IV

Data Wrangling

The following **data wrangling** chapters provide you with the necessary skills for understanding, loading, manipulating, reshaping, and exploring data structures. Perhaps the most time-consuming part of data science is preparing and exploring your data set, and learning how to perform these tasks programmatically can make the process easier and more transparent. Mastering these skills is thus vital to being an effective data scientist.

9

Understanding Data

Previous chapters have introduced the basic programming fundamentals for working with data, detailing how you can tell a computer to do data processing for you. To use a computer to analyze data, you need to both *access* a data set and *interpret* that data set so that you can ask meaningful questions about it. This will enable you to transform raw data into actionable information.

This chapter provides a high-level overview of how to interpret data sets as you get started doing data science—it details the sources of data you might encounter, the formats that data may take, and strategies for determining which questions to ask of that data. Developing a clear mental model of what the values in a data set signify is a necessary prerequisite before you can program a computer to effectively analyze that data.

9.1 The Data Generation Process

Before beginning to work with data, it's important to understand *where data comes from*. There are a variety of processes for capturing events as data, each of which has its own limitations and assumptions. The primary modes of data collection fall into the following categories:

- **Sensors**: The volume of data being collected by sensors has increased dramatically in the last decade. Sensors that automatically detect and record information, such as pollution sensors that measure air quality, are now entering the personal data management sphere (think of FitBits or other step counters). Assuming these devices have been properly calibrated, they offer a reliable and consistent mechanism for data collection.
- **Surveys**: Data that is less externally measurable, such as people's opinions or personal histories, can be gathered from *surveys*. Because surveys are dependent on individuals' self-reporting of their behavior, the quality of data may vary (across surveys, or across individuals). Depending on the domain, people may have poor recall (i.e., people don't remember what they ate last week) or have incentives to respond in a particular way (i.e., people may over-report healthy behaviors). The biases inherent in survey responses should be recognized and, when possible, adjusted for in your analysis.
- **Record keeping**: In many domains, organizations use both automatic and manual processes to keep track of their activities. For example, a hospital may track the length and result of every surgery it performs (and a governing body may require that hospital to report those

results). The reliability of such data will depend on the quality of the systems used to produce it. Scientific experiments also depend on diligent record keeping of results.

- **Secondary data analysis**: Data can be compiled from *existing knowledge artifacts* or measurements, such as counting word occurrences in a historical text (computers can help with this!).

All of these methods of collecting data can lead to potential concerns and biases. For example, sensors may be inaccurate, people may present themselves in particular ways when responding to surveys, record keeping may only focus on particular tasks, and existing artifacts may already exclude perspectives. When working with any data set, it is vital to consider where the data came from (e.g., *who* recorded it, *how*, and *why*) to effectively and meaningfully analyze it.

9.2 Finding Data

Computers' abilities to record and persist data have led to an explosion of available data values that can be analyzed, ranging from personal biological measures (*how many steps have I taken?*) to social network structures (*who are my friends?*) to private information leaked from insecure websites and government agencies (*what are their Social Security numbers?*). In professional environments, you will likely be working with proprietary data collected or managed by your organization. This might be anything from purchase orders of fair trade coffee to the results of medical research—the range is as wide as the types of organizations (since *everyone* now records data and sees a need for data analytics).

Luckily, there are also plenty of free, nonproprietary data sets that you can work with. Organizations will often make large amounts of data available to the public to support experiment duplication, promote transparency, or just see what other people can do with that data. These data sets are great for building your data science skills and portfolio, and are made available in a variety of formats. For example, data may be accessed as downloadable CSV spreadsheets (see Chapter 10), as relational databases (see Chapter 13), or through a web service API (see Chapter 14).

Popular sources of open data sets include:

- **Government publications**: Government organizations (and other bureaucratic systems) produce *a lot* of data as part of their everyday activities, and often make these data sets available in an effort to appear transparent and accountable to the public. You can currently find publicly available data from many countries, such as the United States,[1] Canada,[2] India,[3] and others. Local governments will also make data available: for example, the City of Seattle [4] makes a vast amount of data available in an easy-to-access format. Government data covers a broad range of topics, though it can be influenced by the political situation surrounding its gathering and retention.

[1] **U.S. government's open data:** https://www.data.gov
[2] **Government of Canada open data:** https://open.canada.ca/en/open-data
[3] **Open Government Data Platform India:** https://data.gov.in
[4] **City of Seattle open data portal:** https://data.seattle.gov

- **News and journalism**: Journalism remains one of the most important contexts in which data is gathered and analyzed. Journalists do much of the legwork in producing data—searching existing artifacts, questioning and surveying people, or otherwise revealing and connecting previously hidden or ignored information. News media usually publish the analyzed, summative information for consumption, but they also may make the source data available for others to confirm and expand on their work. For example, the *New York Times*[5] makes much of its historical data available through a web service, while the data politics blog *FiveThirtyEight*[6] makes all of the data behind its articles available on GitHub (invalid models and all).
- **Scientific research**: Another excellent source of data is ongoing scientific research, whether performed in academic or industrial settings. Scientific studies are (in theory) well grounded and structured, providing meaningful data when considered within their proper scope. Since science needs to be disseminated and validated by others to be usable, research is often made publicly available for others to study and critique. Some scientific journals, such as the premier journal *Nature*, require authors to make their data available for others to access and investigate (check out its list[7] of scientific data repositories!).
- **Social networks and media organizations**: Some of the largest quantities of data produced occur online, automatically recorded from people's usage of and interactions with social media applications such as Facebook, Twitter, or Google. To better integrate these services into people's everyday lives, social media companies make much of their data programmatically available for other developers to access and use. For example, it is possible to access live data from Twitter,[8] which has been used for a variety of interesting analyses. Google also provides programmatic access[9] to most of its many services (including search and YouTube).
- **Online communities**: As data science has rapidly increased in popularity, so too has the community of data science practitioners. This community and its online spaces are another great source for interesting and varied data sets and analysis. For example, *Kaggle*[10] hosts a number of data sets as well as "challenges" to analyze them. *Socrata*[11] (which powers the Seattle data repository), also collects a variety of data sets (often from professional or government contributors). Somewhat similarly, the *UCI Machine Learning Repository*[12] maintains a collection of data sets used in machine learning, drawn primarily from academic sources. And there are many other online lists of data sources as well—including a dedicated Subreddit `/r/Datasets`.[13]

In short, there are a huge number of real-world data sets available for you to work with—whether you have a specific question you would like to answer, or just want to explore and be inspired.

---

[5] ***New York Times* Developer Network**: https://developer.nytimes.com
[6] ***FiveThirtyEight:* Our Data:** https://data.fivethirtyeight.com
[7] **Nature: Recommended Data Repositories:** https://www.nature.com/sdata/policies/repositories
[8] **Twitter developer platform:** https://developer.twitter.com/en/docs
[9] **Google APIs Explorer:** https://developers.google.com/apis-explorer/
[10] **Kaggle**: "the home of data science and machine learning": https://www.kaggle.com
[11] **Socrata**: data as a service platform: https://opendata.socrata.com
[12] **UCI Machine Learning Repository:** https://archive.ics.uci.edu/ml/index.php
[13] **/r/DataSets:** https://www.reddit.com/r/datasets/

9.3 Types of Data

Once you acquire a data set, you will have to understand its structure and content before (programmatically) investigating it. Understanding the *types* of data you will encounter depends on your ability to discern the level of measurement for a given piece of data, as well as the different structures that are used to hold that data.

9.3.1 Levels of Measurement

Data can be made up of a variety of types of values (represented by the concept of "data type" in R). More generally, data values can also be discussed in terms of their **level of measurement**[14]—a way of classifying data values in terms of how they can be measured and compared to other values.

The field of statistics commonly classifies values into one of four levels, described in Table 9.1.

Nominal data (often equivalently **categorical data**) is data that has no implicit ordering. For example, you cannot say that "apples are more than oranges," though you can indicate that a particular fruit either is an apple or an orange. Nominal data is commonly used to indicate that an observation belongs in a particular category or group. You do not usually perform mathematical analysis on nominal data (e.g., you can't find the "average" fruit), though you can discuss counts or distributions. Nominal data can be represented by strings (such as the name of the fruit), but also by numbers (e.g., "fruit type #1", "fruit type #2"). Just because a value in a data set is a number, that does not mean you can do math upon it! Note that boolean values (`TRUE` or `FALSE`) are a type of nominal value.

Ordinal data establishes an *order* for nominal categories. Ordinal data may be used for classification, but it also establishes that some groups are *greater than* or *less than* others. For example, you may have classifications of hotels or restaurants as *5-star, 4-star,* and so on. There is an ordering to these categories, but the distances between the values may vary. You are able to find the minimum, maximum, and even median values of ordinal variables, but you can't compute a statistical mean (since ordinal values do not define *how much* greater one value is than another). Note that it is possible to treat nominal variables as ordinal by enforcing an ordering, though in

Table 9.1 **Levels of measurement**

Level	Example	Operations
Nominal unordered; used for classification	*Fruits*: apples, bananas, oranges, etc.	`==`, `!=` "*same or different*"
Ordinal ordered; can sort	*Hotel rating*: 5-star, 4-star, etc.	`==`, `!=`, `<`, `>` "*bigger or smaller*"
Ratio ordered, fixed "zero"	*Lengths*: 1 inch, 1.5 inches, 2 inches, etc.	`==`, `!=`, `<`, `<`, `+`, `-`, `*`, `/` "*twice as big*"
Interval ordered, no set "zero"	*Dates*: 05/15/2012, 04/17/2015, etc.	`==`, `!=`, `<`, `>`, `+`, `-` "*3 units bigger*"

[14] Stevens, S. S. (1946). On the theory of scales of measurement. *Science, 103*(2684), 677–680. https://doi.org/10.1126/science.103.2684.677

effect this changes the measurement level of the data. For example, colors are usually *nominal* data—you cannot say that "red is greater than blue." This is despite the conventional ordering based on the colors of a rainbow; when you say that "red comes before blue (in the rainbow)," you're actually replacing the nominal color value with an ordinal value representing its *position in a rainbow* (which itself is dependent on the *ratio* value of its wavelength)! Ordinal data is also considered categorical.

Ratio data (often equivalently **continuous data**) is the most common level of measurement in real-world data: data based on population counts, monetary values, or amount of activity is usually measured at the ratio level. With ratio data, you can find averages, as well as measure the distance between different values (a feature also available with interval data). As you might expect, you can also compare the ratio of two values when working with ratio data (i.e., value x is twice as great as value y).

Interval data is similar to ratio data, except there is no fixed zero point. For example, dates cannot be discussed in *proportional* terms (i.e., you wouldn't say that *Wednesday is twice as Monday*). Therefore, you can compute the distance (interval) between two values (i.e., *2 days apart*), but you cannot compute the *ratio* between two values. Interval data is also considered continuous.

Identifying and understanding the level of measurement of a particular data feature is important when determining how to analyze a data set. In particular, you need to know what kinds of statistical analysis will be valid for that data, as well as how to interpret what that data is measuring.

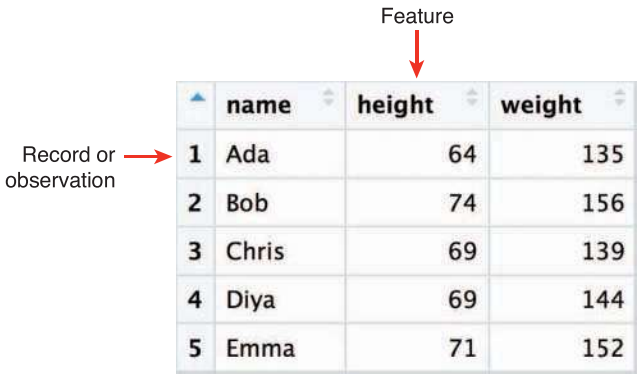
9.3.2 Data Structures

In practice, you will need to organize the numbers, strings, vectors, and lists of values described in the previous chapters into more complex formats. Data is organized into more robust **structures**—particularly as the data set gets large—to better signify what those numbers and strings represent. To work with real-world data, you will need to be able to understand these structures and the terminology used to discuss them.

In practice, most data sets are structured as **tables** of information, with individual data values arranged into *rows* and *columns* (see Figure 9.1). These tables are similar to how data may be recorded in a spreadsheet (using a program such as Microsoft Excel). In a table, each row represents a **record** or **observation**: an instance of a single thing being measured (e.g., a person, a sports match). Each column represents a **feature**: a particular property or aspect of the thing being measured (e.g., the person's height or weight, the scores in a sports game). Each data value can be referred to as a **cell** in the table.

Viewed in this way, a table is a collection of "things" being measured, each of which has a particular value for a characteristic of that thing. And, because all the observations share the same characteristics (features), it is possible to analyze them comparatively. Moreover, by organizing data into a table, each data value (cell) can be automatically given two associated meanings: which observation it is from as well as which feature it represents. This structure allows you to discern semantic meaning from the numbers: the number 64 in figure Figure 9.1 is not just some value; it's "Ada's height."

The table in Figure 9.1 represents a *small* (even tiny) data set, in that it contains just five observations (rows). The size of a data set is generally measured in terms of its number of



	name	height	weight
1	Ada	64	135
2	Bob	74	156
3	Chris	69	139
4	Diya	69	144
5	Emma	71	152

Figure 9.1 A table of data (of people's weights and heights). Rows represent *observations*, while columns represent *features*.

observations: a small data set may contain only a few dozen observations, while a large data set may contain thousands or hundreds of thousands of records. Indeed, "Big Data" is a term that, in part, refers to data sets that are so large that they can't be loaded into the computer's memory without special handling, and may have billions or even trillions of rows! Yet, even a data set with a relatively small number of observations can contain a large number of cells if they record a lot of features per observations (though these tables can often be "inverted" to have more rows and fewer columns; see Chapter 12). Overall, the number of observations and features (rows and columns) is referred to as the **dimensions** of the data set—not to be confused with referring to a table's "two-dimensional" data structure (because each data value has *two* meanings: observation and feature).

Although it is commonly structured in this way, data need not be represented as a single table. More complex data sets may spread data values across multiple tables (such as in a database; see Chapter 13). In other complex data structures, each individual cell in the table may hold a vector or even its own data table. This can cause the table to no longer be two-dimensional, but three- or more-dimensional. Indeed, many data sets available from web services are structured as "nested tables"; see Chapter 14 for details.

9.4 Interpreting Data

The first thing you will need to do upon encountering a data set (whether one you found online or one that was provided by your organization) is to understand the meaning of the data. This requires understanding the domain you are working in, as well as the specific data schema you are working with.

9.4.1 Acquiring Domain Knowledge

The first step toward being able to understand a data set is to research and understand the data's problem domain. The **problem domain** is the set of topics that are relevant to the problem—that is, the context for that data. Working with data requires **domain knowledge**: you need to have a

basic level of understanding of that problem domain to do any sensible analysis of that data. You will need to develop a mental model of what the data values mean. This includes understanding the significance and purpose of any features (so you're not doing math on contextless numbers), the range of expected values for a feature (to detect outliers and other errors), and some of the subtleties that may not be explicit in the data set (such as biases or aggregations that may hide important causalities).

As a specific example, if you wanted to analyze the table shown in Figure 9.1, you would need to first understand what is meant by "height" and "weight" of a person, the implied units of the numbers (inches, centimeters, ... or something else?), an expected range (does Ada's height of 64 mean she is short?), and other external factors that may have influenced the data (e.g., age).

> **Remember**: You do not need to necessarily be an expert in the problem domain (though it wouldn't hurt); you just need to acquire *sufficient* domain knowledge to work within that problem domain!

While people's heights and other data sets discussed in this text should be familiar to most readers, in practice you are quite likely to come across data from problem domains that are outside of your personal domain expertise. Or, more problematically, the data set may be from a problem domain that you *think* you understand but actually have a flawed mental model of (a failure of *meta-cognition*).

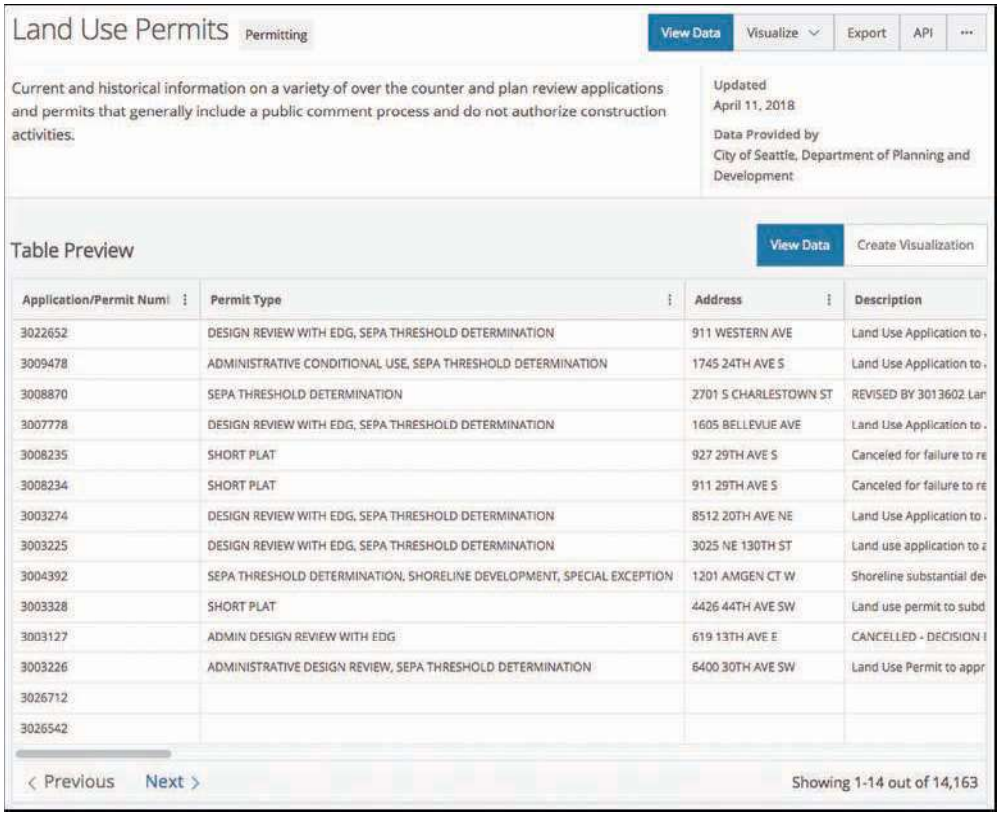
For example, consider the data set shown in Figure 9.2, a screenshot taken from the City of Seattle's data repository. This data set presents information on *Land Use Permits*, a somewhat opaque bureaucratic procedure with which you may be unfamiliar. The question becomes: how would you acquire sufficient domain knowledge to understand and analyze this data set?

Gathering domain knowledge almost always requires outside research—you will rarely be able to understand a domain just by looking at a spreadsheet of numbers. To gain general domain knowledge, we recommend you start by consulting a general knowledge reference: *Wikipedia* provides easy access to basic descriptions. Be sure to read any related articles or resources to improve your understanding: sifting through the vast amount of information online requires cross-referencing different resources, and mapping that information to your data set.

That said, the best way to learn about a problem is to find a *domain expert* who can help explain the domain to you. If you want to know about land use permits, try to find someone who has used one in the past. The second best solution is to ask a librarian—librarians are specifically trained to help people discover and acquire basic domain knowledge. Libraries may also support access to more specialized information sources.

9.4.2 Understanding Data Schemas

Once you have a general understanding of the context for a data set, you can begin interpreting the data set itself. You will need to focus on understanding the **data schema** (e.g., what is represented by the rows and columns), as well as the specific context for those values. We suggest you use the following questions to guide your research:



Land Use Permits Permitting

View Data | Visualize | Export | API | ...

Current and historical information on a variety of over the counter and plan review applications and permits that generally include a public comment process and do not authorize construction activities.

Updated
April 11, 2018

Data Provided by
City of Seattle, Department of Planning and Development

Table Preview

View Data | Create Visualization

Application/Permit Numl	Permit Type	Address	Description
3022652	DESIGN REVIEW WITH EDG, SEPA THRESHOLD DETERMINATION	911 WESTERN AVE	Land Use Application to
3009478	ADMINISTRATIVE CONDITIONAL USE, SEPA THRESHOLD DETERMINATION	1745 24TH AVE S	Land Use Application to
3008870	SEPA THRESHOLD DETERMINATION	2701 S CHARLESTOWN ST	REVISED BY 3013602 Lar
3007778	DESIGN REVIEW WITH EDG, SEPA THRESHOLD DETERMINATION	1605 BELLEVUE AVE	Land Use Application to
3008235	SHORT PLAT	927 29TH AVE S	Canceled for failure to re
3008234	SHORT PLAT	911 29TH AVE S	Canceled for failure to re
3003274	DESIGN REVIEW WITH EDG, SEPA THRESHOLD DETERMINATION	8512 20TH AVE NE	Land Use Application to
3003225	DESIGN REVIEW WITH EDG, SEPA THRESHOLD DETERMINATION	3025 NE 130TH ST	Land use application to a
3004392	SEPA THRESHOLD DETERMINATION, SHORELINE DEVELOPMENT, SPECIAL EXCEPTION	1201 AMGEN CT W	Shoreline substantial de
3003328	SHORT PLAT	4426 44TH AVE SW	Land use permit to subd
3003127	ADMIN DESIGN REVIEW WITH EDG	619 13TH AVE E	CANCELLED - DECISION I
3003226	ADMINISTRATIVE DESIGN REVIEW, SEPA THRESHOLD DETERMINATION	6400 30TH AVE SW	Land Use Permit to appr
3026712			
3026542			

< Previous Next >

Showing 1-14 out of 14,163

Figure 9.2 A preview of land use permits data from the City of Seattle.[15] Content has been edited for display in this text.

"What meta-data *is available for the data set?"*

Many publicly available data sets come with summative explanations, instructions for access and usage, or even descriptions of individual features. This **meta-data** (data about the data) is the best way to begin to understand what value is represented by each cell in the table, since the information comes directly from the source.

For example, Seattle's land use permits page has a short summary (though you would want to look up what an "over-the-counter review application" is), provides a number of categories and tags, lists the dimensions of the data set (14,200 rows as of this writing), and gives a quick description of each column.

A particularly important piece of meta-data to search for is:

"Who created the data set? Where does it come from?"

[15] **City of Seattle: Land Use Permits** (access requires a free account): https://data.seattle.gov/Permitting/Land-Use-Permits/uyyd-8gak

Understanding who generated the data set (and how they did so!) will allow you to know where to find more information about the data—it will let you know who the domain experts are. Moreover, knowing the source and methodology behind the data can help you uncover hidden biases or other subtleties that may not be obvious in the data itself. For example, the Land Use Permits page notes that the data was provided by the "City of Seattle, Department of Planning and Development" (now the Department of Construction & Inspections). If you search for this organization, you can find its website.[16] This website would be a good place to gain further information about the specific data found in the data set.

Once you understand this meta-data, you can begin researching the data set itself:

> *"What features does the data set have?"*

Regardless of the presence of meta-data, you will need to understand the columns of the table to work with it. Go through each column and check if you understand:

1. What "real-world" aspect does each column attempt to capture?
2. For continuous data: what units are the values in?
3. For categorical data: what different categories are represented, and what do those mean?
4. What is the possible range of values?

If the meta-data provides a *key* to the data table, this becomes an easy task. Otherwise, you may need to study the source of the data to determine how to understand the features, sparking additional domain research.

> **Tip:** As you read through a data set—or anything really—you should *write down* the terms and phrases you are not familiar with to look up later. This will discourage you from (inaccurately) guessing a term's meaning, and will help delineate between terms you have and have not yet clarified.

For example, the Land Use Permits data set provides clear descriptions of the columns in the meta-data, but looking at the sample data reveals that some of the values may require additional research. For example, what are the different Permit Types and Decision Types? By going back to the source of the data (the Department of Construction home page), you can navigate to the Permits page and then to the "Permits We Issue (A-Z)" to see a full list of possible permit types. This will let you find out, for example, that "PLAT" refers to "creating or modifying individual parcels of property"—in other words, adjusting lot boundaries.

To understand the features, you will need to look at some sample observations. Open up the spreadsheet or table and look at the first few rows to get a sense for what kind of values they have and what that may say about the data.

Finally, throughout this process, you should continually consider:

> *"What terms do you not know or understand?"*

[16] **Seattle Department of Construction & Inspections (access requires a free account)**: http://www.seattle.gov/dpd/

Depending on the problem domain, a data set may contain a large amount of *jargon*, both to explain the data and inside the data itself. Making sure you understand all the technical terms used will go a long way toward ensuring you can effectively discuss and analyze the data.

> **Caution**: Watch out for acronyms you are not familiar with, and be sure to look them up!

For example, looking at the "Table Preview," you may notice that many of the values for the "Permit Type" feature use the term "SEPA." Searching for this acronym would lead you to a page describing the *State Policy Environmental Act* (requiring environmental impact to be considered in how land is used), as well as details on the "Threshold Determination" process.

Overall, interpreting a data set will require research and work that is *not* programming. While it may seem like such work is keeping you from making progress in processing the data, having a valid mental model of the data is both useful and necessary to perform data analysis.

9.5 Using Data to Answer Questions

Perhaps the most challenging aspect of data analysis is effectively applying questions of interest to the data set to construct the desired information. Indeed, as a data scientist, it will often be your responsibility to translate from various domain questions to specific observations and features in your data set. Take, for example, a question like:

> *"What is the worst disease in the United States?"*

To answer this question, you will need to understand the problem domain of disease burden measurement and acquire a data set that is well positioned to address the question. For example, one appropriate data set would be the *Global Burden of Disease*[17] study performed by the Institute for Health Metrics and Evaluation, which details the *burden of disease* in the United States and around the world.

Once you have acquired this data set, you will need to **operationalize** the motivating question. Considering each of the key words, you will need to identify a set of *diseases*, and then quantify what is meant by "worst." For example, the question could be more concretely phrased as any of these interpretations:

- Which disease *causes the largest number of deaths* in the United States?
- Which disease *causes the most premature deaths* in the United States?
- Which disease *causes the most disability* in the United States?

Depending on your definition of "worst," you will perform very different computations and analysis, possibly arriving at different answers. You thus need to be able to decide *what precisely is meant by a question*—a task that requires understanding the nuances found in the question's problem domain.

Figure 9.3 shows visualizations that try to answer this very question. The figure contains screenshots of *treemaps* from an online tool called *GBD Compare*.[18] A treemap is like a pie chart that is built with

[17] **IHME: Global Burden of Disease:** http://www.healthdata.org/node/835
[18] **GBD Compare**: visualization for global burden of disease: https://vizhub.healthdata.org/gbd-compare/

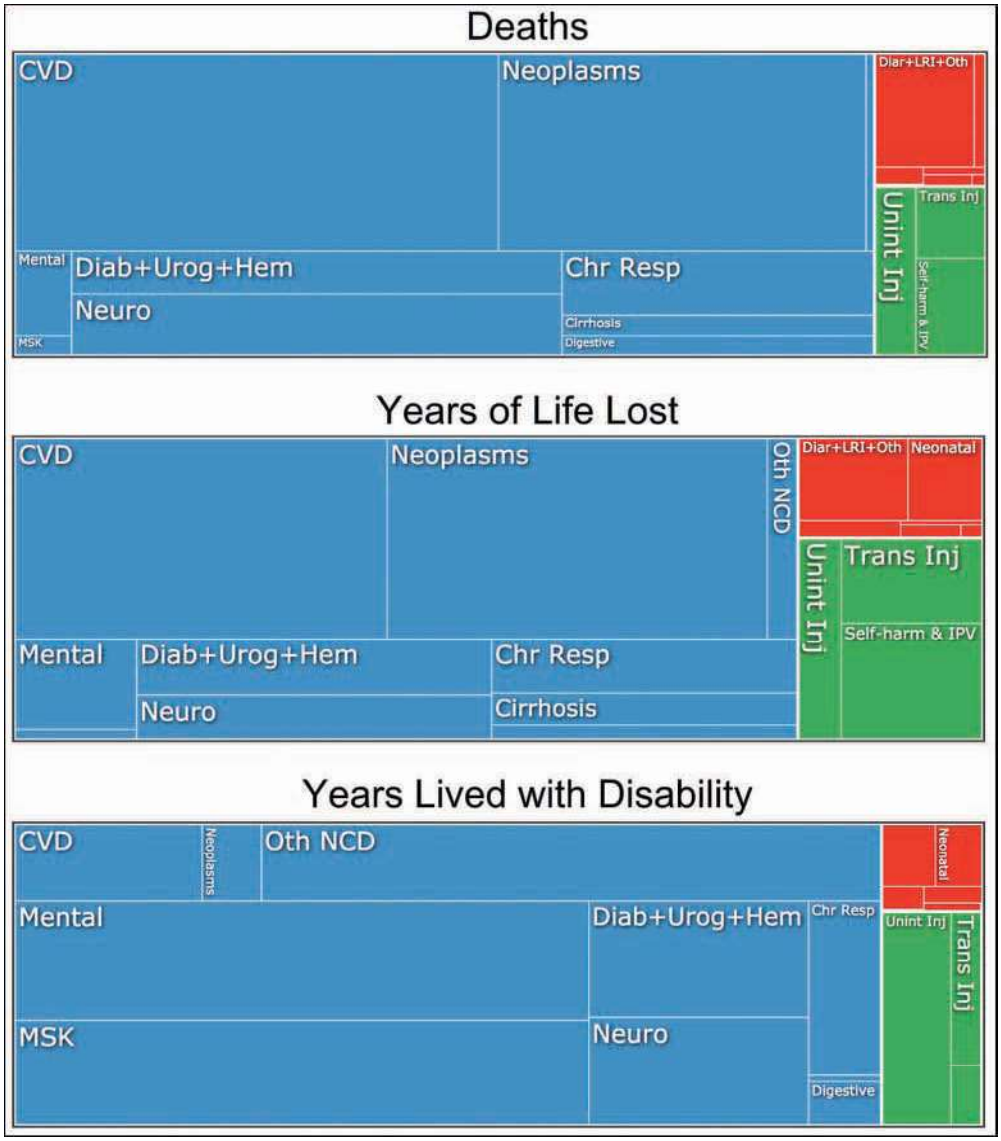


Figure 9.3 Treemaps from the GBD Compare tool showing the proportion of deaths (top), years of life lost (middle), and years lived with disability (bottom) attributable to each disease in the United States.

rectangles: the area of each segment is drawn proportionally to an underlying piece of data. The additional advantage of the treemap is that it can show *hierarchies* of information by *nesting* different levels of rectangles inside of one another. For example, in Figure 9.3, the disease burden from each communicable disease (shown in red) is nested within the same segment of each chart.

Depending on how you choose to operationalize the idea of the "worst disease," different diseases stand out as the most impactful. As you can see in Figure 9.3, almost 90% of all deaths are caused by non-communicable diseases such as cardiovascular diseases (*CVD*) and cancers (*Neoplasms*), shown in blue. When you consider the age of death for each person (computing a metric called *Years of Life Lost*), this value drops to 80%. Moreover, this metric enables you to identify causes of death that disproportionately affect young people, such as traffic accidents (*Trans Inj*) and self-harm, shown in green (see the middle chart in Figure 9.3). Finally, if you consider the "worst" disease to be that currently causing the most physical disability in the population (as in the bottom chart in Figure 9.3), the impacts of musculoskeletal conditions (*MSK*) and mental health issues (*Mental*) are exposed.

Because data analysis is about identifying answers to questions, the first step is to ensure you have a strong understanding of the question of interest and how it is being measured. Only after you have mapped from your questions of interest to specific features (columns) of your data can you perform an effective and meaningful analysis of that data.

10 Data Frames

This chapter introduces data frame values, which are the primary two-dimensional data storage type used in R. In many ways, data frames are similar to the row-and-column table layout that you may be familiar with from spreadsheet programs like Microsoft Excel. Rather than interact with this data structure through a user interface (UI), you will learn how to programmatically and reproducibly perform operations on this data type. This chapter covers ways of creating, describing, and accessing data from data frames in R.

10.1 What Is a Data Frame?

At a practical level, **data frames** act like tables, where data is organized into rows and columns. For example, reconsider the table of names, weights, and heights from Chapter 9, shown in Figure 10.1. In R, you can use data frames to represent these kinds of tables.

Data frames are really just lists (see Chapter 8) in which each element is a vector of the same length. Each vector represents a column, not a row. The elements at corresponding indices in the vectors are considered part of the same row (record). This structure makes sense because each row may have different types of data—such as a person's `name` (string) and `height` (number)—and vector elements must all be of the same type.

	name	height	weight
1	Ada	64	135
2	Bob	74	156
3	Chris	69	139
4	Diya	69	144
5	Emma	71	152

Figure 10.1 A table of data (of people's weights and heights) when viewed as a data frame in RStudio.

For example, you can think of the data shown in Figure 10.1 as a *list* of three *vectors*: `name`, `height`, and `weight`. The name, height, and weight of the first person measured are represented by the first elements of the `name`, `height`, and `weight` vectors, respectively.

You can work with data frames as if they were lists, but data frames have additional properties that make them particularly well suited for handling tables of data.

10.2 Working with Data Frames

Many data science questions can be answered by honing in on the desired subset of your data. In this section, you will learn how to create, describe, and access data from data frames.

10.2.1 Creating Data Frames

Typically you will *load* data sets from some external source (see Section 10.3), rather than writing out the data by hand. However, it is also possible to construct a data frame by combining multiple vectors. To accomplish this, you can use the **`data.frame()`** function, which accepts vectors as arguments, and creates a table with a column for each vector. For example:

```
# Create a data frame by passing vectors to the `data.frame()` function

# A vector of names
name <- c("Ada", "Bob", "Chris", "Diya", "Emma")

# A vector of heights
height <- c(64, 74, 69, 69, 71)

# A vector of weights
weight <- c(135, 156, 139, 144, 152)

# Combine the vectors into a data frame
# Note the names of the variables become the names of the columns!
people <- data.frame(name, height, weight, stringsAsFactors = FALSE)
```

The last argument to the `data.frame()` function is included because one of the vectors contains strings; it tells R to treat that vector as a typical vector, instead of another data type called a **factor** when constructing the data frame. This is usually what you will want to do—see Section 10.3.2 for more information.

You can also specify data frame column names using the `key = value` syntax used by *named lists* when you create your data frame:

```
# Create a data frame of names, weights, and heights,
# specifying column names to use
people <- data.frame(
  name = c("Ada", "Bob", "Chris", "Diya", "Emma"),
  height = c(64, 74, 69, 69, 71),
  weight = c(135, 156, 139, 144, 152)
)
```

Because data frame elements are lists, you can access the values from `people` using the same dollar notation and double-bracket notation as you use with lists:

```
# Retrieve information from a data frame using list-like syntax

# Create the same data frame as above
people <- data.frame(name, height, weight, stringsAsFactors = FALSE)

# Retrieve the `weight` column (as a list element); returns a vector
people_weights <- people$weight

# Retrieve the `height` column (as a list element); returns a vector
people_heights <- people[["height"]]
```

For more flexible approaches to accessing data from data frames, see section 10.2.3.

10.2.2 Describing the Structure of Data Frames

While you can interact with data frames as lists, they also offer a number of additional capabilities and functions. For example, Table 10.1 presents a few functions you can use to *inspect* the structure and content of a data frame:

Table 10.1 **Functions for inspecting data frames**

Function	Description
`nrow(my_data_frame)`	Returns the number of rows in the data frame
`ncol(my_data_frame)`	Returns the number of columns in the data frame
`dim(my_data_frame)`	Returns the dimensions (rows, columns) in the data frame
`colnames(my_data_frame)`	Returns the names of the columns of the data frame
`rownames(my_data_frame)`	Returns the names of the rows of the data frame
`head(my_data_frame)`	Returns the first few rows of the data frame (as a new data frame)
`tail(my_data_frame)`	Returns the last few rows of the data frame (as a new data frame)
`View(my_data_frame)`	Opens the data frame in a spreadsheet-like viewer (only in RStudio)

```
# Use functions to describe the shape and structure of a data frame

# Create the same data frame as above
people <- data.frame(name, height, weight, stringsAsFactors = F)

# Describe the structure of the data frame
nrow(people) # [1] 5
```

```
ncol(people) # [1] 3
dim(people) # [1] 5 3
colnames(people) # [1] "name" "height" "weight"
rownames(people) # [1] "1" "2" "3" "4" "5"

# Create a vector of new column names
new_col_names <- c("first_name", "how_tall", "how_heavy")

# Assign that vector to be the vector of column names
colnames(people) <- new_col_names
```

Many of these description functions can also be used to modify the structure of a data frame. For example, you can use the `colnames()` functions to assign a new set of column names to a data frame.

10.2.3 Accessing Data Frames

As stated earlier, since data frames are lists, it's possible to use dollar notation (`my_df$column_name`) or double-bracket notation (`my_df[["column_name"]]`) to access entire columns. However, R also uses a variation of single-bracket notation that allows you to filter for and access individual data elements (cells) in the table. In this syntax, you put two values separated by a comma (`,`) inside of single square brackets—the first argument specifies which row(s) you want to extract, while the second argument specifies which column(s) you want to extract.

Table 10.2 summarizes how single-bracket notation can be used to access data frames. Take special note of the fourth option's syntax (for retrieving rows): you still include the comma (`,`), but because you leave the *which column* value blank, you get all of the columns!

Table 10.2 **Accessing a data frame with single bracket notation**

Syntax	Description	Example
`my_df[row_name, col_name]`	Element(s) by row and column names	`people["Ada", "height"]` (element in row *named* `Ada` and column *named* `height`)
`my_df[row_num, col_num]`	Element(s) by row and column indices	`people[2, 3]` (element in the second row, third column)
`my_df[row, col]`	Element(s) by row and column; can mix names and indices	`people[2, "height"]` (second element in the `height` column)
`my_df[row, ]`	All elements (columns) in row name or index	`people[2, ]` (all columns in the second row)
`my_df[, col]`	All elements (rows) in a column name or index	`people[, "height"]` (all rows in the `height` column; equivalent to list notations)

```
# Assign a set of row names for the vector
# (using the values in the `name` column)
rownames(people) <- people$name

# Extract the row with the name "Ada" (and all columns)
people["Ada", ] # note the comma, indicating all columns

# Extract the second column as a vector
people[, "height"] # note the comma, indicating all rows

# Extract the second column as a data frame (filtering)
people["height"] # without a comma, it returns a data frame
```

Of course, because numbers and strings are stored in vectors, you're actually specifying vectors of names or indices to extract. This allows you to get multiple rows or columns:

```
# Get the `height` and `weight` columns
people[, c("height", "weight")] # note the comma, indicating all rows

# Get the second through fourth rows
people[2:4, ] # note the comma, indicating all columns
```

Additionally, you can use a vector of boolean values to specify your indices of interest (just as you did with vectors):

```
# Get rows where `people$height` is greater than 70 (and all columns)
people[people$height > 70, ] # rows for which `height` is greater than 70
```

Remember: The type of data that is returned when selecting data using single brackets depends on *how many columns* you are selecting. Extracting values from more than one column will produce a data frame; extracting from just one column will produce a vector.

Tip: In general, it's easier, cleaner, and less buggy to filter by column name (character string), rather than by column number, because it's not unusual for column order to change in a data frame. You should almost *never* access data in a data frame by its positional index. Instead, you should use the column name to specify columns, and a filter to specify rows of interest.

Going Further: While data frames are the two-dimensional data structure suggested by this book, they are not the only 2D data structure in R. For example, a *matrix* is a two-dimensional data structure in which all of the values have the same type (usually numeric).

To use all the syntax and functions described in this chapter, first confirm that a data object is a data frame (using `is.data.frame()`), and if necessary, *convert* an object to a data frame (such as by using the `as.data.frame()` function).

10.3 Working with CSV Data

Section 10.2 demonstrated constructing your own data frames by "hard-coding" the data values. However, it is much more common to load data from somewhere else, such as a separate file on your computer or a data resource on the internet. R is also able to ingest data from a variety of sources. This section focuses on reading tabular data in **comma-separated value** (CSV) format, usually stored in a file with the extension `.csv`. In this format, each line of the file represents a record (row) of data, while each feature (column) of that record is separated by a comma:

```
name, weight, height
Ada, 64, 135
Bob, 74, 156
Chris, 69, 139
Diya, 69, 144
Emma, 71, 152
```

Most spreadsheet programs, such as Microsoft Excel, Numbers, and Google Sheets, are just interfaces for formatting and interacting with data that is saved in this format. These programs easily import and export `.csv` files. But note that `.csv` files are unable to save the formatting and calculation formulas used in those programs—a `.csv` file stores only the data!

You can load the data from a `.csv` file into R by using the **`read.csv()`** function:

```
# Read data from the file `my_file.csv` into a data frame `my_df`
my_df <- read.csv("my_file.csv", stringsAsFactors = FALSE)
```

Again, use the `stringsAsFactors` argument to make sure string data is stored as a vector rather than as a *factor* (see Section 10.3.2 for details). This function will return a data frame just as if you had created it yourself.

> **Remember**: If an element is missing from a data frame (which is very common with real-world data), R will fill that cell with the logical value `NA`, meaning "**n**ot **a**vailable." There are multiple ways[a] to handle this in an analysis; you can filter for those values using bracket notation to replace them, exclude them from your analysis, or impute them using more sophisticated techniques.
>
> [a] See, for example, http://www.statmethods.net/input/missingdata.html

Conversely, you can *write* data to a `.csv` file using the **`write.csv()`** function, in which you specify the data frame you want to write, the filename of the file you want to write the data to, and other optional arguments:

```
# Write the data in `my_df` to the file `my_new_file.csv`
# The `row.names` argument indicates if the row names should be
# written to the file (usually not)
write.csv(my_df, "my_new_file.csv", row.names = FALSE)
```

Additionally, there are many data sets you can explore that ship with the R software. You can see a list of these data sets using the **`data()`** function, and begin working with them directly (try

`View(mtcars)` as an example). Moreover, many packages include data sets that are well suited for demonstrating their functionality. For a robust (though incomplete) list of more than 1,000 data sets that ship with R packages, see this webpage.[1]

10.3.1 Working Directory

The biggest complication when working with `.csv` files is that the `read.csv()` function takes as an argument a **path** to a file. Because you want this script to work on any computer (to support collaboration, or so you can code from your personal computer or a computer at a library), you need to be sure to use a **relative path** to the file. The question is: *relative to what*?

Like the command line, the R interpreter (running inside RStudio) has a **current working directory** from which all file paths are relative. The trick is that *the working directory is not necessarily the directory of the current script file!* This makes sense, as you may have many files open in RStudio at the same time, and your R interpreter can have only one working directory.

Just as you can view the current working directory when on the command line (using `pwd`), you can use an R function to view the current working directory when in R:

```
# Get the absolute path to the current working directory
getwd() # returns a path like /Users/YOUR_NAME/Documents/projects
```

You often will want to change the working directory to be your project's directory (wherever your scripts and data files happen to be; often the root of your project *repository*). It is possible to change the current working directory using the `setwd()` function. However, this function also takes an absolute path, so doesn't fix the problem of working across machines. You *should not* include this absolute path in your script (though you could use it from the console).

A better solution is to use RStudio itself to change the working directory. This is reasonable because the working directory is a property of the *current running environment*, which is what RStudio makes accessible. The easiest way to do this is to use the `Session > Set Working Directory` menu option (see Figure 10.2): you can either set the working directory `To Source File Location` (the folder containing whichever `.R` script you are currently editing; this is usually what you want), or you can browse for a particular directory with `Choose Directory`.

As a specific example, consider trying to load the `my-data.csv` file from the `analysis.R` script, given the folder structure illustrated in Figure 10.3. In your `analysis.R` script you want to be able to use a relative path to access your data (`my-data.csv`). In other words, you don't want to have to specify the **absolute path** (`/Users/YOUR_NAME/Documents/projects/analysis-project/data/my-data.csv`) to find this. Instead, you want to provide instructions on how your program can find your data file *relative* to where you are working (in your `analysis.R` file). After setting the session's path to the working directory, you will be able to use the **relative path** to find it:

```
# Load the data using a relative path
# (this works only after setting the working directory,
# most easily with the RStudio UI)
my_data <- read.csv("data/my-data.csv", stringsAsFactors = FALSE)
```

[1] **R Package Data Sets:** https://vincentarelbundock.github.io/Rdatasets/datasets.html

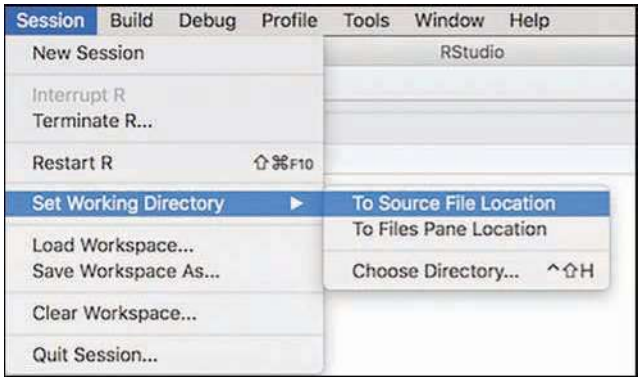


Figure 10.2 Use `Session > Set Working Directory` to change the working directory through RStudio.

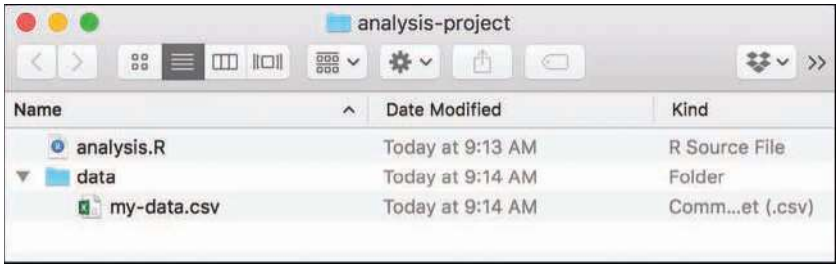


Figure 10.3 The folder structure for a sample project. Once you set the working directory in RStudio, you can access the `my-data.csv` file from the `analysis.R` script using the *relative path* `data/my-data.csv`.

10.3.2 Factor Variables

> **Remember**: You should always include a `stringsAsFactors = FALSE` argument when either loading or creating data frames. This section explains why you need to do that.

Factors are a data structure for *optimizing* variables that consist of a finite set of categories (i.e., they are categorical variables). For example, imagine that you had a vector of shirt sizes that could take on only the values `small`, `medium`, or `large`. If you were working with a large data set (thousands of shirts!), it would end up taking up a lot of memory to store the character strings (5+ letters per word at 1 or more bytes per letter) for each of those variables.

A factor would instead store a number (called a **level**) for each of these character strings—for example, `1` for `small`, `2` for `medium`, or `3` for `large` (though the order of the numbers may vary). `R` will remember the relationship between the integers and their labels (the strings). Since each number takes just 2–4 bytes (rather than 1 byte per letter), factors allow `R` to keep much more information in memory.

To see how factor variables appear similar to (but are actually different from) vectors, you can create a factor variable using **`as.factor()`**:

```
# Demonstrate the creation of a factor variable

# Start with a character vector of shirt sizes
shirt_sizes <- c("small", "medium", "small", "large", "medium", "large")

# Create a factor representation of the vector
shirt_sizes_factor <- as.factor(shirt_sizes)

# View the factor and its levels
print(shirt_sizes_factor)
# [1] small  medium small  large  medium large
# Levels: large medium small

# The length of the factor is still the length of the vector,
# not the number of levels
length(shirt_sizes_factor) # 6
```

When you print out the `shirt_sizes_factor` variable, R still (intelligently) prints out the labels that you are presumably interested in. It also indicates the levels, which are the only possible values that elements can take on.

It is worth restating: factors are not vectors. This means that most all the operations and functions you want to use on vectors will not work:

```
# Attempt to apply vector methods to factors variables: it doesn't work!

# Create a factor of numbers (factors need not be strings)
num_factors <- as.factor(c(10, 10, 20, 20, 30, 30, 40, 40))

# Print the factor to see its levels
print(num_factors)
# [1] 10 10 20 20 30 30 40 40
# Levels: 10 20 30 40

# Multiply the numbers by 2
num_factors * 2 # Warning Message: '*' not meaningful for factors
# Returns vector of NA instead

# Changing entry to a level is fine
num_factors[1] <- 40

# Change entry to a value that ISN'T a level fails
num_factors[1] <- 50 # Warning Message: invalid factor level, NA generated
# num_factors[1] is now NA
```

If you create a data frame with a string vector as a column (as happens with `read.csv()`), it will automatically be treated as a factor unless you explicitly tell it not to be:

```
# Attempt to replace a factor with a (new) string: it doesn't work!

# Create a vector of shirt sizes
shirt_size <- c("small", "medium", "small", "large", "medium", "large")

# Create a vector of costs (in dollars)
cost <- c(15.5, 17, 17, 14, 12, 23)

# Data frame of inventory (by default, stringsAsFactors is set to TRUE)
shirts_factor <- data.frame(shirt_size, cost)

# Confirm that the `shirt_size` column is a factor
is.factor(shirts_factor$shirt_size) # TRUE

# Therefore, you are unable to add a new size like "extra-large"
shirts_factor[1, 1] <- "extra-large"
# Warning: invalid factor level, NA generated
```

The NA produced in the preceding example can be avoided if the `stringsAsFactors` option is set to `FALSE`:

```
# Avoid the creation of factor variables using `stringsAsFactors = FALSE`

# Set `stringsAsFactors` to `FALSE` so that new shirt sizes can be introduced
shirts <- data.frame(shirt_size, cost, stringsAsFactors = FALSE)

# The `shirt_size` column is NOT a factor
is.factor(shirts$shirt_size) # FALSE

# It is possible to add a new size like "extra-large"
shirts[1, 1] <- "extra-large" # no problem!
```

This is not to say that factors can't be useful (beyond just saving memory)! They offer easy ways to group and process data using specialized functions:

```
# Demonstrate the value of factors for "splitting" data into groups
# (while valuable, this is more clearly accomplished through other methods)

# Create vectors of sizes and costs
shirt_size <- c("small", "medium", "small", "large", "medium", "large")
cost <- c(15.5, 17, 17, 14, 12, 23)
```

```
# Data frame of inventory (with factors)
shirts_factor <- data.frame(shirt_size, cost)

# Produce a list of data frames, one for each factor level
#   first argument is the data frame to split
#   second argument the data frame to is the factor to split by
shirt_size_frames <- split(shirts_factor, shirts_factor$shirt_size)

# Apply a function (mean) to each factor level
#   first argument is the vector to apply the function to
#   second argument is the factor to split by
#   third argument is the name of the function
tapply(shirts_factor$cost, shirts_factor$shirt_size, mean)
       # large medium  small
       # 18.50  14.50  16.25
```

While this is a handy use of factors, you can easily do the same type of aggregation without them (as shown in Chapter 11).

In general, the skills associated with this text are more concerned with working with data as vectors. Thus you should always use `stringsAsFactors = FALSE` when creating data frames or loading `.csv` files that include strings.

This chapter has introduced the data frame as the primary data structure for working with two-dimensional data in R. Moving forward, almost all analysis and visualization work will depend on working with data frames. For practice working with data frames, see the set of accompanying book exercises.[2]

[2] **Data frame exercises:** https://github.com/programming-for-data-science/chapter-10-exercises

This page intentionally left blank