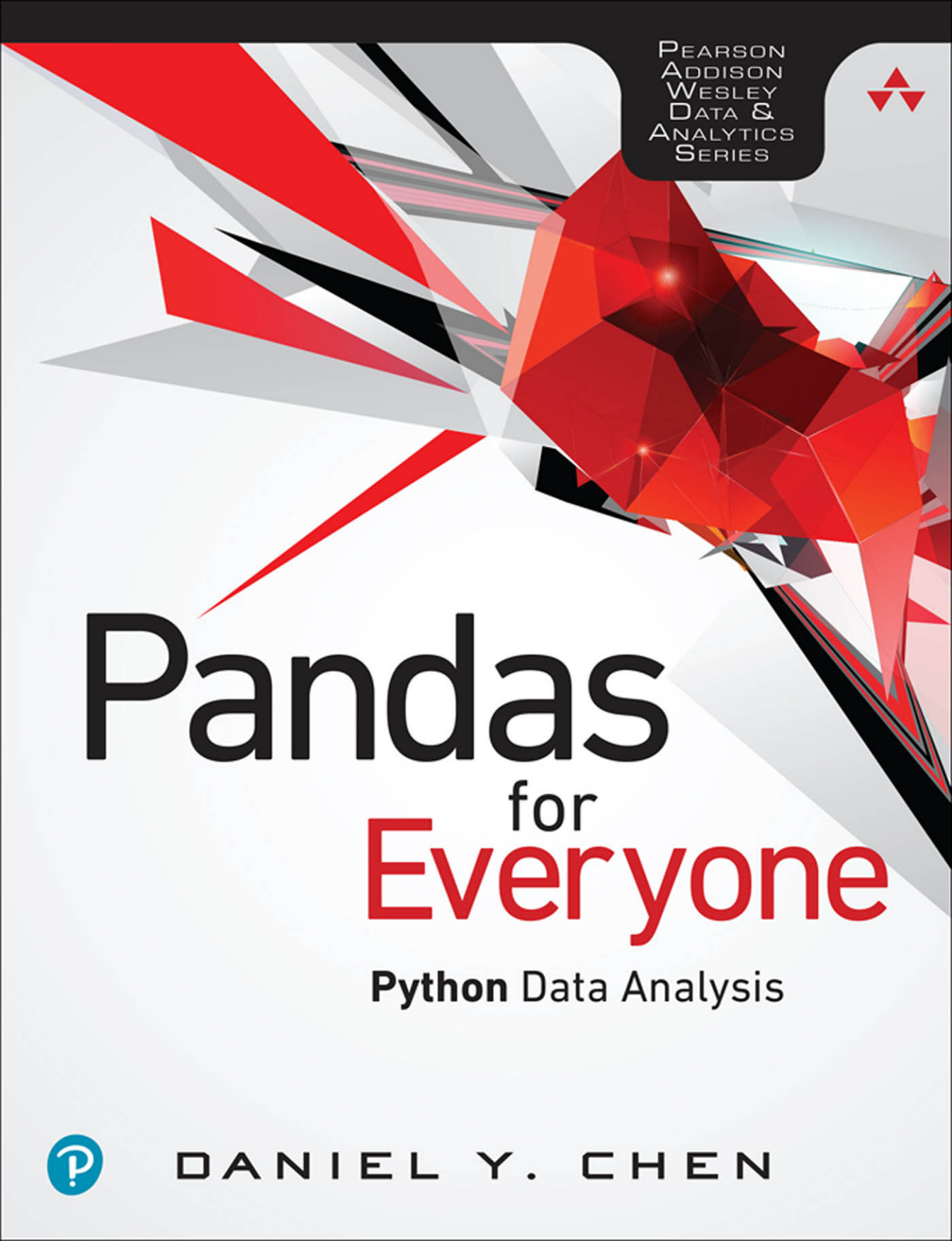
Pearson
Addison
Wesley
Data &
Analytics
Series
Pandas
for
Everyone
Python Data Analysis
Daniel Y. Chen

Pandas for Everyone

Part II

Data Manipulation

This page intentionally left blank

4

Data Assembly

4.1 Introduction

By now, you should be able to load data into `Pandas` and do some basic visualizations. This part of the book focuses on various data cleaning tasks. We begin with assembling a data set for analysis by combining various data sets together.

Concept Map

1. Prior knowledge
    a. loading data
    b. subsetting data
    c. functions and class methods

Objectives

This chapter will cover:

1. Tidy data
2. Concatenating data
3. Merging data sets

4.2 Tidy Data

Hadley Wickham,[1] one of the more prominent members of the R community, talks about the idea of *tidy* data. In fact, he's written a paper about this concept in the *Journal of Statistical Software*.[2] Tidy data is a framework to structure data sets so they can be easily analyzed. It is mainly used as a goal one should aim for when cleaning data. Once you understand what tidy data is, that knowledge will make data collection much easier.

1. Hadley Wickham's homepage: `http://hadley.nz`
2. Tidy data paper: `http://vita.had.co.nz/papers/tidy-data.pdf`

So what is *tidy* data? Hadley Wickham's paper defines it as meeting the following criteria:

- Each row is an observation.
- Each column is a variable.
- Each type of observational unit forms a table.

4.2.1 Combining Data Sets

We begin with Hadley Wickham's last tidy data point: "Each type of observational unit forms a table." When data is tidy, you need to combine various tables together to answer a question. For example, there may be a separate table holding company information and another table holding stock prices. If we want to look at all the stock prices within the tech industry, we may first have to find all the tech companies from the company information table, and then combine that data with the stock price data to get the data we need for our question. The data may have been split up into separate tables to reduce the amount of redundant information (we don't need to store the company information with each stock price entry), but this arrangement means we as data analysts must combine the relevant data ourselves to answer our question.

At other times, a single data set may be split into multiple parts. For example, with time-series data, each date may be in a separate file. In another case, a file may have been split into parts to make the individual files smaller. You may also need to combine data from multiple sources to answer a question (e.g., combine latitudes and longitudes with zip codes). In both cases, you will need to combine data into a single dataframe for analysis.

4.3 Concatenation

One of the (conceptually) easier ways to combine data is with concatenation. Concatenation can be thought of appending a row or column to your data. This approach is possible if your data was split into parts or if you performed a calculation that you want to append to your existing data set.

Concatenation is accomplished by using the `concat` function from Pandas.

4.3.1 Adding Rows

Let's begin with some example data sets so you can see what is actually happening.

```
import pandas as pd

df1 = pd.read_csv('../data/concat_1.csv')
df2 = pd.read_csv('../data/concat_2.csv')
df3 = pd.read_csv('../data/concat_3.csv')

print(df1)
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

```
print(df2)
```

```
    A   B   C   D
0  a4  b4  c4  d4
1  a5  b5  c5  d5
2  a6  b6  c6  d6
3  a7  b7  c7  d7
```

```
print(df3)
```

```
     A    B    C    D
0   a8   b8   c8   d8
1   a9   b9   c9   d9
2  a10  b10  c10  d10
3  a11  b11  c11  d11
```

Stacking the dataframes on top of each other uses the `concat` function in `Pandas`. All of the dataframes to be concatenated are passed in a `list`.

```
row_concat = pd.concat([df1, df2, df3])
print(row_concat)
```

```
     A    B    C    D
0   a0   b0   c0   d0
1   a1   b1   c1   d1
2   a2   b2   c2   d2
3   a3   b3   c3   d3
0   a4   b4   c4   d4
1   a5   b5   c5   d5
2   a6   b6   c6   d6
3   a7   b7   c7   d7
0   a8   b8   c8   d8
1   a9   b9   c9   d9
2  a10  b10  c10  d10
3  a11  b11  c11  d11
```

As you can see, `concat` blindly stacks the dataframes together. If you look at the row names (i.e., the row indices), they are also simply a stacked version of the original row indices. If we apply the various subsetting methods from Table 2.3, the table will be subsetted as expected.

```
# subset the fourth row of the concatenated dataframe
print(row_concat.iloc[3, ])
```

```
A    a3
B    b3
C    c3
D    d3
Name: 3, dtype: object
```

Question

What happens when you use `loc` or `ix` to subset the new dataframe?

Section 2.2.1 showed the process for creating a `Series`. However, if we create a new series to append to a dataframe, it does not append correctly.

```
# create a new row of data
new_row_series = pd.Series(['n1', 'n2', 'n3', 'n4'])
print(new_row_series)
```

```
0    n1
1    n2
2    n3
3    n4
dtype: object
```

```
# attempt to add the new row to a dataframe
print(pd.concat([df1, new_row_series]))
```

```
     A    B    C    D    0
0   a0   b0   c0   d0  NaN
1   a1   b1   c1   d1  NaN
2   a2   b2   c2   d2  NaN
3   a3   b3   c3   d3  NaN
0  NaN  NaN  NaN  NaN   n1
1  NaN  NaN  NaN  NaN   n2
2  NaN  NaN  NaN  NaN   n3
3  NaN  NaN  NaN  NaN   n4
```

The first things you may notice are the `NaN` values. This is simply Python's way of representing a "missing value" (see Chapter 5, "Missing Data"). We were hoping to append our new values as a row, but that didn't happen. In fact, not only did our code not append the values as a row, but it also created a new column completely misaligned with everything else.

If we pause to think about what is happening here, we can see that the results actually make sense. First, if we look at the new indices that were added, we notice that they are very similar to the results we obtained when we concatenated dataframes earlier. The indices of the `new_row series` object are analogs to the row numbers of the dataframe. Also, since our series did not have a matching column, our `new_row` was added to a new column.

To fix this problem, we can turn our series into a dataframe. This data frame contains one row of data, and the column names are the ones the data will bind to.

```
                          # note the double brackets
new_row_df = pd.DataFrame([['n1', 'n2', 'n3', 'n4']],
                          columns=['A', 'B', 'C', 'D'])
print(new_row_df)
```

```
    A   B   C   D
0  n1  n2  n3  n4
```