

7

Exploring, Cleaning, Refining, and Blending Datasets

In the previous chapter, we learned about the power of data visualizations, and the importance of having good-quality, consistent data defined with dimensions and measures.

Now that we understand *why* that's important, we are going to focus on the *how* throughout this chapter by working hands-on with data. Most of the examples provided so far included data that was already *prepped* (prepared) ahead of time for easier consumption. We are now switching gears by learning the skills that are necessary to be comfortable working with data to increase your data literacy.

A key concept of this chapter is cleaning, filtering, and refining data. In many cases, the reason why you need to perform these actions is the source data does not provide high-quality analytics *as is*. Throughout my career, high-quality data is not the norm and data gaps are common. As good data analysts, we need to work with what we have available. We will cover some techniques to enrich the quality of the data so you can provide quality insights and answer questions from the data even when the source does not include all of the information required.

In my experience, highlighting the poor quality of the source data is the insight because not enough transparency exists and key stakeholders are unaware of the challenges of using the data. The bottom line is poor quality should not stop you from proceeding with working with data. My goal is to demonstrate a repeatable technique and workflow to improve data quality for analysis.

We will cover the following topics in this chapter:

- Retrieving, viewing, and storing tabular data
- Learning how to restrict, sort, and sift through data
- Cleaning, refining, and purifying data using Python
- Combining and binning data

Technical requirements

Here's the GitHub repository of this book: <https://github.com/PacktPublishing/Practical-Data-Analysis-using-Jupyter-Notebook/tree/master/Chapter07>.

You can download and install the required software from the following link: <https://www.anaconda.com/products/individual>.

Retrieving, viewing, and storing tabular data

The ability to retrieve and view tabular data has been covered multiple times in prior chapters; however, those examples were focused on the perspective of the consumer. We learned the skills necessary to understand what structured data is in, the many different forms it can take, and how to answer some questions from data. Our data literacy has increased during this time but we have relied on the producers of data sources to make it easier to read using a few Python commands or SQL commands. In this chapter, we are switching gears from being exclusively a **consumer** to now a **producer** of data by learning skills to manipulate data for analysis.

As a good data analyst, you will need both sides of the consumer and producer spectrum of skills to solve more complicated questions with data. For example, a common measure requested by businesses with web or mobile users is called **usage analytics**. This means counting the number of users over snapshots of time, such as by day, week, month, and year. More importantly, you want to better understand whether those users are new, returning, or lost.

Common questions related to usage analytics are as follows:

- How many new users have hit the website this day, week, or month?
- How many returning users have accessed the website this day, week, or month?
- How many users have we lost (inactive for more than 60 days) this week, month, or year?

To answer these types of questions, your data source must have, at a minimum, `timestamp` and unique `user_id` fields available. In many cases, this data will have high volume and velocity, so analyzing this information will require the right combination of people, processes, and technology, which I have had the pleasure of working with. Data engineering teams build out ingestion pipelines to make this data accessible for reporting and analytics.

You may need to work with the data engineering team to apply the business rules and summary levels (also known as aggregates) to the data that include additional fields required to answer the user analytics questions. For our examples, I have provided a much smaller sample of data and we are going to derive new data fields from the existing source data file provided.

I find the best way to learn is to walk through the steps together, so let's create a new Jupyter notebook named `user_churn_prep`. We will begin with retrieving data from SQL against a database and loading it into a `DataFrame`, similar to the steps outlined in [Chapter 5, *Gathering and Loading Data in Python*](#). To keep it simple, we are using another SQLite database to retrieve the source data.



If you would like more details about connecting to SQL data sources, please refer to [Chapter 5, *Gathering and Loading Data in Python*](#).

Retrieving

To create a connection and use SQLite, we have to import a new library using the code. For this example, I have provided the database file named `user_hits.db`, so be sure to download it from my GitHub repository beforehand:

1. To load a SQLite database connection, you just need to add the following command in your Jupyter notebook and run the cell. I have placed a copy on GitHub for reference:

```
In[]: import sqlite3
```

2. Next, we need to assign a connection to a variable named `conn` and point to the location of the database file, which is named `user_hits.db`:

```
In[]: conn = sqlite3.connect('user_hits.db')
```



Be sure that you have copied the `user_hits.db` file to the correct Jupyter folder directory to avoid errors with the connection.

3. Import the `pandas` library so you can create a `DataFrame`:

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

4. Run a SQL statement and assign the results to a `DataFrame`:

```
In[]: df_user_churn = pd.read_sql_query("SELECT * FROM  
tbl_user_hits;", conn)
```

5. Now that we have the results in a `DataFrame`, we can use all of the available `pandas` library commands against this data without going back to the database. Your code should look similar to the following screenshot:

```
In [8]: import sqlite3  
  
In [9]: conn = sqlite3.connect('user_hits.db')  
  
In [10]: import pandas as pd  
  
In [11]: df_user_churn = pd.read_sql_query("SELECT * FROM tbl_user_hits;", conn)
```

Viewing

Perform the following steps to view the results of the retrieved data:

1. To view the results, we can just run the `head()` command against this `DataFrame` using this code:

```
In[]: df_user_churn.head()
```

The output will look like the following table, where the `tbl_user_hits` table has been loaded into a `DataFrame` with a labeled header row with the index column to the left starting with a value of 0:

```
In [7]: df_user_churn.head()
```

Out[7]:

	userid	date
0	1	1/1/2017
1	2	1/2/2017
2	3	1/3/2017
3	4	1/1/2018
4	5	1/2/2018

Before we move on to the next step, let's verify the data we loaded with a few metadata commands.

2. Type in `df_user_churn.info()` in the next In[] : cell and run the cell:

```
In[]: df_user_churn.info()
```

Verify that the output cell displays Out []. There will be multiple rows, including data types for all columns, similar to the following screenshot:

```
In [13]: df_user_churn.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9 entries, 0 to 8
Data columns (total 2 columns):
userid    9 non-null int64
date      9 non-null object
dtypes: int64(1), object(1)
memory usage: 224.0+ bytes
```



Storing

Now that we have the data available to work with as a DataFrame in Jupyter, let's run a few commands to store it as a file for reference. Storing data as a snapshot for analysis is a useful technique to learn, and while our example is simplistic, the concept will help in future data analysis projects.

To store your DataFrame into a CSV file, you just have to run the following command:

```
In[]: df_user_churn.to_csv('user_hits_export.csv')
```

The results will look similar to the following screenshot, where a new CSV file is created in the same project folder as your current Jupyter notebook. Based on the OS you are using on your workstation, the results will vary:

Name	Date modified	Type	Size
 ch_07_retrieve_sql_and_create_dataframe.ipynb	2/20/2020 4:39 PM	IPYNB File	4 KB
 user_hits_export.csv	2/20/2020 4:38 PM	Microsoft Excel Comma Separated Values File	1 KB



There are other formats you can export your DataFrame to, including Excel. You should also note the file path from which you are exporting the data file. Check out the *Further reading* section for more information.

Learning how to restrict, sort, and sift through data

Now that we have the data available in a DataFrame, we can walk through how to restrict, sort, and sift through data with a few Python commands. The concepts we are going to walk through using pandas are also common using SQL, so I will also include the equivalent SQL commands for reference.

Restricting

The concept of restricting data, which is also known as filtering data, is all about isolating one or more records based on conditions. Simple examples are when you are only retrieving results based on matching a specific field and value. For example, you only want to see results for one user or a specific point in time. Other requirements for restricting data can be more complicated, including explicit conditions that require elaborate logic, business rules, and multiple steps. I will not be covering complex examples that require complex logic but will add some references in the *Further reading* section. However, the concepts covered will teach you essential skills to satisfy many common use cases.

For our first example, let's isolate one specific user from our DataFrame. Using pandas commands, that is pretty easy, so let's start up a new Jupyter notebook named `user_churn_restricting`:

1. Import the pandas library so you can create a DataFrame:

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

2. Create a new DataFrame by loading the data from the CSV file we created in the prior example:

```
In[]: df_user_churn = pd.read_csv('user_hits_export.csv');
```



The file path and filename must be the same as those you used in the prior example.

Now that we have all user data loaded into a single DataFrame, we can easily reference the source dataset to restrict results. It is a best practice to keep this source DataFrame intact so you can reference it for other purposes and analysis. It is also common during analysis to need to make adjustments based on changing requirements, or that you will only gain insights by making adjustments.

In my career, I follow a common practice of *you don't know what you don't know* while working with data, so having the flexibility to easily reference the source data without undoing your changes is important. This is commonly known as snapshotting your analysis and having the ability to roll back changes as needed.



When working with big data sources where the sources are larger than a billion rows, snapshots will require a large number of resources where RAM and CPU will be impacted. You may be required to snapshot incrementally for a specific date or create a rolling window of time to limit the amount of data you can work with at one time.

To restrict our data to a specific user, we will be creating a new DataFrame from the source DataFrame. That way, if we need to make adjustments to the filters used to create the new DataFrame, we don't have to rerun all of the steps from the beginning.

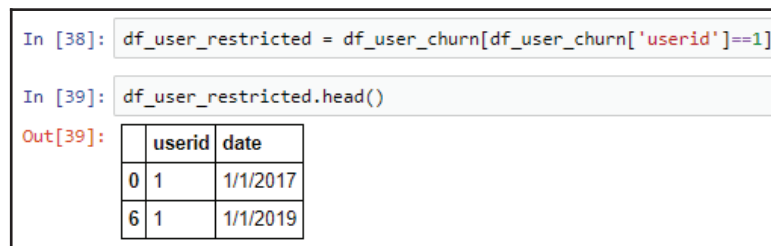
3. Create a new DataFrame by loading the data from the source DataFrame. The syntax is nested so you are actually calling the same `df_user_churn` DataFrame within itself and filtering results only for the explicit value where `userid` is equal to 1:

```
In[]: df_user_restricted =  
df_user_churn[df_user_churn['userid']==1]
```

4. To view and verify the results, you can run a simple `head()` command:

```
In[]: df_user_restricted.head()
```

The results will look similar to the following screenshot, where the only two rows in the new `df_user_restricted` DataFrame have a value where `userid` is 1:



```
In [38]: df_user_restricted = df_user_churn[df_user_churn['userid']==1]  
  
In [39]: df_user_restricted.head()  
Out[39]:
```

	userid	date
0	1	1/1/2017
6	1	1/1/2019

Restricting data helps to isolate records for specific types of analysis to help to answer additional questions. In the next step, we can start answering questions related to usage patterns.

Sorting

Now that we have isolated a specific user by creating a new DataFrame, which is now available for reference, we can enhance our analysis by asking questions such as the following:

- When did a specific user start using our website?
- How frequently does this user access our website?
- When was the last time this user accessed our website?

All of these questions can be answered with a few simple Python commands focused on sorting commands. Sorting data is a skill that computer programmers of any programming language are familiar with. It's easily done with SQL by adding an `order by` command. Many third-party software, such as Microsoft Excel, Google Sheets, or Qlik Sense, has a sorting feature built in. The concept of sorting data is well known, so I will not go into a detailed definition; rather, I will focus on important features and best practices when performing data analysis.

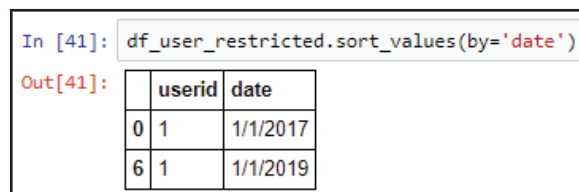
With structured data, sorting is commonly understood to be row-level by specific columns, which will be defined by ordering the sequence of the values from either low to high or high to low. The default is low to high unless you explicitly change it. If the values have a data type that is numeric, such as integer or float, the sort order sequence will be easy to identify. For textual data, the values are sorted alphabetically, and, depending on the technology used, mixed case text will be handled differently. In Python and pandas, we have specific functions available along with parameters to handle many different use cases and needs.

Let's start answering some of the questions we outlined previously using the `sort()` function:

1. To answer the question *When did a specific user start using our website?*, we just need to run the following command:

```
In[]: df_user_restricted.sort_values(by='date')
```

The results will look similar to the following screenshot, where the results are sorted in ascending order by the `date` field:



The screenshot shows a Jupyter Notebook interface. The input cell contains the command `df_user_restricted.sort_values(by='date')`. The output cell displays a DataFrame with two columns, `userid` and `date`, and two rows of data. The first row has `userid` 1 and `date` 1/1/2017. The second row has `userid` 1 and `date` 1/1/2019. The index values 0 and 6 are visible on the left of the rows.

	userid	date
0	1	1/1/2017
6	1	1/1/2019

2. To answer the question *"When is the last time this user accessed our website?"*, we just need to run the following command:

```
In[]: df_user_restricted.sort_values(by='date', ascending=False)
```

The results will look similar to the following screenshot, where the same records are displayed as the previous one; however, the values are sorted in descending order by last date available for this specific `userid`:

```
In [42]: df_user_restricted.sort_values(by='date', ascending=False)
```

```
Out[42]:
```

	userid	date
6	1	1/1/2019
0	1	1/1/2017

Sifting

The concept of sifting through data means we are isolating specific columns and/or rows from a dataset based on one or more conditions. There are nuanced differences between sifting versus restricting, so I would distinguish sifting as the need to include additional business rules or conditions applied to a population of data to isolate a subset of that data. Sifting data usually requires creating new derived columns from the source data to answer more complex questions. For our next example, a good question about usage would be: *Do the same users who hit our website on Monday also return during the same week?*

To answer this question, we need to isolate the usage patterns for a specific day of the week. This process requires a few steps, which we will outline together from the original DataFrame we created previously:

1. Create a new DataFrame by loading the data from the source file:

```
In[]: df_user_churn_cleaned =  
pd.read_csv('user_hits_binning_import.csv', parse_dates=['date'])
```

Next, we need to extend the DataFrame by adding new derived columns to help to make the analysis easier. Since we have a `Timestamp` field available, the pandas library has some very useful functions available to help the process. Standard SQL has built-in features as well and will vary depending on which RDMS is used, so you will have to reference the date/time functions available. For example, a Postgres database uses the syntax of `select to_char(current_date, 'Day')`; to convert a date field into the current day of the week.

2. Import a new `datetime` library for easy reference to date and time functions:

```
In[]: import datetime
```

3. Assign a variable to the current `datetime` for easier calculation of the age from today:

```
In[]: now = pd.to_datetime('now')
```

4. Add a new derived column called `age` that is calculated from the current date minus the date value per user:

```
In[]: df_user_churn_cleaned['age'] = now -  
df_user_churn_cleaned['date']
```



If you receive `datetime` errors in your notebook, you may need to upgrade your `pandas` library.

Cleaning, refining, and purifying data using Python

Data quality is highly important for any data analysis and analytics. In many cases, you will not understand how good or bad the data quality is until you start working with it. I would define good-quality data as information that is well structured, defined, and consistent, where almost all of the values in each field are defined as expected. In my experience, data warehouses will have high-quality data because it has been reported on across the organization. In my experience, bad data quality occurs where a lack of transparency exists against the data source. Bad data quality examples are a lack of conformity and inconsistency in the expected data type or any consistent pattern of values in delimited datasets. To help to solve these data quality issues, you can begin to understand your data with the concepts and questions we covered in [Chapter 1, *Fundamentals of Data Analysis*](#), with **Know Your Data (KYD)**. Since the quality of data will vary by source, some specific questions you can ask to understand data quality are as follows:

- Is the data structured or unstructured?
- Does the data lineage trace back to a system or application?
- Does the data get transformed and stored in a warehouse?
- Does the data have a schema with each field having a defined data type?
- Do you have a data dictionary available with business rules documented?

Receiving answers to these questions ahead of time would be a luxury; uncovering them as you go is more common for a data analyst. During this process, you will still find the need to clean, refine, and purify your data for analysis purposes. How much time you need to spend will vary on many different factors, and the true cost of quality will be the time and effort required to improve data quality.

Cleaning data can take on many different forms and has been a common practice for decades for data engineers and analytic practitioners. There are many different technologies and skillsets required for enterprise and big data cleansing. Data cleaning is an industry within **Information Technology (IT)** because good-quality data is worth the price of outsourcing.

A common definition of data cleansing is the process of removing or resolving poor-quality data records from the source, which can vary based on the technology used to persist the data, such as a database table or encoded file. Poor-quality data can be identified as any data that does not match the producers' intended and defined requirements. This can include the following:

- Missing or null (NaN) values from the fields of one or more rows
- Orphan records where the primary or foreign keys cannot be found in any referenced source tables
- Corrupted records where one or more records cannot be read by any reporting or analysis technology

For our example, let's look at our usage data again and see whether we can find any issues by profiling it to see whether we can find any anomalies:

1. Import the CSV file and run the `info()` command to confirm the data types and row counts and profile the DataFrame for more information:

```
In[: df_usage_patterns = pd.read_csv('user_hits_import.csv')
df_usage_patterns.info()
```

The results will look similar to the following screenshot, where metadata of the DataFrame is presented:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12 entries, 0 to 11
Data columns (total 2 columns):
userid    9 non-null float64
date      12 non-null object
dtypes: float64(1), object(1)
memory usage: 272.0+ bytes
```

One anomaly that is uncovered is that the number of values is different between the two fields. For `userid`, there are 9 non-null values and for the `date` field, there are 12 non-null values. For this dataset, we expect each row to have one value for both fields, but this command is telling us there are missing values. Let's run another command to identify which index/row has the missing data.

2. Run the `isnull()` command to confirm the data types and row counts and profile the DataFrame for more information:

```
In[]: pd.isnull(df_usage_patterns)
```

The results will look similar to the following table, where a list of `True` and `False` values is displayed by row and column:

Out[26]:		
	userid	date
0	False	False
1	False	False
2	False	False
3	False	False
4	False	False
5	False	False
6	False	False
7	False	False
8	False	False
9	True	False
10	True	False
11	True	False

The record count looks okay but notice that there are null values (NaN) that exist in the `userid` field. A unique identifier for each row to help us to identify each user is critical for accurate analysis of this data. The reason why `userid` is blank would have to be explained by the producer of this data and may require additional engineering resources to help to investigate and troubleshoot the root cause of the issue. In some cases, it may be a simple technical hiccup during data source creation that requires a minor code change and reprocessing.



I always recommend cleaning data as close to the source as possible, which saves time by avoiding reworking by other data analysts or reporting systems.

Having nulls included in our analysis will impact our summary statistics and metrics. For example, the count of the average daily users would be lower on the dates where the null values exist. For the user churn analysis, the measure of the frequency of reporting users would be skewed because the NaN values could be one of the returning `user_ids` or a new user.

With any high volume transaction-based system, there could be a margin of error that you may need to account for. As a good data analyst, ask the question, *what is the cost of quality and of being one hundred percent accurate?* If the price is too high due to the time and resources required to change it, a good alternative is to exclude and isolate the missing data so it can be investigated later.



Note that if you end up adding isolated data back into your analysis, you will have to restate results and inform any consumers of the change to your metrics.

Let's walk through an example of how to isolate and exclude any missing data by identifying the NaN records and creating a new DataFrame that has them removed:

1. Create a new DataFrame by loading the data from the source DataFrame, except we will exclude the null values by adding the `dropna()` command:

```
In[]: df_user_churn_cleaned = df_usage_patterns.dropna()
```

2. To view and verify the results, you can run a simple `head()` command and confirm the NaN/null values have been removed:

```
In[]: df_user_churn_cleaned.head(10)
```

The results will look similar to the following table, where the new DataFrame has complete records with no missing values in either `userid` or `date`:

Out[11]:

	userid	date
0	1.0	1/1/2017
1	2.0	1/2/2017
2	3.0	1/3/2017
3	4.0	1/1/2018
4	5.0	1/2/2018
5	6.0	1/3/2018
6	1.0	1/1/2019
7	3.0	1/2/2019
8	6.0	1/3/2019

Combining and binning data

Combining multiple data sources is sometimes necessary for multiple reasons, which include the following:

- The source data is broken up into many different files with the same defined schema (tables and field names), but the number of rows will vary slightly. A common reason is for storage purposes, where it is easier to maintain multiple smaller file sizes versus one large file.
- The data is partitioned where one field is used to break apart the data for faster response time reading or writing to the source data. For example, HIVE/HDFS recommends storing data by a single date value so you can easily identify when it was processed and quickly extract data for a specific day.
- Historical data is stored in a different technology than more current data. For example, the engineering team changed the technology being used to manage the source data and it was decided not to import historical data beyond a specific date.

For any of the reasons defined here, combining data is a common practice in data analysis. I would define the process of combining data as when you are layering two or more data sources into one where the same fields/columns from all sources align. In SQL, this would be known as `UNION ALL` and in `pandas`, we use the `concat()` function to bring all of the data together.

A good visual example of how data is combined is in the following screenshot, where multiple source files are named `user_data_YYYY.csv` and each year is defined as `YYYY`. These three files, which all have the same field names of `userid`, `date`, and `year`, are imported into one SQL table named `tbl_user_data_stage`, which is shown in the following screenshot. The target table that stores this information also includes a new field named `filesource` so the data lineage is more transparent to both the producer and the consumer:

user_data_2017.csv		
userid	date	year
1	1/1/2017	2017
2	1/2/2017	2017
3	1/3/2017	2017

user_data_2018.csv		
userid	date	year
4	1/1/2018	2018
5	1/2/2018	2018
6	1/3/2018	2018

user_data_2019.csv		
userid	date	year
1	1/1/2019	2019
3	1/2/2019	2019
6	1/3/2019	2019

Once the data has been processed and persisted into a table named `tbl_user_data_stage`, all of the records from the three files are preserved as displayed in the following table. In this example, any duplicates would be preserved between what existed in the source files and the target table:

tbl_user_data_stage			
userid	date	year	filesource
1	1/1/2017	2017	user_data_2017.csv
2	1/2/2017	2017	user_data_2017.csv
3	1/3/2017	2017	user_data_2017.csv
4	1/1/2018	2018	user_data_2018.csv
5	1/2/2018	2018	user_data_2018.csv
6	1/3/2018	2018	user_data_2018.csv
1	1/1/2019	2019	user_data_2019.csv
3	1/2/2019	2019	user_data_2019.csv
6	1/3/2019	2019	user_data_2019.csv



One of the reasons data engineering teams create `stage` tables is to help to build data ingestion pipelines and create business rules where duplicate records are removed.

To recreate the example in Jupyter, let's create a new notebook and name it `ch_07_combining_data`. There are more efficient ways to import multiple files but, in our example, we will import each one in separate DataFrames and then combine them into one:

1. Import the pandas library:

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



You will also need to copy the three CSV files to your local folder.

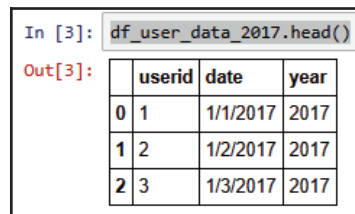
2. Import the first CSV file named `user_data_2017.csv`:

```
In[]: df_user_data_2017 = pd.read_csv('user_data_2017.csv')
```

3. Run the `head()` command to verify the results:

```
In[]: df_user_data_2017.head()
```

The results will look similar to the following screenshot, where the rows are displayed with a header row and index added starting with a value of 0:



The screenshot shows a Jupyter Notebook cell with the input `In [3]: df_user_data_2017.head()` and the output `Out[3]:`. The output is a table with 4 columns: `userid`, `date`, and `year`. The first column is an index starting from 0. The data shows three rows of user data for the year 2017.

	userid	date	year
0	1	1/1/2017	2017
1	2	1/2/2017	2017
2	3	1/3/2017	2017

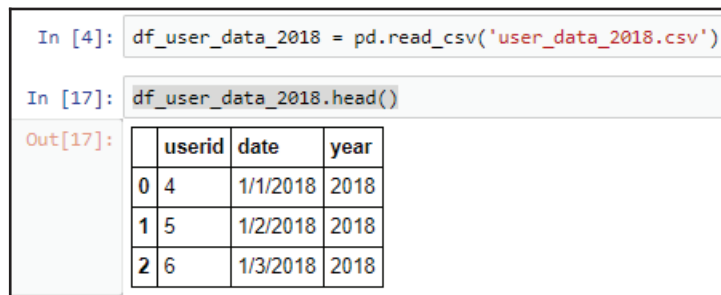
4. Repeat the process for the next CSV file, which is named `user_data_2018.csv`:

```
In[]: df_user_data_2018 = pd.read_csv('user_data_2018.csv')
```

5. Run the `head()` command to verify the results:

```
In[]: df_user_data_2018.head()
```

The results will look similar to the following screenshot, where the rows are displayed with a header row and index added starting with a value of 0:



The screenshot shows two Jupyter Notebook cells. The first cell has the input `In [4]: df_user_data_2018 = pd.read_csv('user_data_2018.csv')`. The second cell has the input `In [17]: df_user_data_2018.head()` and the output `Out[17]:`. The output is a table with 4 columns: `userid`, `date`, and `year`. The first column is an index starting from 0. The data shows three rows of user data for the year 2018.

	userid	date	year
0	4	1/1/2018	2018
1	5	1/2/2018	2018
2	6	1/3/2018	2018

6. Repeat the process for the next CSV file, which is named `user_data_2019.csv`:

```
In[]: df_user_data_2019 = pd.read_csv('user_data_2019.csv')
```

7. Run the `head()` command to verify the results:

```
In[]: df_user_data_2019.head()
```

The results will look similar to the following screenshot, where the rows are displayed with a header row and index added starting with a value of 0:

In [5]:	df_user_data_2019 = pd.read_csv('user_data_2019.csv')																		
In [18]:	df_user_data_2019.head()																		
Out[18]:	<table border="1"> <thead> <tr> <th></th><th>userid</th><th>date</th><th>year</th></tr> </thead> <tbody> <tr> <td>0</td><td>1</td><td>1/1/2019</td><td>2019</td></tr> <tr> <td>1</td><td>3</td><td>1/2/2019</td><td>2019</td></tr> <tr> <td>2</td><td>6</td><td>1/3/2019</td><td>2019</td></tr> </tbody> </table>				userid	date	year	0	1	1/1/2019	2019	1	3	1/2/2019	2019	2	6	1/3/2019	2019
	userid	date	year																
0	1	1/1/2019	2019																
1	3	1/2/2019	2019																
2	6	1/3/2019	2019																

8. The next step is to merge the DataFrames using the `concat()` function. We include the `ignore_index=True` parameter to create a new index value for all of the results:

```
In[]: df_user_data_combined = pd.concat([df_user_data_2017,
df_user_data_2018, df_user_data_2019], ignore_index=True)
```

9. Run the `head()` command to verify the results:

```
In[]: df_user_data_combined.head(10)
```

The results will look similar to the following screenshot, where the rows are displayed with a header row and index added starting with a value of 0:

In [19]:	df_user_data_combined = pd.concat([df_user_data_2017, df_user_data_2018, df_user_data_2019], ignore_index=True)																																										
In [20]:	df_user_data_combined.head(10)																																										
Out[20]:	<table border="1"> <thead> <tr> <th></th><th>userid</th><th>date</th><th>year</th></tr> </thead> <tbody> <tr><td>0</td><td>1</td><td>1/1/2017</td><td>2017</td></tr> <tr><td>1</td><td>2</td><td>1/2/2017</td><td>2017</td></tr> <tr><td>2</td><td>3</td><td>1/3/2017</td><td>2017</td></tr> <tr><td>3</td><td>4</td><td>1/1/2018</td><td>2018</td></tr> <tr><td>4</td><td>5</td><td>1/2/2018</td><td>2018</td></tr> <tr><td>5</td><td>6</td><td>1/3/2018</td><td>2018</td></tr> <tr><td>6</td><td>1</td><td>1/1/2019</td><td>2019</td></tr> <tr><td>7</td><td>3</td><td>1/2/2019</td><td>2019</td></tr> <tr><td>8</td><td>6</td><td>1/3/2019</td><td>2019</td></tr> </tbody> </table>				userid	date	year	0	1	1/1/2017	2017	1	2	1/2/2017	2017	2	3	1/3/2017	2017	3	4	1/1/2018	2018	4	5	1/2/2018	2018	5	6	1/3/2018	2018	6	1	1/1/2019	2019	7	3	1/2/2019	2019	8	6	1/3/2019	2019
	userid	date	year																																								
0	1	1/1/2017	2017																																								
1	2	1/2/2017	2017																																								
2	3	1/3/2017	2017																																								
3	4	1/1/2018	2018																																								
4	5	1/2/2018	2018																																								
5	6	1/3/2018	2018																																								
6	1	1/1/2019	2019																																								
7	3	1/2/2019	2019																																								
8	6	1/3/2019	2019																																								

Binning

Binning is a very common analysis technique that allows you to group numeric data values based on one or more criteria. These groups become named categories; they are ordinal in nature and can have equal widths between the ranges or customized requirements. A good example is age ranges, which you commonly see on surveys such as that seen in the following screenshot:

Q: What is your age?

- ☐ Under 18 years old
- ☐ 18 – 24
- ☐ 25 – 44
- ☐ 45 – 74
- ☐ 75 years or older

In this example, a person's age range is the input, but what if we actually had the birthdate of each person available in the data? Then, we could calculate the age as of today and assign an age band based on the criteria in the preceding screenshot. This is the process of binning your data.

Another common example is weather data, where the assigned categories of *hot*, *warm*, or *cold* are assigned to ranges of Fahrenheit or Celsius temperature. Each bin value is defined by a condition that is arbitrarily decided by the data analyst.

For our user data, let's assign age bins based on when the user first appeared in our dataset. We will define three bins based on the requirements, which allows us the flexibility to adjust the assigned ranges. For our example, we define the bins as follows:

- Less than 1 year
- 1 to 2 years
- Greater than 3 years



The specific conditions on how to create the bins will be evident once we walk through the code.

Another cool feature of this type of analysis is the fact that our calculated age is based on the usage data and a point in time that's calculated each time we run the code. For example, if the date of the first time a user hits the website is *1/1/2017* and we did this analysis on December 3, 2018, the age in days of the user would be 360, which would be assigned to the *Less than 1 year* bin.

If we rerun this analysis at a later date such as November 18, 2019, the calculated age would change, so the new assigned bin would be *1 to 2 years*.

The decision on where to add the logic for each bin will vary. The most flexible to make changes to the assigned bins is to add the logic where you deliver the analytics. In our examples, that would be directly in the Jupyter notebook. However, in enterprise environments where many different technologies could be used to deliver the same analysis, it makes sense to move the binning logic closer to the source. In some cases of very large datasets stored in databases, using SQL or even having the schema changed in the table is a better option.

If you have the luxury of a skilled data engineering team and experience of working with big data like I had, the decision to move the binning logic closer to the data tables is easy. In SQL, you could use `CASE Statement` or if/then logic. Qlik has a function called `class()`, which will bin values based on a linear scale. In Microsoft Excel, a nested formula can be used to assign bins based on a mix of functions.

So, the concept of binning can be applied across different technologies and as a good data analyst, you now have a foundation of understanding how it can be done.

Let's reinforce the knowledge by walking through an example using our usage data and Jupyter Notebook.



Remember to copy any dependency CSV files into the working folder before walking through the following steps.

To recreate the example in Jupyter, let's create a new notebook and name it `ch_07_sifting_and_binning_data`:

1. Import the pandas library:

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

2. Read in the CSV file provided that includes additional data for this example and create a new DataFrame named `df_user_churn_cleaned`. We are also converting the `date` field found in the source CSV file into a data type of `datetime64` while importing using the `parse_dates` parameter. This will make it easier to manipulate in the next few steps:

```
In[]: df_user_churn_cleaned =  
pd.read_csv('user_hits_binning_import.csv', parse_dates=['date'])
```

3. Verify the DataFrame is valid using the `head()` function:

```
In[]: df_user_churn_cleaned.head(10)
```

The output of the function will look similar to the following table, where the DataFrame is loaded with two fields with the correct data types and is available for analysis:

Out[231]:		userid	date
	0	1	2017-01-01
	1	2	2017-01-02
	2	3	2017-01-03
	3	4	2018-01-01
	4	5	2018-01-02
	5	6	2018-10-03
	6	1	2019-10-01
	7	3	2019-10-02
	8	7	2019-10-03
	9	8	2020-01-01

4. Import the `datetime` and `numpy` libraries for reference later to calculate the age value of the `userid` field:

```
In[]: from datetime import datetime  
import numpy as np
```

5. Create a new derived column named `age` by calculating the difference between the current date and time using the `now` function and the `date` field. To format the `age` field in days, we include the `dt.days` function, which will convert the values into a clean `"%d"` format:

```
In[]: #df_user_churn_cleaned['age'] = (datetime.now() -
pd.to_datetime(df_user_churn_cleaned['date'])).dt.days
df_user_churn_cleaned['age'] = (datetime(2020, 2, 28) -
pd.to_datetime(df_user_churn_cleaned['date'])).dt.days
```



To match the screenshots, I explicitly defined the date value to 2020-02-28 with a date format of YYYY-MM-DD. You can uncomment the preceding line to calculate the current timestamp. Since the timestamp changes every time you run the function, the results will not match exactly to any image.

6. Verify that the new `age` column has been included in your DataFrame:

```
In[]: df_user_churn_cleaned.head()
```

The output of the function will look similar to the following table, where the DataFrame has been modified from its original import and includes a new field called `age`:

Out[214]:

	userid	date	age
0	1	2017-01-01	1153
1	2	2017-01-02	1152
2	3	2017-01-03	1151
3	4	2018-01-01	788
4	5	2018-01-02	787
5	6	2018-10-03	513
6	1	2019-10-01	150
7	3	2019-10-02	149
8	7	2019-10-03	148
9	8	2020-01-01	58
10	1	2020-01-02	57
11	2	2020-01-03	56

7. Create a new DataFrame called `df_ages` that groups the dimensions from the existing DataFrame and calculates the max age value by `userid`:

```
In[]: df_ages = df_user_churn_cleaned.groupby('userid').max()
```

The output will look similar to the following screenshot, where the number of rows has decreased from the source DataFrame. Only a distinct list of `userid` values will be displayed along with the maximum age when the first record was created by `userid`:

Out[216]:

	date	age
userid		
1	2020-01-02	1153
2	2020-01-03	1152
3	2019-10-02	1151
4	2018-01-01	788
5	2018-01-02	787
6	2018-10-03	513
7	2019-10-03	148
8	2020-01-01	58

8. Create a new `age_bin` column by using the pandas library's `cut()` function. This will thread each value from the `age` field between one of the assigned `bins` range we have assigned. We use the `labels` parameter to make the analysis easier to consume for any audience. Note that the value of 9999 was chosen to create a maximum boundary for the age value:

```
In[]: df_ages['age_bin'] = pd.cut(x=df_ages['age'], bins=[1, 365, 730, 9999], labels=['< 1 year', '1 to 2 years', '> 3 years'])
```

9. Display the DataFrame and validate the bin values displayed:

```
In[]: df_ages
```


The output of the function will look similar to the following screenshot, where the DataFrame has been modified and we now see the values in the `age_bin` field:

Out[218]:

	date	age	age_bin
userid			
1	2020-01-02	1153	> 3 years
2	2020-01-03	1152	> 3 years
3	2019-10-02	1151	> 3 years
4	2018-01-01	788	> 3 years
5	2018-01-02	787	> 3 years
6	2018-10-03	513	1 to 2 years
7	2019-10-03	148	< 1 year
8	2020-01-01	58	< 1 year

Summary

Congratulations, you have now increased your data literacy skills by working with data as both a consumer and producer of analytics. We covered some important topics, including essential skills to manipulate data by creating views of data, sorting, and querying tabular data from a SQL source. You now have a repeatable workflow for combining multiple data sources into one refined dataset.

We explored additional features of working with `pandas` DataFrames, showing how to restrict and sift data. We walked through real-world practical examples using the concept of *user churn* to answer key business questions about usage patterns by isolating specific users and dealing with missing values from the source data.

Our next chapter is [Chapter 8, Understanding Joins, Relationships, and Data Aggregates](#). Along with creating a summary analysis using a concept called aggregation, we will also go into detail on how to join data with defined relationships.

Further reading

You can refer to the following links for more information on the topics of this chapter:

- A nice walk-through of filtering and grouping using DataFrames: https://github.com/bhavaniravi/pandas_tutorial/blob/master/Pandas_Basics_To_Beyond.ipynb
- Comparison of SQL features and their equivalent pandas functions: https://pandas.pydata.org/pandas-docs/stable/getting_started/comparison/comparison_with_sql.html
- Additional information on exporting data to Excel: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_excel.html#pandas.DataFrame.to_excel
- Examples of SQL date and time functions: <https://www.postgresql.org/docs/8.1/functions-datetime.html>

8

Understanding Joins, Relationships, and Aggregates

I'm really excited about this chapter because we are going to learn about the foundation of blending multiple datasets. This concept has been around for decades using SQL and other technologies including R, pandas, Excel, Cognos, and Qlikview.

The ability to merge data is a powerful skill that applies across different technologies and helps you to answer complex questions such as how product sales can be impacted by weather forecasts. The data sources are mutually exclusive, but today, access to weather data can be added to your data model with a few joins based on geographic location and time of day. We will be covering how this can be done along with the different types of joins. Once exposed to this concept, you will learn what questions can be answered depending on the granularity of data available. For our weather and sales data example, the details become important to understand the level of analysis that can be done. If you wanted to know whether rain impacts sales, the more common fields available, such as date, day, and time, and geographic location tags, such as latitude and longitude, must be available in both sources for you to be accurate in your conclusions after joining the data together.

In this chapter, we will learn how to construct high-quality datasets for further analysis. We will continue to advance your hands-on data literacy skills by learning how to work with join relationships and how to create aggregate data for analysis.

In this chapter, we will cover the following topics:

- Foundations of join relationships
- Join types in action
- Explaining data aggregation
- Summary statistics and outliers