Chapter4(4.7~) & 5

4.7 Summarizing data

An important part of exploratory data analysis is summarizing data. The average and standard deviation are two examples of widely used summary statistics. More informative summaries can often be achieved by first splitting data into groups. In this section, we cover two new dplyr verbs that make these computations easier: summarize and group_by. We learn to access resulting values using the pull function.

4.7.1 summarize

The summarize function in dplyr provides a way to compute summary statistics with intuitive and readable code. We start with a simple example based on heights. The heights dataset includes heights and sex reported by students in an in-class survey.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(dslabs)
data(heights)
```

The following code computes the average and standard deviation for females:

```
s<-heights %>%
  filter(sex=='Female')%>%
  summarize(average=mean(height),standard_deviation=sd(height))
s
```

```
##    average standard_deviation
## 1 64.93942           3.760656
```

This takes our original data table as input, filters it to keep only females, and then produces a new summarized table with just the average and the standard deviation of heights. We get to choose the names of the columns of the resulting table. For example, above we decided to use average and standard_deviation, but we could have used other names just the same.

Because the resulting table stored in s is a data frame, we can access the components with the accessor $:

```
s$average
```

```
## [1] 64.93942
```

```
s$standard_deviation
```

```
## [1] 3.760656
```

s with most other dplyr functions, summarize is aware of the variable names and we can use them directly. So when inside the call to the summarize function we write mean(height), the function is accessing the column with the name “height” and then computing the average of the resulting numeric vector. We can compute any other summary that operates on vectors and returns a single value.

For another example of how we can use the summarize function, let’s compute the average murder rate for the United States. Remember our data table includes total murders and population size for each state and we have already used dplyr to add a murder rate column:

```
murders<-murders%>%mutate(rate=total/population*100000)
```

Remember that the US murder rate is **not** the average of the state murder rates:

```
summarize(murders, mean(rate))
```

```
##   mean(rate)
## 1   2.779125
```

This is because in the computation above the small states are given the same weight as the large ones. The US murder rate is the total number of murders in the US divided by the total US population. So the correct computation is:

```
us_murder_rate <- murders %>%
  summarize(rate=sum(total)/sum(population)*100000)
us_murder_rate
```

```
##       rate
## 1 3.034555
```

This computation counts larger states proportionally to their size which results in a larger value.

4.7.2 Multiple summaries

Suppose we want three summaries from the same variable such as the median, minimum, and maximum heights. We can use summarize like this:

```
heights %>%
  summarize(median=median(height), minimun=min(height), maximum=max(height))
```

```
##   median minimun  maximum
## 1   68.5      50 82.67717
```

But we can obtain these three values with just one line using the quantile function: quantile(x, c(0.5, 0, 1)) returns the median (50th percentile), the min (0th percentile), and max (100th percentile) of the vector x. We can use it with summarize like this:

```
heights %>%
  filter(sex == "Female") %>%
  summarize(median_min_max = quantile(height, c(0.5, 0, 1)))
```

```
##   median_min_max
## 1       64.98031
## 2       51.00000
## 3       79.00000
```

However, notice that the summaries are returned in a row each. To obtain the results in different columns, we have to define a function that returns a data frame like this:

```
median_min_max <- function(x){
  qs<-quantile(x,c(0.5,0,1))
  data.frame(median=qs[1], minimum=qs[2], maximum=qs[3])
}
heights %>%
  filter(sex=="Female")%>%
  summarize(median_min_max(height))
```

```
##     median minimum maximum
## 1 64.98031      51      79
```

In the next section we learn how useful this approach can be when summarizing by group.

4.7.3 Group then summarize with group_by

A common operation in data exploration is to first split data into groups and then compute summaries for each group. For example, we may want to compute the average and standard deviation for men's and women's heights separately. The group_by function helps us do this.

If we type this:

```
heights %>% group_by(sex)
```

```
## # A tibble: 1,050 x 2
## # Groups:   sex [2]
##    sex    height
##    <fct>   <dbl>
##  1 Male       75
##  2 Male       70
##  3 Male       68
##  4 Male       74
##  5 Male       61
##  6 Female     65
##  7 Female     66
##  8 Female     62
##  9 Female     66
## 10 Male       67
## # ... with 1,040 more rows
```

The result does not look very different from heights, except we see Groups: sex [2] when we print the object. Although not immediately obvious from its appearance, this is now a special data frame called a grouped data frame, and dplyr functions, in particular summarize, will behave differently when acting on this object. Conceptually, you can think of this table as many tables, with the same columns but not necessarily the same number of rows, stacked together in one object. When we summarize the data after grouping, this is what happens:

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.3     v stringr 1.4.0
## v tidyr   1.1.3     v forcats 0.5.1
## v readr   2.0.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
heights %>%
  group_by(sex) %>%
  summarize(average=mean(height), standard_deviation = sd(height))
```

```
## # A tibble: 2 x 3
##   sex    average standard_deviation
##   <fct>    <dbl>              <dbl>
## 1 Female    64.9               3.76
## 2 Male      69.3               3.61
```

The summarize function applies the summarization to each group separately.

For another example, let’s compute the median, minimum, and maximum murder rate in the four regions of the country using the median_min_max defined above:

```
murders %>%
  group_by(region)%>%
  summarize(median_min_max(rate))
```

```
## # A tibble: 4 x 4
##   region        median minimum maximum
##   <fct>          <dbl>   <dbl>   <dbl>
## 1 Northeast       1.80   0.320    3.60
## 2 South           3.40   1.46    16.5
## 3 North Central   1.97   0.595    5.36
## 4 West            1.29   0.515    3.63
```

4.8 pull

The us_murder_rate object defined above represents just one number. Yet we are storing it in a data frame:

```
class(us_murder_rate)
```

```
## [1] "data.frame"
```

since, as most dplyr functions, summarize always returns a data frame.

This might be problematic if we want to use this result with functions that require a numeric value. Here we show a useful trick for accessing values stored in data when using pipes: when a data object is piped that object and its columns can be accessed using the pull function. To understand what we mean take a look at this line of code:

```
us_murder_rate %>% pull(rate)
```

```
## [1] 3.034555
```

This returns the value in the rate column of us_murder_rate making it equivalent to us_murder_rate$rate.

To get a number from the original data table with one line of code we can type:

```
us_murder_rate <- murders %>%
  summarize(rate=sum(total)/sum(population)*100000) %>%
  pull()

us_murder_rate
```

```
## [1] 3.034555
```

4.9 Sorting data frames

When examining a dataset, it is often convenient to sort the table by the different columns. We know about the order and sort function, but for ordering entire tables, the dplyr function arrange is useful. For example, here we order the states by population size:

```
murders%>%
  arrange(population)%>%
  head()
```

```
##                  state abb        region population total       rate
## 1              Wyoming  WY          West     563626     5  0.8871131
## 2 District of Columbia  DC         South     601723    99 16.4527532
## 3              Vermont  VT     Northeast     625741     2  0.3196211
## 4         North Dakota  ND North Central     672591     4  0.5947151
## 5               Alaska  AK          West     710231    19  2.6751860
## 6         South Dakota  SD North Central     814180     8  0.9825837
```

With arrange we get to decide which column to sort by. To see the states by murder rate, from lowest to highest, we arrange by rate instead:

```
murders%>%
  arrange(rate)%>%
  head()
```

```
##           state abb        region population total      rate
## 1       Vermont  VT     Northeast     625741     2 0.3196211
## 2 New Hampshire  NH     Northeast    1316470     5 0.3798036
## 3        Hawaii  HI          West    1360301     7 0.5145920
## 4  North Dakota  ND North Central     672591     4 0.5947151
## 5          Iowa  IA North Central    3046355    21 0.6893484
## 6         Idaho  ID          West    1567582    12 0.7655102
```

Note that the default behavior is to order in ascending order. In dplyr, the function desc transforms a vector so that it is in descending order. To sort the table in descending order, we can type:

```
murders%>%
  arrange(desc(rate))
```

```
##                   state abb        region population total       rate
## 1  District of Columbia  DC         South     601723    99 16.4527532
## 2             Louisiana  LA         South    4533372   351  7.7425810
## 3              Missouri  MO North Central    5988927   321  5.3598917
## 4              Maryland  MD         South    5773552   293  5.0748655
## 5        South Carolina  SC         South    4625364   207  4.4753235
## 6              Delaware  DE         South     897934    38  4.2319369
## 7              Michigan  MI North Central    9883640   413  4.1786225
## 8           Mississippi  MS         South    2967297   120  4.0440846
## 9               Georgia  GA         South    9920000   376  3.7903226
## 10              Arizona  AZ          West    6392017   232  3.6295273
## 11         Pennsylvania  PA     Northeast   12702379   457  3.5977513
## 12            Tennessee  TN         South    6346105   219  3.4509357
## 13              Florida  FL         South   19687653   669  3.3980688
## 14           California  CA          West   37253956  1257  3.3741383
## 15           New Mexico  NM          West    2059179    67  3.2537239
## 16                Texas  TX         South   25145561   805  3.2013603
## 17             Arkansas  AR         South    2915918    93  3.1893901
## 18             Virginia  VA         South    8001024   250  3.1246001
## 19               Nevada  NV          West    2700551    84  3.1104763
## 20       North Carolina  NC         South    9535483   286  2.9993237
## 21             Oklahoma  OK         South    3751351   111  2.9589340
## 22             Illinois  IL North Central   12830632   364  2.8369608
## 23              Alabama  AL         South    4779736   135  2.8244238
## 24           New Jersey  NJ     Northeast    8791894   246  2.7980319
## 25          Connecticut  CT     Northeast    3574097    97  2.7139722
## 26                 Ohio  OH North Central   11536504   310  2.6871225
## 27               Alaska  AK          West     710231    19  2.6751860
## 28             Kentucky  KY         South    4339367   116  2.6732010
## 29             New York  NY     Northeast   19378102   517  2.6679599
## 30               Kansas  KS North Central    2853118    63  2.2081106
## 31              Indiana  IN North Central    6483802   142  2.1900730
## 32        Massachusetts  MA     Northeast    6547629   118  1.8021791
## 33             Nebraska  NE North Central    1826341    32  1.7521372
## 34            Wisconsin  WI North Central    5686986    97  1.7056487
## 35         Rhode Island  RI     Northeast    1052567    16  1.5200933
## 36        West Virginia  WV         South    1852994    27  1.4571013
## 37           Washington  WA          West    6724540    93  1.3829942
## 38             Colorado  CO          West    5029196    65  1.2924531
## 39              Montana  MT          West     989415    12  1.2128379
## 40            Minnesota  MN North Central    5303925    53  0.9992600
## 41         South Dakota  SD North Central     814180     8  0.9825837
## 42               Oregon  OR          West    3831074    36  0.9396843
## 43              Wyoming  WY          West     563626     5  0.8871131
## 44                Maine  ME     Northeast    1328361    11  0.8280881
## 45                 Utah  UT          West    2763885    22  0.7959810
## 46                Idaho  ID          West    1567582    12  0.7655102
## 47                 Iowa  IA North Central    3046355    21  0.6893484
## 48         North Dakota  ND North Central     672591     4  0.5947151
## 49               Hawaii  HI          West    1360301     7  0.5145920
## 50        New Hampshire  NH     Northeast    1316470     5  0.3798036
## 51              Vermont  VT     Northeast     625741     2  0.3196211
```

4.9.1 Nested sorting

If we are ordering by a column with ties, we can use a second column to break the tie. Similarly, a third column can be used to break ties between first and second and so on. Here we order by region, then within region we order by murder rate:

```
murders %>%
  arrange(region,rate)%>%
  head()
```

```
##             state abb    region population total      rate
## 1       Vermont  VT Northeast     625741     2 0.3196211
## 2 New Hampshire  NH Northeast    1316470     5 0.3798036
## 3         Maine  ME Northeast    1328361    11 0.8280881
## 4  Rhode Island  RI Northeast    1052567    16 1.5200933
## 5 Massachusetts  MA Northeast    6547629   118 1.8021791
## 6      New York  NY Northeast   19378102   517 2.6679599
```

4.9.2 The top *n*

In the code above, we have used the function head to avoid having the page fill up with the entire dataset. If we want to see a larger proportion, we can use the top_n function. This function takes a data frame as it's first argument, the number of rows to show in the second, and the variable to filter by in the third. Here is an example of how to see the top 5 rows:

```
murders %>% top_n(5,rate)
```

```
##                  state abb        region population total      rate
## 1 District of Columbia  DC         South     601723    99 16.452753
## 2            Louisiana  LA         South    4533372   351  7.742581
## 3             Maryland  MD         South    5773552   293  5.074866
## 4             Missouri  MO North Central    5988927   321  5.359892
## 5       South Carolina  SC         South    4625364   207  4.475323
```

Note that rows are not sorted by rate, only filtered. If we want to sort, we need to use arrange. Note that if the third argument is left blank, top_n filters by the last column.

4.10 Exercises

For these exercises, we will be using the data from the survey collected by the United States National Center for Health Statistics (NCHS). This center has conducted a series of health and nutrition surveys since the 1960's. Starting in 1999, about 5,000 individuals of all ages have been interviewed every year and they complete the health examination component of the survey. Part of the data is made available via the NHANES package. Once you install the NHANES package, you can load the data like this:

```
library(NHANES)
data(NHANES)
```

The NHANES data has many missing values. The mean and sd functions in R will return NA if any of the entries of the input vector is an NA. Here is an example:

```
library(dslabs)
data(na_example)
mean(na_example)
```

```
## [1] NA
```

To ignore the NAs we can use the na.rm argument:

```
mean(na_example, na.rm=TRUE)
```

```
## [1] 2.301754
```

```
sd(na_example, na.rm=TRUE)
```

```
## [1] 1.22338
```

Let's now explore the NHANES data.

1. We will provide some basic facts about blood pressure. First let's select a group to set the standard. We will use 20-to-29-year-old females. AgeDecade is a categorical variable with these ages. Note that the category is coded like " 20-29", with a space in front! What is the average and standard deviation of systolic blood pressure as saved in the BPSysAve variable? Save it to a variable called ref.

```
ref<-NHANES %>%
  filter(AgeDecade == " 20-29",Gender=="female")%>%
  summarize(average=mean(BPSysAve,na.rm=T), standard_deviation=sd(BPSysAve,na.rm=T))

ref
```

```
## # A tibble: 1 x 2
##   average standard_deviation
##     <dbl>              <dbl>
## 1    108.               10.1
```

Hint: Use filter and summarize and use the na.rm = TRUE argument when computing the average and standard deviation. You can also filter the NA values using filter.

2. Using a pipe, assign the average to a numeric variable ref_avg. Hint: Use the code similar to above and then pull.

```
ref_avg <- NHANES%>%
  filter(AgeDecade==" 20-29",Gender=="female")%>%
  summarize(average=mean(BPSysAve,na.rm=T))%>%
  pull()

ref_avg
```

```
## [1] 108.4224
```

3. Now report the min and max values for the same group.

```
NHANES %>%
  filter(AgeDecade==" 20-29", Gender=="female")%>%
  summarize(max=max(BPSysAve, na.rm=T), min=min(BPSysAve,na.rm=T))
```

```
## # A tibble: 1 x 2
##     max   min
##   <int> <int>
## 1   179    84
```

4. Compute the average and standard deviation for females, but for each age group separately rather than a selected decade as in question 1. Note that the age groups are defined by AgeDecade. Hint: rather than filtering by age and gender, filter by Gender and then use group_by.

```
NHANES %>%
  filter(Gender=="female")%>%
  group_by(AgeDecade)%>%
  summarize(average=mean(BPSysAve,na.rm=T), standard_deviation=sd(BPSysAve,na.rm=T))
```

```
## # A tibble: 9 x 3
##   AgeDecade average standard_deviation
##   <fct>       <dbl>              <dbl>
## 1 " 0-9"       100.               9.07
## 2 " 10-19"     104.               9.46
## 3 " 20-29"     108.              10.1
## 4 " 30-39"     111.              12.3
## 5 " 40-49"     115.              14.5
## 6 " 50-59"     122.              16.2
## 7 " 60-69"     127.              17.1
## 8 " 70+"       134.              19.8
## 9  <NA>        142.              22.9
```

5. Repeat exercise 4 for males.

```
NHANES %>%
  filter(Gender=="male")%>%
  group_by(AgeDecade)%>%
  summarize(average=mean(BPSysAve,na.rm=T), standard_deviation=sd(BPSysAve,na.rm=T))
```

```
## # A tibble: 9 x 3
##   AgeDecade average standard_deviation
##   <fct>       <dbl>              <dbl>
## 1 " 0-9"       97.4               8.32
## 2 " 10-19"    110.               11.2
## 3 " 20-29"    118.               11.3
## 4 " 30-39"    119.               12.3
## 5 " 40-49"    121.               14.0
## 6 " 50-59"    126.               17.8
## 7 " 60-69"    127.               17.5
## 8 " 70+"      130.               18.7
## 9  <NA>       136.               23.5
```

6. We can actually combine both summaries for exercises 4 and 5 into one line of code. This is because group_by permits us to group by more than one variable. Obtain one big summary table using group_by(AgeDecade, Gender).

```
NHANES %>%
  group_by(AgeDecade, Gender)%>%
  summarize(average=mean(BPSysAve,na.rm=T), standard_deviation=sd(BPSysAve,na.rm=T))
```

```
## ‘summarise()‘ has grouped output by ’AgeDecade’. You can override using the ‘.groups‘ argument.
```

```
## # A tibble: 18 x 4
## # Groups:   AgeDecade [9]
##    AgeDecade Gender average standard_deviation
##    <fct>     <fct>    <dbl>              <dbl>
##  1 " 0-9"    female   100.                9.07
##  2 " 0-9"    male      97.4               8.32
##  3 " 10-19"  female   104.                9.46
##  4 " 10-19"  male     110.               11.2
##  5 " 20-29"  female   108.               10.1
##  6 " 20-29"  male     118.               11.3
##  7 " 30-39"  female   111.               12.3
##  8 " 30-39"  male     119.               12.3
##  9 " 40-49"  female   115.               14.5
## 10 " 40-49"  male     121.               14.0
## 11 " 50-59"  female   122.               16.2
## 12 " 50-59"  male     126.               17.8
## 13 " 60-69"  female   127.               17.1
## 14 " 60-69"  male     127.               17.5
## 15 " 70+"    female   134.               19.8
## 16 " 70+"    male     130.               18.7
## 17  <NA>     female   142.               22.9
## 18  <NA>     male     136.               23.5
```

7. For males between the ages of 40-49, compare systolic blood pressure across race as reported in the Race1 variable. Order the resulting table from lowest to highest average systolic blood pressure.

```
NHANES%>%
  filter(AgeDecade==" 40-49", Gender=="male")%>%
  group_by(Race1)%>%
  summarize(systolic_blood_pressure = mean(BPSysAve, na.rm=T))%>%
  arrange(systolic_blood_pressure)
```

```
## # A tibble: 5 x 2
##   Race1    systolic_blood_pressure
##   <fct>                      <dbl>
## 1 White                       120.
## 2 Other                       120.
## 3 Hispanic                    122.
## 4 Mexican                     122.
## 5 Black                       126.
```

4.11 Tibbles

Tidy data must be stored in data frames. We introduced the data frame in Section 2.4.1 and have been using the murders data frame throughout the book. In Section 4.7.3 we introduced the group_by function, which permits stratifying data before computing summary statistics. But where is the group information stored in the data frame?

```
murders%>%group_by(region)
```

```
## # A tibble: 51 x 6
## # Groups:   region [4]
##    state                abb   region    population total  rate
##    <chr>                <chr> <fct>          <dbl> <dbl> <dbl>
##  1 Alabama              AL    South        4779736   135  2.82
##  2 Alaska               AK    West          710231    19  2.68
##  3 Arizona              AZ    West         6392017   232  3.63
##  4 Arkansas             AR    South        2915918    93  3.19
##  5 California           CA    West        37253956  1257  3.37
##  6 Colorado             CO    West         5029196    65  1.29
##  7 Connecticut          CT    Northeast    3574097    97  2.71
##  8 Delaware             DE    South         897934    38  4.23
##  9 District of Columbia DC    South         601723    99 16.5
## 10 Florida              FL    South       19687653   669  3.40
## # ... with 41 more rows
```

Notice that there are no columns with this information. But, if you look closely at the output above, you see the line A tibble followd by dimensions. We can learn the class of the returned object using:

```
murders%>%group_by(region)%>%class()
```

```
## [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

The tbl, pronounced tibble, is a special kind of data frame. The functions group_by and summarize always return this type of data frame. The group_by function returns a special kind of tbl, the grouped_df. We will say more about these later. For consistency, the dplyr manipulation verbs (select, filter, mutate, and arrange) preserve the class of the input: if they receive a regular data frame they return a regular data frame, while if they receive a tibble they return a tibble. But tibbles are the preferred format in the tidyverse and as a result tidyverse functions that produce a data frame from scratch return a tibble. For example, in Chapter 5 we will see that tidyverse functions used to import data create tibbles.

Tibbles are very similar to data frames. In fact, you can think of them as a modern version of data frames. Nonetheless there are three important differences which we describe next.

4.11.1 Tibbles display better

The print method for tibbles is more readable than that of a data frame. To see this, compare the outputs of typing murders and the output of murders if we convert it to a tibble. We can do this using as_tibble(murders). If using RStudio, output for a tibble adjusts to your window size. To see this, change the width of your R console and notice how more/less columns are shown.

4.11.2 Subsets of tibbles are tibbles

If you subset the columns of a data frame, you may get back an object that is not a data frame, such as a vector or scalar. For example:

```
class(murders[,4])
```

```
## [1] "numeric"
```

is not a data frame. With tibbles this does not happen:

```
class(as_tibble(murders)[,4])
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

This is useful in the tidyverse since functions require data frames as input.

With tibbles, if you want to access the vector that defines a column, and not get back a data frame, you need to use the accessor $:

```
class(as_tibble(murders)$population)
```

```
## [1] "numeric"
```

A related feature is that tibbles will give you a warning if you try to access a column that does not exist. If we accidentally write Population instead of population this:

```
as_tibble(murders)$Population
```

```
## Warning: Unknown or uninitialised column: ‘Population‘.
```

```
## NULL
```

4.11.3 Tibblees can have complex entries

While data frame columns need to be vectors of numbers, strings, or logical values, tibbles can have more complex objects, such as lists or functions. Also, we can create tibbles with functions:

```
tibble(id=c(1,2,3), func=c(mean,median, sd))
```

```
## # A tibble: 3 x 2
##      id func
##   <dbl> <list>
## 1     1 <fn>
## 2     2 <fn>
## 3     3 <fn>
```

4.11.4 Tibbles can be grouped

The function group_by returns a special kind of tibble: a grouped tibble. This class stores information that lets you know which rows are in which groups. The tidyverse functions, in particular the summarize function, are aware of the group information.

4.11.5 Create a *tibble* using tibble instead of *data.frame*

It is sometimes useful for us to create our own data frames. To create a data frame in the tibble format, you can do this by using the tibble function.

```
grades <-tibble(names=c("John", "Juan","Jean","Yao"), exam_1 =c(95,80,90,85), exam_2=c(90,85,85,90))
```

Note that base R (without packages loaded) has a function with a very similar name, data.frame, that can be used to create a regular data frame rather than a tibble.

```
grades<-data.frame(names=c("John", "Juan","Jean","Yao"), exam_1 =c(95,80,90,85), exam_2=c(90,85,85,90))
```

To convert a regular data frame to a tibble, you can use the as_tibble function.

```
as_tibble(grades)%>% class()
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

4.12 The dot operator

One of the advantages of using the pipe %>% is that we do not have to keep naming new objects as we manipulate the data frame. As a quick reminder, if we want to compute the median murder rate for states in the southern states, instead of typing:

```
tab_1 <- filter(murders, region == "South")
tab_2 <- mutate(tab_1, rate = total / population * 10^5)
rates <- tab_2$rate
median(rates)
```

```
## [1] 3.398069
```

We cna avoid defining any new intermediate objects by instead typing:

```
filter(murders, region == "South") %>%
  mutate(rate = total / population * 10^5) %>%
  summarize(median = median(rate)) %>%
  pull(median)
```

```
## [1] 3.398069
```

We can do this because each of these functions takes a data frame as the first argument. But what if we want to access a component of the data frame. For example, what if the pull function was not available and we wanted to access tab_2$rate? What data frame name would we use? The answer is the dot operator.

For example to access the rate vector without the pull function we could use

```
rates <-   filter(murders, region == "South") %>%
  mutate(rate = total / population * 10^5) %>%
  .$rate
median(rates)
```

```
## [1] 3.398069
```

4.13 The purrr package

In Section 3.5 we learned about the sapply function, which permitted us to apply the same function to each element of a vector. We constructed a function and used sapply to compute the sum of the first n integers for several values of n like this:

```
compute_s_n <- function(n){
  x<-1:n
  sum(x)
}
n<-1:25
s_n <- sapply(n,compute_s_n)
```

This type of operation, applying the same function or procedure to elements of an object, is quite common in data analysis. The purrr package includes functions similar to sapply but that better interact with other tidyverse functions. The main advantage is that we can better control the output type of functions. In contrast, sapply can return several different object types; for example, we might expect a numeric result from a line of code, but sapply might convert our result to character under some circumstances. purrr functions will never do this: they will return objects of a specified type or return an error if this is not possible.

The first purrr function we will learn is map, which works very similar to sapply but always, without exception, returns a list:

```
library(purrr)
s_n <- map(n,compute_s_n)
class(s_n)
```

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

If we want a numeric vector, we can instead use map_dbl which always returns a vector of numeric values.

```
s_n <- map_dbl(n, compute_s_n)
class(s_n)
```

```
## [1] "numeric"
```

This produces the same results as the sapply call shown above.

A particularly useful purrr function for interacting with the rest of the tidyverse is map_df, which always returns a tibble data frame. However, the function being called needs to return a vector or a list with names. For this reason, the following code would result in a Argument 1 must have names error:

```
# s_n <- map_df(n, compute_s_n)
```

We need to change the function to make this work:

```
compute_s_n <- function(n){
  x<-1:n
  tibble(sum = sum(x))
}
s_n <- map_df(n,compute_s_n)
```

The purrr package provides much more functionality not covered here. For more details you can consult this online resource.

4.14 Tidyverse conditionals

A typical data analysis will often involve one or more conditional operations. In Section 3.1 we described the ifelse function, which we will use extensively in this book. In this section we present two dplyr functions that provide further functionality for performing conditional operations.

4.14.1 case_when

The case_when function is useful for vectorizing conditional statements. It is similar to ifelse but can output any number of values, as opposed to just TRUE or FALSE. Here is an example splitting numbers into negative, positive, and 0:

```
x <- c(-2, -1, 0, 1, 2)
case_when(x < 0 ~ "Negative",
          x > 0 ~ "Positive",
          TRUE  ~ "Zero")
```

```
## [1] "Negative" "Negative" "Zero"     "Positive" "Positive"
```

A common use for this function is to define categorical variables based on existing variables. For example, suppose we want to compare the murder rates in four groups of states: New England, West Coast, South, and other. For each state, we need to ask if it is in New England, if it is not we ask if it is in the West Coast, if not we ask if it is in the South, and if not we assign other. Here is how we use case_when to do this:

```
murders%>%
  mutate(group = case_when(
    abb %in% c("ME", "NH", "VT", "MA", "RI", "CT") ~ "New England",
    abb %in% c("WA", "OR", "CA") ~ "West Coast",
    region == "South" ~ "South",
    TRUE ~ "Other")) %>%
  group_by(group) %>%
  summarize(rate = sum(total) / sum(population) * 10^5)
```

```
## # A tibble: 4 x 2
##   group        rate
##   <chr>       <dbl>
## 1 New England  1.72
## 2 Other        2.71
## 3 South        3.63
## 4 West Coast   2.90
```

4.14.2 between

A common operation in data analysis is to determine if a value falls inside an interval. We can check this using conditionals. For example, to check if the elements of a vector x are between a and b we can type

```
## x >= a & x<= b
```

However, this can become cumbersome, especially within the tidyverse approach. The between function performs the same operation.

```
# between(x,a,b)
```

4.15 Exercises

1. Load the murders dataset. Which of the following is true?

```
library(dslabs)
data(murders)
# The answer is b. murders is in tidy format and is stored in a data frame.
```

2. Use as_tibble to convert the murders data table into a tibble and save it in an object called murders_tibble.

```
murders_tibble <- as_tibble(murders)
```

3. Use the group_by function to convert murders into a tibble that is grouped by region.

```
as_tibble(murders)%>%
  group_by(region)
```

```
## # A tibble: 51 x 5
## # Groups:   region [4]
##    state                abb   region    population total
##    <chr>                <chr> <fct>          <dbl> <dbl>
##  1 Alabama              AL    South        4779736   135
##  2 Alaska               AK    West          710231    19
##  3 Arizona              AZ    West         6392017   232
##  4 Arkansas             AR    South        2915918    93
##  5 California           CA    West        37253956  1257
##  6 Colorado             CO    West         5029196    65
##  7 Connecticut          CT    Northeast    3574097    97
##  8 Delaware             DE    South         897934    38
##  9 District of Columbia DC    South         601723    99
## 10 Florida              FL    South       19687653   669
## # ... with 41 more rows
```

4. Write tidyverse code that is equivalent to this code:

```
# exp(mean(log(murders$population)))murders%>%
```

Write it using the pipe so that each function is called without arguments. Use the dot operator to access the population. Hint: The code should start with murders %>%.

```
murders%>%
  .$population%>%
  log()%>%
  mean()%>%
  exp()
```

```
## [1] 3675209
```

5. Use the map_df to create a data frame with three columns named n, s_n, and s_n_2. The first column should contain the numbers 1 through 100. The second and third columns should each contain the sum of 1 through n with n the row number.

```
x<-1:100
compute_s_n <-function(n){
  x<-1:n
  tibble(n=n,s_n=sum(x),s_n_2=sum(x))
}
map_df(x,compute_s_n)
```

```
## # A tibble: 100 x 3
##        n   s_n s_n_2
##    <int> <int> <int>
##  1     1     1     1
##  2     2     3     3
##  3     3     6     6
##  4     4    10    10
##  5     5    15    15
##  6     6    21    21
##  7     7    28    28
##  8     8    36    36
##  9     9    45    45
## 10    10    55    55
## # ... with 90 more rows
```

Chapter5 Importing data

We have been using data sets already stored as R objects. A data scientist will rarely have such luck and will have to import data into R from either a file, a database, or other sources. Currently, one of the most common ways of storing and sharing data for analysis is through electronic spreadsheets. A spreadsheet stores data in rows and columns. It is basically a file version of a data frame. When saving such a table to a computer file, one needs a way to define when a new row or column ends and the other begins. This in turn defines the cells in which single values are stored.

When creating spreadsheets with text files, like the ones created with a simple text editor, a new row is defined with return and columns are separated with some predefined special character. The most common characters are comma (,), semicolon (;), space ( ), and tab (a preset number of spaces or)͡. Here is an example of what a comma separated file looks like if we open it with a basic text editor:

The first row contains column names rather than data. We call this a header, and when we read-in data from a spreadsheet it is important to know if the file has a header or not. Most reading functions assume there is a header. To know if the file has a header, it helps to look at the file before trying to read it. This can be done with a text editor or with RStudio. In RStudio, we can do this by either opening the file in the editor or navigating to the file location, double clicking on the file, and hitting View File.

However, not all spreadsheet files are in a text format. Google Sheets, which are rendered on a browser, are an example. Another example is the proprietary format used by Microsoft Excel. These can't be viewed with a text editor. Despite this, due to the widespread use of Microsoft Excel software, this format is widely used.

We start this chapter by describing the difference between text (ASCII), Unicode, and binary files and how this affects how we import them. We then explain the concepts of file paths and working directories, which are essential to understand how to import data effectively. We then introduce the readr and readxl package and the functions that are available to import spreadsheets into R. Finally, we provide some recommendations on how to store and organize data in files. More complex challenges such as extracting data from web pages or PDF documents are left for the Data Wrangling part of the book.

5.1 Paths and the working directory

The first step when importing data from a spreadsheet is to locate the file containing the data. Although we do not recommend it, you can use an approach similar to what you do to open files in Microsoft Excel by clicking on the RStudio "File" menu, clicking "Import Dataset," then clicking through folders until you find the file. We want to be able to write code rather than use the point-and-click approach. The keys and concepts we need to learn to do this are described in detail in the Productivity Tools part of this book. Here we provide an overview of the very basics.

The main challenge in this first step is that we need to let the R functions doing the importing know where to look for the file containing the data. The simplest way to do this is to have a copy of the file in the folder in which the importing functions look by default. Once we do this, all we have to supply to the importing function is the filename.

A spreadsheet containing the US murders data is included as part of the dslabs package. Finding this file is not straightforward, but the following lines of code copy the file to the folder in which R looks in by default. We explain how these lines work below.

```
filename <- "murders.csv"
dir <- system.file("extdata", package="dslabs")
fullpath<-file.path(dir,filename)
file.copy(fullpath,"murders.csv")
```

```
## [1] FALSE
```

This code does not read the data into R, it just copies a file. But once the file is copied, we can import the data with a simple line of code. Here we use the read_csv function from the readr package, which is part of the tidyverse.

```
library(tidyverse)
dat <- read_csv(filename)
```

```
## Rows: 51 Columns: 5
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (3): state, abb, region
## dbl (2): population, total

##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

The data is imported and stored in dat. The rest of this section defines some important concepts and provides an overview of how we write code that tells R how to find the files we want to import. Chapter 38 provides more details on this topic.

5.1.1 The filesystem

You can think of your computer's filesystem as a series of nested folders, each containing other folders and files. Data scientists refer to folders as directories. We refer to the folder that contains all other folders as the root directory. We refer to the directory in which we are currently located as the working directory. The working directory therefore changes as you move through folders: think of it as your current location.

5.1.2 Relative and full paths

The path of a file is a list of directory names that can be thought of as instructions on what folders to click on, and in what order, to find the file. If these instructions are for finding the file from the root directory we refer to it as the *full path*. If the instructions are for finding the file starting in the working directory we refer to it as a *relative path*. Section 38.3 provides more details on this topic.

To see an example of a full path on your system type the following:

```
system.file(package="dslabs")
```

```
## [1] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library/dslabs"
```

The strings separated by slashes are the directory names. The first slash represents the root directory and we know this is a full path because it starts with a slash. If the first directory name appears without a slash in front, then the path is assumed to be relative. We can use the function list.files to see examples of relative paths.

```
dir <- system.file(package="dslabs")
list.files(path=dir)
```

```
##  [1] "data"        "DESCRIPTION" "extdata"     "help"        "html"
##  [6] "INDEX"       "Meta"        "NAMESPACE"   "R"           "script"
```

These relative paths give us the location of the files or directories if we start in the directory with the full path. For example, the full path to the help directory in the example above is **/Library/Frameworks/R.framework/Versions/3.5/Resources/library/dslabs/help**

Note: You will probably not make much use of the system.file function in your day-to-day data analysis work. We introduce it in this section because it facilitates the sharing of spreadsheets by including them in the dslabs package. You will rarely have the luxury of data being included in packages you already have installed. However, you will frequently need to navigate full and relative paths and import spreadsheet formatted data.

5.1.3 The working directory

We highly recommend only writing relative paths in your code. The reason is that full paths are unique to your computer and you want your code to be portable. You can get the full path of your working directory without writing out explicitly by using the getwd function.

```
wd<-getwd()
wd
```

```
## [1] "/Users/choeseunghwan/Documents/GitHub/bsms222_105_choi/assignment"
```

5.1.4 Generating path names

Another example of obtaining a full path without writing out explicitly was given above when we created the object fullpath like this:

```
filename<-"murders.csv"
dir <- system.file("extdata", package = "dslabs")
fullpath <- file.path(dir, filename)
```

The function system.file provides the full path of the folder containing all the files and directories relevant to the package specified by the package argument. By exploring the directories in dir we find that the extdata contains the file we want:

```
dir <- system.file(package = "dslabs")
filename %in% list.files(file.path(dir, "extdata"))
```

```
## [1] TRUE
```

The system.file function permits us to provide a subdirectory as a first argument, so we can obtain the fullpath of the extdata directory like this:

```
dir <- system.file("extdata", package = "dslabs")
```

The function file.path is used to combine directory names to produce the full path of the file we want to import.

```
fullpath <- file.path(dir, filename)
```

5.1.5 Copying files using paths

The final line of code we used to copy the file into our home directory used the function file.copy. This function takes two arguments: the file to copy and the name to give it in the new directory.

```
file.copy(fullpath, "murders.csv", overwrite=T)
```

```
## [1] TRUE
```

If a file is copied successfully, the file.copy function returns TRUE. Note that we are giving the file the same name, murders.csv, but we could have named it anything. Also note that by not starting the string with a slash, R assumes this is a relative path and copies the file to the working directory.

You should be able to see the file in your working directory and can check by using:

```
list.files()
```

```
##  [1] "[Tutorial] SCN2A mutations.nb.html" "[Tutorial] SCN2A mutations.Rmd"
##  [3] "Chapter2 assignment.nb.html"        "Chapter2 assignment.Rmd"
##  [5] "Chapter2-assignment.html"           "Chapter3 assignment.nb.html"
##  [7] "Chapter3 assignment.Rmd"            "Chapter3-assignment.html"
##  [9] "Chapter4 assignment_4.6.Rmd"        "Chapter4-assignment_4.6.html"
## [11] "Chapter4-assignment_4.6.pdf"        "Chapter4,5.html"
## [13] "Chapter4,5.nb.html"                 "Chapter4,5.pdf"
## [15] "Chapter4,5.Rmd"                     "murders.csv"
## [17] "test2.html"                         "test2.nb.html"
## [19] "test2.Rmd"
```

```
#"murders.csv" is copied to current directory.
```

5.2 The readr and readxl packages

In this section we introduce the main tidyverse data importing functions. We will use the murders.csv file provided by the dslabs package as an example. To simplify the illustration we will copy the file to our working directory using the following code:

```
filename <- "murders.csv"
dir <- system.file("extdata", package = "dslabs")
fullpath <- file.path(dir, filename)
file.copy(fullpath, "murders.csv")
```

```
## [1] FALSE
```

```
# In current directory, there is already identical name
# file, 'murders.csv' that I copied before. That's why
# >file.copy(fullpath,'murders.csv') return FALSE: not
# successful.
```

5.2.1 readr

The **readr** library includes functions for reading data stored in text file spreadsheets into R. **readr** is part of the **tidyverse** package, or you can load it directly:

```
library(readr)
```

The following functions are available to read-in spreadsheets:

```
library(readxl)
print(readxl::read_xlsx("/Users/choeseunghwan/Desktop/test.xlsx" ))
```

```
## # A tibble: 5 x 3
##   Function   Format                                            ‘Typical suffix‘
##   <chr>      <chr>                                             <chr>
## 1 read_table white space separated values                      txt
## 2 read_csv   comma separated values                            csv
## 3 read_csv2  semicolon separated values                        csv
## 4 read_tsv   tab delimited separated values                    tsv
## 5 read_delim general text file format, must define delimiter txt
```

Although the suffix usually tells us what type of file it is, there is no guarantee that these always match. We can open the file to take a look or use the function read_lines to look at a few lines:

```
read_lines("murders.csv",n_max=3)
```

```
## [1] "state,abb,region,population,total" "Alabama,AL,South,4779736,135"
## [3] "Alaska,AK,West,710231,19"
```

This also shows that there is a header. Now we are ready to read-in the data into R. From the .csv suffix and the peek at the file, we know to use read_csv:

```
dat <- read_csv(filename)
```

```
## Rows: 51 Columns: 5

## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (3): state, abb, region
## dbl (2): population, total

##
## i Use ‘spec()‘ to retrieve the full column specification for this data.
## i Specify the column types or set ‘show_col_types = FALSE‘ to quiet this message.
```

Note that we receive a message letting us know what data types were used for each column. Also note that dat is a tibble, not just a data frame. This is because read_csv is a tidyverse parser. We can confirm that the data has in fact been read-in with:

```
# View(dat)
```

Finally, note that we can also use the full path for the file:

```
dat<-read_csv(fullpath)
```

```
## Rows: 51 Columns: 5

## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (3): state, abb, region
## dbl (2): population, total

##
## i Use ‘spec()‘ to retrieve the full column specification for this data.
## i Specify the column types or set ‘show_col_types = FALSE‘ to quiet this message.
```

5.2.2 readxl

You can load the readxl package using

```
library(readxl)
```

The package provides functions to read-in Microsoft Excel formats:

```
print(readxl::read_xlsx("/Users/choeseunghwan/Desktop/test2.xlsx"))
```

```
## # A tibble: 3 x 3
##   Function   Format                 `Typical suffix`
##   <chr>      <chr>                  <chr>
## 1 read_excel auto detect the format xls, xlsx
## 2 read_xls   original format        xls
## 3 read_xlsx  new format             xlsx
```

The Microsoft Excel formats permit you to have more than one spreadsheet in one file. These are referred to as sheets. The functions listed above read the first sheet by default, but we can also read the others. The excel_sheets function gives us the names of all the sheets in an Excel file. These names can then be passed to the sheet argument in the three functions above to read sheets other than the first.

5.3 Exercises

1. Use the read_csv function to read each of the files that the following code saves in the *files* object:

```
path <- system.file("extdata", package = "dslabs")
files <- list.files(path)
files
```

```
## [1] "2010_bigfive_regents.xls"
## [2] "carbon_emissions.csv"
## [3] "fertility-two-countries-example.csv"
## [4] "HRlist2.txt"
## [5] "life-expectancy-and-fertility-two-countries-example.csv"
## [6] "murders.csv"
## [7] "olive.csv"
## [8] "RD-Mortality-Report_2015-18-180531.pdf"
## [9] "ssa-death-probability.csv"
```

```
filespath <- file.path(path,files)
filespath <- filespath[c(2,3,5,6,7,9)] #To filter .csv file using subsetting
sapply(filespath, read_csv)
```

```
## Rows: 264 Columns: 2
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (2): Year, Total carbon emissions from fossil fuel consumption and cemen...
```

```
##
## i Use ‘spec()‘ to retrieve the full column specification for this data.
## i Specify the column types or set ‘show_col_types = FALSE‘ to quiet this message.
```

```
## Rows: 2 Columns: 57
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (1): country
## dbl (56): 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, ...
```

```
##
## i Use ‘spec()‘ to retrieve the full column specification for this data.
## i Specify the column types or set ‘show_col_types = FALSE‘ to quiet this message.
```

```
## Rows: 2 Columns: 113
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr   (1): country
## dbl (112): 1960_fertility, 1960_life_expectancy, 1961_fertility, 1961_life_e...
```

```
##
## i Use ‘spec()‘ to retrieve the full column specification for this data.
## i Specify the column types or set ‘show_col_types = FALSE‘ to quiet this message.
```

```
## Rows: 51 Columns: 5
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (3): state, abb, region
## dbl (2): population, total
```

```
##
## i Use ‘spec()‘ to retrieve the full column specification for this data.
## i Specify the column types or set ‘show_col_types = FALSE‘ to quiet this message.
```

```
## New names:
## * ‘‘ -> ...1
```

```
## Rows: 572 Columns: 11
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): Region, eicosenoic
## dbl (9): ...1, Area, palmitic, palmitoleic, stearic, oleic, linoleic, linole...
```

```
##
## i Use ‘spec()‘ to retrieve the full column specification for this data.
## i Specify the column types or set ‘show_col_types = FALSE‘ to quiet this message.
## Rows: 240 Columns: 5

## -- Column specification ------------------------------------------------------
## Delimiter: ","
## chr (1): Sex
## dbl (3): Age, DeathProb, LifeExp


##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

## $`/Library/Frameworks/R.framework/Versions/4.1/Resources/library/dslabs/extdata/carbon_emissions.csv
## # A tibble: 264 x 2
##     Year `Total carbon emissions from fossil fuel consumption and cement produc~
##    <dbl>                                                                   <dbl>
##  1  1751                                                                       3
##  2  1752                                                                       3
##  3  1753                                                                       3
##  4  1754                                                                       3
##  5  1755                                                                       3
##  6  1756                                                                       3
##  7  1757                                                                       3
##  8  1758                                                                       3
##  9  1759                                                                       3
## 10  1760                                                                       3
## # ... with 254 more rows
##
## $`/Library/Frameworks/R.framework/Versions/4.1/Resources/library/dslabs/extdata/fertility-two-countri
## # A tibble: 2 x 57
##   country  `1960` `1961` `1962` `1963` `1964` `1965` `1966` `1967` `1968` `1969`
##   <chr>     <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
## 1 Germany    2.41   2.44   2.47   2.49   2.49   2.48   2.44   2.37   2.28   2.17
## 2 South K~   6.16   5.99   5.79   5.57   5.36   5.16   4.99   4.85   4.73   4.62
## # ... with 46 more variables: 1970 <dbl>, 1971 <dbl>, 1972 <dbl>, 1973 <dbl>,
## #   1974 <dbl>, 1975 <dbl>, 1976 <dbl>, 1977 <dbl>, 1978 <dbl>, 1979 <dbl>,
## #   1980 <dbl>, 1981 <dbl>, 1982 <dbl>, 1983 <dbl>, 1984 <dbl>, 1985 <dbl>,
## #   1986 <dbl>, 1987 <dbl>, 1988 <dbl>, 1989 <dbl>, 1990 <dbl>, 1991 <dbl>,
## #   1992 <dbl>, 1993 <dbl>, 1994 <dbl>, 1995 <dbl>, 1996 <dbl>, 1997 <dbl>,
## #   1998 <dbl>, 1999 <dbl>, 2000 <dbl>, 2001 <dbl>, 2002 <dbl>, 2003 <dbl>,
## #   2004 <dbl>, 2005 <dbl>, 2006 <dbl>, 2007 <dbl>, 2008 <dbl>, 2009 <dbl>, ...
##
## $`/Library/Frameworks/R.framework/Versions/4.1/Resources/library/dslabs/extdata/life-expectancy-and-
## # A tibble: 2 x 113
##   country     `1960_fertility` `1960_life_expe~ `1961_fertility` `1961_life_expe~
##   <chr>                  <dbl>            <dbl>            <dbl>            <dbl>
## 1 Germany                 2.41             69.3             2.44             69.8
## 2 South Korea             6.16             53.0             5.99             53.8
## # ... with 108 more variables: 1962_fertility <dbl>,
## #   1962_life_expectancy <dbl>, 1963_fertility <dbl>,
## #   1963_life_expectancy <dbl>, 1964_fertility <dbl>,
## #   1964_life_expectancy <dbl>, 1965_fertility <dbl>,
## #   1965_life_expectancy <dbl>, 1966_fertility <dbl>,
## #   1966_life_expectancy <dbl>, 1967_fertility <dbl>,
## #   1967_life_expectancy <dbl>, 1968_fertility <dbl>, ...
##
## $`/Library/Frameworks/R.framework/Versions/4.1/Resources/library/dslabs/extdata/murders.csv`
## # A tibble: 51 x 5
##    state                abb   region    population total
##    <chr>                <chr> <chr>          <dbl> <dbl>
##  1 Alabama              AL    South        4779736   135
##  2 Alaska               AK    West          710231    19
##  3 Arizona              AZ    West         6392017   232
##  4 Arkansas             AR    South        2915918    93
##  5 California           CA    West        37253956  1257
##  6 Colorado             CO    West         5029196    65
##  7 Connecticut          CT    Northeast    3574097    97
##  8 Delaware             DE    South         897934    38
##  9 District of Columbia DC    South         601723    99
## 10 Florida              FL    South       19687653   669
## # ... with 41 more rows
##
## $`/Library/Frameworks/R.framework/Versions/4.1/Resources/library/dslabs/extdata/olive.csv`

## Warning: One or more parsing issues, see `problems()` for details

## # A tibble: 572 x 11
##     ...1 Region        Area palmitic palmitoleic stearic oleic linoleic linolenic
##    <dbl> <chr>        <dbl>    <dbl>       <dbl>   <dbl> <dbl>    <dbl>     <dbl>
##  1     1 North-Apulia     1        1        1075      75   226     7823       672
##  2     2 North-Apulia     1        1        1088      73   224     7709       781
##  3     3 North-Apulia     1        1         911      54   246     8113       549
##  4     4 North-Apulia     1        1         966      57   240     7952       619
##  5     5 North-Apulia     1        1        1051      67   259     7771       672
##  6     6 North-Apulia     1        1         911      49   268     7924       678
##  7     7 North-Apulia     1        1         922      66   264     7990       618
##  8     8 North-Apulia     1        1        1100      61   235     7728       734
##  9     9 North-Apulia     1        1        1082      60   239     7745       709
## 10    10 North-Apulia     1        1        1037      55   213     7944       633
## # ... with 562 more rows, and 2 more variables: arachidic <dbl>,
## #   eicosenoic <chr>
##
## $`/Library/Frameworks/R.framework/Versions/4.1/Resources/library/dslabs/extdata/ssa-death-probability
## # A tibble: 240 x 5
##      Age Sex   DeathProb NumberOfLives LifeExp
##    <dbl> <chr>     <dbl>         <dbl>   <dbl>
##  1     0 Male   0.00638         100000    76.2
##  2     1 Male   0.000453         99362    75.6
##  3     2 Male   0.000282         99317    74.7
##  4     3 Male   0.00023          99289    73.7
##  5     4 Male   0.000169         99266    72.7
##  6     5 Male   0.000155         99249    71.7
##  7     6 Male   0.000145         99234    70.7
##  8     7 Male   0.000135         99219    69.7
##  9     8 Male   0.00012          99206    68.8
## 10     9 Male   0.000105         99194    67.8
## # ... with 230 more rows
```

2. Note that the last one, the olive file, gives us a warning. This is because the first line of the file is missing the header for the first column.

Read the help file for read_csv to figure out how to read in the file without reading this header. If you skip the header, you should not get this warning. Save the result to an object called dat.

```
dat <- read_csv("/Library/Frameworks/R.framework/Versions/4.1/Resources/library/dslabs/extdata/olive.csv
    col_names = F, skip = 1)
```

```
## Rows: 572 Columns: 12
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (1): X2
## dbl (11): X1, X3, X4, X5, X6, X7, X8, X9, X10, X11, X12
```

```
##
## i Use ‘spec()‘ to retrieve the full column specification for this data.
## i Specify the column types or set ‘show_col_types = FALSE‘ to quiet this message.
```

3. A problem with the previous approach is that we don’t know what the columns represent. Type:

```
names(dat)
```

```
##  [1] "X1"  "X2"  "X3"  "X4"  "X5"  "X6"  "X7"  "X8"  "X9"  "X10" "X11" "X12"
```

to see that the names are not informative.

Use the readLines function to read in just the first line (we later learn how to extract values from the output).

```
readLines("/Library/Frameworks/R.framework/Versions/4.1/Resources/library/dslabs/extdata/olive.csv",
    n = 1)
```

```
## [1] ",Region,Area,palmitic,palmitoleic,stearic,oleic,linoleic,linolenic,arachidic,eicosenoic"
```

5.4 Downloading files

Another common place for data to reside is on the internet. When these data are in files, we can download them and then import them or even read them directly from the web. For example, we note that because our dslabs package is on GitHub, the file we downloaded with the package has a url:

```
url <-"https://raw.githubusercontent.com/rafalab/dslabs/master/inst/extdata/murders.csv"
```

The read_csv file can read these files directly:

```
dat<-read_csv(url)
```

```
## Rows: 51 Columns: 5
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (3): state, abb, region
## dbl (2): population, total
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

If you want to have a local copy of the file, you can use the download.file function:

```
download.file(url, "murders.csv")
```

This will download the file and save it on your system with the name murders.csv. You can use any name here, not necessarily murders.csv. Note that when using download.file you should be careful as **it will overwrite existing files without warning**.

Two functions that are sometimes useful when downloading data from the internet are tempdir and tempfile. The first creates a directory with a random name that is very likely to be unique. Similarly, tempfile creates a character string, not a file, that is likely to be a unique filename. So you can run a command like this which erases the temporary file once it imports the data:

```
tmp_filename <- tempfile()
download.file(url,tmp_filename)
dat <- read_csv(tmp_filename)
```

```
## Rows: 51 Columns: 5
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (3): state, abb, region
## dbl (2): population, total
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
file.remove(tmp_filename)
```

```
## [1] TRUE
```

5.5 R-base importing functions

R-base also provides import functions. These have similar names to those in the tidyverse, for example read.table, read.csv and read.delim. You can obtain an data frame like dat using:

```
dat2<-read.csv(filename)
```

An often useful R-base importing function is scan, as it provides much flexibility. When reading in spreadsheets many things can go wrong. The file might have a multiline header, be missing cells, or it might use an unexpected encoding17. We recommend you read this post about common issues found here: https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/.

With experience you will learn how to deal with different challenges. Carefully reading the help files for the functions discussed here will be useful. With scan you can read-in each cell of a file. Here is an example:

```
path<-system.file("extdata", package="dslabs")
filename<- "murders.csv"
x <- scan(file.path(path,filename), sep=",", what="c")
x[1:10]
```

```
##  [1] "state"      "abb"        "region"     "population" "total"
##  [6] "Alabama"    "AL"         "South"      "4779736"    "135"
```

Note that the tidyverse provides read_lines, a similarly useful function.

5.6 Text versus binary files

For data science purposes, files can generally be classified into two categories: text files (also known as ASCII files) and binary files. You have already worked with text files. All your R scripts are text files and so are the R markdown files used to create this book. The csv tables you have read are also text files. One big advantage of these files is that we can easily "look" at them without having to purchase any kind of special software or follow complicated instructions. Any text editor can be used to examine a text file, including freely available editors such as RStudio, Notepad, textEdit, vi, emacs, nano, and pico. To see this, try opening a csv file using the "Open file" RStudio tool. You should be able to see the content right on your editor. However, if you try to open, say, an Excel xls file, jpg or png file, you will not be able to see anything immediately useful. These are binary files. Excel files are actually compressed folders with several text files inside. But the main distinction here is that text files can be easily examined.

Although R includes tools for reading widely used binary files, such as xls files, in general you will want to find data sets stored in text files. Similarly, when sharing data you want to make it available as text files as long as storage is not an issue (binary files are much more efficient at saving space on your disk). In general, plain-text formats make it easier to share data since commercial software is not required for working with the data.

Extracting data from a spreadsheet stored as a text file is perhaps the easiest way to bring data from a file to an R session. Unfortunately, spreadsheets are not always available and the fact that you can look at text files does not necessarily imply that extracting data from them will be straightforward. In the Data Wrangling part of the book we learn to extract data from more complex text files such as html files.

5.7 Unicodde versus ASCII

A pitfall in data science is assuming a file is an ASCII text file when, in fact, it is something else that can look a lot like an ASCII text file: a Unicode text file.

To understand the difference between these, remember that everything on a computer needs to eventually be converted to 0s and 1s. ASCII is an encoding that maps characters to numbers. ASCII uses 7 bits (0s and 1s) which results in 2^7=128 unique items, enough to encode all the characters on an English language keyboard. However, other languages use characters not included in this encoding. For example, the é in México is not encoded by ASCII. For this reason, a new encoding, using more than 7 bits, was defined: Unicode. When using Unicode, one can chose between 8, 16, and 32 bits abbreviated UTF-8, UTF-16, and UTF-32 respectively. RStudio actually defaults to UTF-8 encoding.

Although we do not go into the details of how to deal with the different encodings here, it is important that you know these different encodings exist so that you can better diagnose a problem if you encounter it. One way problems manifest themselves is when you see "weird looking" characters you were not expecting. This StackOverflow discussion is an example: https://stackoverflow.com/questions/18789330/r-on-windows-character-encoding-hell.

5.8 Organizaing data with spreadsheets

Although this book focuses almost exclusively on data analysis, data management is also an important part of data science. As explained in the introduction, we do not cover this topic. However, quite often data analysts needs to collect data, or work with others collecting data, in a way that is most conveniently stored in a spreadsheet. Although filling out a spreadsheet by hand is a practice we highly discourage, we instead recommend the process be automatized as much as possible, sometimes you just have to do it. Therefore, in this section, we provide recommendations on how to organize data in a spreadsheet. Although there are R packages designed to read Microsoft Excel spreadsheets, we generally want to avoid this format. Instead, we recommend Google Sheets as a free software tool. Below we summarize the recommendations made in paper by Karl Broman and Kara Woo18. Please read the paper for important details.

- **Be Consistent** - Before you commence entering data, have a plan. Once you have a plan, be consistent and stick to it.
- **Choose Good Names for Things** - You want the names you pick for objects, files, and directories to be memorable, easy to spell, and descriptive. This is actually a hard balance to achieve and it does require time and thought. One important rule to follow is do not use spaces, use underscores _ or dashes instead -. Also, avoid symbols; stick to letters and numbers.
- **Write Dates as YYYY-MM-DD** - To avoid confusion, we strongly recommend using this global ISO 8601 standard.
- **No Empty Cells** - Fill in all cells and use some common code for missing data.
- **Put Just One Thing in a Cell** - It is better to add columns to store the extra information rather than having more than one piece of information in one cell.
- **Make It a Rectangle** - The spreadsheet should be a rectangle.
- **Create a Data Dictionary** - If you need to explain things, such as what the columns are or what the labels used for categorical variables are, do this in a separate file.
- **No Calculations in the Raw Data Files** - Excel permits you to perform calculations. Do not make this part of your spreadsheet. Code for calculations should be in a script.
- **Do Not Use Font Color or Highlighting as Data** - Most import functions are not able to import this information. Encode this information as a variable instead.
- **Make Backups** - Make regular backups of your data.
- **Use Data Validation to Avoid Errors** - Leverage the tools in your spreadsheet software so that the process is as error-free and repetitive-stress-injury-free as possible.
- **Save the Data as Text Files** - Save files for sharing in comma or tab delimited format.

5.9 Exercises

1. Pick a measurement you can take on a regular basis. For example, your daily weight or how long it takes you to run 5 miles. Keep a spreadsheet that includes the date, the hour, the measurement, and any other informative variable you think is worth keeping. Do this for 2 weeks. Then make a plot.