resources, or even cause the code to end prematurely once RAM usage exceeds a certain amount of your resources. It is always a good idea to make sure that your data files store data as efficiently as possible. For example, downcasting numeric values and making character data categorical when appropriate are good practices.

## See also

We went over the powerful pandas `groupby` method in some detail in *Chapter 9, Fixing Messy Data When Aggregating*.

We examined NumPy's `where` function in *Chapter 6, Cleaning and Exploring Data with Series Operations*.

## Doing one-to-one merges

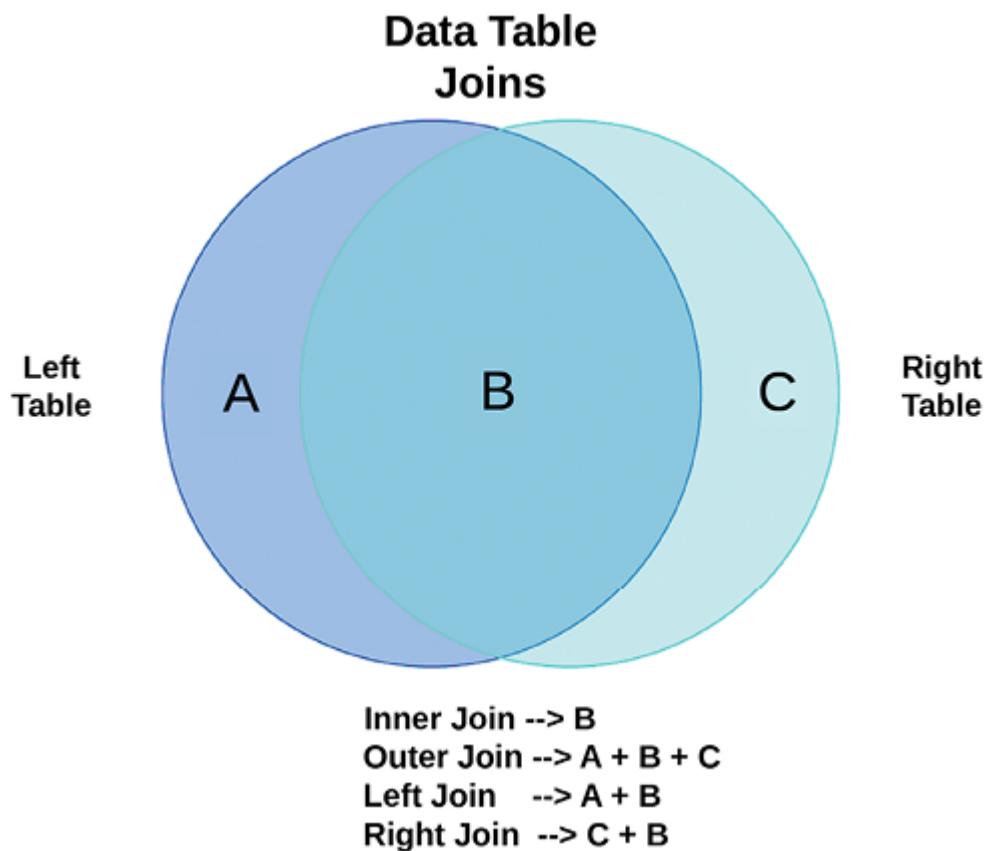
The remainder of this chapter will explore combining data horizontally; that is, merging columns from a data table with columns from another data table. Borrowing from SQL development, we typically talk about such operations as join operations: left joins, right joins, inner joins, and outer joins. This recipe examines one-to-one merges, where the merge-by values are unduplicated in both files. Subsequent recipes will demonstrate one-to-many merges, where the merge-by values are duplicated on the *right* data

table, and many-to-many merges, where merge-by values are duplicated on both the *left* and *right* data tables.

We often speak of the left and right sides of a merge, a convention that we will follow throughout this chapter. But this is of no real consequence, other than for clarity of exposition. We can accomplish exactly the same thing with a merge if A were the left data table and B were the right data table, as we could if the reverse were true.

I am using the expressions merge-by column and merge-by value in this chapter, rather than key column or index column. This avoids possible confusion with pandas index alignment. An index may be used as the merge-by column, but other columns may also be used. I also want to avoid relying on relational database concepts such as primary or foreign keys in this discussion. It is helpful to be aware of which data columns function as primary or foreign keys when we're extracting data from relational systems, and we should take this into account when setting indexes in pandas. But the merging we do for most data cleaning projects often goes beyond these keys.

In the straightforward case of a one-to-one merge, each row in the left data table is matched with one (and only one) row on the right data table, according to the merge-by value. What happens when a merge-by value appears on one, but not the other, data table is determined by the type of join that's specified. The following diagram illustrates the four different types of joins:



*Figure 10.1: A diagram illustrating the four different types of joins*

When two data tables are merged with an inner join, rows are retained when the merge-by values appear in both the left and right data tables. This is the intersection of the left and right data tables, represented by **B** in the preceding diagram. Outer joins return all rows; that is, rows where the merge-by values appear in both data tables, rows where those values appear in the left data table but not the right, and rows where those values appear in the right but not the left—**B**, **A**, and **C**, respectively. This is known as the union. Left joins return rows where the merge-by values are present on the left data table, regardless of whether they are present on the right data table. This is **A** and **B**. Right joins return rows where the merge-by values are present on the right data table, regardless of whether they are present on the left data table.

Missing values may result from outer joins, left joins, or right joins. This is because the returned merged data table will have missing values for columns when the merge-by value is not found. For example, when performing a left join, there may be merge-by values from the left dataset that do not appear on the right dataset. In this case, the columns from the right dataset will all be missing. (I say *may* here because it is possible to do an outer, left, or right join that returns the same results as an inner join because the same merge-by values appear on both sides. Sometimes, a left join is done so that we're certain that all the rows on the left dataset, and only those rows, are returned.)

We will look at all four types of joins in this recipe.

## Getting ready

We will work with two files from the **National Longitudinal Surveys (NLS)**. Both files contain one row per person. One contains employment, educational attainment, and income data, while the other file contains data on the income and educational attainment of the respondents' parents.

### Data note

The **National Longitudinal Surveys (NLS)**, administered by the United States Bureau of Labor Statistics, are longitudinal surveys of individuals who were in high school in 1997 when the surveys started. Participants were surveyed each year through 2023. The surveys are available for public use at [nlsinfo.org](https://nlsinfo.org).

# How to do it...

In this recipe, we will perform left, right, inner, and outer joins on two DataFrames that have one row for each merge-by value. Let's get started:

1. Import `pandas` and load the two NLS DataFrames:

```
import pandas as pd
nls97 = pd.read_csv("data/nls97f.csv", low_memory=False)
nls97.set_index("personid", inplace=True)
nls97add = pd.read_csv("data/nls97add.csv")
```

2. Look at some of the NLS data:

```
nls97.head()
```

```
      gender birthmonth birthyear ...
personid
135335    Female        9      1981 ...
999406     Male         7      1982 ...
151672    Female        9      1983 ...
750699    Female        2      1981 ...
781297     Male        10      1982 ...
                   colenrfeb22  colenroct22   originalid
personid
135335            NaN        NaN        1
999406            NaN        NaN        2
151672  1. Not enrolled        NaN        3
750699            NaN        NaN        4
781297            NaN        NaN        5
[5 rows x 106 columns]
```

```
nls97.shape
```

```
(8984, 106)
```

```
nls97add.head()
```

```
originalid    motherage    parentincome \
0            1            26             -3
1            2            19             -4
2            3            26            63000
3            4            33            11700
4            5            34             -3
fatherhighgrade    motherhighgrade
0                  16                  8
1                  17                 15
2                  -3                 12
3                  12                 12
4                  12                 12
```

```
nls97add.shape
```

```
(8984, 5)
```

3. Check that the number of unique values for `originalid` is equal to the number of rows.

We will use `originalid` for our merge-by column later:

```
nls97.originalid.nunique()==nls97.shape[0]
```

```
True
```

```
nls97add.originalid.nunique()==nls97add.shape[0]
```

```
True
```

#### 4. Create some mismatched IDs.

Unfortunately, the NLS data is a little too clean for our purposes. Due to this, we will mess up a couple of values for `originalid`:

```
nls97 = nls97.sort_values('originalid')
nls97add = nls97add.sort_values('originalid')
nls97.loc[[135335, 999406], "originalid"] = \
    nls97.originalid+10000
nls97.originalid.head(2)
```

```
personid
135335      10001
999406      10002
Name: originalid, dtype: int64
```

```
nls97add.loc[[0,1], "originalid"] = \
    nls97add.originalid+20000
nls97add.originalid.head(2)
```

```
0      20001
1      20002
Name: originalid, dtype: int64
```

#### 5. Use `join` to perform a left join.

`nls97` is the left DataFrame and `nls97add` is the right DataFrame when we use `join` in this way. Show the values for the mismatched IDs. Notice that the values for the columns from the right DataFrame are all missing when there is no matching ID on that DataFrame (the

`originalid` values 10001 and 10002 appear on the left DataFrame but not on the right DataFrame):

```
nls97.set_index("originalid", inplace=True)
nls97add.set_index("originalid", inplace=True)
nlsnew = nls97.join(nls97add)
nlsnew.loc[nlsnew.index>9999, ['originalid', 'gender', 'birthyear', 'motherage', 'parentinr']]
```

| originalid | gender | birthyear | motherage | parentinr |
|------------|--------|-----------|-----------|-----------|
| 10001      | Female | 1981      | NaN       |           |
| 10002      | Male   | 1982      | NaN       |           |

## 6. Perform a left join with `merge`.

The first DataFrame is the left DataFrame, while the second DataFrame is the right DataFrame. Use the `on` parameter to indicate the merge-by column. Set the value of the `how` parameter to `left` to do a left join. We get the same results that we get when using `join`:

```
nlsnew = pd.merge(nls97, nls97add, on=['originalid'], how='left')
nlsnew.loc[nlsnew.index>9999, ['gender', 'birthyear', 'motherage', 'parentinr']]
```

| originalid | gender | birthyear | motherage | parentinr |
|------------|--------|-----------|-----------|-----------|
| 10001      | Female | 1981      | NaN       |           |
| 10002      | Male   | 1982      | NaN       |           |

## 7. Perform a right join.

With a right join, the values from the left DataFrame are missing when there is no matching ID on the left DataFrame:

```
nlsnew = pd.merge(nls97, nls97add, on=['originalid'], how='right')
nlsnew.loc[nlsnew.index>9999, ['gender', 'birthyear', 'motherage', 'parentincome']]
```

| originalid | gender | birthyear | motherage | parentincome |
|------------|--------|-----------|-----------|--------------|
| 20001      | NaN    | NaN       | 26        |              |
| 20002      | NaN    | NaN       | 19        |              |

## 8. Perform an inner join.

None of the mismatched IDs (that have values over 9999) appear after the inner join. This is because they do not appear on both DataFrames:

```
nlsnew = pd.merge(nls97, nls97add, on=['originalid'], how='inner')
nlsnew.loc[nlsnew.index>9999, ['gender', 'birthyear', 'motherage', 'parentincome']]
```

```
Empty DataFrame
Columns: [gender, birthyear, motherage, parentincome]
Index: []
```

## 9. Perform an outer join.

This retains all the rows, so rows with merge-by values in the left DataFrame but not in the right are retained (`originalid` values 10001 and 10002), and rows with merge-by values in the right DataFrame

but not in the left are also retained (`originalid` values 20001 and 20002):

```
nlsnew = pd.merge(nls97, nls97add, on=['originalid'], how=
```

```
nlsnew.loc[nlsnew.index>9999, ['gender','birthyear','mothe
```

| originalid | gender | birthyear | motherage | parent |
|------------|--------|-----------|-----------|--------|
| 10001      | Female | 1,981     | NaN       |        |
| 10002      | Male   | 1,982     | NaN       |        |
| 20001      | NaN    | NaN       | 26        |        |
| 20002      | NaN    | NaN       | 19        |        |

## 10. Create a function to check for ID mismatches.

The function takes a left and right DataFrame, as well as a merge-by column. It performs an outer join because we want to see which merge-by values are present in either DataFrame, or both of them:

```
def checkmerge(dfleft, dfright, idvar):
```

```
...     dfleft['inleft'] = "Y"
```

```
...     dfright['inright'] = "Y"
```

```
...     dfboth = pd.merge(dfleft[[idvar, 'inleft']], \
```

```
...         dfright[[idvar, 'inright']], on=[idvar], how="outer")
```

```
...     dfboth.fillna('N', inplace=True)
```

```
...     print(pd.crosstab(dfboth.inleft, dfboth.inright))
```

```
...
```

```
checkmerge(nls97.reset_index(),nls97add.reset_index(), "or
```

| inright | N | Y |
|---------|---|---|
| inleft  |   |   |

|   |   |      |
|---|---|------|
| N | 0 | 2    |
| Y | 2 | 8982 |

With that, we have demonstrated how to perform the four types of joins with a one-to-one merge.

## How it works...

One-to-one merges are fairly straightforward. The merge-by column values only appear once on the left and right DataFrames. However, some merge-by column values may appear on only one DataFrame. This is what makes the type of join important. If all merge-by column values appeared on both DataFrames, then a left join, right join, inner join, or outer join would return the same result. We took a look at the two DataFrames in the first few steps.

In *step 3*, we confirmed that the number of unique values for the merge-by column (`originalid`) is equal to the number of rows in both DataFrames. This tells us that we will be doing a one-to-one merge.

If the merge-by column is the index, then the easiest way to perform a left join is to use the `join` DataFrame method. We did this in *step 5*. We passed the right DataFrame to the `join` method of the left DataFrame. The same result was returned when we performed a left join using the pandas `merge` function in *step 6*. We used the `how` parameter to specify a left join and indicated the merge-by column using `on`.

In *steps 7 to 9*, we performed the right, inner, and outer joins, respectively. This is specified by the `how` value, which is the only part of the code that is different across these steps.

The simple `checkmerge` function we created in *step 10* counted the number of rows with merge-by column values on one DataFrame but not the other, and the number of values on both. Passing copies of the two DataFrames to this function tells us that two rows are in the left DataFrame and not in the right, two rows are in the right DataFrame but not the left, and 8,982 rows are in both.

## There's more...

You should run a function similar to the `checkmerge` function we created in *step 10* before you do any non-trivial merge—which, in my opinion, is pretty much all merges.

The `merge` function is more flexible than the examples I have used in this recipe suggest. For example, in *step 6*, we did not have to specify the left DataFrame as the first parameter. I could have indicated the left and right DataFrames explicitly, like so:

```
nlsnew = pd.merge(right=nls97add, left=nls97, on=['originalid'])
```

We can also specify different merge-by columns for the left and right DataFrames by using `left_on` and `right_on` instead of `on`:

```
nlsnew = pd.merge(nls97, nls97add, left_on=['originalid'], righ
```

The flexibility of the `merge` function makes it a great tool any time we need to combine data horizontally.

# Doing one-to-one merges by multiple columns

The same logic we used to perform one-to-one merges with one merge-by column applies to merges we perform with multiple merge-by columns. Inner, outer, left, and right joins work the same way when you have two or more merge-by columns. We will demonstrate this in this recipe.

## Getting ready

We will work with the NLS data in this recipe, specifically weeks worked and college enrollment from 2017 through 2021. Both the weeks worked and college enrollment files contain one row per person, per year.

## How to do it...

We will do a one-to-one merge with two DataFrames using multiple merge-by columns on each DataFrame. Let's get started:

1. Import `pandas` and load the NLS weeks worked and college enrollment data:

```
import pandas as pd
nls97weeksworked = pd.read_csv("data/nls97weeksworked")
nls97colenr = pd.read_csv("data/nls97colenr.csv")
```

2. Look at some of the NLS weeks worked data:

```
nls97weeksworked.loc[nls97weeksworked.\n    originalid.isin([2,3])]
```

|    | originalid | year | weeksworked |
|----|------------|------|-------------|
| 5  | 2          | 2017 | 52          |
| 6  | 2          | 2018 | 52          |
| 7  | 2          | 2019 | 52          |
| 8  | 2          | 2020 | 52          |
| 9  | 2          | 2021 | 46          |
| 10 | 3          | 2017 | 52          |
| 11 | 3          | 2018 | 52          |
| 12 | 3          | 2019 | 9           |
| 13 | 3          | 2020 | 0           |
| 14 | 3          | 2021 | 0           |

```
nls97weeksworked.shape
```

```
(44920, 3)
```

```
nls97weeksworked.originalid.unique()
```

```
8984
```

3. Look at some of the NLS college enrollment data:

```
nls97colenr.loc[nls97colenr.\n    originalid.isin([2,3])]
```

|       | originalid | year | colenr          |
|-------|------------|------|-----------------|
| 1     | 2          | 2017 | 1. Not enrolled |
| 2     | 3          | 2017 | 1. Not enrolled |
| 8985  | 2          | 2018 | 1. Not enrolled |
| 8986  | 3          | 2018 | 1. Not enrolled |
| 17969 | 2          | 2019 | 1. Not enrolled |
| 17970 | 3          | 2019 | 1. Not enrolled |
| 26953 | 2          | 2020 | 1. Not enrolled |
| 26954 | 3          | 2020 | 1. Not enrolled |

|       |   |      |                 |
|-------|---|------|-----------------|
| 35937 | 2 | 2021 | 1. Not enrolled |
| 35938 | 3 | 2021 | 1. Not enrolled |

```
nls97colenr.shape
```

|            |
|------------|
| (44920, 3) |
|------------|

```
nls97colenr.originalid.nunique()
```

|      |
|------|
| 8984 |
|------|

4. Check for unique values in the merge-by columns.

We get the same number of merge-by column value combinations (44,920) as the number of rows in both DataFrames:

```
nls97weeksworked.groupby(['originalid', 'year'])\n...    ['originalid'].count().shape
```

|           |
|-----------|
| (44920, ) |
|-----------|

```
nls97colenr.groupby(['originalid', 'year'])\n...    ['originalid'].count().shape
```

|           |
|-----------|
| (44920, ) |
|-----------|

5. Check for mismatches in the merge-by columns. All `originalid` and `year` combinations appear on both files:

```

def checkmerge(dfleft, dfright, idvar):
    ... dfleft['inleft'] = "Y"
    ... dfright['inright'] = "Y"
    ... dfboth = pd.merge(dfleft[idvar + ['inleft']], \
    ...     dfright[idvar + ['inright']], on=idvar, how=""
    ... dfboth.fillna('N', inplace=True)
    ... print(pd.crosstab(dfboth.inleft, dfboth.inright
    ...
checkmerge(nls97weeksworked.copy(), nls97colenr.copy())

```

|         |       |
|---------|-------|
| inright | Y     |
| inleft  |       |
| Y       | 44920 |

## 6. Perform a merge with multiple merge-by columns:

```

nls97workschool = \
pd.merge(nls97weeksworked, nls97colenr,
        on=['originalid', 'year'], how="inner")
nls97workschool.shape

```

(44920, 4)

```

nls97workschool.loc[nls97workschool.\
originalid.isin([2,3])]

```

| originalid | year | weeksworked | colenr          |
|------------|------|-------------|-----------------|
| 5          | 2017 | 52          | 1. Not enrolled |
| 6          | 2018 | 52          | 1. Not enrolled |
| 7          | 2019 | 52          | 1. Not enrolled |
| 8          | 2020 | 52          | 1. Not enrolled |
| 9          | 2021 | 46          | 1. Not enrolled |
| 10         | 2017 | 52          | 1. Not enrolled |
| 11         | 2018 | 52          | 1. Not enrolled |
| 12         | 2019 | 9           | 1. Not enrolled |

|    |   |      |   |                 |
|----|---|------|---|-----------------|
| 13 | 3 | 2020 | 0 | 1. Not enrolled |
| 14 | 3 | 2021 | 0 | 1. Not enrolled |

These steps demonstrate that the syntax for running merges changes very little when there are multiple merge-by columns.

## How it works...

Every person in the NLS data has five rows for both the weeks worked and college enrollment DataFrames, with one for each year between 2017 and 2021. Both files contain 44,920 rows with 8,984 unique individuals (indicated by `originalid`). This all makes sense ( $8,984 \times 5 = 44,920$ ).

*Step 4* confirmed that the combination of columns we will be using for the merge-by columns will not be duplicated, even if individuals are duplicated. Each person has only one row for each year. This means that the merging of the weeks worked and college enrollment data will be a one-to-one merge.

In *step 5*, we checked to see whether there were any individual and year combinations that were in one DataFrame but not the other. There were none.

Finally, we were ready to do the merge in *step 6*. We set the `on` parameter to a list with two column names (`['originalid', 'year']`) to tell the merge function to use both columns in the merge. We specified an inner join, even though we would get the same results with any join. This is because the same merge-by values are present in both files.

## There's more...

All the logic and potential issues in merging data that we discussed in the previous recipe apply, regardless of whether we are merging with one merge-by column or several. Inner, outer, right, and left joins work the same way. We can still calculate the number of rows that will be returned before doing the merge. We should also check for the number of unique merge-by values and for matches between the DataFrames.

If you have worked with recipes in earlier chapters that used the NLS weeks worked and college enrollment data, you probably noticed that it is structured differently here. In previous recipes, there was one row per person, with multiple columns for weeks worked and college enrollment, representing weeks worked and college enrollment for multiple years. For example, `weeksworked21` is the number of weeks worked in 2021. The structure of the weeks worked and college enrollment DataFrames we used in this recipe is considered *tidier* than the NLS DataFrame we used in earlier recipes. We'll learn how to tidy data in *Chapter 11, Tidying and Reshaping Data*.

## Doing one-to-many merges

In one-to-many merges, there are unduplicated values for the merge-by column or columns on the left data table and duplicated values for those columns on the right data table. For these merges, we usually do either an inner join or a left join. Which of those two join types we use matters when merge-by values are missing on the right data table. When performing a left join, all the rows that would be returned from an inner join will be returned, plus one row for each merge-by value present on the left dataset, but not the right. For those additional rows, values for all the columns on the right dataset will be missing in the resulting merged data. This relatively

straightforward fact ends up mattering a fair bit and should be thought through carefully before you code a one-to-many merge.

This is where I start to get nervous, and where I think it makes sense to be a little nervous. When I do workshops on data cleaning, I pause before starting this topic and say, *“Do not start a one-to-many merge until you are able to bring a friend with you.”*

I am joking, of course... mostly. The point I am trying to make is that something should happen to get us to pause before doing a non-trivial merge, and one-to-many merges are never trivial. Too much about the structure of our data can change.

Specifically, there are several things we want to know about the two DataFrames we will be merging before starting. First, we should know what columns make sense as merge-by columns on each DataFrame. One-to-many merges are often used to recapture relationships from an enterprise database system and need to be consistent with the primary keys and foreign keys used. (The primary key on the left data table is often linked to the foreign key on the right data table in a relational database.) Second, we should know what kind of join we will be using and why.

Third, we should know how many rows are on both data tables. Fourth, we should have a good idea of how many rows will be retained based on the type of join, the number of rows in each dataset, and preliminary checks on how many of the merge-by values will match. If all the merge-by values are present on both datasets or if we are doing an inner join, then the number of rows will be equal to the number of rows of the right dataset of a one-to-many merge. But it is often not as straightforward as that. We frequently perform left joins with one-to-many merges. With a left join, the number of retained rows will be equal to the number of rows in the right dataset with a

matching merge-by value, plus the number of rows in the left dataset with non-matching merge-by values.

This should be clearer once we've worked through the examples in this recipe.

## Getting ready

We will be working with data based on weather stations from the Global Historical Climatology Network integrated database for this recipe. One of the DataFrames contains one row for each country. The other contains one row for each weather station. There are typically many weather stations for each country.

## How to do it...

In this recipe, we will do a one-to-many merge of data for countries, which contains one row per country, with weather station data, which contains multiple stations for each country. Let's get started:

1. Import `pandas` and load the weather station and country data:

```
import pandas as pd
countries = pd.read_csv("data/ltcountries.csv")
locations = pd.read_csv("data/ltlocations.csv")
```

2. Set the index for the weather station (`locations`) and country data.

Confirm that the merge-by values for the `countries` DataFrame are unique:

```
countries.set_index(['countryid'], inplace=True)
locations.set_index(['countryid'], inplace=True)
countries.head()
```

```
country
countryid
AC           Antigua and Barbuda
AE           United Arab Emirates
AF           Afghanistan
AG           Algeria
AJ           Azerbaijan
```

```
countries.index.nunique() == countries.shape[0]
```

```
True
```

```
locations[['locationid', 'latitude', 'stnelev']].head(10)
```

```
locationid      latitude    stnelev
countryid
AC          ACW00011604      58        18
AE          AE000041196       25        34
AE          AEM00041184      26        31
AE          AEM00041194      25        10
AE          AEM00041216      24         3
AE          AEM00041217      24        27
AE          AEM00041218      24       265
AF          AF000040930      35   3,366
AF          AFM00040911      37        378
AF          AFM00040938      34       977
```

3. Perform a left join of countries and locations using `join`:

```
stations = countries.join(locations)
stations[['locationid','latitude',
...      'stnelev','country']].head(10)
```

| countryid | locationid           | latitude | stnelev | \ |
|-----------|----------------------|----------|---------|---|
| AC        | ACW00011604          | 58       | 18      |   |
| AE        | AE000041196          | 25       | 34      |   |
| AE        | AEM00041184          | 26       | 31      |   |
| AE        | AEM00041194          | 25       | 10      |   |
| AE        | AEM00041216          | 24       | 3       |   |
| AE        | AEM00041217          | 24       | 27      |   |
| AE        | AEM00041218          | 24       | 265     |   |
| AF        | AF000040930          | 35       | 3,366   |   |
| AF        | AFM00040911          | 37       | 378     |   |
| AF        | AFM00040938          | 34       | 977     |   |
| countryid | country              |          |         |   |
| AC        | Antigua and Barbuda  |          |         |   |
| AE        | United Arab Emirates |          |         |   |
| AE        | United Arab Emirates |          |         |   |
| AE        | United Arab Emirates |          |         |   |
| AE        | United Arab Emirates |          |         |   |
| AE        | United Arab Emirates |          |         |   |
| AE        | United Arab Emirates |          |         |   |
| AF        | Afghanistan          |          |         |   |
| AF        | Afghanistan          |          |         |   |
| AF        | Afghanistan          |          |         |   |

The join seemed to work fine. But let's try using merge instead.

#### 4. Check for merge-by column mismatches before doing the merge.

First, reload the DataFrames since we have made some changes. The `checkmerge` function shows that there are 27,472 rows with merge-by values (from `countryid`) in both DataFrames and 2 in `countries` (the left DataFrame) but not in `locations`. This indicates that an inner join would return 27,472 rows and a left join would return 27,474

rows. The last statement in the function identifies the `countryid` values that appear in one DataFrame but not the other:

```
countries = pd.read_csv("data/ltcountries.csv")
locations = pd.read_csv("data/ltlocations.csv")
def checkmerge(dfleft, dfright, idvar):
    ... dfleft['inleft'] = "Y"
    ... dfright['inright'] = "Y"
    ... dfboth = pd.merge(dfleft[[idvar, 'inleft']], \
        ... dfright[[idvar, 'inright']], on=[idvar], how="outer")
    ... dfboth.fillna('N', inplace=True)
    ... print(pd.crosstab(dfboth.inleft, dfboth.inright))
    ... print(dfboth.loc[(dfboth.inleft=='N') | (dfboth.inri])
    ...
checkmerge(countries.copy(), locations.copy(), "countryid")
```

| inright | N         | Y      |         |
|---------|-----------|--------|---------|
| inleft  |           |        |         |
| N       | 0         | 1      |         |
| Y       | 2         | 27472  |         |
|         | countryid | inleft | inright |
| 9715    | LQ        | Y      | N       |
| 13103   | ST        | Y      | N       |
| 27474   | FO        | N      | Y       |

## 5. Show the rows in one file but not the other.

The last statement in the previous step displays the two values of `countryid` in `countries` but not in `locations`, and the one in `locations` but not in `countries`:

```
countries.loc[countries.countryid.isin(["LQ", "ST"])]
```

|     | countryid | country                       |
|-----|-----------|-------------------------------|
| 124 | LQ        | Palmyra Atoll [United States] |

|     |    |             |
|-----|----|-------------|
| 195 | ST | Saint Lucia |
|-----|----|-------------|

```
locations.loc[locations.countryid=="FO"]
```

|      | locationid  | latitude | longitude | stnelev | station  |
|------|-------------|----------|-----------|---------|----------|
| 7363 | FOM00006009 | 61       | -7        | 102     | AKRABERG |

## 6. Merge the `locations` and `countries` DataFrames.

Perform a left join. Also, count the number of missing values for each column, where merge-by values are present in the countries data but not in the weather station data:

```
stations = pd.merge(countries, locations, on=["countryid"])
stations[['locationid', 'latitude',
...         'stnelev', 'country']].head(10)
```

|   | locationid  | latitude | stnelev | cc |
|---|-------------|----------|---------|----|
| 0 | ACW00011604 | 58       | 18      | Ar |
| 1 | AE000041196 | 25       | 34      | Ur |
| 2 | AEM00041184 | 26       | 31      | Ur |
| 3 | AEM00041194 | 25       | 10      | Ur |
| 4 | AEM00041216 | 24       | 3       | Ur |
| 5 | AEM00041217 | 24       | 27      | Ur |
| 6 | AEM00041218 | 24       | 265     | Ur |
| 7 | AF000040930 | 35       | 3,366   | A1 |
| 8 | AFM00040911 | 37       | 378     | A1 |
| 9 | AFM00040938 | 34       | 977     | A1 |

```
stations.shape
```

```
(27474, 7)
```

```
stations.loc[stations.countryid.isin(["LQ", "ST"])].isnull()
```

```
countryid      0  
country 0  
locationid     2  
latitude        2  
longitude       2  
stnelev 2  
station 2  
dtype: int64
```

The one-to-many merge returns the expected number of rows and new missing values.

## How it works...

In *step 3*, we used the `join` DataFrame method to perform a left join of the `countries` and `locations` DataFrames. This is the easiest way to do a merge. Since the `join` method uses the index of the DataFrames for the merge, we need to set the index first. We then passed the right DataFrame to the `join` method of the left DataFrame.

Although `join` is a little more flexible than this example suggests (you can specify the type of join, for example), I prefer the more verbose pandas `merge` function for all but the simplest of merges. I can be confident when using the `merge` function that all the options I need are available to me.

Before we used the `merge` function, we did some checks in *step 4*. This told us how many rows to expect in the merged DataFrame if we were to do an inner or left join; there would be 27,472 or 27,474 rows, respectively.

We also displayed the rows with merge-by values in one DataFrame but not the other. If we are going to do a left join, we need to decide what to do with the missing values that will result from the right DataFrame. In this case, there were two merge-by values that were not found on the right DataFrame, giving us missing values for those columns.

## There's more...

You may have noticed that in our call to `checkmerge`, we passed copies of the `countries` and `locations` DataFrames:

```
checkmerge(countries.copy(), locations.copy(), "countryid")
```

We use `copy` here because we do not want the `checkmerge` function to make any changes to our original DataFrames.

## See also

We discussed join types in detail in the *Doing one-to-one merges* recipe in this chapter.

## Doing many-to-many merges

Many-to-many merges have duplicate merge-by values in both the left and right DataFrames. We should only rarely need to do a many-to-many merge. Even when data comes to us in that form, it is often because we are missing

the central file in multiple one-to-many relationships. For example, there are donor, donor contributions, and donor contact information data tables, and the last two files contain multiple rows per donor. However, in this case, we do not have access to the donor file, which has a one-to-many relationship with both the contributions and contact information files. This happens more frequently than you may think. People sometimes give us data with little awareness of the underlying structure. When I do a many-to-many merge, it is typically because I am missing some key information rather than because that was how the database was designed.

Many-to-many merges return the Cartesian product of the merge-by column values. So, if a donor ID appears twice on the donor contact information file and five times on the donor contributions file, then the merge will return 10 rows. This is often quite problematic analytically. In this example, a many-to-many merge will duplicate the donor contributions, once for each address.

Often, when faced with a potential many-to-many merge situation, the solution is not to do it. Instead, we can recover the implied one-to-many relationships. With the donor example, we could remove all the rows except for the most recent contact information, thus ensuring that there is one row per donor. We could then do a one-to-many merge with the donor contributions file. But we are not always able to avoid doing a many-to-many merge. Sometimes, we must produce an analytical or flat file that keeps all of the data, without regard for duplication. This recipe demonstrates how to do those merges when that is required.

## Getting ready

We will work with data based on the Cleveland Museum of Art's collections. We will use two CSV files: one containing each media citation for each item in the collection and another containing the creator(s) of each item.



### Data note

The Cleveland Museum of Art provides an API for public access to this data: <https://openaccess-api.clevelandart.org/>. Much more than the citations and creators data is available in the API. The data in this recipe was downloaded in April 2024.

## How to do it...

Follow these steps to complete this recipe:

1. Load `pandas` and the **Cleveland Museum of Art (CMA)** collections data:

```
import pandas as pd
cmacitations = pd.read_csv("data/cmacitations.csv")
cmacreators = pd.read_csv("data/cmacreators.csv")
```

2. Look at the `citations` data. The `itemid` is the identifier for a collection item. The first 10 citations are all for collection item `94979`:

```
cmacitations['citation'] = cmacitations.citation.str[cmacitations.head(10)]
```

```
      itemid      citation
0    94979  Perkins, August
1    94979   Bayley, Frank W
2    94979  W. H. D. "The F
3    94979  <em>The America
4    94979  "Clevel'd Gets
5    94979  "The Inaugurati
6    94979   Bell, Hamilton.
7    94979  Cleveland Museu
8    94979  "Special Exhibi
9    94979   Dunlap, William
```

```
cmacitations.shape
```

```
(16053, 2)
```

```
cmacitations.itemid.unique()
```

```
974
```

3. Look at the `creators` data. The `creatorid` is the identifier of the creator:

```
cmacreators['creator'] = cmacreators.creator.str[0:15
cmacreators.loc[:, ['itemid', 'creator', 'birth_year',
'creatorid']].head(10)
```

```
      itemid      creator  birth_year  creatorid
0    94979  John Singleton      1738       2409
1   102578  William Merritt      1849       3071
2    92937  George Bellows      1882       3005
3   151904  Thomas Eakins (      1844       4037
4   141639  Frederic Edwin      1826       2697
5   110180  Albert Pinkham      1847       3267
```

|   |        |                 |      |      |
|---|--------|-----------------|------|------|
| 6 | 149112 | Charles Sheeler | 1883 | 889  |
| 7 | 126769 | Henri Rousseau  | 1844 | 1804 |
| 8 | 149410 | Paul Gauguin (F | 1848 | 1776 |
| 9 | 135299 | Vincent van Gog | 1853 | 1779 |

```
cmacreators.shape
```

```
(694, 8)
```

```
cmacreators.itemid.nunique()
```

```
618
```

```
cmacreators.creatorid.nunique()
```

```
486
```

4. Show duplicates of merge-by values in the `citations` data.

There are 182 media citations for collection item 148758:

```
cmacitations.itemid.value_counts().head(10)
```

| itemid |     |
|--------|-----|
| 148758 | 182 |
| 113164 | 127 |
| 122351 | 125 |
| 155783 | 119 |
| 151904 | 112 |
| 124245 | 108 |
| 92937  | 104 |

```
123168      98
94979       98
149112      97
Name: count, dtype: int64
```

## 5. Show duplicates of the merge-by values in the `creators` data:

```
cmacreators.itemid.value_counts().head(10)
```

```
itemid
149386      4
142753      3
112932      3
149042      3
114538      3
140001      3
146797      3
149041      3
140427      3
109147      2
Name: count, dtype: int64
```

## 6. Check the merge.

Use the `checkmerge` function we used in the *Doing one-to-many merges* recipe:

```
def checkmerge(dfleft, dfright, idvar):
...     dfleft['inleft'] = "Y"
...     dfright['inright'] = "Y"
...     dfboth = pd.merge(dfleft[[idvar, 'inleft']], \
...                       dfright[[idvar, 'inright']], on=[idvar], how="outer")
...     dfboth.fillna('N', inplace=True)
...     print(pd.crosstab(dfboth.inleft, dfboth.inright))
```

```
...  
checkmerge(cmacitations.copy(), cmaccreators.copy(), "itemid")
```

|   | inright | N    | Y     |
|---|---------|------|-------|
|   | inleft  |      |       |
| N |         | 0    | 14    |
| Y |         | 4277 | 12710 |

## 7. Show a merge-by value duplicated in both DataFrames:

```
cmacitations.loc[cmacitations.itemid==124733]
```

|       | itemid | citation        |
|-------|--------|-----------------|
| 14533 | 124733 | Weigel, J. A. G |
| 14534 | 124733 | Winkler, Friedr |
| 14535 | 124733 | Francis, Henry  |
| 14536 | 124733 | Kurz, Otto. <em |
| 14537 | 124733 | Minneapolis Ins |
| 14538 | 124733 | Pilz, Kurt. "Ha |
| 14539 | 124733 | Koschatzky, Wal |
| 14540 | 124733 | Johnson, Mark M |
| 14541 | 124733 | Kaufmann, Thoma |
| 14542 | 124733 | Koreny, Fritz.  |
| 14543 | 124733 | Achilles-Syndra |
| 14544 | 124733 | Schoch, Rainer, |
| 14545 | 124733 | DeGrazia, Diane |
| 14546 | 124733 | Dunbar, Burton  |

```
cmaccreators.loc[cmaccreators.itemid==124733,  
...     ['itemid', 'creator', 'birth_year', 'title']]
```

|     | itemid | creator         | birth_year |        |
|-----|--------|-----------------|------------|--------|
| 591 | 124733 | Hans Hoffmann ( | 1545       | Dead E |

|     |        |                |      |        |
|-----|--------|----------------|------|--------|
| 592 | 124733 | Albrecht Dürer | 1471 | Dead E |
|-----|--------|----------------|------|--------|

8. Do a many-to-many merge:

```
cma = pd.merge(cmacitations, cmacreators, on=['itemid']
cma.set_index("itemid", inplace=True)
cma.loc[124733, ['citation', 'creator', 'birth_year']]
```

| itemid                                 | citation        | creator         | birth_year |
|----------------------------------------|-----------------|-----------------|------------|
| 124733                                 | Weigel, J. A. G | Hans Hoffmann ( | 154        |
| 124733                                 | Weigel, J. A. G | Albrecht Dürer  | 147        |
| 124733                                 | Winkler, Friedr | Hans Hoffmann ( | 154        |
| 124733                                 | Winkler, Friedr | Albrecht Dürer  | 147        |
| 124733                                 | Francis, Henry  | Hans Hoffmann ( | 154        |
| 124733                                 | Francis, Henry  | Albrecht Dürer  | 147        |
| 124733                                 | Kurz, Otto. <em | Hans Hoffmann ( | 154        |
| 124733                                 | Kurz, Otto. <em | Albrecht Dürer  | 147        |
| 124733                                 | Minneapolis Ins | Hans Hoffmann ( | 154        |
| 124733                                 | Minneapolis Ins | Albrecht Dürer  | 147        |
| 124733                                 | Pilz, Kurt. "Ha | Hans Hoffmann ( | 154        |
| 124733                                 | Pilz, Kurt. "Ha | Albrecht Dürer  | 147        |
| 124733                                 | Koschatzky, Wal | Hans Hoffmann ( | 154        |
| 124733                                 | Koschatzky, Wal | Albrecht Dürer  | 147        |
| ... last 14 rows removed to save space |                 |                 |            |

Now that I have taken you through the messiness of a many-to-many merge, I'll say a little more about how it works.

## How it works...

*Step 2* told us that there were 16,053 citations for 974 unique items. There is a unique ID, `itemid`, for each item in the museum's collection. On average, each item has 16 media citations (16,053/974). *Step 3* told us that there are

694 creators over 618 items that have a creator, so there is only one creator for the overwhelming majority of pieces. But the fact that there are duplicated `itemid`s (our merge-by value) on both the `citations` and `creators` DataFrames means that our merge will be a many-to-many merge.

*Step 4* gave us a sense of which `itemid`s are duplicated on the `citations` DataFrame. Some items in the museum's collection have more than 100 citations. It is worth taking a closer look at the citations for those items to see whether they make sense. *Step 5* showed us that even when there is more than one creator, there are rarely more than three. In *step 6*, we saw that most `itemid`s appear in both the `citations` file and the `creators` file, but a fair number have `citations` rows but no `creators` rows. We will lose those 4,277 rows if we do an inner join or a right join, but not if we do a left join or an outer join. (This assumes that the `citations` DataFrame is the left DataFrame and the `creators` DataFrame is the right one.)

We looked at an `itemid` value that is duplicated in both DataFrames in *step 7*. There are 14 rows for this collection item in the `citations` DataFrame and 2 in the `creators` DataFrame. This will result in 28 rows ( $2 * 14$ ) with that `itemid` in the merged DataFrame. The `citations` data will be repeated for each row in `creators`.

This was confirmed when we looked at the results of the merge in *step 8*. We performed an outer join with `itemid` as the merge-by column. When we displayed the rows in the merged file for the same ID we used in *step 7*, we got the 28 rows we were expecting (I removed the last 14 rows of output to save space).

## There's more...

It is good to understand what to expect when we do a many-to-many merge because there are times when it cannot be avoided. But even in this case, we can tell that the many-to-many relationship is really just two one-to-many relationships with the data file missing from the one side. There is likely a data table that contains one row per collection item that has a one-to-many relationship with both the `citations` data and the `creators` data. When we do not have access to a file like that, it is probably best to try to reproduce a file with that structure. With this data, we could have created a file containing `itemid` and maybe `title`, and then done one-to-many merges with the `citations` and `creators` data.

However, there are occasions when we must produce a flat file for subsequent analysis. We might need to do that when we, or a colleague who is getting the cleaned data from us, are using software that cannot handle relational data well. For example, someone in another department might do a lot of data visualization work with Excel. As long as that person knows which analyses require them to remove duplicated rows, a file with a structure like the one we produced in *step 8* might work fine.

## Developing a merge routine

I find it helpful to think of merging data as the parking lot of the data cleaning process. Merging data and parking may seem routine, but they are where a disproportionate number of accidents occur. One approach to getting in and out of parking lots without an incident occurring is to use a similar strategy each time you go to a particular lot. It could be that you always go to a relatively low-traffic area and you get to that area the same way most of the time.

I think a similar approach can be applied to getting in and out of merges with our data relatively unscathed. If we choose a general approach that works for us 80 to 90 percent of the time, we can focus on what is most important—the data, rather than the techniques for manipulating that data.

In this recipe, I will demonstrate the general approach that works for me, but the particular techniques I will use are not very important. I think it is just helpful to have an approach that you understand well and that you become comfortable using.

## Getting ready

We will return to the objectives we focused on in the *Doing one-to-many merges* recipe of this chapter. We want to do a left join of the `countries` data with the `locations` data from the Global Historical Climatology Network integrated database.

## How to do it...

In this recipe, we will do a left join of the `countries` and `locations` data after checking for merge-by value mismatches. Let's get started:

1. Import `pandas` and load the weather station and country data:

```
import pandas as pd
countries = pd.read_csv("data/ltcountries.csv")
locations = pd.read_csv("data/ltlocations.csv")
```

2. Check the merge-by column matches:

```
def checkmerge(dfleft, dfright, mergebyleft, mergebyr
...     dfleft['inleft'] = "Y"
```

```

...     dfright['inright'] = "Y"
...     dfboth = \
...         pd.merge(dfleft[[mergebyleft, 'inleft']], \
...             dfright[[mergebyright, 'inright']], \
...             left_on=[mergebyleft], \
...             right_on=[mergebyright], how="outer")
...     dfboth.fillna('N', inplace=True)
...     print(pd.crosstab(dfboth.inleft,
...                         dfboth.inright))
...     print(dfboth.loc[(dfboth.inleft=='N') | \
...                      (dfboth.inright=='N')].head(20))
checkmerge(countries.copy(), locations.copy(), "count"

```

| inright   | N      | Y       |   |
|-----------|--------|---------|---|
| inleft    |        |         |   |
| N         | 0      | 1       |   |
| Y         | 2      | 27472   |   |
| countryid | inleft | inright |   |
| 7363      | F0     | N       | Y |
| 9716      | LQ     | Y       | N |
| 13104     | ST     | Y       | N |

### 3. Merge the country and location data:

```

stations = pd.merge(countries, locations, left_on=["c
stations[['locationid','latitude',
...      'stnelev','country']].head(10)

```

|   | locationid  | latitude | stnelev | country        |
|---|-------------|----------|---------|----------------|
| 0 | ACW00011604 | 57.7667  | 18.0    | Antigua and E  |
| 1 | AE000041196 | 25.3330  | 34.0    | United Arab Em |
| 2 | AEM00041184 | 25.6170  | 31.0    | United Arab Em |
| 3 | AEM00041194 | 25.2550  | 10.4    | United Arab Em |
| 4 | AEM00041216 | 24.4300  | 3.0     | United Arab Em |
| 5 | AEM00041217 | 24.4330  | 26.8    | United Arab Em |
| 6 | AEM00041218 | 24.2620  | 264.9   | United Arab Em |
| 7 | AF000040930 | 35.3170  | 3366.0  | Afgha          |

|   |             |         |       |       |
|---|-------------|---------|-------|-------|
| 8 | AFM00040911 | 36.7000 | 378.0 | Afghā |
| 9 | AFM00040938 | 34.2100 | 977.2 | Afghā |

```
stations.shape
```

```
(27474, 7)
```

Here, we got the expected number of rows from a left join: 27,472 rows with merge-by values in both DataFrames and two rows with merge-by values in the left DataFrame, but not the right.

## How it works...

For the overwhelming majority of merges I do, something like the logic used in *steps 2 and 3* works well. We added a fourth argument to the `checkmerge` function we used in the previous recipe. This allows us to specify different merge-by columns for the left and right DataFrames. We do not need to recreate this function every time we do a merge. We can just include it in a module that we import. (We'll go over adding helper functions to modules in the final chapter of this book.)

Calling the `checkmerge` function before running a merge gives us enough information so that we know what to expect when running the merge with different join types. We will know how many rows will be returned from an inner, outer, left, or right join. We will also know where new missing values will be generated before we run the actual merge. Of course, this is a fairly expensive operation, requiring us to run a merge twice each time—one diagnostic outer join followed by whatever join we subsequently choose.

But I would argue that it is usually worth it, if for no other reason than that it helps us to stop and think about what we are doing.

Finally, we performed the merge in *step 3*. This is my preferred syntax. I always use the left DataFrame for the first argument and the right DataFrame for the second argument, though `merge` allows us to specify the left and right DataFrames in different ways. I also set values for `left_on` and `right_on`, even if the merge-by column is the same and I could use `on` instead (as we did in the previous recipe). This is so I will not have to change the syntax in cases where the merge-by column is different, and I like that it makes the merge-by column explicit for both DataFrames.

A somewhat more controversial routine is that I default to a left join, setting the `how` parameter to `left` initially. I make that my starting assumption and then ask myself if there is any reason to do a different join. The rows in the left DataFrame often represent my unit of analysis (students, patients, customers, and so on) and I am adding supplemental data from the right DataFrame (GPA, blood pressure, zip code, and so on). It may be problematic to remove rows from the unit of analysis because the merge-by value is not present on the right DataFrame, as would happen if I did an inner join instead. For example, in the *Doing one-to-one merges* recipe of this chapter, it probably would not have made sense to remove rows from the main NLS data because they did not appear on the supplemental data we have for parents.

## See also

We will create modules with useful data cleaning functions in *Chapter 12, Automate Data Cleaning with User-Defined Functions, Classes and Pipelines*.

We have discussed the types of joins in the *Doing one-to-one merges* recipe in this chapter.

## Summary

We carefully examined combining data vertically, also known as concatenating, and combining data horizontally, also known as merging, in this chapter. We went over key data issues when concatenating data, including different columns across files. We also considered key issues with merging data, such as missing merge-by column values and the unexpected duplication of data. We looked at how those issues change with the type of join used. In the next chapter, we will learn about tidying and reshaping messy data.

## Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review.  
Scan the QR code below to get a free eBook of your choice.



# 11

## Tidying and Reshaping Data

Echoing Leo Tolstoy’s wisdom (“Happy families are all alike; every unhappy family is unhappy in its own way.”), Hadley Wickham tells us, all tidy data is fundamentally alike, but all untidy data is messy in its own special way. How many times have we all stared at some rows of data and thought, “*What... how... why did they do that?*” This overstates the case somewhat. Although there are many ways that data can be poorly structured, there are limits to human creativity in this regard. It is possible to categorize the most frequent ways in which datasets deviate from normalized or tidy forms.

This was Hadley Wickham’s observation in his seminal work on tidy data. We can lean on that work, and our own experiences with oddly structured data, to prepare for the reshaping we have to do. Untidy data often has one or more of the following characteristics: a lack of clarity about merge-by column relationships; data redundancy on the *one* side of one-to-many relationships; data redundancy due to many-to-many relationships; values stored in column names; multiple values stored in one variable value; and data not being structured at the unit of analysis. (Although the last category is not necessarily a case of untidy data, some of the techniques we will review in the next few recipes are applicable to common unit-of-analysis problems.)

We use powerful tools in this chapter to deal with the challenges of data cleaning like the preceding. Specifically, we'll go over the following:

- Removing duplicated rows
- Fixing many-to-many relationships
- Using `stack` and `melt` to reshape data from wide to long format
- Melting multiple groups of columns
- Using `unstack` and `pivot` to reshape data from long to wide format

## Technical requirements

You will need pandas, NumPy, and Matplotlib to complete the recipes in this chapter. I used pandas 2.1.4, but the code will run on pandas 1.5.3 or later.

The code in this chapter can be downloaded from the book's GitHub repository, <https://github.com/PacktPublishing/Python-Data-Cleaning-Cookbook-Second-Edition>.

## Removing duplicated rows

There are several reasons why we might have data duplicated at the unit of analysis:

- The existing DataFrame may be the result of a one-to-many merge, and the one side is the unit of analysis.
- The DataFrame is repeated measures or panel data collapsed into a flat file, which is just a special case of the first situation.
- We may be working with an analysis file where multiple one-to-many relationships have been flattened, creating many-to-many

relationships.

When the *one* side is the unit of analysis, data on the *many* side may need to be collapsed in some way. For example, if we are analyzing outcomes for a cohort of students at a college, students are the unit of analysis; but we may also have course enrollment data for each student. To prepare the data for analysis, we might need to first count the number of courses, sum the total credits, or calculate the GPA for each student, before ending up with one row per student. To generalize from this example, we often need to aggregate the information on the *many* side before removing duplicated data.

In this recipe, we look at the pandas techniques for removing duplicate rows, and consider when we do and don't need to do aggregation during that process. We address duplication in many-to-many relationships in the next recipe.

## Getting ready

We will work with the COVID-19 daily case data in this recipe. It has one row per day per country, each row having the number of new cases and new deaths for that day. There are also demographic data for each country, and running totals for cases and deaths, so the last row for each country provides total cases and total deaths.

### Data note

Our World in Data provides COVID-19 public use data at  
<https://ourworldindata.org/covid-cases>.

The dataset includes total cases and deaths, tests



administered, hospital beds, and demographic data such as median age, gross domestic product, and diabetes prevalence. The dataset used in this recipe was downloaded on March 3, 2024.

## How to do it...

We use `drop_duplicates` to remove duplicated demographic data for each country in the COVID-19 daily data. We explore `groupby` as an alternative to `drop_duplicates` when we need to do some aggregation before removing duplicated data:

1. Import `pandas` and the COVID-19 daily cases data:

```
import pandas as pd
covidcases = pd.read_csv("data/covidcases.csv")
```

2. Create lists for the daily cases and deaths columns, case total columns, and demographic columns (the `total_cases` and `total_deaths` columns are the running totals for cases and deaths respectively for that country):

```
dailyvars = ['casedate', 'new_cases', 'new_deaths']
totvars = ['location', 'total_cases', 'total_deaths']
demovars = ['population', 'population_density',
...     'median_age', 'gdp_per_capita',
...     'hospital_beds_per_thousand', 'region']
covidcases[dailyvars + totvars + demovars].head(2).T
```

|           | 0          | 1          |
|-----------|------------|------------|
| casedate  | 2020-03-01 | 2020-03-15 |
| new_cases | 1          | 6          |

|                            |             |             |
|----------------------------|-------------|-------------|
| new_deaths                 | 0           | 0           |
| location                   | Afghanistan | Afghanistan |
| total_cases                | 1           | 7           |
| total_deaths               | 0           | 0           |
| population                 | 41128772    | 41128772    |
| population_density         | 54          | 54          |
| median_age                 | 19          | 19          |
| gdp_per_capita             | 1,804       | 1,804       |
| hospital_beds_per_thousand | 0           | 0           |
| region                     | South Asia  | South Asia  |

3. Create a DataFrame with just the daily data:

```
coviddaily = covidcases[['location'] + dailyvars]  
coviddaily.shape
```

```
(36501, 4)
```

```
coviddaily.head()
```

|   | location    | casedate   | new_cases | new_deaths |
|---|-------------|------------|-----------|------------|
| 0 | Afghanistan | 2020-03-01 | 1         | 0          |
| 1 | Afghanistan | 2020-03-15 | 6         | 0          |
| 2 | Afghanistan | 2020-03-22 | 17        | 0          |
| 3 | Afghanistan | 2020-03-29 | 67        | 2          |
| 4 | Afghanistan | 2020-04-05 | 183       | 3          |

4. Select one row per country.

Check to see how many countries (location) to expect by getting the number of unique locations. Sort by location and casedate. Then use `drop_duplicates` to select one row per location, and use the `keep` parameter to indicate that we want the last row for each country:

```
covidcases.location.nunique()
```

```
231
```

```
covidddemo = \  
    covidcases[['casedate'] + totvars + demovars].\  
    sort_values(['location', 'casedate']).\  
    drop_duplicates(['location'], keep='last').\  
    rename(columns={'casedate':'lastdate'})  
covidddemo.shape
```

```
(231, 10)
```

```
covidddemo.head(2).T
```

|                            |             |                                |
|----------------------------|-------------|--------------------------------|
| lastdate                   | 204         | 37                             |
| location                   | 2024-02-04  | 2024-01-2                      |
| total_cases                | Afghanistan | Albania                        |
| total_deaths               | 231,539     | 334,86                         |
| population                 | 7,982       | 3,60                           |
| population_density         | 41128772    | 284231                         |
| median_age                 | 54          | 16                             |
| gdp_per_capita             | 19          | 3                              |
| hospital_beds_per_thousand | 1,804       | 11,80                          |
| region                     | 0           | South Asia      Eastern Europe |

## 5. Sum the values for each group.

Use the pandas DataFrame groupby method to sum total cases and deaths for each country. (We calculate sums for cases and deaths here rather than using the running total of cases and deaths already in the

DataFrame.) Also, get the last value for some of the columns that are duplicated across all rows for each country: `median_age`, `gdp_per_capita`, `region`, and `casedate`. (We select only a few columns from the DataFrame.) Notice that the numbers match those from *step 4*:

```
covidtotals = covidcases.groupby(['location'],
...     as_index=False).\
...     agg({'new_cases':'sum', 'new_deaths':'sum',
...         'median_age':'last', 'gdp_per_capita':'last',
...         'region':'last', 'casedate':'last',
...         'population':'last'}).\
...     rename(columns={'new_cases':'total_cases',
...         'new_deaths':'total_deaths',
...         'casedate':'lastdate'})
covidtotals.head(2).T
```

|                | 0           | 1              |
|----------------|-------------|----------------|
| location       | Afghanistan | Albania        |
| total_cases    | 231,539     | 334,863        |
| total_deaths   | 7,982       | 3,605          |
| median_age     | 19          | 38             |
| gdp_per_capita | 1,804       | 11,803         |
| region         | South Asia  | Eastern Europe |
| lastdate       | 2024-02-04  | 2024-01-28     |
| population     | 41128772    | 2842318        |

The choice of `drop_duplicates` or `groupby` to eliminate data redundancy comes down to whether we need to do any aggregation before collapsing the *many* side.

## How it works...

The COVID-19 data has one row per country per day, but very little of the data is actually daily data. Only `casedate`, `new_cases`, and `new_deaths` can be considered daily data. The other columns show cumulative cases and deaths, or demographic data. The cumulative data is redundant since we have the actual values for `new_cases` and `new_deaths`. The demographic data has the same values for each country across all days.

There is an implied one-to-many relationship between the country (and its associated demographic data) on the *one* side and the daily data on the *many* side. We can recover that structure by creating a DataFrame with the daily data, and another DataFrame with the demographic data. We do that in *steps 3 and 4*. When we need totals across countries we can generate those ourselves, rather than storing redundant data.

The running totals variables are not completely useless, however. We can use them to check our calculations of total cases and total deaths. *Step 5* shows how we can use `groupby` to restructure data when we need to do more than drop duplicates. In this case, we want to summarize `new_cases` and `new_deaths` for each country.

## There's more...

I can sometimes forget a small detail. When changing the structure of data, the meaning of certain columns can change. In this example, `casedate` becomes the date for the last row for each country. We rename that column `lastdate`.

## See also

We explore `groupby` in more detail in *Chapter 9, Fixing Messy Data When Aggregating*.

Hadley Wickham's *Tidy Data* paper is available at  
<https://vita.had.co.nz/papers/tidy-data.pdf>.

## Fixing many-to-many relationships

We sometimes have to work with a data table that was created from a many-to-many merge. This is a merge where merge-by column values are duplicated on both the left and right sides. As we discussed in the previous chapter, many-to-many relationships in a data file often represent multiple one-to-many relationships where the *one* side has been removed. There is a one-to-many relationship between dataset A and dataset B, and also a one-to-many relationship between dataset A and dataset C. The problem we sometimes have is that we receive a data file with B and C merged but with A excluded.

The best way to work with data structured in this way is to recreate the implied one-to-many relationships, if possible. We do this by first creating a dataset structured like A; that is, how A is likely structured given the many-to-many relationship we see between B and C. The key to being able to do this is to identify a good merge-by column for the data on both sides of the many-to-many relationship. This column, or these columns, will be duplicated in both the B and C datasets, but will be unduplicated in the theoretical A dataset.

The data we use in this recipe is a good example. We have data from the Cleveland Museum of Art on its collections. We have multiple rows for every item in the museum's collection. Those rows have data on the

collection item (including title and creation date); the creator (including years of birth and death); and citations of the work in the press. When there are multiple creators and multiple citations, which is often, rows are duplicated. More precisely, the number of rows for each collection item is the Cartesian product of the number of citations and creations. So, if there are 5 citations and 2 creators, we see 10 rows for that item.

What we want instead is a collections file with one row (and a unique identifier) for each item in the collection, a creators file with one row per creator for each item, and a citations file with one row per citation of each item. We will create those files in this recipe.

Some of you will have noticed that there is still more tidying up to do here. We ultimately want a separate creator file with one row for every creator, and another file with just a creator id and a collection item id. We need this structure because a creator may be the creator for multiple items. We ignore that added complication in this recipe.

I should add that this situation is not the fault of the Cleveland Museum of Art, which generously provides an API that returns collections data as a JSON file. It is the responsibility of individuals who use the API to create data files that are most appropriate for their research purposes. It is also possible, and often a good choice, to work directly from the more flexible structure of a JSON file. We demonstrate how to do that in *Chapter 12, Automate Data Cleaning with User-Defined Functions, Classes, and Pipelines*.

## Getting ready

We will work with data on the Cleveland Museum of Art's collections. The CSV file has data on both creators and citations, merged by an `itemid`

column that identifies the collection item. There are one or many rows for citations and creators for each item.



### Data note

The Cleveland Museum of Art provides an API for public access to this data: <https://openaccess-api.clevelandart.org/>. Much more than the citations and creators data is available in the API. The data in this recipe were downloaded in April 2024.

## How to do it...

We handle many-to-many relationships between DataFrames by recovering the multiple implied one-to-many relationships in the data:

1. Import `pandas` and the museum's `collections` data. Let's also limit the length of values in the `collection` and `title` columns for easier display:

```
import pandas as pd
cma = pd.read_csv("data/cmacollections.csv")
cma['category'] = cma.category.str.strip().str[0:15]
cma['title'] = cma.title.str.strip().str[0:30]
```

2. Show some of the museum's `collections` data. Notice that almost all of the data values are redundant, except for `citation`.

Also, show the number of unique `itemid`, `citation`, and `creator` values. There are 986 unique collection items, 12,941 citations, and 1,062 item/creator combinations:

```
cma.shape
```

```
(17001, 9)
```

```
cma.head(4).T
```

```
          0  
itemid      75551      75551  
citation    Purrmann, Hans. <em>Henri Mati  
creatorid   2,130      2,130  
creator     Henri Matisse ( Henri Matisse  
title       Tulips      Tulip  
birth_year  1869       1869  
death_year  1954       1954  
category    Mod Euro - Pain Mod Euro - Pai  
creation_date 1914      1914  
              2  
itemid      75551      75551  
citation    Flam, Jack D. <em>Masters of  
creatorid   2,130      2,130  
creator     Henri Matisse ( Henri Matisse  
title       Tulips      Tulip  
birth_year  1869       1869  
death_year  1954       1954  
category    Mod Euro - Pain Mod Euro - Pai  
creation_date 1914      1914
```

```
cma.itemid.nunique()
```

```
986
```

```
cma.drop_duplicates(['itemid','citation']).\
```

```
itemid.count()
```

```
12941
```

```
cma.drop_duplicates(['itemid', 'creatorid']).\n    itemid.count()
```

```
1062
```

### 3. Show a collection item with duplicated citations and creators.

Only show the first 6 rows (there are actually 28 in total). Notice that the citation data is duplicated for every creator:

```
cma.set_index(['itemid'], inplace=True)\n    cma.loc[124733, ['title', 'citation',\n        'creation_date', 'creator', 'birth_year']].head(6)
```

|        | title            | citation        | \          |
|--------|------------------|-----------------|------------|
| itemid |                  |                 |            |
| 124733 | Dead Blue Roller | Weigel, J. A. G |            |
| 124733 | Dead Blue Roller | Weigel, J. A. G |            |
| 124733 | Dead Blue Roller | Winkler, Friedr |            |
| 124733 | Dead Blue Roller | Winkler, Friedr |            |
| 124733 | Dead Blue Roller | Francis, Henry  |            |
| 124733 | Dead Blue Roller | Francis, Henry  |            |
|        | creation_date    | creator         | birth_year |
| itemid |                  |                 |            |
| 124733 | 1583             | Hans Hoffmann ( | 1545       |
| 124733 | 1583             | Albrecht Dürer  | 1471       |
| 124733 | 1583             | Hans Hoffmann ( | 1545       |
| 124733 | 1583             | Albrecht Dürer  | 1471       |

```
124733      1583      Hans Hoffmann (      154€  
124733      1583      Albrecht Dürer      147€
```

4. Create a collections DataFrame. `title`, `category`, and `creation_date` should be unique to a collection item, so we create a DataFrame with just those columns, along with the `itemid` index. We get the expected number of rows, 986:

```
collectionsvars = \  
    ['title','category','creation_date']  
cmacollections = cma[collectionsvars].\  
    reset_index().\  
    drop_duplicates(['itemid']).\  
    set_index(['itemid'])  
cmacollections.shape
```

```
(986, 3)
```

```
cmacollections.head()
```

```
          title \  
itemid  
75551           Tulips  
75763 Procession or Pardon at Perros  
78982   The Resurrection of Christ  
84662       The Orange Christ  
86110 Sunset Glow over a Fishing Vil  
            category creation_date  
itemid  
75551 Mod Euro - Pain      1914  
75763 Mod Euro - Pain      1891  
78982 P - German befo      1622
```

```
84662    Mod Euro - Pain          1889
86110    ASIAN - Hanging   1460s-1550s
```

5. Let's look at the row in the new DataFrame, `cmacollections`, for the same item we displayed in *step 3*:

```
cmacollections.loc[124733]
```

```
title           Dead Blue Roller
category        DR - German
creation_date   1583
Name: 124733, dtype: object
```

6. Create a citations DataFrame.

This will just have `itemid` and `citation`:

```
cmacitations = cma[['citation']].\
    reset_index().\
    drop_duplicates(['itemid','citation']).\
    set_index(['itemid'])
cmacitations.loc[124733]
```

```
            citation
itemid
124733      Weigel, J. A. G
124733      Winkler, Friedr
124733      Francis, Henry
124733      Kurz, Otto. <em
124733      Minneapolis Ins
124733      Pilz, Kurt. "Ha
124733      Koschatzky, Wal
124733      Johnson, Mark M
124733      Kaufmann, Thoma
124733      Koreny, Fritz.
```

```
124733      Achilles-Syndra
124733      Schoch, Rainer,
124733      DeGrazia, Diane
124733      Dunbar, Burton
```

## 7. Create a creators DataFrame:

```
creatorsvars = \
    ['creator', 'birth_year', 'death_year']
cmacreators = cma[creatorsvars].\
    reset_index().\
    drop_duplicates(['itemid', 'creator']).\
    set_index(['itemid'])
cmacreators.loc[124733]
```

|        | creator         | birth_year | death_year |
|--------|-----------------|------------|------------|
| itemid |                 |            |            |
| 124733 | Hans Hoffmann ( | 1545       | 1592       |
| 124733 | Albrecht Dürer  | 1471       | 1528       |

## 8. Count the number of collection items with a creator born after 1950.

First, convert the `birth_year` values from string to numeric. Then, create a DataFrame with just young artists:

```
cma_creators['birth_year'] = \
    cma_creators.birth_year.str.\
    findall("\d+").str[0].astype(float)
youngartists = \
    cma_creators.loc[cma_creators.birth_year > 1950,\n        ['creator']].assign(creatorbornafter1950='Y')
youngartists.shape[0]==youngartists.index.nunique()
```

```
True
```

```
youngartists
```

| itemid | creator         | creatorbornafter1950 |
|--------|-----------------|----------------------|
| 168529 | Richard Barnes  | Y                    |
| 369885 | Simone Leigh (A | Y                    |
| 371392 | Belkis Ayón (Cu | Y                    |
| 378931 | Teresa Margolle | Y                    |

9. Now, we can merge the `youngartists` DataFrame with the `collections` DataFrame to create a flag for collection items that have at least one creator born after 1950:

```
cmacollections = \
    pd.merge(cmacollections, youngartists,
             left_on=['itemid'], right_on=['itemid'], how='left')
cmacollections.fillna({'creatorbornafter1950':'N'}, i
cmacollections.shape
```

```
(986, 9)
```

```
cmacollections.creatorbornafter1950.value_counts()
```

```
creatorbornafter1950
N      982
Y       4
Name: count, dtype: int64
```

Now we have three DataFrames—collection items (`cmacollections`), citations (`cmacitations`), and creators (`cmacreators`)—instead of one. `cmacollections` has a one-to-many relationship with both `cmacitations` and `cmacreators`.

## How it works...

If you mainly work directly with enterprise data, you probably rarely see a file with this kind of structure, but many of us are not so lucky. If we requested data from the museum on both the media citations and creators of their collections, it would not be completely surprising to get a data file similar to this one, with duplicated data for citations and creators. But the presence of what looks like a unique identifier of collection items gives us some hope of recovering the one-to-many relationships between a collection item and its citations, and a collection item and its creators.

*Step 2* shows that there are 986 unique `itemid` values. This suggests that there are probably only 986 collection items represented in the 17,001 rows of the DataFrame. There are 12,941 unique `itemid` and `citation` pairs, or about 13 citations per collection item on average. There are 1,062 `itemid` and `creator` pairs.

*Step 3* shows the duplication of collection item values such as `title`. The number of rows returned is equal to the Cartesian product of the merge-by values on the left and right sides of a merge. For the *Dead Blue Roller* item, there are 28 rows (we only show six of them in *step 3*), since there were 14 citations and 2 creators. The row for each creator is duplicated 14 times; once for each citation. Each citation is there twice; once for each creator. There are very few use cases for which it makes sense to leave the data in this state.

Our North Star to guide us in getting this data into better shape is the `itemid` column. We use it to create a collections DataFrame in *step 4*. We keep only one row for each value of `itemid`, and get other columns associated with a collection item, rather than a citation or creator—`title`, `category`, and `creation_date` (since `itemid` is the index, we need to first reset the index before dropping duplicates).

We follow the same procedure to create `citations` and `creators` DataFrames in *steps 6* and *7*. We use `drop_duplicates` to keep unique combinations of `itemid` and `citation`, and unique combinations of `itemid` and `creator`, respectively. This gives us the expected number of rows in the example case: 14 `citations` rows and 2 `creators` rows.

*Step 8* demonstrates how we can now work with these DataFrames to construct new columns and do analysis. We want the number of collection items that have at least one creator born after 1950. The unit of analysis is the collection items, but we need information from the creators DataFrame for the calculation. Since the relationship between `cmacollections` and `cmacreators` is one-to-many, we make sure that we are only retrieving one row per `itemid` in the creators DataFrame, even if more than one creator for an item was born after 1950:

```
youngartists.shape[0]==youngartists.index.nunique()
```

## There's more...

The duplication that occurs with many-to-many merges is most problematic when we are working with quantitative data. If the original file had the assessed value of each item in the collection, it would be duplicated in much the same way as `title` is duplicated. Any descriptive statistics we

generated on the assessed value would be off by a fair bit. For example, if the *Dead Blue Roller* item had an assessed value of \$1,000,000, we would get \$28,000,000 when summarizing the assessed value, since there are 28 duplicated values.

This shows the importance of normalized and tidy data. If there were an assessed value column, we would have included it in the `cmacollections` DataFrame we created in *step 4*. This value would be unduplicated and we would be able to generate summary statistics for collections.

I find it helpful to always return to the unit of analysis, which overlaps with the tidy data concept but is different in some ways. The approach in *step 8* would have been very different if we were just interested in the number of creators born after 1950, instead of the number of collection items with a creator born after 1950. In that case, the unit of analysis would be the creator and we would just use the creators DataFrame.

## See also

We examine many-to-many merges in the *Doing many-to-many merges* recipe in *Chapter 10, Addressing Data Issues When Combining DataFrames*.

We demonstrate a very different way to work with data structured in this way in *Chapter 12, Automate Data Cleaning with User-Defined Functions, Classes and Pipelines*, in the *Classes that handle non-tabular data structures* recipe.

# Using stack and melt to reshape data from wide to long format

One type of untidiness that Wickham identified is variable values embedded in column names. Although this rarely happens with enterprise or relational data, it is fairly common with analytical or survey data. Variable names might have suffixes that indicate a time period, such as a month or year. Or similar variables on a survey might have similar names, such as

`familymember1age` and `familymember2age`, because that is convenient and consistent with the survey designers' understanding of the variable.

One reason why this messiness happens relatively frequently with survey data is that there can be multiple units of analysis on one survey instrument. An example is the United States decennial census, which asks both household and personal questions. Survey data is also sometimes made up of repeated measures or panel data, but nonetheless often has only one row per respondent. When this is the case, new measurements or responses are stored in new columns rather than new rows, and the column names will be similar to column names for responses from earlier periods, except for a change in suffix.

The United States **National Longitudinal Survey of Youth (NLS)** is a good example of this. It is panel data, where each individual is surveyed each year. However, there is just one row of data per respondent in the analysis file provided. Responses to questions such as the number of weeks worked in a given year are placed in new columns. Tidying the NLS data means converting columns such as `weeksworked17` through `weeksworked21` (for weeks worked in 2017 through 2021) to just one column for weeks worked, another column for year, and five rows for each person (one for

each year) rather than one. This is sometimes referred to as converting data from *wide to long* format.

Amazingly, pandas has several functions that make transformations like this relatively easy: `stack`, `melt`, and `wide_to_long`. We use `stack` and `melt` in this recipe, and explore `wide_to_long` in the next.

## Getting ready

We will work with the NLS data on the number of weeks worked and college enrollment status for each year. The DataFrame has one row per survey respondent.

### Data note



The **National Longitudinal Surveys (NLS)**, administered by the United States Bureau of Labor Statistics, are longitudinal surveys of individuals who were in high school in 1997 when the surveys started. Participants were surveyed each year through 2023. The surveys are available for public use at [nlsinfo.org](https://nlsinfo.org).

## How to do it...

We will use `stack` and `melt` to transform the NLS weeks worked data from wide to long, pulling out year values from the column names as we do so:

1. Import `pandas` and the NLS data:

```
import pandas as pd
```

```
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
```

## 2. View some of the values for the number of weeks worked.

First, set the index:

```
nls97.set_index(['originalid'], inplace=True)
weeksworkecols = ['weeksworke17', 'weeksworke18',
                  'weeksworke19', 'weeksworke20', 'weeksworke21']
nls97.loc[[2,3], weeksworkecols].T
```

|              |    |    |
|--------------|----|----|
| originalid   | 2  | 3  |
| weeksworke17 | 52 | 52 |
| weeksworke18 | 52 | 52 |
| weeksworke19 | 52 | 9  |
| weeksworke20 | 52 | 0  |
| weeksworke21 | 46 | 0  |

```
nls97.shape
```

```
(8984, 110)
```

## 3. Use `stack` to transform the data from wide to long.

First, select only the `weeksworke##` columns. Use `stack` to move each column name in the original DataFrame into the index and move the `weeksworke##` values into the associated row. Reset the `index` so that the `weeksworke##` column names become the values for the `level_1` column (which we rename `year`), and the `weeksworke##` values become the values for the 0 column (which we rename `weeksworke`):



## Data note

For future upgrades to `pandas 3.0`, we would have to mention keyword argument as `(future_stack=True)` in the `stack` function.

```
weeksworked = nls97[weeksworkedcols].\  
    stack().\  
    reset_index().\  
    rename(columns={'level_1':'year',0:'weeksworked'})  
weeksworked.loc[weeksworked.originalid.isin([2,3])]
```

|    | originalid | year          | weeksworked |
|----|------------|---------------|-------------|
| 5  | 2          | weeksworked17 | 52          |
| 6  | 2          | weeksworked18 | 52          |
| 7  | 2          | weeksworked19 | 52          |
| 8  | 2          | weeksworked20 | 52          |
| 9  | 2          | weeksworked21 | 46          |
| 10 | 3          | weeksworked17 | 52          |
| 11 | 3          | weeksworked18 | 52          |
| 12 | 3          | weeksworked19 | 9           |
| 13 | 3          | weeksworked20 | 0           |
| 14 | 3          | weeksworked21 | 0           |

### 4. Fix the `year` values.

Get the last digits of the year values, convert them to integers, and add 2,000:

```
weeksworked['year'] = \  
    weeksworked.year.str[-2:].astype(int)+2000  
weeksworked.loc[weeksworked.originalid.isin([2,3])]
```

|    | originalid | year | weeksworked |
|----|------------|------|-------------|
| 5  | 2          | 2017 | 52          |
| 6  | 2          | 2018 | 52          |
| 7  | 2          | 2019 | 52          |
| 8  | 2          | 2020 | 52          |
| 9  | 2          | 2021 | 46          |
| 10 | 3          | 2017 | 52          |
| 11 | 3          | 2018 | 52          |
| 12 | 3          | 2019 | 9           |
| 13 | 3          | 2020 | 0           |
| 14 | 3          | 2021 | 0           |

```
weeksworked.shape
```

```
(44920, 3)
```

5. Alternatively, use `melt` to transform the data from wide to long.

First, reset the `index` and select the `originalid` and `weeksworked##` columns. Use the `id_vars` and `value_vars` parameters of `melt` to specify `originalid` as the `ID` variable and the `weeksworked##` columns as the columns to be rotated, or melted. Use the `var_name` and `value_name` parameters to rename the columns as `year` and `weeksworked` respectively. The column names in `value_vars` become the values for the new `year` column (which we convert to an integer using the original suffix). The values for the `value_vars` columns are assigned to the new `weeksworked` column for the associated row:

```
weeksworked = nls97.reset_index().\  
    loc[:,['originalid'] + weeksworkedcols].\  
    melt(id_vars=['originalid'],  
        value_vars=weeksworkedcols,  
        var_name='year', value_name='weeksworked')
```

```

weeksworked['year'] = \
    weeksworked.year.str[-2:].astype(int)+2000
weeksworked.set_index(['originalid'], inplace=True)
weeksworked.loc[[2,3]]

```

|            | year | weeksworked |
|------------|------|-------------|
| originalid |      |             |
| 2          | 2017 | 52          |
| 2          | 2018 | 52          |
| 2          | 2019 | 52          |
| 2          | 2020 | 52          |
| 2          | 2021 | 46          |
| 3          | 2017 | 52          |
| 3          | 2018 | 52          |
| 3          | 2019 | 9           |
| 3          | 2020 | 0           |
| 3          | 2021 | 0           |

## 6. Reshape the college enrollment columns with `melt`.

This works the same way as the `melt` function for the weeks worked columns:

```

colenrcols = \
    ['colenroct17','colenroct18','colenroct19',
     'colenroct20','colenroct21']
colenr = nls97.reset_index().\
    loc[:,['originalid'] + colenrcols].\
    melt(id_vars=['originalid'], value_vars=colenrcols,
         var_name='year', value_name='colenr')
colenr['year'] = colenr.year.str[-2:].astype(int)+2000
colenr.set_index(['originalid'], inplace=True)
colenr.loc[[2,3]]

```

|            | year | colenr |
|------------|------|--------|
| originalid |      |        |

|   |      |                 |
|---|------|-----------------|
| 2 | 2017 | 1. Not enrolled |
| 2 | 2018 | 1. Not enrolled |
| 2 | 2019 | 1. Not enrolled |
| 2 | 2020 | 1. Not enrolled |
| 2 | 2021 | 1. Not enrolled |
| 3 | 2017 | 1. Not enrolled |
| 3 | 2018 | 1. Not enrolled |
| 3 | 2019 | 1. Not enrolled |
| 3 | 2020 | 1. Not enrolled |
| 3 | 2021 | 1. Not enrolled |

7. Merge the weeks worked and college enrollment data:

```
workschool = \
    pd.merge(weeksworked, colenr, on=['originalid', 'year'])
workschool.shape
```

(44920, 3)

```
workschool.loc[[2,3]]
```

| originalid | year | weeksworked | olenr           |
|------------|------|-------------|-----------------|
| 2          | 2017 | 52          | 1. Not enrolled |
| 2          | 2018 | 52          | 1. Not enrolled |
| 2          | 2019 | 52          | 1. Not enrolled |
| 2          | 2020 | 52          | 1. Not enrolled |
| 2          | 2021 | 46          | 1. Not enrolled |
| 3          | 2017 | 52          | 1. Not enrolled |
| 3          | 2018 | 52          | 1. Not enrolled |
| 3          | 2019 | 9           | 1. Not enrolled |
| 3          | 2020 | 0           | 1. Not enrolled |
| 3          | 2021 | 0           | 1. Not enrolled |

This gives us one DataFrame from the melting of both the weeks worked and the college enrollment columns.

## How it works...

We can use `stack` or `melt` to reshape data from wide to long format, but `melt` provides more flexibility. `stack` will move all of the column names into the index. We see in *step 4* that we get the expected number of rows after stacking, `44920`, which is  $5 * 8984$ , the number of rows in the initial data.

With `melt`, we can rotate the column names and values based on an `ID` variable other than the index. We do this with the `id_vars` parameter. We specify which variables to melt by using the `value_vars` parameter.

In *step 6*, we also reshape the college enrollment columns. To create one DataFrame with the reshaped weeks worked and college enrollment data, we merge the two DataFrames we created in *steps 5* and *6*. We will see in the next recipe how to accomplish what we did in *steps 5* through *7* in one step.

## Melting multiple groups of columns

When we needed to melt multiple groups of columns in the previous recipe, we used `melt` twice and then merged the resulting DataFrames. That worked fine, but we can accomplish the same tasks in one step with the `wide_to_long` function. `wide_to_long` has more functionality than `melt`, but is a bit more complicated to use.

# Getting ready

We will work with the weeks worked and college enrollment data from the NLS in this recipe.

## How to do it...

We will transform multiple groups of columns at once using `wide_to_long`:

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

```
import pandas as pd
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index('personid', inplace=True)
```

2. View some of the weeks worked and college enrollment data:

```
weeksworkecols = ['weeksworke17', 'weeksworke18',
                  'weeksworke19', 'weeksworke20', 'weeksworke21']
colenrcols = ['colenroct17', 'colenroct18',
              'colenroct19', 'colenroct20', 'colenroct21']
nls97.loc[nls97.originalid.isin([2,3]),
          ['originalid'] + weeksworkecols + colenrcols].T
```

|              |                 |                 |
|--------------|-----------------|-----------------|
| personid     | 999406          | 151672          |
| originalid   | 2               | 3               |
| weeksworke17 | 52              | 52              |
| weeksworke18 | 52              | 52              |
| weeksworke19 | 52              | 9               |
| weeksworke20 | 52              | 0               |
| weeksworke21 | 46              | 0               |
| colenroct17  | 1. Not enrolled | 1. Not enrolled |
| colenroct18  | 1. Not enrolled | 1. Not enrolled |
| colenroct19  | 1. Not enrolled | 1. Not enrolled |

|             |                 |                 |
|-------------|-----------------|-----------------|
| colenroct20 | 1. Not enrolled | 1. Not enrolled |
| colenroct21 | 1. Not enrolled | 1. Not enrolled |

### 3. Run the `wide_to_long` function.

Pass a list to `stubnames` to indicate the column groups that are wanted. (All columns starting with the same characters as each item in the list will be selected for melting.) Use the `i` parameter to indicate the ID variable (`originalid`), and use the `j` parameter to name the column (`year`) that is based on the column suffixes—[17](#), [18](#), and so on:

```
workschool = pd.wide_to_long(nls97[['originalid']]
...     + weeksworkecols + colenrcols],
...     stubnames=['weeksworke', 'colenroct'],
...     i=['originalid'], j='year').reset_index()
workschool['year'] = workschool.year+2000
workschool = workschool.\
    ... sort_values(['originalid', 'year'])
workschool.set_index(['originalid'], inplace=True)
workschool.loc[[2,3]]
```

| originalid | year | weeksworke | colenr       |
|------------|------|------------|--------------|
| 2          | 2017 | 52         | 1. Not enro] |
| 2          | 2018 | 52         | 1. Not enro] |
| 2          | 2019 | 52         | 1. Not enro] |
| 2          | 2020 | 52         | 1. Not enro] |
| 2          | 2021 | 46         | 1. Not enro] |
| 3          | 2017 | 52         | 1. Not enro] |
| 3          | 2018 | 52         | 1. Not enro] |
| 3          | 2019 | 9          | 1. Not enro] |
| 3          | 2020 | 0          | 1. Not enro] |
| 3          | 2021 | 0          | 1. Not enro] |

`wide_to_long` accomplishes in one step what it took us several steps to accomplish in the previous recipe using `melt`.

## How it works...

The `wide_to_long` function does almost all of the work for us, though it takes more effort to set it up than for `stack` or `melt`. We need to provide the function with the characters (`weeksworke`d and `colenroct` in this case) of the column groups. Since our variables are named with suffixes indicating the year, `wide_to_long` translates the suffixes into values that make sense and melts them into the column that is named with the `j` parameter. It's almost magic!

## There's more...

The suffixes of the `stubnames` columns in this recipe are the same: 17 through 21. But that does not have to be the case. When suffixes are present for one column group, but not for another, the values for the latter column group for that suffix will be missing. We can see that by excluding `weeksworke`d17 from the DataFrame and adding `weeksworke`d16:

```
weeksworkecols = ['weeksworke16', 'weeksworke18',
                  'weeksworke19', 'weeksworke20', 'weeksworke21']
workschool = pd.wide_to_long(nls97[['originalid']]
    ... + weeksworkecols + colenrcols],
    ... stubnames=['weeksworke', 'colenroct'],
    ... i=['originalid'], j='year').reset_index()
workschool['year'] = workschool.year+2000
workschool = workschool.sort_values(['originalid', 'year'])
workschool.set_index(['originalid'], inplace=True)
workschool.loc[[2,3]]
```

| originalid | year | weeksworked | coenroct     |
|------------|------|-------------|--------------|
| 2          | 2016 | 53          |              |
| 2          | 2017 | NaN         | 1. Not enro] |
| 2          | 2018 | 52          | 1. Not enro] |
| 2          | 2019 | 52          | 1. Not enro] |
| 2          | 2020 | 52          | 1. Not enro] |
| 2          | 2021 | 46          | 1. Not enro] |
| 3          | 2016 | 53          |              |
| 3          | 2017 | NaN         | 1. Not enro] |
| 3          | 2018 | 52          | 1. Not enro] |
| 3          | 2019 | 9           | 1. Not enro] |
| 3          | 2020 | 0           | 1. Not enro] |
| 3          | 2021 | 0           | 1. Not enro] |

The `weeksworked` values for 2017 are now missing, as are the `coenroct` values for 2016.

## Using unstack and pivot to reshape data from long to wide format

Sometimes, we actually have to move data from a tidy to an untidy structure. This is often because we need to prepare the data for analysis with software packages that do not handle relational data well, or because we are submitting data to some external authority that has requested it in an untidy format. `unstack` and `pivot` can be helpful when we need to reshape data from long to wide format. `unstack` does the opposite of what we did with `stack`, and `pivot` does the opposite of `melt`.

## Getting ready

We continue to work with the NLS data on weeks worked and college enrollment in this recipe.

## How to do it...

We use `unstack` and `pivot` to return the melted NLS DataFrame to its original state:

1. Import `pandas` and load the stacked and melted NLS data:

```
import pandas as pd
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index(['originalid'], inplace=True)
```

2. Stack the data again.

This repeats the stack from an earlier recipe in this chapter:

```
weeksworkecols = ['weeksworke17', 'weeksworke18',
                  'weeksworke19', 'weeksworke20', 'weeksworke21']
weeksworkestacked = nls97[weeksworkecols].\
    stack()
weeksworkestacked.loc[[2, 3]]
```

| originalid |              |    |
|------------|--------------|----|
| 2          | weeksworke17 | 52 |
|            | weeksworke18 | 52 |
|            | weeksworke19 | 52 |
|            | weeksworke20 | 52 |
|            | weeksworke21 | 46 |
| 3          | weeksworke17 | 52 |
|            | weeksworke18 | 52 |
|            | weeksworke19 | 9  |
|            | weeksworke20 | 0  |

```
weeksworked21          0  
dtype: float64
```

### 3. Melt the data again.

This repeats the `melt` from an earlier recipe in this chapter:

```
weeksworkedmelted = nls97.reset_index().\  
...     loc[:,['originalid']] + weeksworkedcols].\  
...     melt(id_vars=['originalid'],  
...           value_vars=weeksworkedcols,  
...           var_name='year', value_name='weeksworked')  
weeksworkedmelted.loc[weeksworkedmelted.\  
    originalid.isin([2,3])].\  
    sort_values(['originalid','year'])
```

|       | originalid | year          | weekswor |
|-------|------------|---------------|----------|
| 1     | 2          | weeksworked17 |          |
| 8985  | 2          | weeksworked18 |          |
| 17969 | 2          | weeksworked19 |          |
| 26953 | 2          | weeksworked20 |          |
| 35937 | 2          | weeksworked21 |          |
| 2     | 3          | weeksworked17 |          |
| 8986  | 3          | weeksworked18 |          |
| 17970 | 3          | weeksworked19 |          |
| 26954 | 3          | weeksworked20 |          |
| 35938 | 3          | weeksworked21 |          |

### 4. Use `unstack` to convert the stacked data from long to wide:

```
weeksworked = weeksworkedstacked.unstack()  
weeksworked.loc[[2,3]].T
```

|               |    |    |
|---------------|----|----|
| originalid    | 2  | 3  |
| weeksworked17 | 52 | 52 |
| weeksworked18 | 52 | 52 |
| weeksworked19 | 52 | 9  |
| weeksworked20 | 52 | 0  |
| weeksworked21 | 46 | 0  |

5. Use `pivot` to convert the melted data from long to wide.

`pivot` is slightly more complicated than `unstack`. We need to pass arguments to do the reverse of `melt`, telling `pivot` the column to use for the column name suffixes (`year`) and where to grab the values to be unmelted (from the `weeksworked` columns, in this case):

```
weeksworked = weeksworkedmelted.\n...     pivot(index='originalid',\n...     columns='year', values=['weeksworked']).\\n\nreset_index()\nweeksworked.columns = ['originalid'] + \\n    [col[1] for col in weeksworked.columns[1:]]\nweeksworked.loc[weeksworked.originalid.isin([2,3])].T
```

|               | 1  | 2  |
|---------------|----|----|
| originalid    | 2  | 3  |
| weeksworked17 | 52 | 52 |
| weeksworked18 | 52 | 52 |
| weeksworked19 | 52 | 9  |
| weeksworked20 | 52 | 0  |
| weeksworked21 | 46 | 0  |

This returns the NLS data back to its original untidy form.

## How it works...

We first do a `stack` and a `melt` in *steps 2* and *3* respectively. This rotates the DataFrames from wide to long format. We then `unstack` (*step 4*) and `pivot` (*step 5*) those DataFrames to rotate them back from long to wide.

`unstack` uses the multi-index that is created by the `stack` to figure out how to rotate the data.

The `pivot` function needs us to indicate the index column (`originalid`), the column whose values will be appended to the column names (`year`), and the name of the columns with the values to be unmelted (`weeksworked`).

`Pivot` will return multilevel column names. We fix that by pulling from the second level with `[col[1] for col in weeksworked.columns[1:]]`.

## Summary

We explored key tidy data topics in this chapter. These topics included handling duplicated data, either by dropping rows where the data are redundant, or aggregating by group. We also restructured data stored in a many-to-many format into a tidy format. Finally, we stepped through several ways of converting data from wide to long format, and back to wide when necessary. Up next is the final chapter of the book, where we will learn to automate data cleaning with user-defined functions, classes and pipelines.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.gg/p8uSgEAETX>



# 12

## Automate Data Cleaning with User-Defined Functions, Classes, and Pipelines

There are a number of great reasons to write code that is reusable. When we step back from the particular data-cleaning problem at hand and consider its relationship to very similar problems, we can actually improve our understanding of the key issues involved. We are also more likely to address a task systematically when we set our sights more on solving it for the long term than on the before-lunch solution. This has the additional benefit of helping us to disentangle the substantive issues from the mechanics of data manipulation.

We will create several modules to accomplish routine data-cleaning tasks in this chapter. The functions and classes in these modules are examples of code that can be reused across `DataFrames`, or for one `DataFrame` over an extended period of time. These functions handle many of the tasks we discussed in the first eleven chapters, but in a manner that allows us to reuse our code.

Specifically, the recipes in this chapter cover the following:

- Functions for getting a first look at our data
- Functions for displaying summary statistics and frequencies

- Functions for identifying outliers and unexpected values
- Functions for aggregating or combining data
- Classes that contain the logic for updating Series values
- Classes that handle non-tabular data structures
- Functions for checking overall data quality
- Pre-processing data with pipelines: a simple example
- Pre-processing data with pipelines: a more complicated example

## Technical requirements

You will need pandas, NumPy, and Matplotlib to complete the recipes in this chapter. I used pandas 2.1.4, but the code will run on pandas 1.5.3 or later.

The code in this chapter can be downloaded from the book's GitHub repository, <https://github.com/PacktPublishing/Python-Data-Cleaning-Cookbook-Second-Edition>.

## Functions for getting a first look at our data

The first few steps we take after we import our data into a pandas DataFrame are pretty much the same regardless of the characteristics of the data. We almost always want to know the number of columns and rows and the column data types, and to see the first few rows. We also might want to view the index and check whether there is a unique identifier for DataFrame rows. These discrete, easily repeatable tasks are good candidates for a collection of functions we can organize into a module.

In this recipe, we will create a module with functions that give us a good first look at any pandas DataFrame. A module is simply a collection of Python code that we can import into another Python program. Modules are easy to reuse because they can be referenced by any program with access to the folder where the module is saved.

## Getting ready

We create two files in this recipe: one with a function we will use to look at our data and another to call that function. Let's call the file with the function we will use `basicdescriptives.py` and place it in a subfolder called `helperfunctions`.

We work with the **National Longitudinal Surveys (NLS)** data in this recipe.

### Data note



The NLS, administered by the United States Bureau of Labor Statistics, is a collection of longitudinal surveys of individuals who were in high school in 1997 when the surveys started. Participants were surveyed each year through 2023. The surveys are available for public use at [nlsinfo.org](http://nlsinfo.org).

## How to do it...

We will create a function to take an initial look at a DataFrame.

1. Create the `basicdescriptives.py` file with the function we want.

The `getfirstlook` function will return a dictionary with summary information on a DataFrame. Save the file in the `helperfunctions` subfolder as `basicdescriptives.py`. (You can also just download the code from the GitHub repository.) Also, create a function (`displaydict`) to pretty up the display of a dictionary:

```
import pandas as pd
def getfirstlook(df, nrows=5, uniqueids=None):
    ...     out = {}
    ...     out['head'] = df.head(nrows)
    ...     out['dtypes'] = df.dtypes
    ...     out['nrows'] = df.shape[0]
    ...     out['ncols'] = df.shape[1]
    ...     out['index'] = df.index
    ...     if (uniqueids is not None):
    ...         out['uniqueids'] = df[uniqueids].nunique()
    ...     return out
def displaydict(dicttodisplay):
    ...     print(*(': '.join(map(str, x)) \
    ...             for x in dicttodisplay.items()), sep='\n\n')
```

2. Create a separate file, `firstlook.py`, to call the `getfirstlook` function.

Import the `pandas`, `os`, and `sys` libraries, and load the NLS data:

```
import pandas as pd
import os
import sys
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index('personid', inplace=True)
```

3. Import the `basicdescriptives` module.

First, append the `helperfunctions` subfolder to the Python path. We can then import `basicdescriptives`. We use the same name as the name of the file to import the module. We create an alias, `bd`, to make it easier to access the functions in the module later. (We can use `importlib`, commented out here, if we need to reload `basicdescriptives` because we have made some changes in the code in that module.)

```
sys.path.append(os.getcwd() + "/helperfunctions")
import basicdescriptives as bd
# import importlib
# importlib.reload(bd)
```

#### 4. Take a first look at the NLS data.

We can just pass the DataFrame to the `getfirstlook` function in the `basicdescriptives` module to get a quick summary of the NLS data. The `displaydict` function gives us prettier printing of the dictionary:

```
dfinfo = bd.getfirstlook(nls97)
bd.displaydict(dfinfo)
```

| head:    | gender          | birthmonth      | birthyear | ... | pe  |
|----------|-----------------|-----------------|-----------|-----|-----|
| personid |                 |                 |           |     | ... |
| 135335   | Female          | 9               | 1981      | ... |     |
| 999406   | Male            | 7               | 1982      | ... |     |
| 151672   | Female          | 9               | 1983      | ... |     |
| 750699   | Female          | 2               | 1981      | ... |     |
| 781297   | Male            | 10              | 1982      | ... |     |
|          | fatherhighgrade | motherhighgrade |           |     |     |
| personid |                 |                 |           |     |     |
| 135335   |                 | 16              |           | 8   |     |
| 999406   |                 | 17              |           | 15  |     |
| 151672   |                 | -3              |           | 12  |     |

```
750699           12           12
781297           12           12
[5 rows x 110 columns]
dtypes: gender      object
birthmonth     int64
birthyear      int64
sampletype     object
ethnicity      object
originalid    int64
motherage      int64
parentincome   int64
fatherhighgrade int64
motherhighgrade int64
Length: 110, dtype: object
nrows: 8984
ncols: 110
index: Index([135335, 999406, 151672, 750699, 781297, 6138,
              474817, 530234, 351406,
              ...
              290800, 209909, 756325, 543646, 411195, 505861, 3682,
              215605, 643085, 713757],
              dtype='int64', name='personid', length=8984)
```

## 5. Pass values to the `nrows` and `uniqueids` parameters of `getfirstlook`.

The two parameters default to values of 5 and `None` respectively, unless we provide values:

```
dfinfo = bd.getfirstlook(nls97, 2, 'originalid')
bd.displaydict(dfinfo)
```

```
head:      gender   birthmonth   birthyear   ...
personid
135335    Female        9          1981       ...
999406    Male         7          1982       ...
                           fatherhighgrade  motherhighgrade
                           ...
personid
```

```
135335           16          8
999406           17         15
[2 rows x 110 columns]
dtypes: gender      object
birthmonth      int64
birthyear       int64
sampletype      object
ethnicity       object
originalid     int64
motherage       int64
parentincome    int64
fatherhighgrade int64
motherhighgrade int64
Length: 110, dtype: object
nrows: 8984
ncols: 110
index: Index([135335, 999406, 151672, 750699, 781297, 6138,
              474817, 530234, 351406,
              ...
              290800, 209909, 756325, 543646, 411195, 505861, 368,
              215605, 643085, 713757],
              dtype='int64', name='personid', length=8984)
uniqueids: 8984
```

## 6. Work with some of the returned dictionary keys and values.

We can also display selected key values from the dictionary returned from `getfirstlook`. Show the number of rows and data types, and check to see whether each row has a `uniqueid` instance (`dfinfo['nrows'] == dfinfo['uniqueids']`):

```
dfinfo['nrows']
```

```
8984
```

```
dfinfo['dtypes']
```

```
gender          object
birthmonth      int64
birthyear       int64
sampletype      object
ethnicity       object
originalid     int64
motherage       int64
parentincome    int64
fatherhighgrade int64
motherhighgrade int64
Length: 110, dtype: object
```

```
dfinfo['nrows'] == dfinfo['uniqueids']
```

```
True
```

Let's take a closer look at how the function works and how we call it.

## How it works...

Almost all of the action in this recipe is in the `getfirstlook` function, which we look at in *step 1*. We place the `getfirstlook` function in a separate file that we name `basicdescriptives.py`, which we can import as a module with that name (minus the extension).

We could have typed the function into the file we were using and called it from there. By putting it in a module instead, we can call it from any file that has access to the folder where the module is saved. When we import the `basicdescriptives` module in *step 3*, we load all of the code in `basicdescriptives`, allowing us to call all functions in that module.

The `getfirstlook` function returns a dictionary with useful information about the DataFrame that is passed to it. We see the first five rows, the number of columns and rows, the data types, and the index. By passing a value to the `uniqueid` parameter, we also get the number of unique values for the column.

By adding keyword parameters (`nrows` and `uniqueid`) with default values, we improve the flexibility of `getfirstlook`, without increasing the amount of effort it takes to call the function when we do not need the extra functionality.

In the first call, in *step 4*, we do not pass values for `nrows` or `uniqueid`, sticking with the default values. In *step 5*, we indicate that we only want two rows displayed and that we want to examine unique values for `originalid`.

## There's more...

The point of this recipe, and the ones that follow it, is not to provide code that you can download and run on your own data, though you are certainly welcome to do that. I am mainly trying to demonstrate how you can collect your favorite approaches to data cleaning in handy modules, and how this allows for easy code reuse. The specific code here is just a serving suggestion, if you will.

Whenever we use a combination of positional and keyword parameters, the positional parameters must go first.

# Functions for displaying summary statistics and frequencies

During the first few days of working with a DataFrame, we try to get a good sense of the distribution of continuous variables and counts for categorical variables. We also often do counts by selected groups. Although pandas and NumPy have many built-in methods for these purposes—`describe`, `mean`, `valuecounts`, `crosstab`, and so on—data analysts often have preferences for how they work with these tools. If, for example, an analyst finds that they usually need to see more percentiles than those generated by `describe`, they can use their own function instead. We will create user-defined functions for displaying summary statistics and frequencies in this recipe.

## Getting ready

We will be working with the `basicdescriptives` module again in this recipe. All of the functions we will define are saved in that module. We will continue to work with the NLS data.

## How to do it...

We will use functions we create to generate summary statistics and counts:

1. Create the `gettots` function in the `basicdescriptives` module.

The function takes a pandas DataFrame and creates a dictionary with selected summary statistics. It returns a pandas DataFrame:

```
def gettots(df):
...     out = {}
```

```
...     out['min'] = df.min()
...     out['per15'] = df.quantile(0.15)
...     out['qr1'] = df.quantile(0.25)
...     out['med'] = df.median()
...     out['qr3'] = df.quantile(0.75)
...     out['per85'] = df.quantile(0.85)
...     out['max'] = df.max()
...     out['count'] = df.count()
...     out['mean'] = df.mean()
...     out['iqr'] = out['qr3']-out['qr1']
... return pd.DataFrame(out)
```

## 2. Import the `pandas`, `os`, and `sys` libraries.

Do this from a different file, which you can call `taking_measure.py`:

```
import pandas as pd
import os
import sys
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index('personid', inplace=True)
```

## 3. Import the `basicdescriptives` module:

```
sys.path.append(os.getcwd() + "/helperfunctions")
import basicdescriptives as bd
```

## 4. Show summary statistics for continuous variables.

Use the `gettots` function from the `basicdescriptives` module that we created in *step 1*:

```
bd.gettots(nls97[['satverbal', 'satmath']]).T
```

|       | satverbal | satmath |
|-------|-----------|---------|
| min   | 14        | 7       |
| per15 | 390       | 390     |
| qr1   | 430       | 430     |
| med   | 500       | 500     |
| qr3   | 570       | 580     |
| per85 | 620       | 621     |
| max   | 800       | 800     |
| count | 1,406     | 1,407   |
| mean  | 500       | 501     |
| iqr   | 140       | 150     |

```
bd.gettots(nls97.filter(like="weeksworked"))
```

|               | min | per15 | qr1 | ... | count | mean | iqr |
|---------------|-----|-------|-----|-----|-------|------|-----|
| weeksworked00 | 0   | 0     | 5   | ... | 8626  | 26   | 45  |
| weeksworked01 | 0   | 0     | 10  | ... | 8591  | 30   | 41  |
| weeksworked02 | 0   | 0     | 13  | ... | 8591  | 32   | 39  |
| weeksworked03 | 0   | 0     | 14  | ... | 8535  | 34   | 38  |
| weeksworked04 | 0   | 1     | 17  | ... | 8513  | 35   | 35  |
| weeksworked05 | 0   | 5     | 22  | ... | 8468  | 37   | 31  |
| weeksworked06 | 0   | 9     | 27  | ... | 8419  | 38   | 25  |
| weeksworked07 | 0   | 10    | 30  | ... | 8360  | 39   | 22  |
| weeksworked08 | 0   | 9     | 30  | ... | 8292  | 39   | 22  |
| weeksworked09 | 0   | 0     | 22  | ... | 8267  | 38   | 30  |
| weeksworked10 | 0   | 0     | 21  | ... | 8198  | 37   | 31  |
| weeksworked11 | 0   | 0     | 22  | ... | 8123  | 38   | 31  |
| weeksworked12 | 0   | 0     | 23  | ... | 7988  | 38   | 29  |
| weeksworked13 | 0   | 0     | 28  | ... | 7942  | 39   | 24  |
| weeksworked14 | 0   | 0     | 26  | ... | 7896  | 39   | 26  |
| weeksworked15 | 0   | 0     | 33  | ... | 7767  | 40   | 19  |
| weeksworked16 | 0   | 0     | 31  | ... | 7654  | 40   | 22  |
| weeksworked17 | 0   | 0     | 38  | ... | 7496  | 40   | 14  |
| weeksworked18 | 0   | 0     | 35  | ... | 7435  | 40   | 17  |
| weeksworked19 | 0   | 4     | 42  | ... | 7237  | 41   | 10  |
| weeksworked20 | 0   | 0     | 21  | ... | 6971  | 38   | 31  |
| weeksworked21 | 0   | 0     | 35  | ... | 6627  | 36   | 15  |

```
weeksworked22      0      0    2   ...   2202    11    17
[23 rows x 10 columns]
```

## 5. Create a function to count missing values by columns and rows.

The `getmissings` function will take a DataFrame and a parameter for showing percentages or counts. It returns two Series, one with the missing values for each column and the other with missing values by row. Save the function in the `basicdescriptives` module:

```
def getmissings(df, byrowperc=False):
    return df.isnull().sum(), \
        df.isnull().sum(axis=1).\
        value_counts(normalize=byrowperc).\
        sort_index()
```

## 6. Call the `getmissings` function.

Call it first with `byrowperc` (the second parameter) set to `True`. This will show the percentage of rows with the associated number of missing values. For example, the `missingbyrows` value shows that 73% of rows have 0 missing values for `weeksworked20` and `weeksworked21`. Call it again, leaving `byrowperc` at its default value of `False`, to get counts instead:

```
missingsbycols, missingsbyrows = \
    bd.getmissings(nls97[['weeksworked20',
    'weeksworked21']], True)
missingsbycols
```

```
weeksworked20      2013
weeksworked21      2357
```

```
dtype: int64
```

```
missingsbyrows
```

```
0    0.73
1    0.05
2    0.22
Name: proportion, dtype: float64
```

```
missingsbycols, missingsbyrows = \
bd.getmissings(nls97[['weeksworked20',
'weeksworked21']])
missingsbyrows
```

```
0    6594
1    410
2   1980
Name: count, dtype: int64
```

## 7. Create a function to calculate frequencies for all categorical variables.

The `makefreqs` function loops through all columns with the category data type in the passed DataFrame, running `value_counts` on each one. The frequencies are saved to the file indicated by `outfile`:

```
def makefreqs(df, outfile):
...     freqout = open(outfile, 'w')
...     for col in df.\
...         select_dtypes(include=["category"]):
...         print(col, "-----",
...               "frequencies",
...               df[col].value_counts().sort_index(),
...               "percentages",
```

```
...         df[col].value_counts(normalize=True).\  
...         sort_index(),  
...         sep="\n\n", end="\n\n\n",  
...         file=freqout)  
... freqout.close()
```

## 8. Call the `makefreqs` function.

First change the data type of each object column to `category`. This call runs `value_counts` on category data columns in the NLS DataFrame and saves the frequencies to `nlsfreqs.txt` in the `views` subfolder of the current folder.

```
nls97.loc[:, nls97.dtypes == 'object'] = \  
...     nls97.select_dtypes(['object']). \  
...     apply(lambda x: x.astype('category'))  
bd.makefreqs(nls97, "views/nlsfreqs.txt")
```

## 9. Create a function to get counts by groups.

The `getcnts` function counts the number of rows for each combination of column values in `cats`, a list of column names. It also counts the number of rows for each combination of column values excluding the final column in `cats`. This provides a total across all values of the final column. (The next step shows what this looks like.)

```
def getcnts(df, cats, rowsel=None):  
...     tots = cats[:-1]  
...     catcnt = df.groupby(cats, dropna=False).size().\  
...             reset_index(name='catcnt')  
...     totcnt = df.groupby(tots, dropna=False).size().\  
...             reset_index(name='totcnt')
```

```

...     percs = pd.merge(catcnt, totcnt, left_on=tots,
...                         right_on=tots, how="left")
...     percs['percent'] = percs.catcnt / percs.totcnt
...     if (rowsel is not None):
...         percs = percs.loc[eval("percs." + rowsel)]
...     return percs

```

10. Pass the `maritalstatus` and `colenroct00` columns to the `getcnts` function.

This returns a DataFrame with counts for each column value combination, as well as counts for all combinations excluding the last column. This is used to calculate percentages within groups. For example, 669 respondents were divorced and 560 of those (or 84%) were not enrolled in college in October 2000:

```

bd.getcnts(nls97,
    ['maritalstatus', 'colenroct00'])

```

|    | maritalstatus | colenroct00       | catcnt | totcnt |
|----|---------------|-------------------|--------|--------|
| 0  | Divorced      | 1. Not enrolled   | 560    | 669    |
| 1  | Divorced      | 2. 2-year college | 50     | 669    |
| 2  | Divorced      | 3. 4-year college | 59     | 669    |
| 3  | Married       | 1. Not enrolled   | 2264   | 3068   |
| 4  | Married       | 2. 2-year college | 236    | 3068   |
| 5  | Married       | 3. 4-year college | 568    | 3068   |
| 6  | Never-married | 1. Not enrolled   | 2363   | 2767   |
| 7  | Never-married | 2. 2-year college | 131    | 2767   |
| 8  | Never-married | 3. 4-year college | 273    | 2767   |
| 9  | Separated     | 1. Not enrolled   | 127    | 148    |
| 10 | Separated     | 2. 2-year college | 13     | 148    |
| 11 | Separated     | 3. 4-year college | 8      | 148    |
| 12 | Widowed       | 1. Not enrolled   | 19     | 23     |
| 13 | Widowed       | 2. 2-year college | 1      | 23     |
| 14 | Widowed       | 3. 4-year college | 3      | 23     |

|    |     |                   |      |      |
|----|-----|-------------------|------|------|
| 15 | NaN | 1. Not enrolled   | 1745 | 2309 |
| 16 | NaN | 2. 2-year college | 153  | 2309 |
| 17 | NaN | 3. 4-year college | 261  | 2309 |
| 18 | NaN |                   | 150  | 2309 |

11. Use the `rowsel` parameter of `getcnts` to limit the output to specific rows. This will show only the not-enrolled-in-college rows:

```
bd.getcnts(nls97,
    ['maritalstatus', 'colenroct20'],
    "colenroct20.str[0:1]=='1'")
```

|    | maritalstatus | colenroct00     | catcnt | totcnt |
|----|---------------|-----------------|--------|--------|
| 0  | Divorced      | 1. Not enrolled | 560    | 669    |
| 3  | Married       | 1. Not enrolled | 2264   | 3068   |
| 6  | Never-married | 1. Not enrolled | 2363   | 2767   |
| 9  | Separated     | 1. Not enrolled | 127    | 148    |
| 12 | Widowed       | 1. Not enrolled | 19     | 23     |
| 15 | NaN           | 1. Not enrolled | 1745   | 2309   |

These steps demonstrate how to create functions and use them to generate summary statistics and frequencies.

## How it works...

In *step 1*, we created a function, `gettots`, that calculated descriptive statistics for all columns in a DataFrame, returning those results in a summary DataFrame. Most of the statistics can be generated with the `describe` method, but we add a few statistics—the 15<sup>th</sup> percentile, the 85<sup>th</sup> percentile, and the interquartile range. We call that function twice in *step 4*,

the first time for the SAT verbal and math scores and the second time for all weeks worked columns.

*Steps 5 and 6* create and call a function that shows the number of missing values for each column in the passed DataFrame. The function also counts missing values for each row, displaying the frequency of missing values. The frequency of missing values by row can also be displayed as a percentage of all rows by passing a value of `True` to the `byrowperc` parameter.

*Steps 7 and 8* produce a text file with frequencies for all categorical variables in the passed DataFrame. We just loop through all columns with the category data type and run `value_counts`. Since often the output is long, we save it to a file. It is also good to have frequencies saved somewhere for later reference.

The `getcnts` function we create in *step 9* and call in *steps 10 and 11* is a tad idiosyncratic. pandas has a very useful `crosstab` function, which I use frequently. But I often need a no-fuss way to look at group counts and percentages for subgroups within groups. The `getcnts` function does that.

## There's more...

A function can be very helpful even when it does not do very much. There is not much code in the `getmissings` function, but I check for missing values so frequently that the small time savings are significant cumulatively. It also reminds me to check for missing values by column and by row.

## See also

We explore pandas' tools for generating summary statistics and frequencies in *Chapter 3, Taking the Measure of Your Data*.

## Functions for identifying outliers and unexpected values

If I had to pick one data-cleaning area where I find reusable code most beneficial, it would be in the identification of outliers and unexpected values. This is because our prior assumptions often lead us to the central tendency of a distribution, rather than to the extremes. Quickly—think of a cat. Unless you were thinking about a particular cat in your life, an image of a generic feline between 8 and 10 pounds probably came to mind; not one that is 6 pounds or 22 pounds.

We often need to be more deliberate to elevate extreme values to consciousness. This is where having a standard set of diagnostic functions to run on our data is very helpful. We can run these functions even if nothing in particular triggers us to run them. This recipe provides examples of functions that we can use regularly to identify outliers and unexpected values.

## Getting ready

We will create two files in this recipe, one with the functions we will use to check for outliers and another with the code we will use to call those functions. Let's call the file with the functions we will use `outliers.py`, and place it in a subfolder called `helperfunctions`.

You will need the `matplotlib` and `scipy` libraries, in addition to `pandas`, to run the code in this recipe. You can install `matplotlib` and `scipy` by entering `pip install matplotlib` and `pip install scipy` in a Terminal client or in Windows PowerShell. You will also need the `pprint` utility, which you can install with `pip install pprint`.

We will work with the NLS and COVID-19 data in this recipe. The COVID-19 data has one row per country, with cumulative cases and deaths for that country.

### Data note



Our World in Data provides COVID-19 public use data at <https://ourworldindata.org/covid-cases>.

The dataset includes total cases and deaths, tests administered, hospital beds, and demographic data such as median age, gross domestic product, and diabetes prevalence. The dataset used in this recipe was downloaded on March 3, 2024.

## How to do it...

We create and call functions to check the distribution of variables, list extreme values, and visualize a distribution:

1. Import the `pandas`, `os`, `sys`, and `pprint` libraries.

Also, load the NLS and COVID-19 data:

```
import pandas as pd  
import os
```

```
import sys
import pprint
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97.set_index('personid', inplace=True)
covidtotals = pd.read_csv("data/covidtotals.csv")
```

## 2. Create a function to show some important properties of a distribution.

The `getdistprops` function takes a Series and generates measures of central tendency, shape, and spread. The function returns a dictionary with these measures. It also handles situations where the Shapiro test for normality does not return a value. It will not add keys for `normstat` and `normpvalue` when that happens. Save the function in a file named `outliers.py` in the `helperfunctions` subfolder of the current directory. (Also load the `pandas`, `matplotlib`, `scipy`, and `math` libraries we will need for this and other functions in this module.)

```
import pandas as pd
import matplotlib.pyplot as plt
import scipy.stats as scistat
import math
def getdistprops(seriestotest):
    ...     out = {}
    ...     normstat, normpvalue = scistat.shapiro(seriestotest)
    ...     if (not math.isnan(normstat)):
    ...         out['normstat'] = normstat
    ...         if (normpvalue>=0.05):
    ...             out['normpvalue'] = str(round(normpvalue, 2)) +
    ...             elif (normpvalue<0.05):
    ...                 out['normpvalue'] = str(round(normpvalue, 2)) +
    ...     out['mean'] = seriestotest.mean()
    ...     out['median'] = seriestotest.median()
    ...     out['std'] = seriestotest.std()
    ...     out['kurtosis'] = seriestotest.kurtosis()
```

```
...     out['skew'] = seriestotest.skew()
...     out['count'] = seriestotest.count()
...     return out
```

3. Pass the total cases per million in population series to the `getdistprops` function.

The `skew` and `kurtosis` values suggest that the distribution of `total_cases_pm` has a positive skew and shorter tails than a normally distributed variable. The Shapiro test of normality (`normpvalue`) confirms this. (Use `pprint` to improve the display of the dictionary returned by `getdistprops`.)

```
dist = ol.getdistprops(covidtotals.total_cases_pm)
pprint.pprint(dist)
```

```
{'count': 231,
 'kurtosis': -0.4280595203351645,
 'mean': 206177.79462337663,
 'median': 133946.251,
 'normpvalue': '0.0: Reject Normal',
 'normstat': 0.8750641345977783,
 'skew': 0.8349032460009967,
 'std': 203858.09625231632}
```

4. Create a function to list the outliers in a DataFrame.

The `getoutliers` function iterates over all columns in `sumvars`. It determines outlier thresholds for those columns, setting them at 1.5 times the interquartile range (the distance between the first and third quartiles) below the first quartile or above the third quartile. It then selects all rows with values above the high threshold or below the

low threshold. It adds columns that indicate the variable examined (`varname`) for outliers and the threshold levels. It also includes columns in the `othervars` list in the DataFrame it returns:

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

## 5. Call the `getoutlier` function.

Pass a list of columns to check for outliers (`sumvars`) and another list of columns to include in the returned DataFrame (`othervars`). Show the count of outliers for each variable and view the outliers for SAT math:

```
sumvars = ['satmath', 'wageincome20']
othervars = ['originalid', 'highestdegree', 'gender', 'maritalstatus']
outliers = ol.getoutliers(nls97, sumvars, othervars)
outliers.varname.value_counts(sort=False)
```

```
varname
satmath          10
wageincome20     234
Name: count, dtype: int64
```

```
outliers.loc[outliers.varname=='satmath', othervars + sum]
```

```
      originalid    highestdegree   ...    satmath
337438        159    2. High School   ...    200.00
448463        326    4. Bachelors    ...     47.00
799095        535    5. Masters     ...     59.00
267254       1622    2. High School   ...     48.00
955430       2547    2. High School   ...    200.00
748274       3394    4. Bachelors    ...     42.00
399109       3883    2. High School   ...     36.00
223058       6696      0. None       ...     46.00
291029       7088    2. High School   ...     51.00
738290       7705    2. High School   ...      7.00
[10 rows x 6 columns]
```

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

## 6. Create a function to generate histograms and boxplots.

The `makeplot` function takes a Series, title, and label for the *x*-axis.  
The default plot is set as a histogram:

```
def makeplot(seriestoplot, title, xlabel, plottype="hist"):
...     if (plottype=="hist"):
...         plt.hist(seriestoplot)
...         plt.axvline(seriestoplot.mean(), color='red', \
...                     linestyle='dashed', linewidth=1)
...         plt.xlabel(xlabel)
...         plt.ylabel("Frequency")
```

```
...     elif (plottype=="box"):
...         plt.boxplot(seriestoplot.dropna(), labels=[xlabel])
...         plt.title(title)
...         plt.show()
```

7. Call the `makeplot` function to create a histogram:

```
ol.makeplot(nls97.satmath, "Histogram of SAT Math", "
```

This generates the following histogram:

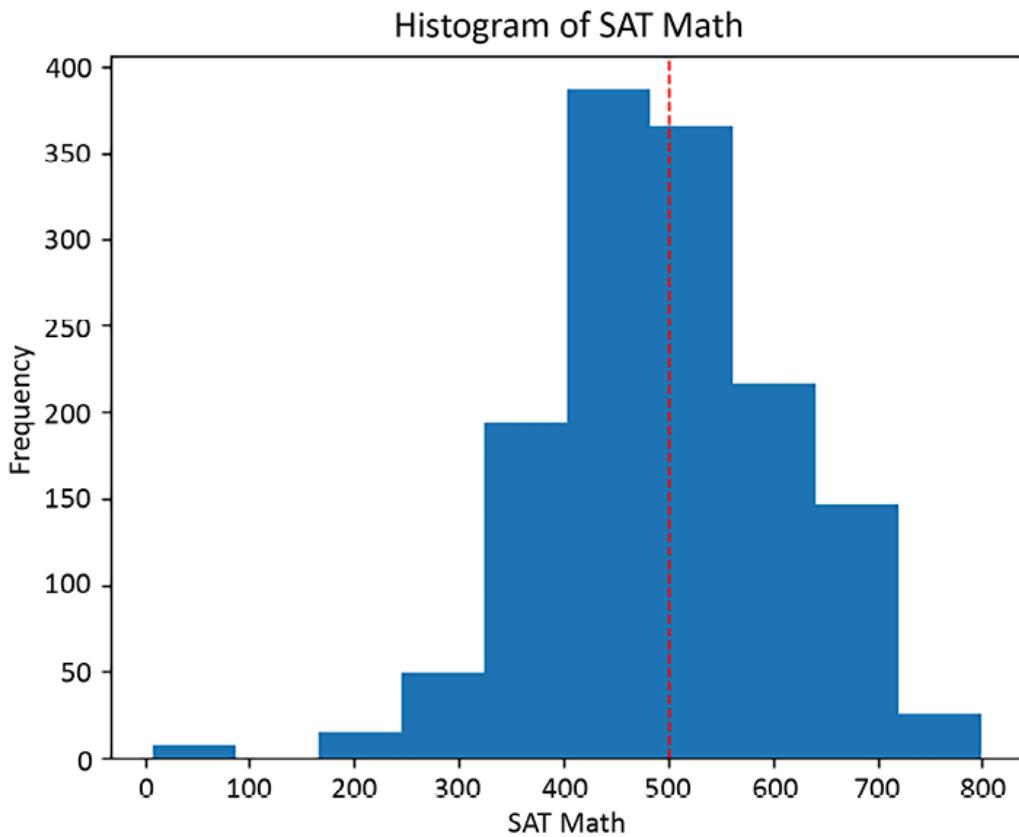


Figure 12.1: Frequencies of SAT math values

8. Use the `makeplot` function to create a boxplot:

```
ol.makeplot(nls97.satmath, "Boxplot of SAT Math", "SA")
```

This generates the following boxplot:

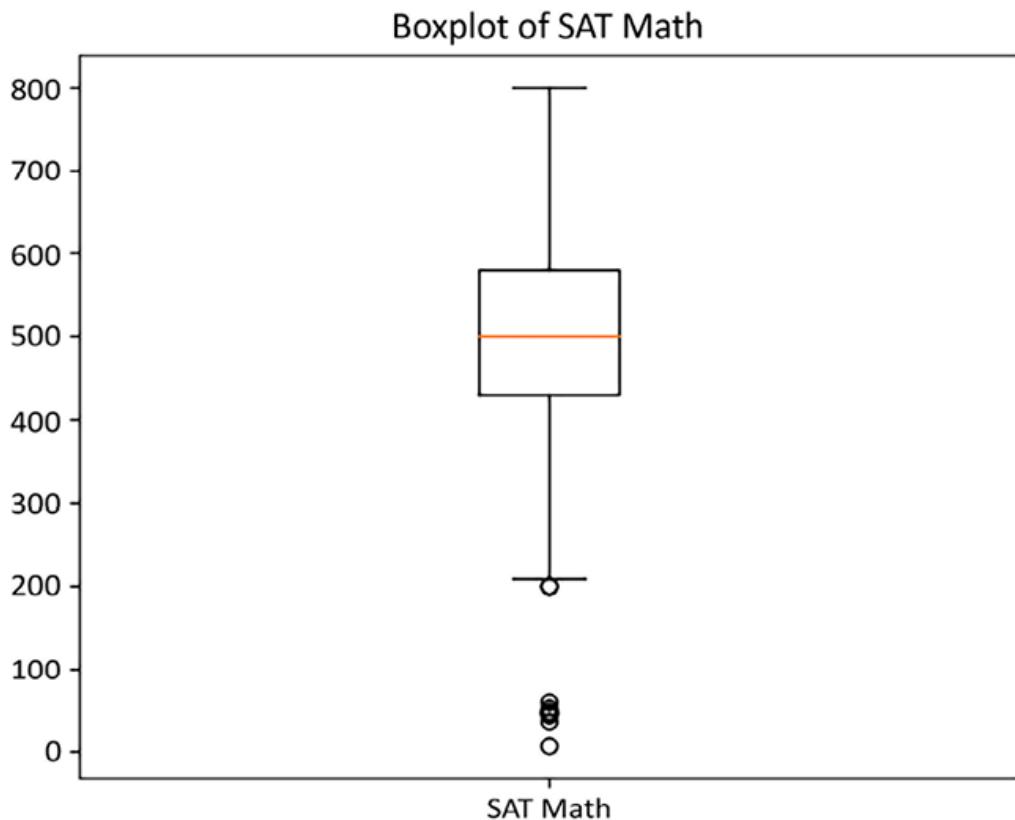


Figure 12.2: Show the median, interquartile range, and outlier thresholds with a boxplot

The preceding steps show how we can develop reusable code to check for outliers and unexpected values.

## How it works...

We start by getting the key attributes of a distribution, including the mean, median, standard deviation, skew, and kurtosis. We do this by passing a Series to the `getdistprop` function in *step 3*, getting back a dictionary with these measures.

The function in *step 4* selects rows where one of the columns in `sumvars` has a value that is an outlier. It also includes the values for the columns in `othervars` and the threshold amounts in the DataFrame it returns.

We create a function in *step 6* that makes it easier to create a simple histogram or boxplot. The functionality of `matplotlib` is great, but it can take a minute to remind ourselves of the syntax when we just want to create a simple histogram or boxplot. We can avoid that by defining a function with a few routine parameters: Series, title, and *x*-label. We call that function in *steps 7* and *8*.

## There's more...

We do not want to do too much work with a continuous variable before getting a good sense of how its values are distributed; what is the central tendency and shape of the distribution? If we run something like the functions in this recipe for key continuous variables, we would be off to a good start.

The relatively painless portability of Python modules makes this pretty easy to do. If we wanted to use the `outliers` module that we used in this example, we would just need to save the `outliers.py` file to a folder that our program can access, add that folder to the Python path, and import it.

Usually, when we are inspecting an extreme value, we want to have a better idea of the context of other variables that might explain why the value is extreme. For example, a height of 178 centimeters is not an outlier for an adult male, but it definitely is for a 9-year-old. The DataFrame produced in *steps 4* and *5* provides us with both the outlier values and other data that

might be relevant. Saving the data to an Excel file makes it easy to inspect outlier rows later or share that data with others.

## See also

We go into a fair bit of detail on detecting outliers and unexpected values in *Chapter 4, Identifying Outliers in Subsets of Data*. We examine histograms, boxplots, and many other visualizations in *Chapter 5, Using Visualizations for the Identification of Unexpected Values*.

## Functions for aggregating or combining data

Most data analysis projects require some reshaping of data. We may need to aggregate by group or combine data vertically or horizontally. We have to do similar tasks each time we prepare our data for this reshaping. We can routinize some of these tasks with functions, improving both the reliability of our code and our efficiency in getting the work done. We sometimes need to check for mismatches in merge-by columns before doing a merge, check for unexpected changes in values in panel data from one period to the next before aggregating, or concatenate a number of files at once and verify that data has been combined accurately.

These are just a few examples of the kind of data aggregation and combining tasks that might lend themselves to a more generalized coding solution. In this recipe, we define functions that can help with these tasks.

## Getting ready

We will work with the COVID-19 daily data in this recipe. This data comprises new cases and new deaths for each country by day. We will also work with land temperature data for several countries in 2023. The data for each country is in a separate file and has one row per weather station in that country for each month.



### Data note

The land temperature DataFrame has the average temperature readings (in °C) in 2023 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.ncei.noaa.gov/products/land-based-station/global-historical-climatology-network-monthly>.

## How to do it...

We will use functions to aggregate data, combine data vertically, and check merge-by values:

1. Import the `pandas`, `os`, and `sys` libraries:

```
import pandas as pd  
import os  
import sys
```

2. Create a function (`adjmeans`) to aggregate values by period for a group.

Sort the values in the passed DataFrame by group (`byvar`) and then `period`. Convert the DataFrame values to a NumPy array. Loop through the values, do a running tally of the `var` column, and set the running tally back to 0 when you reach a new value for `byvar`. Before aggregating, check for extreme changes in values from one period to the next. The `changeexclude` parameter indicates the size of a change from one period to the next that should be considered extreme. The `excludetype` parameter indicates whether the `changeexclude` value is an absolute amount or a percentage of the `var` column's mean. Save the function in a file called `combineagg.py` in the `helperfunctions` subfolder:

```
def adjmeans(df, byvar, var, period, changeexclude=None, ε
...     df = df.sort_values([byvar, period])
...     df = df.dropna(subset=[var])
...     # iterate using numpy arrays
...     prevbyvar = 'ZZZ'
...     prevvarvalue = 0
...     rowlist = []
...     varvalues = df[[byvar, var]].values
...     # convert exclusion ratio to absolute number
...     if (excludetype=="ratio" and changeexclude is not None):
...         changeexclude = df[var].mean()*changeexclude
...     # loop through variable values
...     for j in range(len(varvalues)):
...         byvar = varvalues[j][0]
...         varvalue = varvalues[j][1]
...         if (prevbyvar!=byvar):
...             if (prevbyvar!='ZZZ'):
...                 rowlist.append({'byvar':prevbyvar, 'avgvar':varvalue,
...                               'sumvar':varsum, 'byvarcnt':byvarcnt})
...                 varsum = 0
...                 byvarcnt = 0
```

```

...     prevbyvar = byvar
...     # exclude extreme changes in variable value
...     if ((changeexclude is None) or (0 <= abs(varvalue-
...         <= changeexclude) or (byvarcnt==0)):
...         varsum += varvalue
...         byvarcnt += 1
...         prevvarvalue = varvalue
...     rowlist.append({'byvar':prevbyvar, 'avgvar':varsum/t
...         'sumvar':varsum, 'byvarcnt':byvarcnt})
... return pd.DataFrame(rowlist)

```

### 3. Import the `combineagg` module:

```

sys.path.append(os.getcwd() + "/helperfunctions")
import combineagg as ca

```

### 4. Load the DataFrames:

```

coviddaily = pd.read_csv("data/coviddaily.csv")
ltbrazil = pd.read_csv("data/ltbrazil.csv")
countries = pd.read_csv("data/ltcountries.csv")
locations = pd.read_csv("data/ltlocations.csv")

```

### 5. Call the `adjmeans` function to summarize panel data by group and time period.

Indicate that we want a summary of `new_cases` by `location`:

```
ca.adjmeans(coviddaily, 'location','new_cases','casedate')
```

|   | byvar       | avgvar | sumvar  | by\\ |
|---|-------------|--------|---------|------|
| 0 | Afghanistan | 1,129  | 231,539 |      |
| 1 | Albania     | 1,914  | 334,863 |      |
| 2 | Algeria     | 1,439  | 272,010 |      |

|                        |                   |        |            |
|------------------------|-------------------|--------|------------|
| 3                      | American Samoa    | 144    | 8,359      |
| 4                      | Andorra           | 304    | 48,015     |
| ..                     | ...               | ...    | ...        |
| 226                    | Vietnam           | 60,542 | 11,624,000 |
| 227                    | Wallis and Futuna | 154    | 3,550      |
| 228                    | Yemen             | 98     | 11,945     |
| 229                    | Zambia            | 2,019  | 349,304    |
| 230                    | Zimbabwe          | 1,358  | 266,266    |
| [231 rows x 4 columns] |                   |        |            |

6. Call the `adjmeans` function again, this time excluding values where `new_cases` goes up or down by more than 5,000 from one day to the next. Notice some reduction in the counts for some countries:

|                        | byvar             | avgvar | sumvar  | byvarcr |
|------------------------|-------------------|--------|---------|---------|
| 0                      | Afghanistan       | 1,129  | 231,539 | 26      |
| 1                      | Albania           | 1,855  | 322,772 | 17      |
| 2                      | Algeria           | 1,290  | 239,896 | 18      |
| 3                      | American Samoa    | 144    | 8,359   | 5       |
| 4                      | Andorra           | 304    | 48,015  | 15      |
| ..                     | ...               | ...    | ...     | ..      |
| 226                    | Vietnam           | 6,410  | 967,910 | 15      |
| 227                    | Wallis and Futuna | 154    | 3,550   | 2       |
| 228                    | Yemen             | 98     | 11,945  | 12      |
| 229                    | Zambia            | 1,555  | 259,768 | 16      |
| 230                    | Zimbabwe          | 1,112  | 214,526 | 19      |
| [231 rows x 4 columns] |                   |        |         |         |

7. Create a function to check values for merge-by columns on one file but not another.

The `checkmerge` function does an outer join of two DataFrames passed to it, using the third and fourth parameters for the merge-by columns for the first and second DataFrames respectively. It then does a crosstab that shows the number of rows with merge-by values in both DataFrames and those in one DataFrame but not the other. It also shows up to 20 rows of data for merge-by values found in just one file:

```
def checkmerge(dfleft, dfright, mergebyleft, mergebyright):
...     dfleft['inleft'] = "Y"
...     dfright['inright'] = "Y"
...     dfboth = pd.merge(dfleft[[mergebyleft, 'inleft']], \
...                         dfright[[mergebyright, 'inright']], left_on=[mergebyleft], \
...                         right_on=[mergebyright], how="outer")
...     dfboth.fillna('N', inplace=True)
...     print(pd.crosstab(dfboth.inleft, dfboth.inright))
...     print(dfboth.loc[(dfboth.inleft=='N') | (dfboth.inright=='N')])
```

## 8. Call the `checkmerge` function.

Check a merge between the `countries` land temperatures DataFrame (which has one row per country) and the `locations` DataFrame (which has one row for each weather station in each country). The crosstab shows that 27,472 merge-by column values are in both DataFrames, two are in the `countries` file and not in the `locations` file, and one is in the `locations` file but not the `countries` file:

```
ca.checkmerge(countries.copy(), locations.copy(), \
...             "countryid", "countryid")
```

| inright | N | Y |
|---------|---|---|
| inleft  |   |   |

|       | 0         | 1      |         |
|-------|-----------|--------|---------|
| Y     | 2         | 27472  |         |
|       | countryid | inleft | inright |
| 7363  | F0        | N      | Y       |
| 9716  | LQ        | Y      | N       |
| 13104 | ST        | Y      | N       |

## 9. Create a function that concatenates all CSV files in a folder.

This function loops through all of the filenames in the specified folder. It uses the `endswith` method to check that the filename has a CSV file extension. It then loads the DataFrame and prints out the number of rows. Finally, it uses `concat` to append the rows of the new DataFrame to the rows already appended. If column names on a file are different, it prints those column names:

```
def addfiles(directory):
    ...     dfout = pd.DataFrame()
    ...     columnsmatched = True
    ...     # loop through the files
    ...     for filename in os.listdir(directory):
    ...         if filename.endswith(".csv"):
    ...             fileloc = os.path.join(directory, filename)
    ...             # open the next file
    ...             with open(fileloc) as f:
    ...                 dfnew = pd.read_csv(fileloc)
    ...                 print(filename + " has " + str(dfnew.shape[0]))
    ...                 dfout = pd.concat([dfout, dfnew])
    ...                 # check if current file has any different col
    ...                 columndiff = dfout.columns.symmetric_differenc
    ...                 if (not columndiff.empty):
    ...                     print("", "Different column names for:", fi
    ...                         columndiff, "", sep="\n")
    ...                 columnsmatched = False
```

```
...     print("Columns Matched:", columnsmatched)
...     return dfout
```

10. Use the `addfiles` function to concatenate all of the `countries` land temperature files.

It looks like the file for Oman (`ltoman`) is slightly different. It does not have the `latabs` column. Notice that the counts for each country in the combined DataFrame match the number of rows for each country file:

```
landtemps = ca.addfiles("data/ltcountry")
```

```
ltpoland.csv has 120 rows.
ltcameroon.csv has 48 rows.
ltmexico.csv has 852 rows.
ltjapan.csv has 1800 rows.
ltindia.csv has 1116 rows.
ltoman.csv has 288 rows.
Different column names for:
ltoman.csv
Index(['latabs'], dtype='object')
ltbrazil.csv has 1008 rows.
Columns Matched: False
```

```
landtemps.country.value_counts()
```

```
country
Japan        1800
India         1116
Brazil        1008
Mexico        852
Oman          288
```

```
Poland      120
Cameroon     48
Name: count, dtype: int64
```

The preceding steps demonstrate how we can systematize some of our messy data reshaping work. I am sure you can think of a number of other functions that might be helpful.

## How it works...

You may have noticed that in the `adjmeans` function we define in *step 2*, we actually do not append our summary of the `var` column values until we get to the next `byvar` column value. This is because there is no way to tell that we are on the last row for any `byvar` value until we get to the next `byvar` value. That is not a problem because we append the summary to `rowlist` right before we reset the value to `0`. This also means that we need to do something special to output the totals for the last `byvar` value since no next `byvar` value is reached. We do this with a final append after the loop is complete.

In *step 5*, we call the `adjmeans` function we defined in *step 2*. Since we do not set a value for the `changeexclude` parameter, the function will include all values in the aggregation. This will give us the same results as we would get using `groupby` with an aggregation function. When we pass an argument to `changeexclude`, however, we determine which rows to exclude from the aggregation. In *step 6*, the fifth argument in the call to `adjmeans` indicates that we should exclude new case values that are more than 5,000 cases higher or lower than the value for the previous day.

The function in *step 9* works well when the data files to be concatenated have the same, or nearly the same, structure. We print an alert when the column names are different, as *step 10* shows. The `latabs` column is not in the Oman file. This means that in the concatenated file, `latabs` will be missing for all of the rows for Oman.

## There's more...

The `adjmeans` function does a fairly straightforward check of each new value to be aggregated before including it in the total. But we could imagine much more complicated checks. We could even have made a call to another function within the `adjmeans` function where we are deciding whether to include the row.

## See also

We examine combining DataFrames vertically and horizontally in *Chapter 10, Addressing Data Issues When Combining DataFrames*.

## Classes that contain the logic for updating Series values

We sometimes work with a particular dataset for an extended period of time, occasionally years. The data might be updated regularly, for a new month or year, or with additional individuals, but the data structure might be fairly stable. If that dataset also has a large number of columns, we might be able to improve the reliability and readability of our code by implementing classes.

When we create classes, we define the attributes and methods of objects. When I use classes for my data-cleaning work, I tend to conceptualize a class as representing my unit of analysis. So, if my unit of analysis is a student, then I have a student class. Each instance of a student created by that class might have birth date and gender attributes and a course registration method. I might also create a subclass for alumni that inherits methods and attributes from the student class.

Data cleaning for the NLS DataFrame could be implemented nicely with classes. The dataset has been relatively stable for 25 years, both in terms of the variables and the allowable values for each variable. We explore how to create a respondent class for NLS survey responses in this recipe.

## Getting ready

You will need to create a `helperfunctions` subfolder in your current directory to run the code in this recipe. We will save the file (`respondent.py`) for our new class in that subfolder.

## How to do it...

We will define a respondent class to create several new Series based on the NLS data:

1. Import the `pandas`, `os`, `sys`, and `pprint` libraries.

We store this code in a different file than we will save the respondent class. Let's call this file `class_cleaning.py`. We will instantiate respondent objects from this file:

```
import pandas as pd
import os
import sys
import pprint
```

2. Create a `Respondent` class and save it to `respondent.py` in the `helperfunctions` subfolder.

When we call our class (instantiate a class object), the `__init__` method runs automatically. (There is a double underscore before and after `init`.) The `__init__` method has `self` as the first parameter, as any instance method does. The `__init__` method of this class also has a `respdict` parameter, which expects a dictionary of values from the NLS data. In later steps, we will instantiate a respondent object once for each row of data in the NLS DataFrame.

The `__init__` method assigns the passed `respdict` value to `self.respdict` to create an instance variable that we can reference in other methods. Finally, we increment a counter, `respondentcnt`. We will be able to use this later to confirm the number of instances of `respondent` that we created. We also import the `math` and `datetime` modules because we will need them later. (Notice that class names are capitalized by convention.)

```
import math
import datetime as dt
class Respondent:
...     respondentcnt = 0
...     def __init__(self, respdict):
...         self.respdict = respdict
...         Respondent.respondentcnt+=1
```

### 3. Add a method for counting the number of children.

This is a very simple method that just adds the number of children living with the respondent to the number of children not living with the respondent, to get the total number of children. It uses the `childathome` and `childnotathome` key values in the `self.respdict` dictionary:

```
def childnum(self):
...     return self.respdict['childathome'] + self.respdict[
```

### 4. Add a method for calculating average weeks worked across the 25 years of the survey.

Use dictionary comprehension to create a dictionary (`workdict`) of the weeks-worked keys that do not have missing values. Sum the values in `workdict` and divide that by the length of `workdict`:

```
def avgweeksworked(self):
...     workdict = {k: v for k, v in self.respdict.items() \
...                 if k.startswith('weeksworked') and not math.isnan(
...     nweeks = len(workdict)
...     if (nweeks>0):
...         avgww = sum(workdict.values())/nweeks
...     else:
...         avgww = 0
...     return avgww
```

### 5. Add a method for calculating age as of a given date.

This method takes a date string (`bydatestring`) to use for the end date of the age calculation. We use the `datetime` module to convert

the `date` string to a `datetime` object, `bydate`. We subtract the birth year value in `self.respdict` from the year of `bydate`, subtracting 1 from that calculation if the birth date has not happened yet that year. (We only have birth month and birth year in the NLS data, so we choose 15 as a midpoint.)

```
def ageby(self, bydatestring):
...     bydate = dt.datetime.strptime(bydatestring, '%Y%m%d')
...     birthyear = self.respdict['birthyear']
...     birthmonth = self.respdict['birthmonth']
...     age = bydate.year - birthyear
...     if (bydate.month<birthmonth or (bydate.month==birthmonth
...         and bydate.day<15)):
...         age = age -1
...     return age
```

6. Add a method to create a flag if the respondent ever enrolled at a 4-year college.

Use dictionary comprehension to check whether any college enrollment values are at a 4-year college:

```
def baenrollment(self):
...     colenrdict = {k: v for k, v in self.respdict.items()
...         if k.startswith('colenr') and v=="3. 4-year college"}
...     if (len(colenrdict)>0):
...         return "Y"
...     else:
...         return "N"
```

7. Import the respondent class.

Now we are ready to instantiate some `Respondent` objects! Let's do that from the `class_cleaning.py` file we started in *step 1*. We start by importing the respondent class. (This step assumes that `respondent.py` is in the `helperfunctions` subfolder.)

```
sys.path.append(os.getcwd() + "/helperfunctions")
import respondent as rp
```

## 8. Load the NLS data and create a list of dictionaries.

Use the `to_dict` method to create the list of dictionaries (`nls97list`). Each row from the DataFrame will be a dictionary with column names as keys. Show part of the first dictionary (the first row):

```
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)
nls97list = nls97.to_dict('records')
nls97.shape
```

```
(8984, 111)
```

```
len(nls97list)
```

```
8984
```

```
pprint.pprint(nls97list[0:1])
```

```
[{'birthmonth': 9,
 'birthyear': 1981,
 'childathome': nan,
 'childnotathome': nan,
 'colenrfeb00': '3. 4-year college',
```

```
'colenrfeb01': '3. 4-year college',
...
'weeksworked21': nan,
'weeksworked22': nan}]
```

## 9. Loop through the list, creating a `respondent` instance each time.

We pass each dictionary to the `respondent` class,

`rp.Respondent(respdict)`. Once we have created a `respondent` object (`resp`), we can then use all of the instance methods to get the values we need. We create a new dictionary with those values returned by instance methods. We then append that dictionary to `analysisdict`:

```
analysislist = []
for respdict in nls97list:
    ...
    resp = rp.Respondent(respdict)
    newdict = dict(originalid=respdict['originalid'],
    ...
    childnum=resp.childnum(),
    ...
    avgweeksworked=resp.avgweeksworked(),
    ...
    age=resp.ageby('20201015'),
    ...
    baenrollment=resp.baenrollment())
    ...
    analysislist.append(newdict)
```

## 10. Pass the dictionary to the pandas `DataFrame` method.

First, check the number of items in `analysislist` and the number of instances created:

```
len(analysislist)
```

```
8984
```

```
resp.respondentcnt
```

```
8984
```

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

```
[{'age': 39,
 'avgweeksworked': 48.4375,
 'baenrollment': 'Y',
 'childnum': nan,
 'originalid': 1},
 {'age': 38,
 'avgweeksworked': 49.90909090909091,
 'baenrollment': 'Y',
 'childnum': nan,
 'originalid': 2}]
```

```
analysis = pd.DataFrame(analysislist)
analysis.head(2)
```

|   | originalid | childnum | avgweeksworked | age | baer |
|---|------------|----------|----------------|-----|------|
| 0 | 1          | NaN      | 48             | 39  |      |
| 1 | 2          | NaN      | 50             | 38  |      |

These steps demonstrated how to create a class in Python, how to pass data to a class, how to create an instance of a class, and how to call the methods of the class to update variable values.

## How it works...

The key work in this recipe is done in *step 2*. It creates the respondent class and sets us up well for the remaining steps. We pass a dictionary with the values for each row to the class's `__init__` method. The `__init__` method assigns that dictionary to an instance variable that will be available to all of the class's methods (`self.respdict = respdict`).

*Steps 3* through *6* use that dictionary to calculate the number of children, average weeks worked per year, age, and college enrollment. *Steps 4* and *6* show how helpful dictionary comprehensions are when we need to test for the same value over many keys. The dictionary comprehensions select the relevant keys, `weeksworked##`, `colenroct##`, and `colenrfeb##`, and allow us to inspect the values of those keys. This is incredibly useful when we have data that is untidy in this way, as survey data often is.

In *step 8*, we create a list of dictionaries with the `to_dict` method. It has the expected number of list items, 8,984, the same as the number of rows in the DataFrame. We use `pprint` to show what the dictionary looks like for the first list item. The dictionary has keys for the column names and values for the column values.

We iterate over the list in *step 9*, creating a new respondent object and passing the list item. We call the methods to get the values we want, except for `originalid`, which we can pull directly from the dictionary. We create a dictionary (`newdict`) with those values, which we append to a list (`analysislist`).

In *step 10*, we create a pandas DataFrame from the list (`analysislist`) we created in *step 9*. We do this by passing the list to the pandas DataFrame method.

## There's more...

We pass dictionaries to the class rather than data rows, which is also a possibility. We do this because navigating a NumPy array is more efficient than looping over a DataFrame with `itertuples` or `iterrows`. We do not lose much of the functionality needed for our class when we work with dictionaries rather than DataFrame rows. We are still able to use functions such as `sum` and `mean` and count the number of values meeting certain criteria.

It is hard to avoid having to iterate over data with this conceptualization of a respondent class. This respondent class is consistent with our understanding of the unit of analysis, the survey respondent. That is also, unsurprisingly, how the data comes to us. But iterating over data one row at a time is resource-intensive, even with more efficient NumPy arrays.

I would argue, however, that you gain more than you lose by constructing a class like this one when working with data with many columns and with a structure that does not change much over time. The most important advantage is that it matches our intuition about the data and focuses our work on understanding the data for each respondent. I also think we find that when we construct the class well, we do far fewer passes through the data than we otherwise might.

## See also

We examine navigating over DataFrame rows and NumPy arrays in *Chapter 9, Fixing Messy Data When Aggregating*.

This was a very quick introduction to working with classes in Python. If you would like to learn more about object-oriented programming in Python,

I would recommend *Python 3 Object-Oriented Programming, Third Edition* by Dusty Phillips.

## Classes that handle non-tabular data structures

Data scientists increasingly receive non-tabular data, often in the form of JSON or XML files. The flexibility of JSON and XML allows organizations to capture complicated relationships between data items in one file. A one-to-many relationship stored in two tables in an enterprise data system can be represented well in JSON by a parent node for the one side and child nodes for data on the many side.

When we receive JSON data we often start by trying to normalize it. Indeed, we do that in a couple of recipes in this book. We try to recover the one-to-one and one-to-many relationships in the data obfuscated by the flexibility of JSON. But there is another way to work with such data, one that has many advantages.

Instead of normalizing the data, we can create a class that instantiates objects at the appropriate unit of analysis, and use the methods of the class to navigate the many side of one-to-many relationships. For example, if we get a JSON file that has student nodes and then multiple child nodes for each course taken by a student, we would usually normalize that data by creating a student file and a course file, with student ID as the merge-by column on both files. An alternative, which we explore in this recipe, would be to leave the data as it is, create a student class, and create methods that do calculations on the child nodes, such as calculating total credits taken.

Let's try that with this recipe using data from the Cleveland Museum of Art that has collection items, one or more nodes for media citations for each item, and one or more nodes for each creator of the item.

## Getting ready

This recipe assumes you have the `requests` and `pprint` libraries. If they are not installed, you can install them with `pip`. From Terminal, or PowerShell (in Windows), enter `pip install requests` and `pip install pprint`.

I show here the structure of the JSON file that is created when using the `collections` API of the Cleveland Museum of Art (I have abbreviated the JSON file to save space).

```
{
  "id": 165157,
  "title": "Fulton and Nostrand",
  "creation_date": "1958",
  "citations": [
    {
      "citation": "Annual Exhibition: Sculpture, Paintings, Watercolor and Printmaking, [8]-[12] (1958).",
      "page_number": "Unpaginated, [8], [12]",
      "url": null
    },
    {
      "citation": "Moscow to See Modern U.S. Art, New York (1958).",
      "page_number": "P. 60",
      "url": null
    }
  ],
  "creators": [
    {
      "description": "Jacob Lawrence (American, 1917-2000)",
      "role": "artist",
      "birth_year": "1917",
      "death_year": "2000"
    }
  ]
}
```

```
]  
}
```



## Data note

The Cleveland Museum of Art provides an API for public access to this data: <https://openaccess-api.clevelandart.org/>. Much more than the citations and creators data used in this recipe is available with the API.

## How to do it...

We create a collection item class that summarizes the data we need on creators and media citations:

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

Let's first create a file that we will use to instantiate collection item objects and call it `class_cleaning_json.py`:

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

2. Create a `CollectionItem` class.

We pass a dictionary for each collection item to the `__init__` method of the class, which runs automatically when an instance of the class is created. We assign the collection item dictionary to an instance

variable. Save the class as `collectionitem.py` in the `helperfunctions` folder:

```
class Collectionitem:  
...     collectionitemcnt = 0  
...     def __init__(self, colldict):  
...         self.colldict = colldict  
...         Collectionitem.collectionitemcnt+=1
```

3. Create a method to get the birth year of the first creator for each collection item.

Remember that collection items can have multiple creators. This means that the `creators` key has one or more list items as values, and these items are themselves dictionaries. To get the birth year of the first creator, then, we need `['creators'][0]['birth_year']`. We also need to allow for the birth year key to be missing, so we test for that first:

```
def birthyearcreator1(self):  
...     if ("birth_year" in self.colldict['creators'][0]):  
...         byear = self.colldict['creators'][0]['birth_year']  
...     else:  
...         byear = "Unknown"  
...     return byear
```

4. Create a method to get the birth years for all creators.

Use list comprehension to loop through all the `creators` items. This will return the birth years as a list:

```
def birthyearsall(self):  
...     byearlist = [item.get('birth_year') for item in \
```

```
...     self.colldict['creators']]  
...     return byearlist
```

5. Create a method to count the number of creators:

```
def ncreators(self):  
    ...     return len(self.colldict['creators'])
```

6. Create a method to count the number of media citations:

```
def ncitations(self):  
    ...     return len(self.colldict['citations'])
```

7. Import the `collectionitem` module.

We do this from the `class_cleaning_json.py` file we created in *step 1*:

```
sys.path.append(os.getcwd() + "/helperfunctions")  
import collectionitem as ci
```

8. Load the art museum's collections data.

This returns a list of dictionaries. We just pull a subset of the museum collections data with African American artists:

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

9. Loop through the `camcollections` list.

Create a collection item instance for each item in `camcollections`.

Pass each item, which is a dictionary of collections, creators, and citation keys, to the class. Call the methods we have just created and assign the values they return to a new dictionary (`newdict`). Append that dictionary to a list (`analysislist`). (Some of the values can be pulled directly from the dictionary, such as with

`title=colldict['title']`, since we do not need to change the value in any way.)

```
analysislist = []
for colldict in camcollections:
    ...     coll = ci.Collectionitem(colldict)
    ...     newdict = dict(id=colldict['id'],
    ...                     title=colldict['title'],
    ...                     type=colldict['type'],
    ...                     creationdate=colldict['creation_date'],
    ...                     ncreators=coll.ncreators(),
    ...                     ncitations=coll.ncitations(),
    ...                     birthyearsall=coll.birthyearsall(),
    ...                     birthyear=coll.birthyearcreator1())
    ...     analysislist.append(newdict)
```

## 10. Create an analysis DataFrame with the new list of dictionaries.

Confirm that we are getting the correct counts, and print the dictionary for the first item:

```
len(camcollections)
```

```
1000
```

```
len(analysislist)
```

```
1000
```

```
pprint.pprint(analysislist[0:1])
```

```
[{'birthyear': '1917',
 'birthyearsall': ['1917'],
 'creationdate': '1958',
 'id': 165157,
 'ncitations': 30,
 'ncreators': 1,
 'title': 'Fulton and Nostrand',
 'type': 'Painting'}]
```

```
analysis = pd.DataFrame(analysislist)
analysis.birthyearsall.value_counts().head()
```

```
birthyearsall
[1951]      283
[1953]      119
[1961, None] 105
[1937]      55
[1922]      41
Name: count, dtype: int64
```

```
analysis.head(2).T
```

|              | 0                   | 1             |
|--------------|---------------------|---------------|
| id           | 165157              | 163769        |
| title        | Fulton and Nostrand | Go Down Death |
| type         | Painting            | Painting      |
| creationdate | 1958                | 1934          |
| ncreators    | 1                   | 1             |
| ncitations   | 30                  | 18            |

|               |        |        |
|---------------|--------|--------|
| birthyearsall | [1917] | [1899] |
| birthyear     | 1917   | 1899   |

These steps give a sense of how we can use classes to handle non-tabular data.

## How it works...

This recipe demonstrated how to work directly with a JSON file, or any file with implied one-to-many or many-to-many relationships. We created a class at the unit of analysis (a collection item, in this case) and then created methods to summarize multiple nodes of data for each collection item.

The methods we created in *steps 3 to 6* are satisfyingly straightforward. When we first look at the structure of the data, displayed in the *Getting ready* section of this recipe, it is hard not to feel that it will be really difficult to clean. It looks like anything goes. But it turns out to have a fairly reliable structure. We can count on one or more child nodes for `creators` and `citations`. Each `creators` and `citations` node also has child nodes, which are key and value pairs. These keys are not always present, so we need to first check to see whether they are present before trying to grab their values. We did this in *step 3*.

## There's more...

I go into some detail about the advantages of working directly with JSON files in *Chapter 2, Anticipating Data Cleaning Issues When Working with HTML, JSON, and Spark Data*. I think the museum's collections data is a good example of why we might want to stick with the JSON if we can. The

structure of the data actually makes sense, even if it is in a very different form. There is always a danger when we try to normalize it that we will miss some aspects of its structure.

## Functions for checking overall data quality

We can tighten up our data quality checks by being more explicit and upfront about what we are evaluating. We likely have some expectations about the distribution of variable values, about the range of allowable values, and about the number of missing values very early in a data analysis project. This may come from documentation, our knowledge of the underlying real-world processes represented by the data, or our understanding of statistics. It is a good idea to have a routine for delineating those initial assumptions, testing them, and then revising assumptions throughout a project. This recipe will demonstrate what that process might look like.

We set up data quality targets for each variable of interest. This includes allowable values and thresholds for missing values for categorical variables. It also includes ranges of values; missing value, skewness, and kurtosis thresholds; and checking for outliers for numeric values. We will check unique identifier variables for duplication and for missing values. We start with the assumptions in this CSV file about variables on the NLS file:

|    | A                     | B           | C                                                    | D         | E                     | F              | G                  | H                | I       |
|----|-----------------------|-------------|------------------------------------------------------|-----------|-----------------------|----------------|--------------------|------------------|---------|
| 1  | varname               | type        | categories                                           | range     | missing thresho<br>ld | skewtar<br>get | kurtosis<br>target | showout<br>liers | include |
| 2  | maritalstatus         | categorical | Divorced Married Never-m<br>arried Separated Widowed |           | 0.2                   |                |                    |                  | Y       |
| 3  | colenroct21           | categorical | 1. Not enrolled 2. 2-year c<br>ompleted              |           | 0.2                   |                |                    |                  | N       |
| 4  | personid              | unique      |                                                      |           |                       |                |                    |                  | N       |
| 5  | originalid            | unique      |                                                      |           |                       |                |                    |                  | Y       |
| 6  | birthyear             | numeric     |                                                      | 1980 1984 | 0.3                   | 0              | 0 N                |                  | N       |
| 7  | highestgradecompleted | numeric     |                                                      | 5 16      | 0.3                   | 0              | 3 N                |                  | Y       |
| 8  | childathome           | numeric     |                                                      | 0 7       | 0.3                   | 0              | 3 Y                |                  | N       |
| 9  | childnotathome        | numeric     |                                                      | 0 7       | 0.3                   | 0              | 3 N                |                  | N       |
| 10 | satverbal             | numeric     |                                                      | 200 800   | 0.3                   | 0              | 3 N                |                  | N       |
| 11 | satmath               | numeric     |                                                      | 200 800   | 0.3                   | 0              | 3 N                |                  | N       |
| 12 | gpaoverall            | numeric     |                                                      | 100 400   | 0.3                   | 0              | 3 N                |                  | Y       |
| 13 | gpaenglish            | numeric     |                                                      | 100 400   | 0.3                   | 0              | 3 N                |                  | N       |
| 14 | gpamath               | numeric     |                                                      | 100 400   | 0.3                   | 0              | 3 N                |                  | N       |
| 15 | gpascience            | numeric     |                                                      | 100 400   | 0.3                   | 0              | 3 N                |                  | N       |
| 16 | weeksworked21         | numeric     |                                                      | 0 52      | 0.3                   | 0              | 3 N                |                  | N       |
| 17 | nightlyhrssleep       | numeric     |                                                      | 3 9       | 0.3                   | 0              | 3 Y                |                  | Y       |
| 18 |                       |             |                                                      |           |                       |                |                    |                  |         |

Figure 12.3: Data checks for selected NLS columns

Figure 12.3 shows our initial assumptions. For example, for

`maritalstatus`, we assume the category values *Divorced|Married|Never-married|Separated|Widowed*, and that no more than 20% of values will be missing. For `nightlyhrssleep`, a numeric variable, we assume that values will be between 3 and 9, that no more than 30% of values will be missing, and that it will have a skew and kurtosis close to that of a normal distribution.

We also indicate that we want to check for outliers. The final column is a flag we can use if we only want to do data checks for a few variables. Here we indicate that we want to do checks for `maritalstatus`, `originalid`, `highestgradecompleted`, `gpaenglish`, and `nightlyhrssleep`.

## Getting ready

We will work again with the NLS data in this recipe.

## How to do it...

We use our predefined data-checking targets to analyze selected variables in the NLS data.

1. Create the functions we will need for data checking and save them in the `helperfunctions` subfolder with the name `runchecks.py`. The following two functions, `checkcats` and `checkoutliers`, will be used to test values in a list and outliers respectively. We will see how that works in subsequent steps:

```
def checkcats(cat1,cat2):
    missingcats = \
        set(cat1).symmetric_difference(set(cat2))
    return missingcats
def checkoutliers(values):
    thirdq, firstq = values.\
        quantile(0.75),values.\
        quantile(0.25)
    interquartilerange = 1.5*(thirdq-firstq)
    outlierhigh, outlierlow = \
        interquartilerange+thirdq, \
        firstq-interquartilerange
    return outlierhigh, outlierlow
```

2. We then define a function to run all of our checks, `runchecks`, which will take a DataFrame (`df`), our data targets (`dc`), a list of numerical columns (`numvars`), a list of categorical columns (`catvars`), and a list of identifier columns (`idvars`):

```
def runchecks(df,dc,numvars,catvars,idvars):
```

3. Within the `runchecks` function, we loop over the categorical variable columns in our data checks. We get the values of all targets for the variable with `dcvals = dc.loc[col]`. We create a NumPy array,

`compcat`, from the category values. We then compare that array to all of the values for that column in the passed DataFrame (`(df[col].dropna().str.strip().unique())`). If there is a category in one array but not the other (`valuediff`) we print that to the console. We also calculate the missing-value percentage. If it is beyond the threshold we specified, we print a message:

```
for col in df[catvars]:
    dcvals = dc.loc[col]
    print("\n\nChecks for categorical variable", col)
    compcat = list(dcvals.categories.split(' | '))
    valuediff = checkcats(compcat, df[col].dropna().\
        str.strip().unique())
    if len(valuediff) > 0:
        print("at least one non-matching category:",
              valuediff)

    missingper = df[col].isnull().sum()/df.shape[0]
    if missingper > dcvals.missingthreshold:
        print("missing percent beyond threshold of",
              dcvals.missingthreshold, "is", missingper)
```

4. Let's now look at the loop for checking the numeric variables. We create a NumPy array from the range value in our data-checking targets, `range = np.fromstring(dcvals.range, sep=' | ')`. The first element of `range` is the lower end of the range. The second element is the upper end. We then get the min and max values for the variable from the DataFrame and compare those with the range indicated in the target file.

We calculate the missing-values percentage and print if it exceeds the threshold we set in the data-checking target file.

We show outliers if the `showoutliers` flag is set to `Y`. We use the `checkoutliers` function we set up earlier, which uses a simple interquartile range calculation to determine outliers. Finally, we check the skew and kurtosis to get an indication of how far from normally distributed the variable might be:

```
for col in df[numvars]:
    dcvals = dc.loc[col]
    print("\n\nChecks for numeric variable", col)

    range = np.fromstring(dcvals.range, sep=' | ')
    min = df[col].min()
    max = df[col].max()
    if min < range[0]:
        print("at least one record below range starting at '",
              range[0], "min value is", min)
    if max > range[1]:
        print("at least one record above range ending at ",
              range[1], "max value is", max)
    missingper = df[col].isnull().sum()/df.shape[0]
    if missingper > dcvals.missingthreshold:
        print("missing percent beyond threshold of",
              dcvals.missingthreshold, "is", missingper)

    if dcvals.showoutliers == "Y":
        outlierhigh, outlierlow = checkoutliers(df[col])
        print("\nvalues less than", outlierlow, "\n",
              df.loc[df[col]<outlierlow,col].\
              agg(["min", 'max', 'count']), end="\n")
        print("\nvalues greater than", outlierhigh,
              "\n", df.loc[df[col]>outlierhigh,col].\
              agg(["min", 'max', 'count']), end="\n")
        skewcol = df[col].skew()
        if abs(skewcol-dcvals.skewtarget)>1.2:
            print("skew substantially different from target of",
                  dcvals.skewtarget, "is", skewcol)

    kurtosiscol = df[col].kurtosis()
    if abs(kurtosiscol-dcvals.kurtosistarget)>1.2:
```

```
    print("kurtosis substantially different from target  
dcvals.kurtosistarget, "is", kurtosiscol)
```

5. We do a couple of straightforward checks for variables identified as id variables in the targets file. We look to see if the variable is duplicated and check for missing values:

```
for col in df[idvars]:  
    print("\n\nChecks for id variable", col)  
  
    uniquevals = df[col].nunique()  
    nrows = df.shape[0]  
    if uniquevals != nrows:  
        print("not unique identifier", uniquevals,  
              "unique values not equal to", nrows, "rows.")  
  
    missingvals = df[col].isnull().sum()  
    if missingvals > 0:  
        print("unique value has", missingvals,  
              "missing values")
```

6. Now we are ready to run the data checks. We start by loading the NLS DataFrame and the data-checking targets.

```
import pandas as pd  
import numpy as np  
import os  
import sys  
nls97 = pd.read_csv("data/nls97g.csv", low_memory=False)  
dc = pd.read_csv("data/datacheckingtargets.csv")  
dc.set_index('varname', inplace=True)
```

7. We import the `runchecks` module we just created.

```
    sys.path.append(os.getcwd() + "/helperfunctions")
    import runchecks as rc
```

8. Let's mess up some of the id variable values for testing the code. We also fix logical missing values for `highestgradecompleted`, setting them to actual missing values.

```
nls97.originalid.head(7)
```

```
0    1
1    2
2    3
3    4
4    5
5    6
6    7
Name: originalid, dtype: int64
```

```
nls97.loc[nls97.originalid==2, "originalid"] = 1
nls97.loc[nls97.originalid.between(3,7), "originalid"]
nls97.originalid.head(7)
```

```
0    1.0
1    1.0
2    NaN
3    NaN
4    NaN
5    NaN
6    NaN
Name: originalid, dtype: float64
```

```
nls97["highestgradecompleted"] = nls97.highestgradeco
```

9. We select just those targets flagged to be included. We then create categorical variable, numeric variable, and id variable lists based on the data-checking targets file:

```
dc = dc.loc[dc.include=="Y"]
numvars = dc.loc[dc.type=="numeric"].index.to_list()
catvars = dc.loc[dc.type=="categorical"].index.to_list()
idvars = dc.loc[dc.type=="unique"].index.to_list()
```

10. Now, we are ready to run the checks.

```
rc.runchecks(nls97, dc, numvars, catvars, idvars)
```

This produces the following output:

```

1 Checks for categorical variable maritalstatus
2 missing percent beyond threshold of 0.2 is 0.25701246660730187
3
4
5 Checks for numeric variable highestgradecompleted
6 at least one record above range ending at 16.0 max value is 20.0
7 kurtosis substantially different from target of 3.0 is -0.6076960550376977
8
9
10 Checks for numeric variable gpaoverall
11 at least one record below range starting at 100.0 min value is 10.0
12 at least one record above range ending at 400.0 max value is 417.0
13 missing percent beyond threshold of 0.3 is 0.3317008014247551
14 kurtosis substantially different from target of 3.0 is 0.3125440750593671
15
16
17 Checks for numeric variable nightlyhrssleep
18 at least one record below range starting at 3.0 min value is 0.0
19 at least one record above range ending at 9.0 max value is 20.0
20
21 values less than 3.0
22 min      0.0
23 max      2.0
24 count    31.0
25 Name: nightlyhrssleep, dtype: float64
26
27 values greater than 11.0
28 min      12.0
29 max      20.0
30 count    27.0
31 Name: nightlyhrssleep, dtype: float64
32 kurtosis substantially different from target of 3.0 is 5.087650564985399
33
34
35 Checks for id variable originalid
36 not unique identifier 8978 unique values not equal to 8984 rows.
37 unique value has 5 missing values

```

*Figure 12.4: Running the checks*

We see that `maritalstatus` has more missings (26%) than the 20% threshold we set. `highestgradecompleted` and `gpaoverall` have values that exceed the anticipated range. The kurtosis for both variables is low.

`nightlyhrssleep` has outliers substantially below and above the interquartile range. 31 respondents have `nightlyhrssleep` of 2 or less. 27 respondents have very high `nightlyhrssleep`, of 12 or more.

These steps show how we can use our prior domain knowledge and understanding of statistics to better target our investigation of data quality.

## How it works...

We created a CSV file with our data-checking targets. We used that file in our checks of the NLS data. We did that by passing both the NLS DataFrame and the data-checking targets to `runchecks`. Code in `runchecks` loops through the column names from the data-checking file and does checks based on the type of variable.

The targets for each variable are defined by `dcvals = dc.loc[col]`, which grabs all of the target values for that row in the target file. We can then refer to `dcvals.missingthreshold` to get the missing-values threshold, for example. We then compare the percentage of missing values (`(df[col].isnull().sum()/df.shape[0])`) to the missing threshold, and print a message if the missing-value percentage is greater than the threshold. We do the same type of checking for range of values, skew, outliers, and so on, depending on the type of variable.

We can add new variables to the data-checking targets file without changing any of the code in `runchecks`. We can also change target values.

## There's more...

It is sometimes important to be proactive with our data checks. There is a difference between displaying some sample statistics and frequency distributions to get a general sense of the data, and marshaling our domain knowledge and understanding of statistics for careful examination of data quality. A more intentional approach might require us to occasionally step away from our Python development environment for a moment to reflect on our expectations for data values and their distribution. Setting up initial data quality targets, and revisiting them regularly, can help us do that.

# Pre-processing data with pipelines: a simple example

When doing predictive analysis, we often need to fold all of our pre-processing and feature engineering into a pipeline, including scaling, encoding, and handling outliers and missing values. We discussed the reasons why we might need to incorporate all of our data preparation into a data pipeline in *Chapter 8, Encoding, Transforming, and Scaling Features*. The main takeaway from that chapter is that pipelines are critical when we are building explanatory models and need to avoid data leakage. This can be trickier still when we are using  $k$ -fold cross-validation for model validation, since testing and training DataFrames change during evaluation. Cross-validation has become the norm when constructing predictive models.



**Note**

$k$ -fold cross-validation trains our model on all but one of the  $k$  folds, or parts, leaving one out for testing. This is repeated  $k$  times, each time excluding a different fold for testing. Performance metrics are then based on the average scores across the  $k$  folds.

Another benefit of pipelines is that they help us ensure reproducible results, as they are often intended to take our analysis from raw data to model evaluation.

Although this recipe demonstrates how to use a pipeline all the way through model evaluation, we will not go into detail there. A good resource for both

model evaluation and pipelines with scikit-learn tools is the book *Data Cleaning and Exploration with Machine Learning*, also written by me.

We start with a relatively simple example here, a model with two numeric features and one numeric target. We work on a much more complicated example in the next recipe.

## Getting ready

We will work with scikit-learn's pipeline tool in this recipe, and a few other modules for encoding and scaling the data, and imputing values for missing data. We will work with land temperature data again in this recipe.

## How to do it...

1. We start by loading the `scikit-learn` modules we will be using in this recipe for transforming our data. We will use `StandardScaler` to standardize our features, `SimpleImputer` to impute values for missing data, and `make_pipeline` to pull all of our pre-processing together. We also use `train_test_split` to create training and testing DataFrames. I'll discuss the other modules as we use them:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.impute import SimpleImputer
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_validate
from sklearn.model_selection import KFold
```

2. We load the land temperature data and create training and testing DataFrames. We will try to model temperature as a function of latitude and elevation. Given the very different ranges of the `latabs` and `elevation` variables, scaling will be important:

```
landtemps = pd.read_csv("data/landtemps2023avgs.csv")
feature_cols = ['latabs', 'elevation']
X_train, X_test, y_train, y_test = \
    train_test_split(landtemps[feature_cols], \
        landtemps[['avgtemp']], test_size=0.1, random_state
```

For an introduction to `train_test_split`, see *Chapter 8, Encoding, Transforming, and Scaling Features*.

3. We set up  $k$ -fold cross-validation. We indicate that we want five folds and for the data to be shuffled:

```
kf = KFold(n_splits=5, shuffle=True, random_state=0)
```

4. Now, we are ready to set up our pipeline. The pipeline will do standard scaling, impute the mean when values are missing, and then run a linear regression model. Both features will be handled in the same way.

```
pipeline = \
    make_pipeline(StandardScaler(),
        SimpleImputer(strategy="mean"), LinearRegression())
```

5. After constructing the pipeline, and instantiating a  $k$ -fold cross-validation object, we are ready to run the pre-processing, estimate the model, and generate evaluation metrics. We pass the pipeline to the `cross_validate` function, as well as our training data. We also pass the

`kfold` object we created in *step 3*. We get a pretty decent *R-squared* value.

```
scores = \
    cross_validate(pipeline, X=x_train, y=y_train.value
    cv=kf, scoring=['r2', 'neg_mean_absolute_error'],
    n_jobs=1)
print("Mean Absolute Error: %.2f, R-squared: %.2f" %
    (scores['test_neg_mean_absolute_error'].mean(),
     scores['test_r2'].mean()))
```

```
Mean Absolute Error: -2.53, R-squared: 0.82
```

## How it works...

We used scikit-learn's `make_pipeline` to create a pipeline with just three steps: apply standard scaling, impute values for missing data based on the mean for that variable, and fit a linear regression model. What is so helpful about pipelines is that they automatically feed a transformation from one step into the next step. This is easy to do, once we get the hang of it, despite the complication of *k*-fold cross-validation.

We can imagine for a moment how messy it would be to write our own code to do this when the training and testing `DataFrames` are changing, as with *k*-fold cross-validation. Even something as simple as using the mean for imputation is tricky. We would need to take a new mean for the training data each time the training data changed. Our pipeline handles all of this for us.

That was a relatively straightforward example, with just a couple of features that we could handle in the same way. We also did not bother to check for

outliers or scale the target variable. We use a pipeline to handle a much trickier modeling project in the next recipe.

## Pre-processing data with pipelines: a more complicated example

If you have ever built a data pipeline, you know that it can be a little messy when you are working with several different data types. For example, we might need to impute the median for missing values with continuous features and the most frequent value for categorical features. We might also need to transform our target variable. We explore how to apply different pre-processing to different variables in this recipe.

### Getting ready

We will work with a fair number of scikit-learn modules in this recipe. Although this can be confusing at first, you quickly become grateful that scikit-learn has a tool to do pretty much anything you need. Scikit-learn also allows us to add our own transformations to a pipeline if we need to do so. I demonstrate how to construct our own transformer in this recipe.

We will work with wage and employment data from the NLS.

### How to do it...

1. We start by loading the libraries we used in the previous recipe. Then we add the `ColumnTransformer` and `TransformedTargetRegressor` classes. We will use those classes to transform our features and target respectively.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.impute import SimpleImputer
from sklearn.pipeline import make_pipeline
from feature_engine.encoding import OneHotEncoder
from sklearn.impute import KNNImputer
from sklearn.model_selection import cross_validate, K
from sklearn.compose import ColumnTransformer
from sklearn.compose import TransformedTargetRegresso
```

2. The column transformer is quite flexible. We can even use it with pre-processing functions we have defined ourselves. The code block below imports the `OutlierTrans` class from the `preprocfunc` module in the `helperfunctions` subfolder.

```
import os
import sys
sys.path.append(os.getcwd() + "/helperfunctions")
from preprocfunc import OutlierTrans
```

3. The `OutlierTrans` class identifies missing values by distance from the interquartile range. This is a technique we demonstrated in *Chapter 4, Identifying Outliers in Subsets of Data*, and have used multiple times in this chapter.

To work in a scikit-learn pipeline our class has to have `fit` and `transform` methods. We also need to inherit the `BaseEstimator` and `TransformerMixin` classes.

In this class, almost all of the action happens in the `transform` method. Any value that is more than 1.5 times the interquartile range

above the third quartile or below the first quartile is assigned missing, though that threshold can be changed:

```
class OutlierTrans(BaseEstimator, TransformerMixin):
    def __init__(self, threshold=1.5):
        self.threshold = threshold

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        Xnew = X.copy()
        for col in Xnew.columns:
            thirdq, firstq = Xnew[col].quantile(0.75), \
                Xnew[col].quantile(0.25)
            inlierrange = self.threshold*(thirdq-firstq)
            outlierhigh, outlierlow = inlierrange+thirdq, \
                firstq-inlierrange
            Xnew.loc[(Xnew[col]>outlierhigh) | \
                (Xnew[col]<outlierlow), col] = np.nan
        return Xnew.values
```

Our `OutlierTrans` class can be used later in our pipeline in the same way we use the `StandardScaler` and other transformers. We will do that later.

4. Now we are ready to load the data that needs to be processed. We will work with the NLS wage data. Wage income will be our target, and we will use high school GPA, mother's and father's highest grade completed, parent income, gender, weeks worked, and whether the individual completed a bachelor's degree as features.

We create lists of features to handle in different ways here. That will be helpful later when we instruct our pipeline to carry out different operations on numerical, categorical, and binary features.

```
nls97wages = pd.read_csv("data/nls97wages.csv", low_memory=True)
nls97wages.set_index("personid", inplace=True)
num_cols = ['gpascience', 'gpaenglish', 'gpamath',
            'gpaoverall', 'motherhighgrade', 'fatherhighgrade',
            'parentincome', 'weeksworked20']
cat_cols = ['gender']
bin_cols = ['completedba']
target = nls97wages[['wageincome20']]
features = nls97wages[num_cols + cat_cols + bin_cols]
X_train, X_test, y_train, y_test = \
    train_test_split(features, \
                     target, test_size=0.2, random_state=0)
```

5. Now we can set up a column transformer. We first create pipelines for handling numerical data (`standtrans`), categorical data, and binary data.

For the numerical data (continuous), we want to assign outlier values to missing. Here we pass a value of 2 to the `threshold` parameter of `OutlierTrans`, indicating that we want values 2 times the interquartile range above or below that range to be set to missing. Recall that it is common to use 1.5, so we are being somewhat conservative.

We do one-hot encoding of the `gender` column, essentially creating a dummy variable. We drop the last category to avoid the **dummy variable trap**, as discussed in *Chapter 8, Encoding, Transforming, and Scaling Features*.

We then create a `ColumnTransformer` object, passing to it the three pipelines we just created, and indicating which features to use with which pipeline.

We do not worry about missing values yet for the numeric variables.  
We will handle them later:

```
standtrans = make_pipeline(OutlierTrans(2),
    StandardScaler())
cattrans = \
    make_pipeline(SimpleImputer(strategy=\
        "most_frequent"), OneHotEncoder(drop_last=True))
bintrans = \
    make_pipeline(SimpleImputer(strategy=\
        "most_frequent"))
coltrans = ColumnTransformer(
    transformers=[
        ("stand", standtrans, num_cols),
        ("cat", cattrans, ['gender']),
        ("bin", bintrans, ['completedba'])])
)
```

6. We can now add the column transformer to a pipeline that also includes the linear model that we would like to run. We add KNN imputation to the pipeline to handle missing values for the numeric values. We have already handled missing values for the categorical variables.

We also need to scale the target, which cannot be done in our pipeline. We use scikit-learn's `TransformedTargetRegressor` for that. We pass the pipeline we just created to the target regressor's `regressor` parameter.

```
lr = LinearRegression()
pipe1 = make_pipeline(coltrans,
    KNNImputer(n_neighbors=5), lr)
```

```
ttr=TransformedTargetRegressor(regressor=pipe1,  
                                transformer=StandardScaler())
```

7. Let's do  $k$ -fold cross-validation using this pipeline. We can pass our pipeline, via the target regressor `ttr`, to the `cross_validate` function.

```
kf = KFold(n_splits=10, shuffle=True, random_state=0)  
scores = cross_validate(ttr, X=X_train, y=y_train,  
                       cv=kf, scoring=('r2', 'neg_mean_absolute_error'),  
                       n_jobs=1)  
print("Mean Absolute Error: %.2f, R-squared: %.2f" %  
      (scores['test_neg_mean_absolute_error'].mean(),  
       scores['test_r2'].mean()))
```

```
Mean Absolute Error: -32899.64, R-squared: 0.16
```

These scores are not very good, though that was not quite the point of this exercise. The key takeaway here is that we typically want to fold most of the pre-processing we will do into a pipeline. This is the best way to avoid data leakage. The column transformer is an extremely flexible tool, allowing us to apply different transformations to different features.

## How it works...

We created several different pipelines to pre-process our data before fitting our model, one for numeric data, one for categorical data, and one for binary data. The column transformer helps us by allowing us to apply different pipelines to different columns. We set up the column transformer in *step 5*.

We created another pipeline in *step 6*. That pipeline actually begins with the column transformer. Then, the dataset that results from the column transformer's pre-processing is passed to the KNN imputer to handle missing values from the numeric columns, and then to the linear regression model.

It is good to note that we are able to add transformations to a scikit-learn pipeline, even ones we have designed ourselves, because they inherit the `BaseEstimator` and `TransformerMixin` classes, as we saw in *step 3*.

## There's more...

There is one additional thing that is very cool, and useful, about pipelines that was not demonstrated in this example. If you have ever had to generate predictions based on variables that have been scaled or transformed in some way, you likely remember how much of a nuisance that can be. Well, pipelines handle that for us, generating predictions in the appropriate units.

## See also

This really just scratches the surface of what can be done with pipelines. For a fuller discussion, see the book *Data Cleaning and Exploration with Machine Learning*, also written by me.

## Summary

There was a fair bit packed into this chapter, covering several approaches to automating our data-cleaning work. We created functions for showing the structure of our data and generating descriptive statistics. We created functions for restructuring and aggregating our data. We also developed

Python classes for handling data cleaning when we have a large number of variables, each requiring very different treatment. We also saw how Python classes can make it easier to work directly with a JSON file. We examined being more intentional with our data cleaning by checking our data against predefined targets. Finally, we explored how to automate our data cleaning with pipelines.

## Leave a review!

Enjoyed this book? Help readers like you by leaving an Amazon review.

Scan the QR code below to get a free eBook of your choice.





[packt.com](http://packt.com)

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At [www.packt.com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



## Azure Data Factory Cookbook – Second Edition

Dmitry Foshin

Dimtry Anoshin

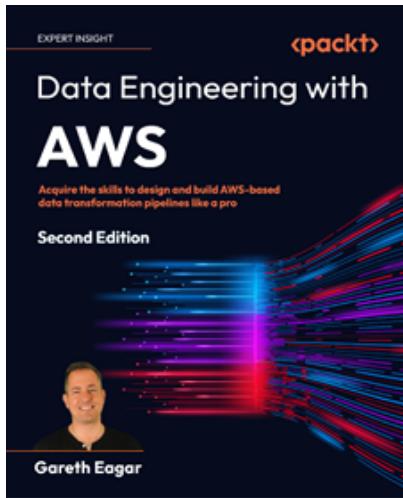
Tonya Chernyshova

Xenia Irton

ISBN: 978-1-80324-659-8

- Create an orchestration and transformation job in ADF
- Develop, execute, and monitor data flows using Azure Synapse

- Create big data pipelines using Databricks and Delta tables
- Work with big data in Azure Data Lake using Spark Pool
- Migrate on-premises SSIS jobs to ADF
- Integrate ADF with commonly used Azure services such as Azure ML, Azure Logic Apps, and Azure Functions
- Run big data compute jobs within HDInsight and Azure Databricks
- Copy data from AWS S3 and Google Cloud Storage to Azure Storage using ADF's built-in connectors



## **Data Engineering with AWS – Second Edition**

Gareth Eagar

ISBN: 978-1-80461-442-6

- Seamlessly ingest streaming data with Amazon Kinesis Data Firehose
- Optimize, denormalize, and join datasets with AWS Glue Studio
- Use Amazon S3 events to trigger a Lambda process to transform a file
- Load data into a Redshift data warehouse and run queries with ease
- Visualize and explore data using Amazon QuickSight
- Extract sentiment data from a dataset using Amazon Comprehend
- Build transactional data lakes using Apache Iceberg with Amazon Athena
- Learn how a data mesh approach can be implemented on AWS

# Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](https://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

# Share your thoughts

Now you've finished *Python Data Cleaning Cookbook, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here to go straight to the Amazon review page](#) for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

# Index

## A

### **anomalies**

finding, with Isolation Forest [152-156](#)

### **anti-pattern [318-322](#)**

### **API**

complicated JSON data, importing from [53-57](#)

### **apply**

using, with groupby [336-341](#)

## B

### **bagging [276](#)**

### **Beautiful Soup [58](#)**

### **binning [308](#)**

### **bivariate relationships**

outliers, identifying [128-135](#)

unexpected values, identifying [128-135](#)

viewing, with scatter plots [191-197](#)

### **boxplots**

used, for identifying outliers for continuous variables [173-179](#)

### **broadcasting [224](#)**

# C

## categorical features

encoding [294-297](#)

encoding, with high cardinality [300-303](#)

encoding, with medium cardinality [300-303](#)

## categorical variables

frequencies, generating for [98-102](#)

## chaining [8](#)

## classes

logic, for updating Series values [429-434](#)

non-tabular data structures, handling [435-439](#)

## Cleveland Museum of Art Open Access API

reference link [54](#), [371](#), [436](#)

## columns [286](#)

organizing [84-89](#)

selecting [84-89](#)

## comma separated values (CSV) [2](#)

## complicated aggregation functions

using, with groupby [330-336](#)

## complicated JSON data

importing, from API [53-57](#)

## continuous variables

summary statistics, generating for [102-106](#)

## correlation matrix

heat map, generating based on [204-208](#)

## CSV files

importing [2-7](#)

# D

## data

importing, from SQL databases [18-25](#)

importing, from web pages [58-63](#)

organizing, by groups with groupby method [326-329](#)

reshaping, from wide to long format with melt [393-398](#)

reshaping, from wide to long format with pivot [401-404](#)

reshaping, from wide to long format with stack [393-398](#)

reshaping, from wide to long format with unstack [401-404](#)

versioning [72-74](#)

## DataCamp

reference link [46](#)

## data, converting from wide to long format [394](#)

## DataFrame [1](#)

combining, vertically [348-354](#)

unit of analysis, modifying with groupby [341-344](#)

unit of analysis, modifying with pivot\_table [344-346](#)

## data leakage [286](#)

avoiding [286-289](#)

## data points, with significant influence

identifying, with linear regression [144-147](#)

## **data, pre-processing with pipelines**

complicated example [449-453](#)

simple example [446-449](#)

## **datasets [78](#)**

### **data types [7](#)**

### **data, with itertuples**

looping through [318-322](#)

## **dates**

working with [241-247](#)

## **dependent/response variables [286, 289](#)**

### **descriptive statistics**

displaying, with generative AI [107-115](#)

## **deterministic regression imputation [267](#)**

### **discretization [308](#)**

### **distribution of continuous variables**

examining, with histograms [166-173](#)

### **distribution shape and outliers**

examining, with violin plots [186-190](#)

### **dummy variables [294](#)**

### **dummy variable trap [296](#)**

### **duplicated rows**

removing [382-385](#)

## E

**equal frequency binning** [308-311](#)

**equal width binning** [308-311](#)

### Excel files

importing [9-17](#)

## F

**feature binning** [308-311](#)

**feature hashing** [302](#)

**feature scaling** [313-316](#)

**filter operator** [90](#)

### frequencies

generating, for categorical variables [98-102](#)

### functions

data, aggregating or combining [423-429](#)

first look of data, obtaining [406-411](#)

outliers and unexpected values, identifying [417-422](#)

overall data quality, checking [440-446](#)

summary statistics and frequencies, displaying [411-416](#)

## G

### generative AI

used, for displaying descriptive statistics [107-115](#)

### Global Historical Climatology Network

reference link [3](#), [40](#), [64](#), [167](#), [226](#), [290](#), [319](#), [348](#), [424](#)

## **Grade Point Average (GPA) [213](#), [287](#), [341](#)**

### **groupby**

apply, using with [336-341](#)

complicated aggregation functions, using with [330-336](#)

used, for modifying unit of analysis of DataFrame [341-344](#)

used, for organizing data by groups [326-329](#)

user-defined functions, using with [336-341](#)

### **grouped boxplots**

used, for uncovering unexpected values in particular group  
[179-185](#)

## **H**

### **hashing trick [300](#)**

### **heat map**

generating, based on correlation matrix [204-208](#)

### **high cardinality [294](#)**

used, for encoding categorical features [300-303](#)

### **histograms**

used, for examining distribution of continuous variables [166-173](#)

### **hyperparameter [276](#)**

## **I**

### **imputation**

k-Nearest Neighbors (KNN), using for [273-276](#)

PandasAI, using for [280-284](#)

random forest, using for [276-279](#)

**independent/predictor variables** [286](#)

**interquartile range (IQR)** [173](#)

## **Isolation Forest**

used, for finding anomalies [152-156](#)

## **J**

**JSON data**

persisting [68-71](#)

serializing, reasons [68](#)

**Jupyter Notebook** [2](#)

## **K**

**k-means binning** [311-313](#)

**k-Nearest Neighbors (KNN)**

used, for finding outliers [148-151](#)

used, for imputation [273-276](#)

**KNN imputation**

limitations [276](#)

## **L**

**lambda function** [230](#)

**linear regression**

used, for identifying data points with significant influence [144-147](#)

## **line plots**

used, for examining trends in continuous variables [198-203](#)

# **M**

## **many-to-many relationships**

fixing [386-392](#)

## **mathematical transformations**

using [303-308](#)

## **Matplotlib [103](#)**

## **medium cardinality**

used, for encoding categorical features [300-303](#)

## **melt**

using, to reshape data from wide to long format [393-398](#)

## **merge routine**

developing [376-378](#)

## **merges**

many-to-many merges, performing [370-375](#)

one-to-many merges, performing [365-370](#)

one-to-one merges, performing [355-361](#)

one-to-one merges, performing by multiple columns [361-364](#)

## **miceforest [279](#)**

imputation results, viewing [280](#)

**Microsoft SQL Server** [18](#)

**min-max scaling** [313](#)

**missing values**

cleaning [260-267](#)

identifying [256-260](#)

**multiple groups of columns**

melting [398-400](#)

**multiple imputations by chained equations (MICE)** [279](#)

**MySQL** [18](#)

## N

**National Longitudinal Surveys (NLS)** [26](#), [78-83](#), [213](#), [256](#), [286](#),  
[330](#), [356](#), [394](#), [406](#)

URL [79](#)

**NLS Investigator**

URL [26](#)

**NumPy arrays**

used, for calculating summaries by group [323-325](#)

## O

**one-hot encoding** [294-297](#)

**OpenAI**

using, for Series operations [248-253](#)

**ordinal encoding** [297-300](#)

**Organisation for Economic Co-operation and Development**

[URL](#) [11](#)

## **Our World in Data, COVID-19 cases**

reference link [58](#), [79](#), [119](#), [167](#), [241](#), [256](#), [300](#), [319](#), [382](#), [417](#)

## **outliers**

finding, with k-nearest neighbors (KNN) [148-151](#)

identifying, in bivariate relationships [128-135](#)

identifying, with PandasAI [156-161](#)

identifying, with single variable [118-128](#)

## **outliers for continuous variables**

identifying, with boxplots [173-179](#)

# **P**

## **PandasAI**

used, for identifying outliers [156-161](#)

using, for imputation [280-284](#)

## **PandasAI library** [107](#)

## **PandasAI SmartDataframe object** [107](#)

## **pandas Series**

summary statistics, displaying for [217-221](#)

values, obtaining from [212-217](#)

## **pivot**

using, to reshape data from long to wide format [401-404](#)

## **pivot\_table**

used, for modifying unit of analysis of DataFrame [344-346](#)

## **Python Dictionary Comprehension Tutorial**

reference link [46](#)

## **Python Outlier Detection (PyOD) [148](#)**

reference link [152](#)

# **R**

## **random forest**

using, for imputation [276-279](#)

## **R data**

importing [34-38](#)

## **redundant/unhelpful features**

removing [289-294](#)

## **regression**

used, for imputing values [267-273](#)

## **rows**

selecting [91-98](#)

# **S**

## **SAS data**

importing [25-33](#)

## **scatter plots**

used, for viewing bivariate relationships [191-197](#)

## **Scholastic Assessment Test (SAT) [174, 287, 332](#)**

## **select\_dtypes [89](#)**

## **Series operations**

OpenAI, using for [248-253](#)

## **Series values**

modifying [221-225](#)

modifying, conditionally [225-232](#)

## **simple JavaScript Object Notation (JSON) data**

importing [46-52](#)

## **Spark data**

working with [63-67](#)

## **Spark, using with Python**

reference link [63](#)

## **SPSS data**

importing [25-33](#)

## **SQL databases**

data, importing from [18-25](#)

## **stack**

using, to reshape data from wide to long format [393-398](#)

## **standard/z-score scaling [313](#)**

## **Stata data**

importing [25-33](#)

## **string Series data**

cleaning [233-240](#)

evaluating [233-240](#)

## **subsetting**

used, for examining logical inconsistencies in variable relationships [136-144](#)

## **summary statistics**

displaying, for pandas Series [217-221](#)

generating, for continuous variables [102-106](#)

# **T**

## **tabular data**

persisting [39-42](#)

## **target variable** [289](#)

## **training datasets**

creating [286-289](#)

## **trends in continuous variables**

examining, with line plots [198-203](#)

# **U**

## **unexpected values**

identifying, in bivariate relationships [128-135](#)

## **unexpected values in particular group**

uncovering, with grouped boxplots [179-185](#)

## **union** [356](#)

## **unit of analysis**

data duplication, reasons [382](#)

## **unstack**

using, to reshape data from long to wide format [401-404](#)

## **user-defined functions**

using, with groupby [336-341](#)

## **V**

### **values**

imputing, with regression [267-273](#)

obtaining, from pandas Series [212-217](#)

### **variable relationships**

subsetting, used for examining logical inconsistencies [136-144](#)

### **View API keys** [107](#)

### **violin plots**

used, for examining distribution shape and outliers [186-190](#)

## **W**

### **web pages**

data, importing from [58-63](#)

### **web scraping** [58](#)

# Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781803239873>

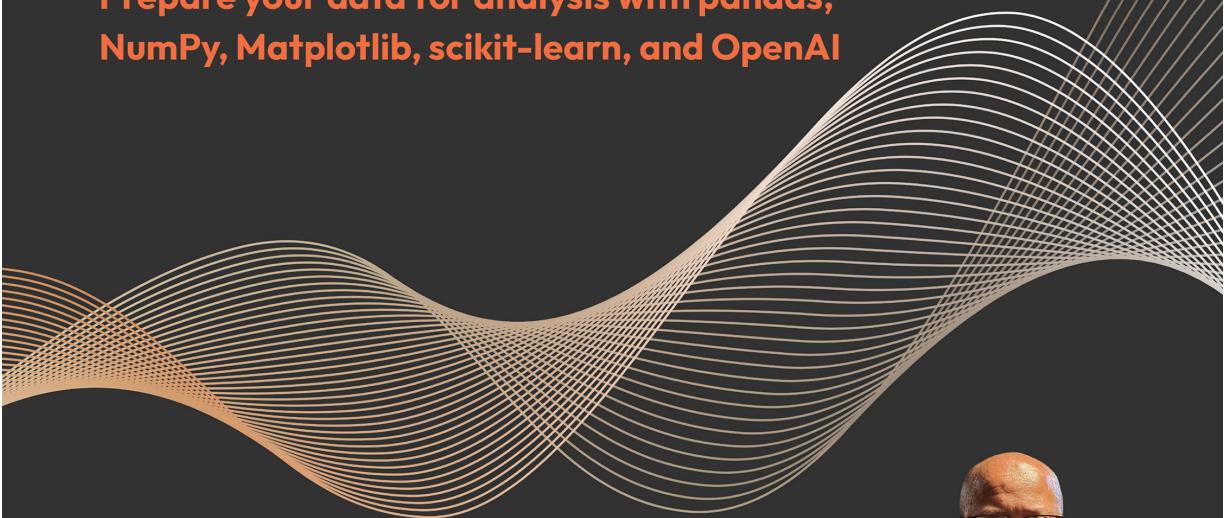
2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.

EXPERT INSIGHT

---

# Python Data Cleaning Cookbook

**Prepare your data for analysis with pandas,  
NumPy, Matplotlib, scikit-learn, and OpenAI**



**Second Edition**

---



**Michael Walker**

**<packt>**