

4

Reading and Exploring Complex Data

CONTENTS

4.1	Introduction	149
4.2	Lists	154
4.2.1	Subsetting Lists	157
4.2.1.1	Accessing An Element of a List	158
4.2.2	Applying Functions to Elements of a List	160
4.2.3	Exploring Rainfall on the Colorado Front Range	161
4.3	Reading Data into a Character Vector	166
4.3.1	A Study of Web Page Updates	166
4.4	Reading Data from a Web Page	171
4.4.1	World Records in the Men's 1500 meter	174
4.5	Reading <i>JSON</i> Formatted Data	175
4.6	Kiva	181
4.6.1	Loan Elements	182
4.6.2	The Payments	186
4.6.3	The Borrowers	186
4.6.4	Final Structure	186
4.7	Summary	187
4.8	Functions for Handling Complex Data Formats	187
4.9	Guided Practice	188
4.10	Exercises	189
	Bibliography	189

4.1 Introduction

Data do not always come in simple rectangular collections of numbers like in Chapter 2. At times the data have a nested structure where data values are at different levels, e.g., loans and payments on these loans, information about weather stations and daily rainfall recordings at the stations, and automobile accidents and the vehicles involved. When we have data at different levels, we typically have a few options for organization. The choice of how to organize the data can depend on how we plan to work with and analyze them. We may even decide to organize the data in two different formats because one format is more conducive to a particular kind of analysis than another. We saw this in Section 2.3.3 with the Clinton emails. The first table we examined had 1 row for each email. Then later in Section 2.6.1, we worked with 2 additional files. One table had 1 record for each person who sent or received one of these emails. The other table had 1 record for each recipient of an email so some emails corresponded to 1 record in this file, i.e., when the email had 1 recipient, and others corresponded to 2 or more records when the email was sent to multiple individuals.

In this chapter, we encounter data from several applications. What they all have in common is that the data do not immediately translate into a data frame or table. In these examples, we demonstrate how to create multiple tables, a collection of numeric vectors of different lengths, and a collection of vectors and data frames.

Reading Complex Data

When data can not be arranged in a simple rectangular form, we must decide how to organize them, i.e., what structure to use to contain the data. This organization depends on several considerations, i.e.,

- What are our plans for analyzing the data?
- What is a convenient structure for this analysis?

Often in these situations, the data are at different levels of granularity. For each level, we need to identify the observations and variables, i.e., we consider the following:

- What entity corresponds to a row? (the data may contain different entities so we can have more than one answer to this question)
- What are the variables?

The structure of the data may be a collection of varying-length vectors, multiple related data frames, or a combination of these. When this occurs, we need to know:

- What is the variable (or variables) that connects the observations across tables?

Example 4-1 Rainfall in the Colorado Front Range

Climate scientists and statisticians study historical patterns in rainfall in an effort to make predictions of extreme weather events. For example, statisticians at the National Center for Atmospheric Research (NCAR) in Boulder, Colorado have worked on the problem of estimating rainfall for a future storm that has a 1% chance of occurring in a year; this is called the 100-year event. To help in this effort, they study recordings of daily precipitation at dozens of weather stations in the Front Range of Colorado. Their work involves collaborations with researchers at other locations, and they share their data in a format that is ready for analysis in *R*. One example is the *FrontRangeWeather.rda* file. An *.rda* file is a binary file so it cannot be viewed in a plain text editor, but we can easily load it into our *R* work space and begin analysis. We load the contents of *FrontRangeWeather.rda* with

```
load("FrontRangeWeather.rda")
```

The functions introduced in Chapter 1 can help us examine the objects in the file. First, we list them with

```
ls()

[1] "FrontRangeWeather"
```

We see the *rda* file contains one object, *FrontRangeWeather*. We determine its class with

```
class(FrontRangeWeather)

[1] "list"
```

`FrontRangeWeather` is a list. This is a different kind of data structure from a data frame. We call `length()` and `names()` to learn more about `FrontRangeWeather`:

```
length(FrontRangeWeather)
```

```
[1] 3
```

```
names(FrontRangeWeather)
```

```
[1] "days"      "precip"     "stations"
```

We find that `FrontRangeWeather` has 3 elements named `days`, `precip`, and `stations`. Lists are more complex and richer in structure than the vectors, data frames, and arrays we worked with in Chapter 2. In order to examine these data, we need to understand more about the list structure in *R*. This is the topic of Section 4.2.

■

Example 4-2 World Record in the Men's 1500 meter

Wikipedia contains a vast amount of information, some of which is presented in table format. One example is the world records in the men's 1500 meter race (see Figure 4.1 and [4]). This table has 5 columns and a row for each time the world record was broken. Notice that this is not a typical data table because, for example, the third column contains three pieces of information: an image of the athlete's home country flag, the athlete's name, and a 3-letter country code in parentheses. Moreover, the table is embedded within a Web page (see Figure 4.2) and there are several other tables on this page. Clearly we need to carry out some special processing to access the records in this table.











Time ↕	Auto ↕	Athlete ↕	Date ↕	Place ↕
3:55.8		 Abel Kiviat (USA)	1912-06-08	Cambridge, Massachusetts, USA
3:54.7		 John Zander (SWE)	1917-08-05	Stockholm, Sweden
3:52.6		 Paavo Nurmi (FIN)	1924-06-19	Helsinki, Finland
3:51.0		 Otto Peltzer (GER)	1926-09-11	Berlin, Germany
3:49.2		 Jules Ladoumegue (FRA)	1930-10-05	Paris, France
3:49.2		 Luigi Beccali (ITA)	1933-09-09	Turin, Italy
3:49.0		 Luigi Beccali (ITA)	1933-09-17	Milan, Italy
3:48.8		 Bill Bonthron (USA)	1934-06-30	Milwaukee, USA
3:47.8		 Jack Lovelock (NZL)	1936-08-06	Berlin, Germany
3:47.6		 Gunder Hägg (SWE)	1941-08-10	Stockholm, Sweden
3:45.8		 Gunder Hägg (SWE)	1942-07-17	Stockholm, Sweden

Figure 4.1: Screen Shot of 1500 meter World Records Wikipedia Table. *This screen shot shows a portion of the Wikipedia page that contains the results for the world records in the men's 1500 meter race. See Figure 4.2 for a screen shot of the top of the page.*

A first impulse might be to copy and paste the table from our Web browser into a text editor and then 'hand' edit the text into a format that we can easily read into *R* with one of the functions we used to read rectangular data in Chapter 2. However, there are problems with this approach: it is error prone and not computationally reproducible. For example, we might accidentally miss the last 2 rows of the table when we copy it. Additionally, if in our editing we unknowingly delete or change a few values, then when we or someone else later

detects the problem, we are not able to retrace our steps to see where the error arose. This approach does not result in a record of the sequence of steps followed to produce the data frame. That is, we cannot check code that we used to create the table, fix that code, and repeat the process. Similarly, if we later decide to alter the processing of the table because, say, the format has changed slightly, then we cannot update our code to reflect the new processing and simply run it to recreate the data frame.



Figure 4.2: Screen Shot of 1500 meter Wikipedia page. This screen shot shows the top of the Wikipedia Web page (https://en.wikipedia.org/wiki/1500_metres_world_record_progression) that contains a table of the world records in the men's 1500 meter race. See Figure 4.1 for a screen shot of the table. This page contains multiple tables, including world records for women and pre-IAAF (International Association of Athletics Federations) records.

We briefly introduce the *HTML* format in Section 4.4, and in Section 4.4.1, we extract this table from the Web page using the `readHTMLTable()` function and create a data frame for analysis. *HTML* is covered in more detail in Chapter 12.

Example 4-3 Caching Web Page Updates

Internet search engines, such as Google, Bing, and Ask, keep copies of Web pages so that when you make a query, they can quickly search their stored pages and return their findings to you. These saved pages are called a Web cache. In order to keep the cache up to date, Web pages need to be visited regularly and the cache updated. To try to determine how often Web pages change and how often a site should be visited to keep the cache fresh, the updating behavior of 1,000 Web pages was studied. Each of these pages was visited every hour for 30 days. The page was compared to the previous visit, and if it had changed, the cache was updated and the time of the visit was recorded.

For example, below are records for the first 4 of these 1,000 Web pages. The 3 pieces of information provided in each row are: the type of domain of the page, the total number

of visits made to the page, and a comma-separated set of visits on which a change was observed. (These 3 pieces of information are separated by tabs.)

```
net      378      35,134,155,157,177,204,314,315,319,350,366,369,371
jp       707      552,604,672
com      418      1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,...,417,418
net      369      30,36,45,65,88,154,157,166,169,197,199,...,367,369
```

We see that the domain of the 1st page is .net, which stands for a network technology; the next domain is .jp, which means the domain is registered in Japan; and the 3rd record is from a .com site.

According to the study design, each page was to be visited hourly for 30 days, which means that 24×30 , or 720 visits, were planned. Since the first visit has no previous version of the page to compare against, we have only 719 subsequent visits (or revisits) where we can observe a change. As noted above, the second value in a record provides the actual number of visits made to the page. These can differ from 719 if visits are unsuccessful. Our first 4 sites were successfully visited 378, 707, 418, and 369 times, respectively.

The last piece of information provides the ‘times’ when a change is observed. Notice that the site in the 2nd row had changed on 3 visits, the 552nd, 604th, and 672nd. In contrast, the 3rd site was visited 418 times, and from the partial display of times when a change was observed, it appears that the site had changed between every visit. If we think of a Web page as a record or observation, then our records do not form a typical rectangular shape with a fixed set of columns corresponding to specific variables. This situation often arises with, e.g., medical records where patients typically have a varying number of visits to a clinic. The notion of non-rectangular data impacts how the data are stored in the computer and how we write code to analyze the data. We address these issues in Section 4.3.1. ■

Example 4-4 Kiva Loans

Kiva [2] is a nonprofit organization whose mission is to “connect people through lending to alleviate poverty.” Essentially, Kiva allows people like you and me to volunteer small amounts of money to loan to people around the world who use it for such things as starting a small business. Like many Internet-based organizations, Kiva makes their data available via a Web service. This includes providing links for downloading the entire database of loans and lenders in both *JSON* and *XML* format. Open data such as Kiva’s create wonderful opportunities for statisticians, social scientists, and the interested public to explore and learn about the world of micro-financing. Moreover, the possibility of merging these data with other publicly available data sets, creates an even greater potential for discovery and understanding.

The API for Kiva’s Web services is described at <http://build.kiva.org/api>. (API stands for ‘a programming interface’, this interface describes how to access the data programmatically.) There we find that the information about the most recent loans is available at <http://api.kivaws.org/v1/loans/newest.json>. The ‘json’ at the end of the URL indicates that this is *JSON* content (another type of plain text file format). When we visit this page, we find the following, which we have formatted to make it easier to see the structure of the information,

```
{ "paging":{
  "page":1, "total":6025, "page_size":20, "pages":302
},
  "loans":[
    { "id":984656,
```

```

    "name": "Khamheang Group",
    "description": { "languages": [ "en" ] },
    "status": "fundraising",
    "funded_amount": 0,
    "basket_amount": 0,
    "image": { "id": 2032579, "template_id": 1 },
    "activity": "Home Appliances",
    "sector": "Personal Use",
    "themes": [ "Water and Sanitation" ],
    "use": "to purchase TerraClear water filters
           so they can have access to safe drinking water.",
    "location": { "country_code": "LA", "country": "Lao PDR",
                  "town": "Laos",
                  "geo": { "level": "town", "pairs": "18 105", "type": "point" } },
    "partner_id": 393,
    "posted_date": "2015-11-28T22:30:08Z",
    "planned_expiration_date": "2015-12-28T22:30:08Z",
    "loan_amount": 325,
    "borrower_count": 7, "lender_count": 0,
    "bonus_credit_eligibility": false,
    "tags": []
  }, .....
}]

```

Notice the use of curly and square brackets, commas, and colons to delimit the various pieces of information. For example, this *JSON* “object” has two fields, `paging` and `loans`. The `paging` field is itself an object with four fields, which are all numeric values. The `loans` field is an array of objects. Each element in this array describes a loan with fields for such things as the loan identifier, the name and location of the lendeer, the specific purpose of the loan, and the amount being sought.

We want to create a data structure that contains information about each loan, but it is not immediately obvious whether or not we can map the *JSON* content into a data frame. We do not know whether or not all the loans have the same fields of information. For example, do all loans have a `country_code` element associated with them? (It turns out they don’t.) Also, each loan contains a record of the payments made. If multiple payments have been made, then the information about each payment forms a ragged array where different loans have a different number of payment fields. More details about the *JSON* format would be helpful in determining how to organize the data in *R*. We cover this in Section 4.5, and we tackle the challenge of organizing these data into convenient data structures in Section 4.6.

■

Lastly, some data are either so large or so complex that they require a more programmatic approach to handling them. In Chapter 14 we consider situations that require programming to acquire and clean the data, in Chapter 9 we work with data in relational databases, and in Chapter 12 we examine data available through Web scraping, forms and APIs.

4.2 Lists

In Chapter 1 and Chapter 2 we examined three kinds of data structures, vectors, data frames, and arrays. We saw that vectors are ordered collections of values of the same type. Arrays are similar to vectors except that they also have shape information, e.g., an r -by- c matrix is a two-dimensional array which we can also treat as a vector of length $r \times c$. The data frame is an ordered collection of vectors of the same length and possibly different types. However, there are many situations where we want to group values together that are not necessarily all of the same length, or not necessarily all vectors. In Q.4-3 (page 152), we saw that each Web page has a record of the visits on which the page changed, and the number of these varies with the page. Also, we will soon see that the information provided for the `FrontRangeWeather` includes a data frame with one row for each station and an object with daily recordings of precipitation at each station. The weather station details and the daily precipitation are very different kinds of objects, and `FrontRangeWeather` is a container that allows us to group these different kinds of elements together. This container is a list data structure. The list structure is also useful as a container for the Web cache data..

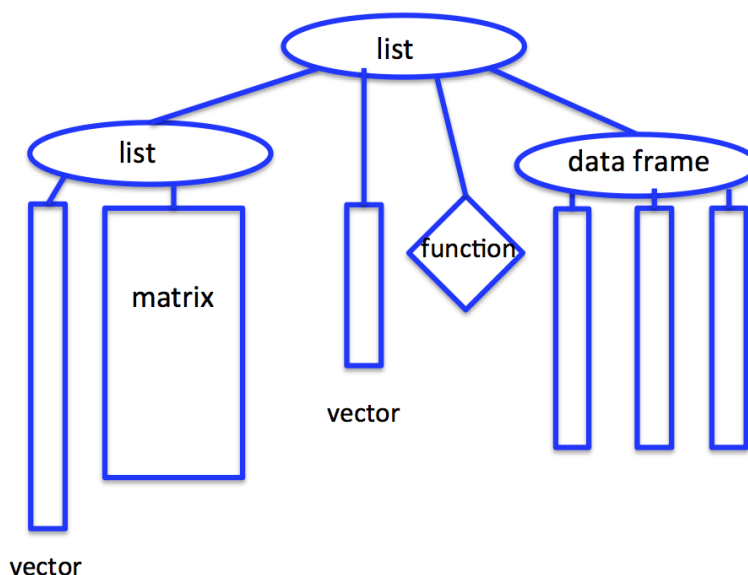


Figure 4.3: Conceptual Diagram of a List. *This diagram provides the basic structure of a simple list. This list has 4 elements: a list, a vector, a function, and a data frame. The list element contains a vector and a matrix. Notice that the data frame contains 3 vectors.*

Figure 4.3 provides a conceptual diagram for a simple list. This list consists of 4 elements: a list, a vector, a function, and a data frame. The 1st element (a list) contains two elements—a vector and a matrix. The 4th element (a data frame) contains 3 vectors which necessarily are all the same length.

We have created a list with the same structure shown in Figure 4.3 for us to explore. We load it into our session with

```
load("exampleList.rda")
```

Many of the functions that we used in Chapter 1 to glean information about vectors and data frames also accept a list as input. That is, we find the length of the list, the names of its elements (if they have names) and confirm its class with calls to `length()`, `names()` and `class()` as follows:

```
class(aList)

[1] "list"

length(aList)

[1] 4

names(aList)

[1] "listToo" "aVec"      "aFunc"      "aDF"
```

We see from these function calls that `aList` has 4 elements, which have been given names that suggest their types, i.e., the first element is another list, the second is a vector, the third is a function and the fourth is a data frame.

We print the contents of `aList` to the console to confirm that this list has the same basic structure as the list in Figure 4.3.

```
aList

$listToo
$listToo$aVec
[1] 1 3 5 7

$listToo$aMat
      [,1] [,2]
[1,]   10   12
[2,]   14   16
[3,]   18   20

$aVec
[1] "a" "b" "c" "d"

$aFunc
function (x)
{
  sum(x^2)
}

$aDF
  id height sex
1  1     60  f
2  2     72  m
3  3     66  f
4  4     70  m
```


Notice that there is a vector within `listToo` and it is called `aVec`, which is the same name as the second element of `aList`. However, these two vectors are not the same objects. One is an element of `aList` and the other is an element of `listToo`, which in turn is an element of `aList`.

The `$`-signs that are printed to the console preceding each element's name suggests that we can access the contents of a list in a similar fashion to vectors in a data frame. Accessing elements of a list and computing subsets on lists is the topic of the next section.

Lastly, we can also use the `str()` function, which is short for structure, to display the structure of `aList` in a compact fashion, e.g.,

```
str(aList)

str(aList)
List of 4
 $ listToo:List of 2
  ..$ aVec: num [1:4] 1 3 5 7
  ..$ aMat: num [1:3, 1:2] 10 14 18 12 16 20
 $ aVec    : chr [1:4] "a" "b" "c" "d"
 $ aFunc   :function (x)
  ..- attr(*, "srcref")=Class 'srcref'
        atomic [1:8] 1 14 1 37 14 37 1 1
  .. .. - attr(*, "srcfile")=Classes 'srcfilecopy',
        'srcfile' <environment: 0x10e0cf4d8>
 $ aDF     :'data.frame':    4 obs. of  3 variables:
  ..$ id    : int [1:4] 1 2 3 4
  ..$ height: num [1:4] 60 72 66 70
  ..$ sex   : Factor w/ 2 levels "f","m": 1 2 1 2
```

The return value contains a lot of information. We see `aList` is a list of 4 elements, and the names, types, and some of the contents of these elements.

4.2.1 Subsetting Lists

With lists, we can use the same style of subsetting as we do for vectors. In fact, a list is a special type of vector. The vectors we have worked with in Chapter 1 are actually atomic vectors, meaning all elements are of the same primitive type. A list is a non-atomic vector in that its elements can be any type of object.

We can subset by position, exclusion, logical, and name. For example, we compute a list of the 2nd and 4th elements of `aList` using the position of these elements in the list with:

```
aList[c(2, 4)]

$aVec
[1] "a" "b" "c" "d"

$aDF
  id height sex
1  1     60  f
2  2     72  m
3  3     66  f
4  4     70  m
```

Similarly, we arrive at the same subset when we use exclusion, logical, and names with the following computations, respectively,

```
aList[ -c(1, 3)]
aList[c(FALSE, TRUE, FALSE, TRUE)]
aList[c("aVec", "aDF")]
```

4.2.1.1 Accessing An Element of a List

When we subset a list, the return value is a list. Even when we ask for one element in the list, we obtain a list as the return value. For example, when we ask for the 2nd element of `aList`, we get a list of length 1 in return, e.g.,

```
aList[2]

$aVec
[1] "a" "b" "c" "d"
```

```
class(aList[2])

[1] "list"
```

This is consistent with subsetting vectors. That is, when we take a subset of a vector, the return value is a vector, even if it has only one element. However, at times, we want to drop the list container and work directly with the element.

If the elements have names, we can use `$`-notation to work directly with the element, e.g.,

```
aList$aVec

[1] "a" "b" "c" "d"
```

Notice that the return value is printed to the console in a slightly different format, which indicates that it is a vector and not a list. We can operate on this vector, just as we do with other vectors. For example we can find its length and class with

```
length(aList$aVec)

[1] 4

class(aList$aVec)

[1] "character"
```

Also, we can use subsetting techniques to access elements in this vector, e.g., we can subset by position to change the 3rd element in `aList$aVec` to "z" with

```
aList$aVec[3] = "z"
```

Now `aList$aVec` has values "a" "b" "z" "d".

We can even invoke the function `aFunc()` in `aList` with the help of the `$`-notation, e.g., `aList$aFunc(1:3)`. Notice the return value (14) is the sum of the squares of 1, 2, and 3.

Another way to extract an individual element from a list uses double-square brackets, e.g.,

```
aList[[2]]
```

```
[1] "a" "b" "z" "d"
```

Again, we can access subsets of this vector with, e.g.,

```
aList[[2]][3:4]
```

```
[1] "z" "d"
```

Notice that with `aList[[2]][1:2]` we obtain a vector of length 2, which consists of the 3rd and 4th elements of the 2nd element of `aList` (which is a vector). We can also use double square brackets to call `aFunc()`, e.g., `aList[["aFunc"]](1:3)` and `aList[[3]](1:3)` both return 14.

We provide a few more examples of subsetting lists with double-square brackets and `$`-signs to help solidify our understanding of this notation. Recall that the first element of `aList` is also a list and its name is `listToo`. Compare the return values from the following commands to help check your understanding of subsetting lists.

```
aList[[1]][1] ❶
```

```
$aVec
```

```
[1] 1 3 5 7
```

```
aList[[1]][[1]] ❷
```

```
[1] 1 3 5 7
```

```
aList$listToo[[1]] ❸
```

```
[1] 1 3 5 7
```

```
aList$listToo$aVec ❹
```

```
[1] 1 3 5 7
```

```
aList[[1]][[2]] ❺
```

```
  [,1] [,2]
```

```
[1,]   10   12
```

```
[2,]   14   16
```

```
[3,]   18   20
```

```
aList[1][[2]] ❻
```

```
Error in aList[1][[2]] : subscript out of bounds
```

```
aList[1][2] ❼
```

```
$<NA>
```

```
NULL
```

- ❶ The double square bracket returns the first element in the list, which is `listToo`. Then, the second subset operation, i.e., the subset with the single square bracket, returns a list with one element, `aVec`.
- ❷ The first use of double square brackets in this expression returns the `listToo` list. Then the second set of double square brackets returns the first element in `listToo`, which is the vector `aVec`.

- ③ When the elements of the list have names, we can use the `$`-notation to access them. This expression is equivalent to the previous expression that uses two sets of double-square brackets, and both are equivalent to `aList[[1]]$aVec`.
- ④ We also can chain `$`-signs together. In this case, we again get the vector `1 3 5 7` as return value.
- ⑤ This sequence of two sets of double-square brackets returns the 2nd element of the 1st element of our list, which is the matrix `aMat`.
- ⑥ The first set of brackets in this expression are single so the return value is a list of length 1. The second set of brackets are double so the return value is the 2nd element of the list. However, this element does not exist because we have a list of length 1, so we get an error.
- ⑦ In contrast to the previous expression, these two sets of single square brackets returns a list of length 1. However, the element in this list is `NULL`.

A Data Frame is a List

It may have already occurred to you that data frames have many similarities to lists. In fact, a data frame is a special case of a list where all the elements are vectors of the same length. This means that we can use the double-square bracket form of subsetting with data frames, e.g., `aList$aDF[["sex"]]` retrieves the vector called `sex` in our example data frame. Additionally, we can compute a subset of the columns of the data frame with, e.g., `aList$aDF[2:3]`. We did not include a comma within our square bracket so *R* interprets this subset command as applying to the columns, i.e., to the elements of the list/data frame. The return value is a 2 column data frame that contains all rows from the original data frame.

The List

The list is an ordered container of heterogeneous objects. That is, it is a vector of heterogeneous objects. A list can include as elements a combination of vectors, multi-dimensional arrays, data frames, lists, functions, etc.

A subset can be computed on a list by position, exclusion, logical, name, and all.

A single element can be extracted from a list with double-square brackets, i.e., `[[`. This is different from a subset of 1 element of a list, which is a list of length 1. If the elements in the list are named, then `$`-notation can be used to extract one element from a list, e.g., `aList$aVec`.

The subset operators can be applied in sequence to compute a subset of elements from an element of a list, e.g., `aList[[1]]$aMat[2, 1]` returns the value in the 2nd row and 1st column of `aMat`, which is an element of the 1st element of `aList`.

The `lapply()` and `sapply()` functions apply a function, supplied as an argument, to each element of a list. The return value is a list, or, with `sapply()` a vector is returned if the return value can be reduced to a vector.

4.2.2 Applying Functions to Elements of a List

As with data frames, we can apply a function to each element of a list. We can use the function `lapply`, which stands for “list apply”. We can also use `sapply()`, which performs the same operations but simplifies the return value to a vector when possible. For example, we can find the mean of each element in the list `listToo` with

```
lapply(aList$listToo, mean)
```

```
$aVec
[1] 4
```

```
$aMat
[1] 15
```

Notice that the mean of the matrix is the average of all elements in the 3 by 2 matrix. Since the return values are single numerics, we can use `sapply()` to “simplify” the return value with

```
sapply(aList$listToo, mean)
```

```
aVec aMat
4    15
```

Additionally, we can find the class of each element in `aList` with

```
sapply(aList, class)
```

```
listToo      aVec      aFunc      aDF
"list"      "character"  "function" "data.frame"
```

We obtain the length of each element with

```
sapply(aList, length)
```

```
listToo      aVec      aFunc      aDF
2            4          1          3
```

We now have the tools and understanding of the list structure to examine and explore the *Rda* file introduced in Q.4-1 (page 150).

4.2.3 Exploring Rainfall on the Colorado Front Range

Let’s begin our exploration of the Front Range weather data by getting a better understanding of its structure. After we load the data into *R*, we can call `class()` to find that `FrontRangeWeather` is a list, as mentioned in Q.4-1 (page 150). That is,

```
load("FrontRangeWeather.rda")
class(FrontRangeWeather)
```

```
[1] "list"
```

Earlier, we found the length and names of the elements in `FrontRangeWeather` with calls to `length()` and `names()`, e.g., `length(FrontRangeWeather)` returns 3 and `names(~FrontRangeWeather)` returns:

```
[1] "days"      "precip"     "stations"
```

We continue our investigation of the contents of `FrontRangeWeather` and apply functions to its elements. For example, we can find the class of each of the 3 elements in `FrontRangeWeather` with

```
sapply(FrontRangeWeather, class)
      days      precip      stations
"list"      "list" "data.frame"
```

Both `days` and `precip` are lists and `stations` is a data frame. We can find the length of each of these objects by applying `length()` to each element of `FrontRangeWeather` with

```
sapply(FrontRangeWeather, length)
      days  precip stations
      56     56         4
```

We find that both `days` and `precip` have length 56 and `stations` has length 4. Recall that the length of a data frame is the number of columns, or vectors in the data frame. The `str()` function provides this information too, but it is not as succinct as we might like because information is given for all 56 vectors in `days` and `precip`.

Let's dig a little deeper to learn more about the `stations` data frame. Given it has only 4 variables, we can use `head()` to view the first few rows of the data frame with

```
head(FrontRangeWeather$stations)
  station   lon   lat elev
1 st050183 -105.53 40.22 8450
2 st050263 -105.88 39.00 8920
3 st050712 -104.32 38.87 6040
4 st050843 -105.27 40.03 5420
5 st050945 -104.33 40.65 4880
6 st051179 -104.13 39.75 5150
```

This data frame has information about the weather stations, including location (latitude and longitude) and elevation, as well as an identifier for the stations.

We can explore these data graphically by plotting latitude against longitude. We provide some context to such a plot by placing the stations on a map that includes the Colorado state boundary, and we color the points according to elevation. Our map appears in Figure 4.4.

We made this map with the `map()` function in the `maps` package as follows:

```
library(maps)
map('state', fill=TRUE, col="gray95",
    xlim = c(-110, -101), ylim = c(36, 42))

elevCut = cut(FrontRangeWeather$stations$elev, breaks = 7)
rcolors = rainbow(n = 7, alpha = 1)

with(FrontRangeWeather$stations,
     points(x = lon, y = lat, pch = 19, col = rcolors[elevCut]))
```

Note that when we call `map()`, we specify a range for the latitude and longitude values that are a bit larger than Colorado so the map shows the boundaries with the neighboring states. Then we add points to this map with `points()`. These are placed at the latitude and longitude of the 56 weather stations. We discretize `elev` (with the `cut()` function) and use `elevCut` to select a color from the rainbow palette (in `rcolors`). Red represents the lowest and blue the highest elevation with orange and yellow falling in between.

From the map we see where the Front Range is within the state of Colorado. Also, the colors indicate that lower elevations are in the eastern region of the Front Range. This makes sense because the Rocky mountains are in the western part of the state and the eastern part corresponds to the high plains.

Now that we have a much better picture of where the weather stations are located, let's further explore the other two elements of `FrontRangeWeather`: `days` and `precip`. We

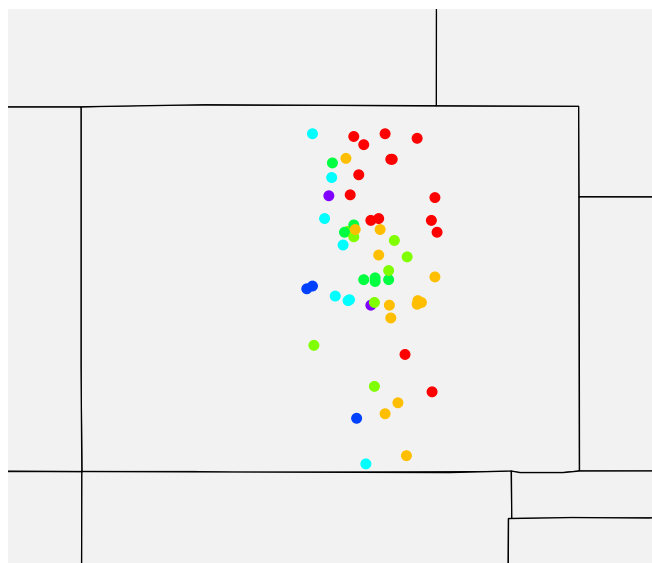


Figure 4.4: Location of Weather Stations in the Colorado Front Range. *The points on this map of Colorado show the locations of the weather stations in `FrontRangeWeather`. The color of the points indicates the elevation of the station. Red represents the lowest elevations, blue the highest, and orange and yellow in between. It is evident from the locations of these points that the Front Range corresponds to a north-south swath of the state, and the higher elevations are in the western portion nearest the Rocky mountains.*

now know that both `days` and `precip` are lists of 56 elements. This count (56) corresponds to the number of weather stations in `station`. You can check this with, e.g., `dim(FrontRangeWeather$stations)`. To find the class of these elements we use `sapply()`, e.g.,

```
sapply(FrontRangeWeather$days, class)

st050183 st050263 st050712 st050843 st050945 st051179
"numeric" "numeric" "numeric" "numeric" "numeric" "numeric" ...
```

These are all numeric vectors, and these vectors have names that match the values in the `station` variable in the `stations` data frame. We check the lengths of these vectors with `sapply(FrontRangeWeather$days, length)` and find that they have different lengths, e.g. the first station has 9878 measurements, the 2nd has 6751, and the 3rd has only 3959. When we apply `class()` and `length()` to the elements of `precip`, we find that its 56 elements also are numeric vectors, and their lengths match the the lengths of the vectors in `days`. That is,

```
all(sapply(FrontRangeWeather$days, length) ==
    sapply(FrontRangeWeather$precip, length))
```

returns TRUE.

We have enough information to sketch a conceptual diagram of `FrontRangeWeather` (see Figure 4.5). From this diagram we see that `FrontRangeWeather` has three elements, two lists (`days` and `precip`) and a data frame (`stations`). The data frame has 56 rows, corresponding to the 56 weather stations, where precipitation measurements are recorded, and 4 variables providing the latitude, longitude, elevation, and station identifier. The two lists (`days` and `precip`) each contain 56 numeric vectors, one for each weather station. Their lengths differ according to the number of days on which a station recorded precipitation.

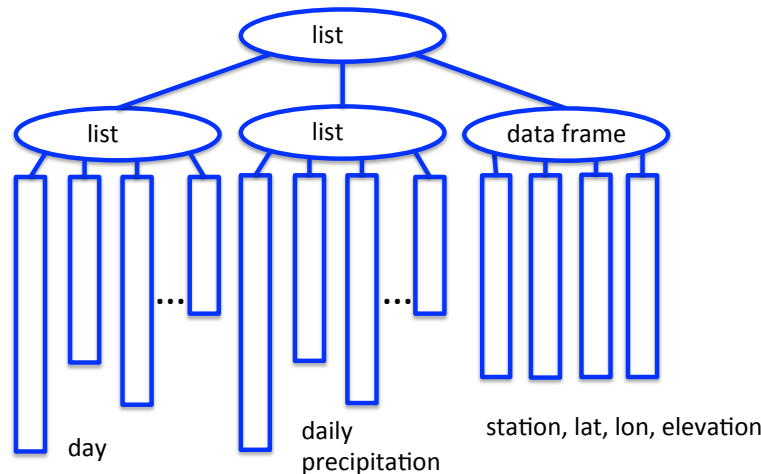


Figure 4.5: Conceptual Diagram of Colorado Front Range Data. *This diagram provides a sketch of the structure of `FrontRangeWeather`. This object is a list with 3 elements, which are two lists and a data frame. The data frame contains information about each of the 56 weather stations (rows), including the station identifier, latitude, longitude, and elevation. For each of the stations, there is a numeric vector in the two lists, i.e., each list contains 56 numeric vectors. These vectors contain the date (in `day`) and the amount of precipitation recorded on that day (in `precip`).*

Let's examine the first few values for the first station with

```
head(FrontRangeWeather$days[[1]])
[1] 1948.585 1948.587 1948.590 1948.593 1948.596 1948.598
```

These numeric values have the format `year + days/365` so, e.g., January 1, 1950 is `1950.003` and February 20, 1950 is `1950 + (31 + 20)/365` or `1950.14`.

The precipitation recorded on the first 6 days at this weather station are

```
head(FrontRangeWeather$precip[[1]])
[1] 0 10 11 1 0 0
```

These measurements are recorded in 100ths of an inch so a value of 11 is 0.11 inches of rain. Let's examine with a line plot the relationship between precipitation and date for the 1st station. We make this plot with

```
with(FrontRangeWeather,
     plot(precip[[1]] ~ days[[1]], type = "l",
          xlab = "Date", ylab = "Precip (100s inch)"))
```


The `with()` function is a convenience function to save us some typing when we specify the elements in a list of data frame. For example, the following expression: `plot~(FrontRangeWeather$precip[[1]] ~ FrontRangeWeather$days[[1]], ...)` is equivalent to the above.

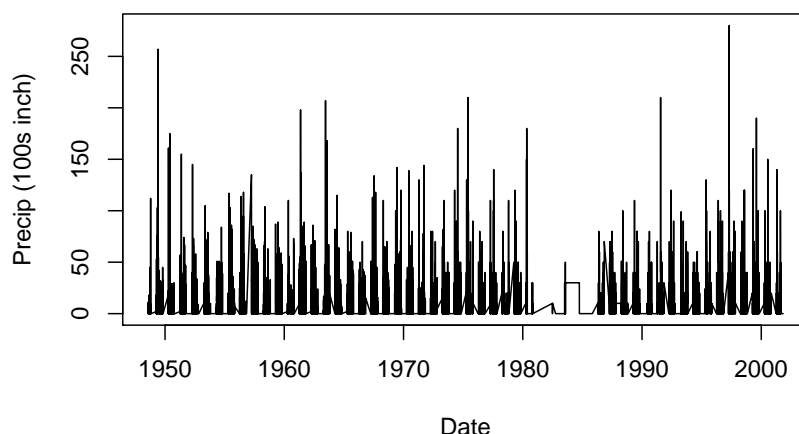


Figure 4.6: Precipitation Recorded at One Weather Station. *This line plot of the daily precipitation (rainfall) measured at the 1st weather station in `FrontRangeWeather` has a gap where only a few measurements were recorded between 1980 and 1985. Additionally, there are regular annual gaps over the entire range of observations. These are due to the winter months when rainfall was not measured.*

The plot appears in Figure 4.6. We make a few observations: a) at the time these data were published, this weather station had been in operation for over 50 years; b) there are very few measurements between 1980 and 1985, which indicates either the equipment/station was not in operation or those measurements were lost; c) there appear to be regular annual gaps in precipitation; d) there are a few spikes in precipitation indicating a few large storms with over 2.5 inches of rainfall.

Let's examine these gaps in operation for all of the stations in more detail. To do this, we make a specialized dot chart. In this plot, we construct a row of dots for each station. The dots are placed at the dates that a recording was made, even if the measurement was 0. We use the `stripchart()` function to make the plot as follows:

```
plot(c(1948, 2002), c(1, 56),
     xlab = "", ylab = "Station", type = "n", axes = FALSE)
stripchart(FrontRangeWeather$days, add = TRUE,
           col = "blue", pch = ".")
axis(1)
```

We see in Figure 4.7 narrow vertical white stripes. These stripes correspond to a consistent lack of measurements; in other words, all stations are missing data for these days. This occurs because we do not have precipitation measurements for the winter months when the precipitation is in the form of snow and there is not a risk of flooding. Longer horizontal

white spaces indicate that a station is not in operation. We confirm that the first row has a white region in the early to mid 1980s. We also see that many stations have not been in operation for 50 years.

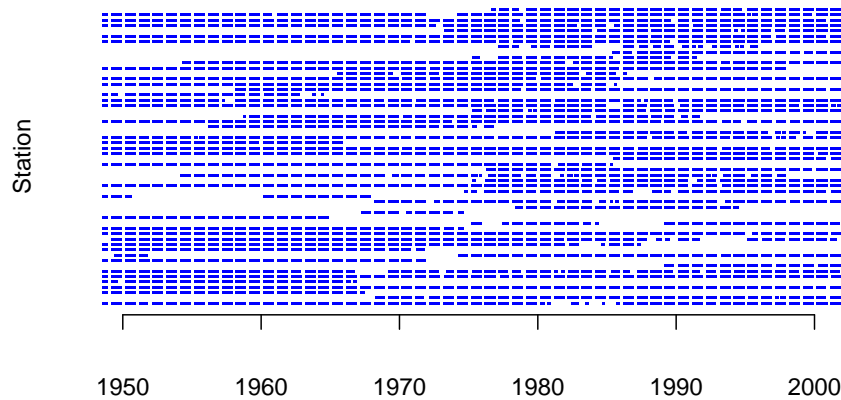


Figure 4.7: Dates of Operation of the Weather Stations. *This strip chart has a blue dot for each measurement taken at each weather station. All dots for one weather station appear on the same horizontal line. The dots for a station are placed at the date when the recording was made. The bottom row of dots corresponds to the 1st weather station. The gap from 1980 through 1985 is apparent from the horizontal gap in the 1st row. The plot has white vertical stripes which make clear the regularity of the lack of measurements in the winter months for all stations.*

We have successfully determined the structure of `FrontRangeWeather` and made a cursory examination of the values to learn about the units of measurement, range of values, and missing-ness of the data. We are ready to more carefully analyze the data, but that is not our goal here.

4.3 Reading Data into a Character Vector

We have the ability to read text data into *R* as a character vector, where each line in the input file becomes one string (element) of the character vector. Then, the length of the vector equals the number of lines read from the file. This approach can be useful when the text needs special processing that is not readily accomplished through the use of functions such as those described in Chapter 2. That is, we can write specialized code to process the strings and create the desired data structure. Of course, we do not want to take this approach unless the more standard approaches are inadequate. The functions such as `read_delim()`, `read_table()`, etc. tend to be more robust than our code and we should use them.

The `readLines()` function reads text into *R* and returns a character vector. In addition to supplying the ‘connection’ (such as a file name or *URL*), we can also specify the number of lines to read with the *n* argument. The following example provides a demonstration of how to use `readLines()` and why we may need to read the contents of a file into a character vector. The processing of the strings in this example are simple. More complex processing of strings is described in Chapter 8 on regular expressions and string manipulation.

4.3.1 A Study of Web Page Updates

Recall from Q.4-3 (page 152) that the Web cache study consists of 3 pieces of information for each Web page: the domain of the page, the total number of visits made to the page, and a comma-separated set of visits when a change was observed. We re-display the first 4 of the 1000 records below for convenience.

```
net      378      35,134,155,157,177,204,314,315,319,350,366,369,371
jp       707      552,604,672
com      418      1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,...,417,418
net      369      30,36,45,65,88,154,157,166,169,197,199,...,367,369
```

As noted earlier, these 3 pieces of information are separated by tab characters in the file. We can use `read_delim()` and provide the tab character as the separator to read these data into *R* as a data frame, e.g.,

```
tryDF = read_delim("webCache.txt", delim = "\t")
dim(tryDF)
```

```
[1] 1000      3
```

The first two rows of this data frame are:

```
tryDF[1:2, ]
      V1  V2                                     V3
1 net 378 35,134,155,157,177,204,314,315,319,350,366,369,371
2  jp 707                                     552,604,672
```

Clearly the data are not in a convenient format for analysis because all of the Web page's changes are in a character string. This data frame is not a workable structure. A better structure is one where the changes are numeric vectors and since these vectors are of different lengths, we want a list of vectors. The domain and the number of visits can remain as a 1000 by 2 data frame, or it might be simpler to keep them as 2 separate vectors. If we want one container for all of 3 pieces of information, then it is simpler to access the domains and the number of visits as vectors in the list rather than as vectors within a data frame within the list. In this case the structure that we want is a list with three elements containing a character vector of domains, a numeric vector of the total number of visits to a page, and a list of 1000 numeric vectors, each containing the visits on which a change was observed. See Figure 4.8 for a diagram of this data structure.

We can continue to work with the `tryDF` data frame to create our desired list but we leave this approach as an exercise. Instead, we use the `readLines()` function to read the file into *R* as a character vector of length 1000. Each line in the file corresponds to a string in the return vector. We do this with

```
txt = readLines("webCache.txt")
```

We confirm that indeed we have a character vector of length 1000 with

```
class(txt)
[1] "character"

length(txt)
```

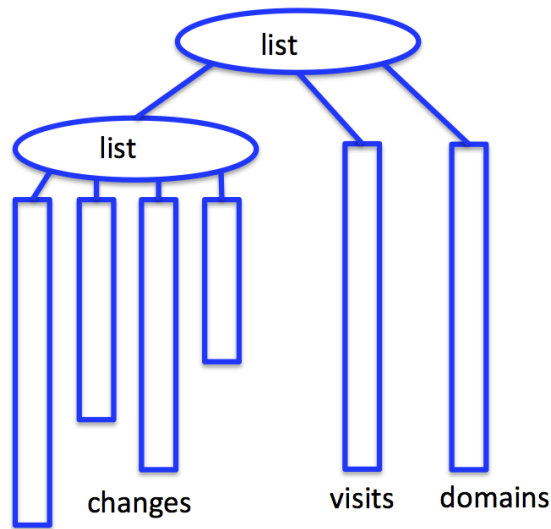


Figure 4.8: Conceptual Diagram of Web Cache Data. *This diagram provides a sketch of the desired structure for the Web cache data. The structure is a list with 3 elements: a list and 2 vectors. The **domains** vector is a character vector with 1,000 strings, each denoting the domain of the corresponding Web page. The **visits** vector is a numeric vector with the total number of successful visits to the page. The **changes** list is a collection of 1,000 numeric vectors. Each vector corresponds to a Web page, and the vector contains the times when a change was observed on the page.*

```
[1] 1000
```

We can split these strings up into the desired three pieces with a call to `strsplit()` as follows

```
els = strsplit(txt,"\\t")
```

This function returns a list, where each element is a character vector containing the sub-strings that result from the splitting action. For example, the first two strings in `txt` are

```
txt[ 1:2 ]
```

```
[1] "net\t378\t35,134,155,157,177,204,314,315,319,350,366,369,371"
[2] "jp\t707\t552,604,672"
```

(These match the first two records in the file.) And, the first two elements of `els` are

```
els[1:2]
```

```
[[1]]
[1] "net"
[2] "378"
[3] "35,134,155,157,177,204,314,315,319,350,366,369,371"
```

```
[[2]]
[1] "jp"          "707"          "552,604,672"
```

We have successfully split each of these strings into 3 substrings with the 1st substring containing the domain, the 2nd the total visits, and the third the comma-separated times. If we can extract the 1st element from each vector then we have all of the domains. We can extract the 1st substring of the first record with

```
els[[1]][1]

[1] "net"
```

Similarly, we extract the domain of the 2nd page with `els[[2]][1]`. Essentially, we want to apply `[1]` to each element of `els`. In fact, the square bracket operator is a function and we can use it in a call to `sapply()`. Before we do, let's try to call the square bracket function using standard function syntax, e.g., `f(x)`. There are two inputs to this function—the vector to be subsetted and the position(s) that we want. We can construct the function call with:

```
"["(els[[2]], 1)

[1] "jp"
```

This is a bit strange looking, but we see that we have provided the `[`-operator with two inputs (`[[2]]` and `1`) and the `[`-operator has returned the desired domain. We needed to put the `[`-operator in quotes to avoid a syntax error. Now we are ready to apply the `[`-operator to each element of the `els` vector. We do this with

```
domains = sapply(els, "[", 1)
```

Notice that we supplied the square bracket function with the additional argument value of `1`. We also use `sapply()` rather than `lapply()` because we know that the return value from each call is a character vector of length 1 and we want these simplified into a vector. Let's check the first few elements of `domains` with

```
head(domain)

[1] "net" "jp" "com" "net" "ca" "de"
```

It appears that our extraction of the domains into a character vector has worked as expected.

We can similarly extract the total number of visits to a page with

```
visits = sapply(els, "[", 2)
head(visits)

[1] "378" "707" "418" "369" "719" "612"
```

We convert these character strings to numeric with

```
visits = as.numeric(visits)
```

Now we have two vectors `domains` and `visits`. Our remaining task is to create the list of vectors of changes.

We begin by extracting the strings of changes from `els` with

```
changes = sapply(els, "[", 3)
changes[1:2]
```

```
[1] "35,134,155,157,177,204,314,315,319,350,366,369,371"
[2] "552,604,672"
```

We can use `strsplit()` to split the string of changes. This time we want the split to be on commas, i.e.,

```
changes = strsplit(changes, ",")
changes[1:2]

[[1]]
 [1] "35"  "134" "155" "157" "177" "204" "314" "315" "319"
[10] "350" "366" "369" "371"

[[2]]
 [1] "552" "604" "672"
```

As with `visits`, we want to convert these substrings into numeric values. We do this with

```
changes = lapply(changes, as.numeric)
changes[1:2]

[[1]]
 [1] 35 134 155 157 177 204 314 315 319 350 366 369 371

[[2]]
 [1] 552 604 672
```

We see that we have created `changes` as a list of numeric vectors.

Let's explore a bit further to check our work. We can examine the distribution of the total number of visits with

```
hist(visits, breaks = 50, main = "")
```

We see in this histogram (Figure 4.9) that the vast number of sites were visited the planned number of times or nearly so. Yet, all did not go as planned and some sites were visited fewer than 100 or 200 times.

Let's also examine the distribution of the number of changes observed for the sites. This information can be obtained from the lengths of the vectors in `changes`. We can find these values with

```
numChanges = sapply(changes, length)
head(numChanges)

[1] 13 3 385 27 185 17
```

Before we make a histogram of these values, let's restrict the sites that we examine to those that have been visited close to the planned number of times (719). We use `visits` to take a subset of `numChanges` and then make the histogram with

```
hist(numChanges[ visits > 700 ], breaks = 25, main = "",
     xlab = "Number of Changes Observed")
```

We can see in this histogram (Figure 4.10) that the distribution of the number of changes is unimodal and skewed right, i.e., many pages change a few times and some pages change a great number of times.

Our final task is to collect these three pieces of data into a list. We do this with

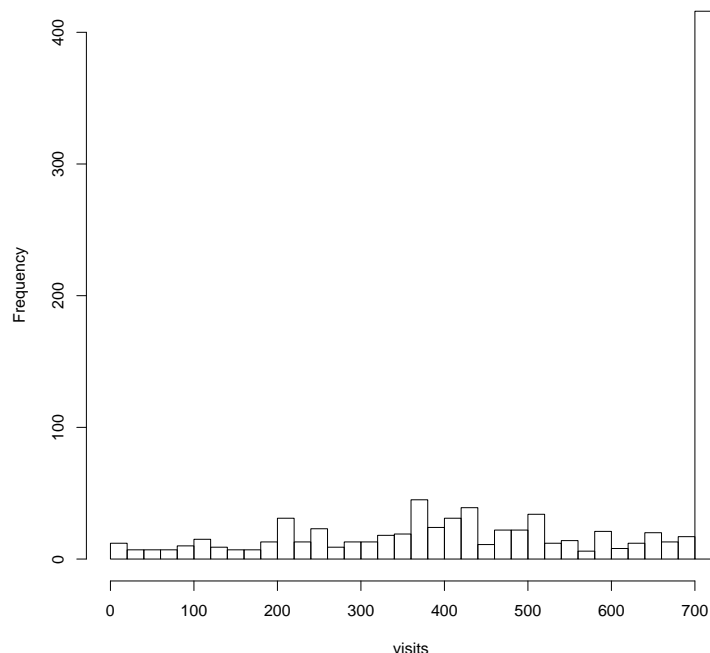


Figure 4.9: Histogram of the Total Number of Visits to Web Pages. *This histogram shows the distribution of the number of visits to a Web page for the 1,000 pages. The plan was for each page to be visited 719 times, and the bar above 700-720 shows that at least 400 pages were visited the planned number of times. However, many sites were visited fewer than 700 times, including a few sites that have less than 100 visits.*

```
cache = list(domains, visits, changes)
class(cache)
```

```
[1] "list"
```

```
length(cache)
```

```
[1] 3
```

```
names(cache)
```

```
[1] "domains" "visits"  "changes"
```

We can continue to analyze these data and we can collaborate with other researchers by sharing `cache`. To do this, we save `cache` to an `.rda` file with, e.g., `save(cache, file = "webCache.rda")`. An `rda` file can contain multiple objects so, e.g., if we find it easier to not group our 3 objects into 1 list, then we can save them in an `rda` file as follows: `save(domains, visits, changes, file = "webCache.rda")`.

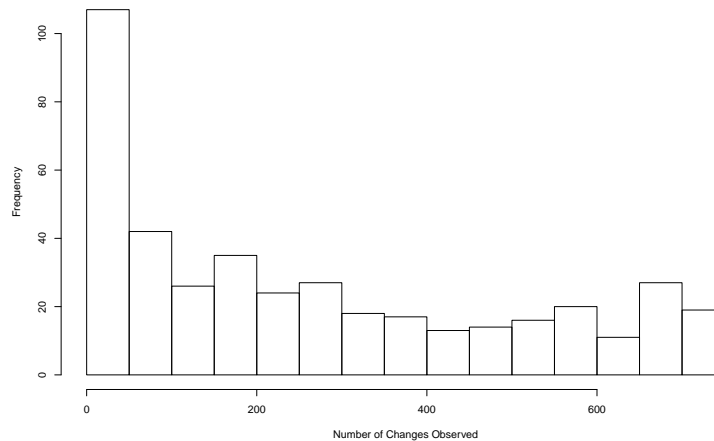


Figure 4.10: Histogram of the Total Number of Changes to Web Pages. *The distribution of the number of changes observed is skewed right with a peak in the 0-25 range and a long right tail with some pages changing on every or nearly every visit. This histogram includes only those pages with at least 700 visits.*

4.4 Reading Data from a Web Page

Web pages often contain tables of data that we want to extract and analyze. Although the data might appear to be a simple plain text table, a table in a Web page is typically marked up with tags that describe the cell and how the information should be rendered in the Web page, e.g., left or right justified, padding around the value, and format of the table header. If we want to extract just the values from the table, we need to separate the content from the mark up. For example, we have created a very simple *HTML* page containing a table of the family data for the family in Q.1-7 (page 21). Figure 4.11 displays a screen shot of this page as it is displayed in a browser. In the following example, we show how to easily extract these values from the table and create a data frame. We use the `readHTMLTable()` function in the XML package to do this.

Example 4-5 Extracting Data From an HTML Table

The actual content of this *HTML* file appears below. The terms such as `<title>` and `<table>` are *HTML* tags, and we describe them in greater detail in Chapter 12. For now, we point out that the value in a cell is between `<td>` and `</td>` tags and each row of the table is begins with `<tr>` and ends with `</tr>`.

```
<html>
<head>
<title>Example table</title>
</head>
<body>
<h2>A Family</h2>

<p>
```


A Family

This is a simple Web page with a table of data.

name	sex	age	height	weight	bmi	overWt
Tom	m	77	70	175	25.16	T
May	f	33	64	125	21.50	F
Joe	m	79	73	185	24.46	F
Bob	m	47	67	156	24.48	F
Sue	f	27	64	105	18.06	F
Liz	f	33	68	190	28.95	T
Jon	m	67	68	185	28.19	T
Sal	f	52	65	124	20.68	F
Tim	m	59	68	175	26.66	T
Tom	m	27	71	215	30.05	T
Ann	f	55	67	166	26.05	T
Dan	m	24	66	140	22.64	F
Art	m	46	66	150	24.26	F
Zoe	f	48	62	125	22.91	F

Figure 4.11: Screen Shot of a Simple *HTML* Web Page. This screen shot shows a very plain Web page with little content except a table of data for the 14-member example family from Chapter 1. Note that the column labeled ‘bmi’ provides values to 2 decimal places, and the ‘overWt’ column has capital T or F for whether or not the person’s BMI is over 25.

```

This is a simple Web page with a table of data.
</p>
<table>
<tr><th>name</th><th>sex</th><th>age</th><th>height</th>
    <th>weight</th><th>bmi</th><th>overWt</th></tr>
<tr><td>Tom</td><td>m</td><td>77</td><td>70</td><td>175</td>
    <td>25.16</td><td>T</td></tr>
<tr><td>May</td><td>f</td><td>33</td><td>64</td><td>125</td>
    <td>21.50</td><td>F</td></tr>
<tr><td>Joe</td><td>m</td><td>79</td><td>73</td><td>185</td>
    <td>24.46</td><td>F</td></tr>
...
</table>
</body>
</html>

```

It you understand a little about *HTML* then it seems possible to extract the information from the cells of a table and convert them into a data frame in *R*. *HTML* files (and the tables they contain) have a lot of structure, which we discuss in greater detail in Chapter 12. For now, we simply use the `readHTMLTable()` function in the XML package, which exploits this structure to find and extract cell values from an *HTML* table. We load the XML package and call `readHTMLTable()` as follows:

```

library(XML)
familyH = readHTMLTable("family.html", header = TRUE, which = 1,
    colClasses = c("character", "factor", "numeric",
        "numeric", "numeric", "numeric", "logical"))

```

Notice that we specified that the table has a header so the first row of the table is used for

variable names. The *which* argument indicates that we want the first (and in this case only) table in the page. If we do not use this argument then all tables in the page are returned as a list of data frames. The *colClasses* argument specifies the class for each of the variables (i.e., columns); we have indicated that the first column is a character vector, the second is a factor, the next four are numeric, and that last is logical. Without these specifications, all the columns are treated as factor vectors.

We confirm that the data frame contains the data from the Web page,

```
head(familyH)
```

	names	sex	age	height	weight	bmi	overWt
1	Tom	m	77	70	175	25.16	TRUE
2	May	f	33	64	125	21.50	FALSE
3	Joe	m	79	73	185	24.46	FALSE
4	Bob	m	47	67	156	24.48	FALSE
5	Sue	f	27	64	105	18.06	FALSE
6	Liz	f	33	68	190	28.95	TRUE

And, we can check the class of the variables with a call to *sapply()*. ■

Web pages often have more complex content than the page shown in Figure 4.11. In the next section, we return to the problem of extracting data from a table on a Wikipedia page. This page has several tables and figures, but the process of extracting the table of interest is nearly as simple as this example.

4.4.1 World Records in the Men's 1500 meter

We saw in Figure 4.2 that the Wikipedia Web page `1500_metres_world_record_progression` contains tables of world records for the 1500 meter race. The particular table that we want is shown in Figure 4.1. It is one of several tables in the page. We access this page from within *R* with the *getURL()* function in the *RCurl* package. We do this with

```
library(RCurl)
wikipedia = "https://en.wikipedia.org/wiki/"
wikipediaPage =
  paste(wikipedia,
        "1500_metres_world_record_progression", sep = "")
htmlContent = getURL(wikipediaPage)
```

The page is now in the character vector *htmlContent*. We use the *readHTMLTable()* function to extract all the tables as a list of data frames with

```
library(XML)
result = readHTMLTable(htmlContent)
```

We examine the first few rows of each data frame with *sapply(result, head, 2)* or with *str(result)* to determine that the desired data frame is the second element of *result*. We can work directly with *result[[2]]* or process *htmlContent* again and this time specify that we want only the 2nd table. We do this with

```
tableWR = readHTMLTable(htmlContent, which = 2,
                        stringsAsFactors = FALSE)
```

We kept the variables as strings because we want to specially process the race time and date. We combine the `min` and `sec` vectors to create `runTime` in seconds. We also convert the `Date` to a special format. The plotting (and other) functions recognize this format and make sensible axes with the dates. A step plot (Figure 4.12) shows the setting/breaking of the world record from 1912 to 2015. We make this plot with

```
tempTime = as.POSIXlt(tableWR$Time, format = "%M:%OS")
finalTable =
  data.frame(runTime = (tempTime$min * 60) + tempTime$sec,
             recordSet = as.Date(tableWR$Date,
                                format = "%Y-%m-%d"))
plot(runTime ~ recordSet, data = finalTable,
     type = "s", xlab = "Year", ylab = "Time (sec)",
     main = "World Records in Men's 1500 meter",
     xlim = c(1912, 2015))
```

The current record was set in 1998, i.e., it has held for 18 years.

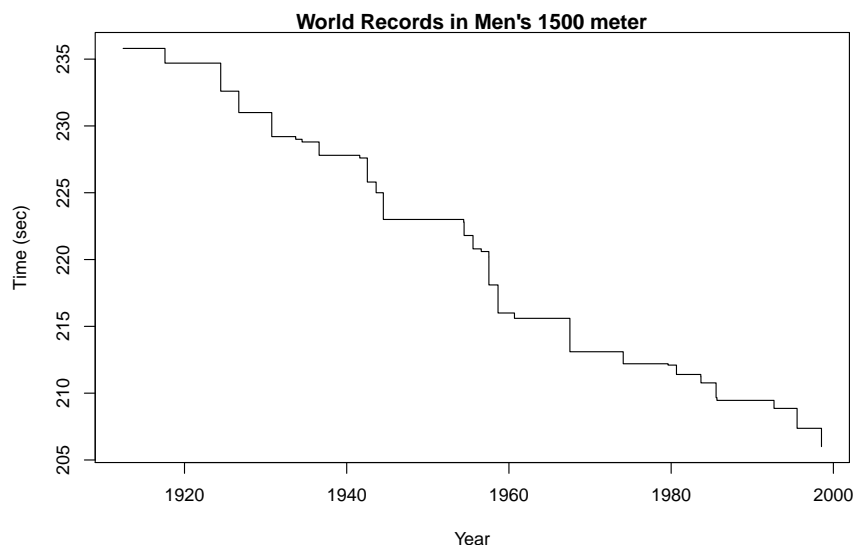


Figure 4.12: World Record in Men's 1500 m. *This step plot has a step for each time the world record for the men's 1500 meter was broken. The horizontal segments indicate the length of time the record stood and the vertical drops show the seconds by which the record was broken.*

4.5 Reading *JSON* Formatted Data

JSON (JavaScript Object Notation) is a simple, lightweight format that originates from the *JavaScript* [1] language syntax for creating objects. Data given in *JSON* format can be used directly in *JavaScript* code such as in Web pages. These characteristics make *JSON* a

valuable tool for working with data, especially data coming from Web services or displayed in browsers.

The *JSON* format is very simple. As with most data-oriented computer languages, *JSON* has common primitive data types: boolean/logical (*true* and *false*), number, and string. Unlike in *R*, these are scalars in *JSON*. Also, with *JSON* there is only one type of number—a real or floating-point value, which is a *numeric* in *R*. *JSON* does not have the notion of a missing value, infinity, or “not a number”, i.e., *R*’s *NA*, *Inf*, *NaN*, respectively. It does have the notion of *null*, the empty “object.”

These four types (boolean, number, string, and *null*) make up the entire collection of scalar values so there is a reasonably obvious mapping between *JSON* and *R* primitive data types. Specifically, *JSON* scalars map to *R* vectors with length 1. The *true* and *false* map to *TRUE* and *FALSE*, respectively; *null* maps to *NULL*; numbers map to *numeric* vectors; and a *JSON* string maps to a character vector of length 1.

In addition to the scalar types, *JSON* has two container data types for collections of zero or more values. In *R*, these correspond to unnamed and named vectors or lists. In other languages, these are often referred to as simple ordered arrays and associative arrays. In *JSON*, these are termed arrays and objects, respectively. The array is an ordered, unnamed collection that is identified by the notation:

```
[ value, value, value, ... ]
```

That is, an opening [and a closing] denote the beginning and end of a comma-separated array of values. Named arrays or “objects” use { and } and have the form

```
{ "name" : value, "name" : value, ... }
```

The quotes are required for the names or “keys,” and again, the elements are comma-separated.

These containers can be nested, i.e., an element of a container can itself be a container (or a scalar). This gives *JSON* the flexibility to represent arbitrary data structures. Unlike *R*, there is no distinction between a collection of homogeneous values and nonhomogeneous values, i.e., the distinction between a *vector* and *list* in *R*. In other words, *JSON* ignores the fact that a collection contains values of the same type and just has the notion of a container, either named (the object) or ordered (the array).

An important caveat is that top-level *JSON* content cannot be a simple primitive value. Instead, it must be a container, either named or not. This means that we would never have *JSON* content of the form *1*, *true*, or “xy” : [1, 2]. Instead, we need to have, respectively, [1] or [true] or { “xy” : [1, 2] }. Note the white-space between values in *JSON* content is ignored, unless of course it is within a string.

***JSON* (JavaScript Object Notation)**

JSON formatted data have the following properties:

- The primitive data types are boolean (*true* and *false*), number, and string. These are scalars.
- *null* stands for the empty “object”.
- The array is a simple, ordered, unnamed collection of values. These are denoted by an opening [and a closing] and the values are comma-separated, i.e.,

```
[ value, value, value, ... ]
```

- The object is a named, unordered collection values. The { and } delimit the object and the values appear as "name": value pairs. That is, an object has the form

```
{ "name" : value, "name" : value, ... }
```

The quotes are necessary for the names (keys).

- Arrays and objects can be nested, i.e., an element of a container can itself be a container (or a scalar). This allows nesting of values and gives *JSON* the flexibility to represent arbitrary data structures.

A *JSON* object can be imported into *R* as a list, and depending on its structure, it can possibly be reduced to a vector or data frame or collection of data frames.

We provide an example of *JSON* content that is available from the *New York Times*.

Example 4-6 JSON Content Available from the New York Times

The following shows data in *JSON* format, taken from *New York Times* Web service documentation for its campaign finance API [3]. These data illustrate the full set of data types and how they are represented in *JSON*.

```
{
  1 "results": [
    2 {
      "city": "NEW YORK",
      3 "address": "34 WEST 38TH ST - FLR 5",
      "name": "AMERICANLP",
      "zip": 10018,
      4 "treasurer": "TJ WALKER",
      "super_pac": true,
      "relative_uri": "/committees/C00507244.json",
      "candidate": null,
      5 "id": "C00507244",
      "leadership": false,
      6 "sponsor_name": null,
      "party": "",
      "fec_uri": "http://query.nictusa.com/.../C00507244/",
      "state": "NY"
      7
    },
    ...
  ],
  8 "base_uri": "http://api.nytimes.com/.../finances/2012/",
  "copyright": "Copyright (c) 2011 The New York Times Company...",
  "cycle": 2012,
  "status": "OK"
}
```

- 1 The { delimits the *JSON* object. We can think of the content of this object, in *R* terms, as a list with five named elements: *results*, *base_uri*, *copyright*, *cycle*, and *status*.

- ❷ The first element, *results*, is an ordered collection of individual result objects. That is, the `[` indicates that *results* is a simple ordered array.
- ❸ Each element of *results* is an object/associative-array with many fields such as `city`, `address`, `name`, `zip`, etc.
- ❹ The `zip` field is a number. In the original source, these were represented as strings (we modified them to display the full spectrum of primitive data types).
- ❺ The `candidate` and `sponsor_name` fields have the special value *null*.
- ❻ The value of each of the `leadership` and `super_pac` fields is a logical value: *true* or *false*.
- ❼ Many of the values are strings.
- ❽ The last four elements of this object are simple strings named `base_uri`, `copyright`, `cycle`, and `status`.

■

The two main operations when dealing with *JSON* in *R*—converting *JSON* content to *R* objects, and converting *R* objects to *JSON*—are handled by functions named `fromJSON()` and `toJSON()`, respectively. The `fromJSON()` function can read content from a file, *URL*, or directly from a string, i.e., in memory. It processes the bytes from this source and converts the values into *R* objects and returns a single *R* object containing the subelements. The `toJSON()` function takes an *R* data object and generates the *JSON* representation as a single string. This can then be, for example, added to a file, sent as part of an *HTTP* request, or sent to another application. These functions are available in 3 packages: `RJSONIO`, `rjsonlite` and `rjson`. In the next example, we show how to read a *JSON* file into *R* and create a data frame.

Example 4-7 A Family's Data in *JSON*

The data for this example family was first introduced in Chapter 1 (see Q.1-1 (page 12)). Here these data have been recast into a *JSON* formatted file. The file appears below.

```
{ "family": [
{"firstName":"Tom" , "sex":"m" , "age":77 , "height":70 ,
 "weight":175 , "bmi":25.16 , "overWt":true },
{"firstName":"May" , "sex":"f" , "age":33 , "height":64 ,
 "weight":125 , "bmi":21.50 , "overWt":false },
{"firstName":"Joe" , "sex":"m" , "age":79 , "height":73 ,
 "weight":185 , "bmi":24.46 , "overWt":false },
{"firstName":"Bob" , "sex":"m" , "age":47 , "height":67 ,
 "weight":156 , "bmi":24.48 , "overWt":false },
...
]}
```

Notice that this *JSON* content consists of an object with one field named `family`. The `family` field is an array of 14 objects, each object corresponds to a family member. The keys in each of these objects are the same: `firstName`, `sex`, `age`, `height`, `weight`, `bmi`, and `overWt`. The value for each key is a scalar and these scalars correspond to the measurement of that variable for the respective family member. We read this file into *R* with

```
familyList = fromJSON("family.json")
```

We confirm that we have a list with one element which itself is a list named `family`.

```
class(familyList)
```

```
[1] "list"
```

```
names(familyList)
```

```
[1] "family"
```

```
class(familyList$family)
```

```
[1] "list"
```

The `family` has 14 unnamed elements, i.e.,

```
names(familyList$family)
```

```
NULL
```

```
length(familyList$family)
```

```
[1] 14
```

Each of the 14 elements in `family` is a list. We assign the first of these to the variable `tom` and further explore this object with

```
tom = familyList$family[[1]]
```

```
class(tom)
```

```
[1] "list"
```

```
names(tom)
```

```
[1] "firstName" "sex"      "age"      "height"
```

```
[5] "weight"    "bmi"      "overWt"
```

Furthermore, each of these 7 named objects in `tom` is a vector of length 1. For example, `tom$age` is the numeric containing the single value 77. Figure 4.13 provides a sketch of the structure of `familyList`.

We want to create a data frame from `familyList`. One approach is to extract the values of each of the vectors in the 14 lists. For example, we can extract the age of each family member by applying the `[[`-operator to each element of `family`. We do this with

```
sapply(familyList$family, "[", "age")
```

```
[1] 77 33 79 47 27 33 67 52 59 27 55 24 46 49
```

Note that we supplied the argument "age" to `[[` in our call to `sapply()`. We can create a vector for each of the variables with

```
name = sapply(familyList$family, "[", "firstName")
```

```
sex = sapply(familyList$family, "[", "sex")
```

```
age = sapply(familyList$family, "[", "age")
```

```
ht = sapply(familyList$family, "[", "height")
```

```
wt = sapply(familyList$family, "[", "weight")
```

```
bmi = sapply(familyList$family, "[", "bmi")
```

```
owt = sapply(familyList$family, "[", "overWt")
```

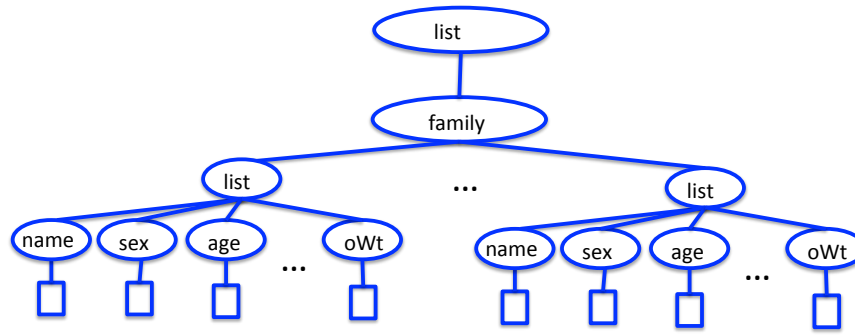


Figure 4.13: Diagram of a *JSON* File Read into *R*.

This diagram shows the structure of the *list* containing the information in the *JSON*-formatted file after being read into *R*. This *list* consists of an object called `family`. The *family* is an array of 14 fields, each of which is an object with 14 scalar fields. These scalars are `firstName`, `sex`, `age`, `height`, `weight`, `bmi`, and `overWt`. Each object and array is converted into a *list* in *R* so the data structure is a *list* of 14 *lists*, one for each family member. Each of these 14 *lists* is a *list* of 7 *lists*, one for each variable. Finally, each of these vectors has length 1 and contains the value for that variable for that member of the family.

We can determine the class of each of these variables by checking the class of the elements of `tom`, e.g.,

```
sapply(tom, class)
```

```

  firstName      gender      age      height
"character" "character" "numeric" "numeric"
   weight      bmi    overWt
"numeric"  "numeric" "logical"

```

Note that the *true* and *false* values were converted to a logical vector. For our final data frame, we want `firstName` to remain a character vector and to convert `sex` to a factor. We combine all of these vectors into a data frame with

```

family = data.frame(name, sex = as.factor(sex), age, height = ht,
                    weight = wt, bmi, overWt = owt,
                    stringsAsFactors = FALSE)

```

Notice that we supplied the value `FALSE` for the `stringsAsFactors` argument to prevent the default conversion of character vectors into factors. This way, `name` remains a character vector. We check the data frame with

```
head(family)
```

```

  name sex age height weight  bmi overWt
1  Tom  m  77     70    175 25.16   TRUE
2  May  f  33     64    125 21.50  FALSE
3  Joe  m  79     73    185 24.46  FALSE
4  Bob  m  47     67    156 24.48  FALSE
5  Sue  f  27     64    105 18.06  FALSE
6  Liz  f  33     68    190 28.95   TRUE

```


One last comment: when we created the 7 vectors `name`, `sex`, etc., the code was repetitive. When this happens, we typically ask ourselves if there is a more concise way to carry out these computations. Indeed there is and we cover this approach in Chapter 5. However, for those who already know a bit about writing functions, we supply the alternative code as a preview.

```
varNames = c("firstName", "sex", "age", "height",
             "weight", "bmi", "overWt")
family = lapply(varNames, function(x) {
  sapply(familyList$family, "[", x)
})
family = as.data.frame(family)
names(family) = varNames
```

Notice the nested apply functions. The outer `lapply()` applies the function to each string in `varNames`. This function is one that we have written ourselves in order to call `sapply()` with the variable name as an argument.

■

4.6 Kiva

Recall that the information about the Kiva loans is formatted in *JSON* as an array of objects (see Q.4-4 (page 153)). Each element in this array describes a loan with fields for such things as the loan identifier, the name and location of the person looking for the loan, the more specific purpose of the loan, and the amount being sought. We want to create a data structure that contains information about each loan, but it is not immediately obvious from the display of a loan field in Q.4-4 (page 153) whether or not we can map the *JSON* content into a data frame. To help us figure this out, we can read the *JSON* content into *R* and explore the structure.

We begin by downloading the *JSON* file and extracting all of the files with

```
download.file("http://s3.kiva.org/snapshots/kiva_ds_json.zip",
             destfile = "kiva_json.zip")
unzip("kiva_json.zip")
```

We can also download the file interactively in our browser or use shell commands to download it and then unzip it (see [?]). When we unzip the file we find a `loans/` directory and within it there are files with names `n.json` for `n` ranging from 1 to 1975.

When we open one of the files, e.g. `1.json` in our text editor to see the structure of the file, we find there is only one line in this file and it contains all the data for 500 loans. The *JSON* format has a very simple, well-defined structure, and most languages have parsers for reading *JSON*. That is, the 1st element is named header and contains fields for the total, page, date and `page_size`. The 2nd element contains information about the loans. Each loan has an identifier (`id`), description, a status, funded amount, and so on. The format of each loan is similar, but we do not know if all loans have all fields. Also, some fields are simple numbers or text strings, but others are collections of sub-values. We need to investigate these formats to determine how we want to organize the data in *R*.

Our first task is to read the data into *R*. As noted in Section 4.5, *R* has packages and functions to convert *JSON* content directly into *R* objects. We can read `1.json` with

```
library(RJSONIO)
loanDoc1 = fromJSON("loans/1.json")
```

Note that to read this file, we can also use either the `rjsonlite` or `rjson` packages and their `fromJSON()` functions.

The first element of `loanDoc1` is the header information, i.e.,

```
names(loanDoc1)

[1] "header" "loans"

loanDoc1$header

$total
[1] 987161

$page
[1] 1

$date
[1] "2015-12-02T18:49:18Z"

$page_size
[1] 500
```

We can access the first loan with

```
loan1 = loanDoc1$loans[[1]]
```

At this point, we have read the first file, `1.json`, into *R* and technically we have the data from this file available to analyze. However, the data are not in a convenient format to work with. If we want to compute the median loan amount, we must loop over each loan and extract the `funded_amount` and then compute the median, e.g.,

```
median(sapply(loanDoc1$loans, '[', 'funded_amount'))
```

If we are interested in the `paid_amount`, things become slightly more complex as some loans do not have this field. For example, the 1st record does not have this field but the 2nd one does, e.g.,

```
loanDoc1$loans[[1]]$paid_amount

NULL
```

```
loanDoc1$loans[[2]]$paid_amount

[1] 500
```

As a result,

```
sapply(loanDoc1$loans, '[', 'paid_amount')
```

returns a list rather than a vector of numbers. We have to `unlist()` this and then compute the median, e.g., `median(unlist(sapply(loanDoc1$loans, '[', 'paid_amount-')))`.

4.6.1 Loan Elements

Let's think about how we want to structure the data. A data frame is a convenient form where each row corresponds to a loan. The variables in this data frame are `id`, `status`, `funded_amount`, etc. In order to determine the variables in this data frame, let's see what elements are in all loans and which are not. We can do this with

```
table(unlist(lapply(loanDoc1$loans, names)))
```

activity	basket_amount
500	500
bonus_credit_eligibility	borrowers
500	500
currency_exchange_loss_amount	delinquent
500	500
...	

This shows that all loans have all fields in the first file. (We need to verify this for the other files.)

Some elements are simple single values and others have list values. Let's examine the elements of the first loan and determine which elements are simple single values. We apply the `is.atomic()` function to each element in `loan1` with

```
atomic1 = sapply(loan1, is.atomic)
```

We also find the length of each element, i.e.,

```
length1 = sapply(loan1, length)
```

Scalar values are atomic and length 1. These are:

```
index1atomic = atomic1 & (length1 == 1)
names(loan1)[indexAtomic]
```

[1] "id"	"name"
[3] "status"	"funded_amount"
[5] "activity"	"sector"
[7] "use"	"partner_id"
[9] "loan_amount"	"lender_count"
[11] "bonus_credit_eligibility"	

We can determine which elements have more complex values from the negation of `indexAtomic`, i.e.,

```
names(loan1)[!indexAtomic]
```

[1] "description"	"basket_amount"
[3] "paid_amount"	"image"
[5] "video"	"themes"
[7] "delinquent"	"location"
[9] "posted_date"	"planned_expiration_date"
[11] "currency_exchange_loss_amount"	"tags"
[13] "borrowers"	"terms"
[15] "payments"	"funded_date"
[17] "paid_date"	"journal_totals"
[19] "translator"	

The `delinquent` field, when not `NULL`, has the value `TRUE`. This suggests that we store the delinquent information as a *logical* vector and map the `NULL` values to `FALSE`. We leave this as an exercise.

Note that there are more efficient way to use `sapply()` to extract the `delinquent` elements from `loanDoc1` and apply the `is.null()` all in one function call without having to save the intermediate delinquent elements (in `dels`). This approach requires us to provide an anonymous function to `sapply`. We cover this topic in Chapter 5.

Journal Totals

We can continue our investigation of these non-atomic elements to determine how to store them in a data frame. For example, the `journal_totals` field is numeric, but it consists of two values named `entries` and `bulkEntries`. One way to incorporate these into our data frame is to flatten the vector by adding each element as its own top-level variable in the data frame. That is, create two variables, `entries` and `bulkEntries` from `journal_totals`. We also leave this task as an exercise.

Loan Themes

As another example, the `themes` field is mostly `NULL` but there are a few character vectors. We confirm this with

```
themes = lapply(loanDoc1$loans, '[', "themes")
table(sapply(themes, class))
```

```
character      NULL
      28         472
```

We can look at the length of these character vectors with

```
table(sapply(themes, length))
```

```
 0    1    2    3
472 20    7    1
```

We see that there are a few loans that have multiple themes. We can combine these multiple themes into a single string to put the themes into a single column. Alternatively, we can create a separate data frame and use the loan identifier to map between the loan and the theme, e.g.,

```
loan_id theme
  1      Underfunded Areas
  1      Rural Exclusion
 97      Conflict Zones
124      Rural Exclusion
```

How we decide to represent these data (if at all) depends on how we plan to use them and which representation makes the computations more convenient overall.

Translator

The `translator` field is sometimes `NULL` but is typically a list, and in some cases it is a character string:

```
translators = lapply(loanDoc1$loans, '[', 'translator')
table(sapply(translators, class))
```

```
character      list      NULL
      5         365      130
```

The list elements have a `byline` and an `image` identifier field. The `byline` is the name of the person. The character elements in `translator` give the name of the translator, i.e., the `byline` field. We can collapse `translator` to the `byline` field as a character string and drop the `image`. We can use NA for those that are NULL. However, do we really need the translator information for a loan? We may want to drop this information all together.

4.6.2 The Payments

The payments field is complicated because it contains information about each installment. We can combine these records into their own data frame for each loan. However, where do we put this data frame in the loans data frame? We can have a payments column each of whose elements is itself a 7 column data frame. This can work, but it is slightly awkward to work with. If we want to find the number of payments made for each loan, we have to loop over each loan and compute the number of rows. To work with the payment amounts or dates, we have to do slightly more. An alternative approach puts all of the payment information for all of the loans into a single data frame that is separate from the loan data frame. That is, each row in this data frame corresponds to a payment. In this case, some loans will have multiple rows and others may not have any. If we take this approach, then we must add the loan identifier (`id`) as an additional column so that we can connect the payment records with the loans records.

Another approach is to use a payment as the entity corresponding to each row in our loans data frame. We then have to repeat all of the fixed information about the loan for each row (e.g. `id`, name, description, status, ...). This leads to quite a lot of data repetition. Importantly, it makes the typical operations we do on loans more complicated. For example, to compute the distribution of loan amounts with this structure, we have to discard all the duplicate loan amounts for the same loan. This is a cumbersome approach. The solution of a separate payment data frame is preferable.

4.6.3 The Borrowers

The `borrowers` field is another *list* that appears in each loan element. This field identifies each of the borrowers for the loan. There are a different number of borrowers for each loan so this is a variable length list. We confirm this with

```
borrowers = lapply(loanDoc1$loans, '[' , "borrowers")
table(sapply(borrowers, length))
```

```
 1    2    3    4    5    6    7    8    9   10   11   12   13   14
419    3   13   10   15    1    4    2    5    5    3    3    5    4
 15   16   19   20   26
  4    1    1    1    1
```

Most loans (419 of the 500) are made to one person. However 81 loans have multiple lendees, including one loan with as many as 26 borrowers.

In our analysis, we may want to see if there are any people who received multiple loans, and whether they paid them back. We also may want to see if the number of lendees indicates whether the loan is paid back and whether different loan uses are associated with more lendees. Therefore, it makes sense to map the lendees to their own table, and use the loan identifier to link between the loan data frame and the lendees table. Additionally, we may want to create a new variable—the number of borrowers for a loan—and add it to our data frame.

4.6.4 Final Structure

Given our explorations and ideas for analysis so far, we want to create three data frames – one for loans, one for payment installments, and one for borrowers. The loans data frame is the primary one. There is one observation for each loan, and all of the atomic elements are included as variables in the data frame. Additionally, we may want to flatten some complex elements, such as `journal_totals` and `translator` to create additional variables in the data frame and derive new variables to include, such as the number of borrowers on a loan. We also need to systematically examine the non-atomic length-one entries. For example, we have seen that with the `delinquent` field we want to create a logical vector where the NULL entries have values of FALSE.

The other two data frames are at different levels of granularity than the `loans` data frame. The `payments` data frame has one row for each loan payment so some loans may not appear in the data frame because no payments have been made, and other loans may have many rows, one for each payment made on the loan. Similarly, the `borrowers` data frame has a row for each borrower of a loan. Most loans have one borrower and consequently one row in the data frame. Those loans with multiple borrowers have a row for each person. In both the `payments` and `borrowers` data frames, we want to include the `id` for the corresponding loan so this information can be linked to the loan information. We create these three data frames in the exercises.

4.7 Summary

4.8 Functions for Handling Complex Data Formats

This chapter introduced many of the functions available in *R* for working with lists and data that do not have simple table-like formats. These are summarized below.

`readHTMLTable()` (in XML package) Return all the tables in the *HTML* document as data frames. The `which` parameter allows us to specify those tables we want to extract from the document. The `colClasses` parameter allows us to specify the classes/types for the columns or a function to convert the content to numbers, percentages, factors, etc.

`str()` Compactly display the structure of an arbitrary *R* object. The display includes the class of the object, at minimum, and may include information such as the object's dimensions, initial elements, or arguments.

`getURL()` (in RCurl package) Retrieve the content of a URL. Note the `getURLContent()` function is a more general function that handles binary or text results.

`fromJSON()` (in RJSONIO package) Parse *JSON* content into a list, and convert the content into the corresponding *R* type.

`toJSON()` (in RJSONIO package) Convert a vector or list to a *JSON* object.

`unlist()` Reduce the supplied list to a vector by concatenating elements in that list. If `recursive` is TRUE (default), the reduction is recursively applied.

`is.atomic()` Return TRUE if the supplied argument is a vector, specifically an atomic vector type (logical, integer, numeric, complex, character, or raw).

list.files() Return a character vector with all directories and files in the supplied directory.

do.call() Construct and execute a function call by passing a list of arguments (*args*) to a function (*what*). The function can be specified as a string.

rbind() Stack a sequence of vectors, matrices, or data frames by rows (on top of each other). The return value is a matrix or data frame with row length equal to the length of the sequence (vectors) or sum of the row lengths of each element in the sequence (matrices and data frames). The inputs must have the same lengths (when stacking vectors) or columns (matrices and data frames).

readLines() Read in a text file from a connection (e.g., a file name or *URL*) and return a character vector with the same length as the number of lines in the text file. To limit the number of lines to be read, use the *n* argument.

The Family of Apply Functions

These functions are variants on the concept of applying a specified function to each element of a list or data frame. The first argument in each of these apply functions is the object on which to operate.

lapply() Apply supplied function to each element in the data frame or list and return the result as a list with the same length as the original data frame or list. The input can also be a vector, in which case the function is applied to each element in the vector.

sapply() Similar to **lapply()** but the result is simplified to a vector or matrix, when possible.

mapply() Multivariate version of **sapply()**. Apply the supplied function and vectorize over multiple arguments. In this case, the function is provided as the first argument to **mapply()** and the multiple inputs follow. Additional arguments can be specified as a list in *MoreArgs*.

apply() Apply supplied function to the specified margins of the data frame, matrix, or array. For example, for matrices, **MARGIN == 1** indicates rows (apply the function across columns for each row) and **MARGIN == 2** indicates columns (apply down columns). As another example, for a three-dimensional array, **MARGIN == c(1, 2)** indicates rows and columns (across pages).

tapply() Apply function to each sub-group of the data vector (first argument). The sub-groups are formed from the unique values of the vector in *INDEX*. A list of vectors can be supplied in *INDEX*; these vectors should have the same length as the data vector.

4.9 Guided Practice

Subsetting lists

This section provides practice on subsetting lists. There are many ways to subset (see Section 4.2.1 for a review). Double square brackets are used to go one level deeper into a list structure and access individual elements of the list. A named element of the list can also be accessed with the *\$*-notation. Since data frames are special kinds of lists, both double square brackets and the *\$*-notation work for data frames.

The following set of questions concern the list *myList*. The list is loaded by `load("~/exampleList.rda")`.

1. Examine `myList`. How many elements are in the list? What are the names and classes of the variables in the list?
2. Create a new vector `check` that contains the elements in the even positions of `chars`. Write the code generally by not using the fact that you know how many elements are in `chars`.
3. The list `myList` contains a function and a data frame. Call this function, passing in the variable `x` from the data frame.

Apply functions

This section provides practice with the family of `apply()` functions, which include `apply()`, `lapply()`, `sapply()`, `tapply()`, and `mapply()`.

In answering the following questions, you will work with a subset of the rainfall data from five weather stations in the Colorado Front Range (see Q.4-1 (page 150) and Section 4.2.3). The data are loaded with `load("rainfallCO.rda")`. The data are in two lists, `rain` and `day`. Each list has five elements; one for each weather station. The goal is to summarize the data and create *R* objects useful for analysis, e.g., determining the average daily rainfall.

1. Examine the two lists, `rain` and `day`. Verify that they each have five elements. What are the names, classes, and dimensions of the elements in the lists?
2. Since `rain` provides rainfall data on the corresponding dates in `day` for each weather station, make sure the station names and dimensions match between the two lists. Use the `sapply()` function in your answer.
3. To help calculate average daily rainfall per *year*, create a new list called `year` that extracts the year from the data in the `day` list. Then, use `year` to find the total number of years that each station was in operation. *Hint*: Use the `floor()` and `unique()` functions in your answer. Why is the function `floor()` necessary?
4. Calculate the average rainfall per year for the third station in the dataset. Use the `tapply()` function in your answer. *Hint*: With the `tapply()` function, the function provided in the `FUN` argument is applied to subsets of the data (`X` argument) where the subsets are determined by the vector supplied in `INDEX` argument.
5. Create a new list `avgRainList` that contains the average daily rainfall per year for each station. Each element in the new list should correspond to one weather station. Use the `mapply()` (multivariate apply) function in your answer. *Hint*: The technique from the last question is useful here.

4.10 Exercises

Bibliography

- [1] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2006.
- [2] Kiva Organization. Kiva: Loans that change lives. <http://www.kiva.org/>, 2011.

- [3] The *New York Times* Company. The Times Developer Network: An API clearinghouse and community. http://developer.nytimes.com/docs/campaign_finance_api/campaign_finance_api_examples, 2012.
- [4] Wikipedia. 1500 metres world record progression. https://en.wikipedia.org/wiki/1500_metres_world_record_progression, 2015.