chapter 4(4.7 - 4.15)

Contents

0.1 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.

0.1.1 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)
```

```
##
##          : '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
```

As 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.

0.1.2 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:

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.

0.1.3 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:

```
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
```

0.2 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(rate)

us_murder_rate
```

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

which is now a numeric:

```
class(us_murder_rate)
```

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

0.3 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
```

0.3.1 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
```

0.3.2 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.

0.4 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
```

```
sd(na_example)
```

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

To ignore the `NA`s 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`.

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`.

```
ref <- NHANES %>% filter(AgeDecade == " 20-29" & Gender == "female") %>% summarize(average=mean(BPSysAve

ref
```

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

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 <- ref %>% .$average

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(min = min(BPSysAve, na.rm = TRUE), max = max(BPSysAve, na.rm = TRUE))
```

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

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=TRUE), standard_deviation = sd(BPSysAve,na.rm=TRUE))
```

```
## # 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=TRUE), standard_deviation = sd(BPSysAve,na.rm=TRUE))
```

```
## # 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(Gender, AgeDecade) %>%
  summarize(average = mean(BPSysAve,na.rm=TRUE), standard_deviation = sd(BPSysAve,na.rm=TRUE))
```

```
## `summarise()` has grouped output by 'Gender'. You can override using the `.groups` argument.
```

```
## # A tibble: 18 x 4
## # Groups:   Gender [2]
##    Gender AgeDecade average standard_deviation
##    <fct>  <fct>       <dbl>              <dbl>
##  1 female " 0-9"      100.                9.07
##  2 female " 10-19"    104.                9.46
##  3 female " 20-29"    108.               10.1
##  4 female " 30-39"    111.               12.3
##  5 female " 40-49"    115.               14.5
##  6 female " 50-59"    122.               16.2
##  7 female " 60-69"    127.               17.1
##  8 female " 70+"      134.               19.8
##  9 female  <NA>       142.               22.9
## 10 male   " 0-9"       97.4               8.32
## 11 male   " 10-19"    110.               11.2
## 12 male   " 20-29"    118.               11.3
## 13 male   " 30-39"    119.               12.3
## 14 male   " 40-49"    121.               14.0
## 15 male   " 50-59"    126.               17.8
## 16 male   " 60-69"    127.               17.5
## 17 male   " 70+"      130.               18.7
## 18 male    <NA>       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(Gender == "male" & AgeDecade == " 40-49") %>%
  group_by(Race1) %>%
  summarize(average = mean(BPSysAve, na.rm = TRUE)) %>%
  arrange(average)
```

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

0.5 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.

0.5.1 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.

0.5.2 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:

```
murders$Population
```

```
## NULL
```

returns a `NULL` with no warning, which can make it harder to debug. In contrast, if we try this with a tibble we get an informative warning:

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

0.5.3 4.11.3 Tibbles 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>
```

0.5.4 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.

0.5.5 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"
```

0.6 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 can 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
```

0.7 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)

s_n
```

```
## # A tibble: 25 x 1
##      sum
##    <int>
##  1     1
##  2     3
##  3     6
##  4    10
##  5    15
##  6    21
##  7    28
##  8    36
##  9    45
## 10    55
## # ... with 15 more rows
```

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

0.8 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.

0.8.1 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
```

0.8.2 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.

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

0.9 4.15 Exercises

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

a. `murders` is in tidy format and is stored in a tibble.
b. `murders` is in tidy format and is stored in a data frame.
c. `murders` is not in tidy format and is stored in a tibble.
d. `murders` is not in tidy format and is stored in a data frame.

```r
murders
```

```
##                   state abb        region population total       rate
## 1               Alabama  AL         South    4779736   135  2.8244238
## 2                Alaska  AK          West     710231    19  2.6751860
## 3               Arizona  AZ          West    6392017   232  3.6295273
## 4              Arkansas  AR         South    2915918    93  3.1893901
## 5            California  CA          West   37253956  1257  3.3741383
## 6              Colorado  CO          West    5029196    65  1.2924531
## 7           Connecticut  CT     Northeast    3574097    97  2.7139722
## 8              Delaware  DE         South     897934    38  4.2319369
## 9  District of Columbia  DC         South     601723    99 16.4527532
## 10              Florida  FL         South   19687653   669  3.3980688
## 11              Georgia  GA         South    9920000   376  3.7903226
## 12               Hawaii  HI          West    1360301     7  0.5145920
## 13                Idaho  ID          West    1567582    12  0.7655102
## 14             Illinois  IL North Central   12830632   364  2.8369608
## 15              Indiana  IN North Central    6483802   142  2.1900730
## 16                 Iowa  IA North Central    3046355    21  0.6893484
## 17               Kansas  KS North Central    2853118    63  2.2081106
## 18             Kentucky  KY         South    4339367   116  2.6732010
## 19            Louisiana  LA         South    4533372   351  7.7425810
## 20                Maine  ME     Northeast    1328361    11  0.8280881
## 21             Maryland  MD         South    5773552   293  5.0748655
## 22        Massachusetts  MA     Northeast    6547629   118  1.8021791
## 23             Michigan  MI North Central    9883640   413  4.1786225
## 24            Minnesota  MN North Central    5303925    53  0.9992600
## 25          Mississippi  MS         South    2967297   120  4.0440846
## 26             Missouri  MO North Central    5988927   321  5.3598917
## 27              Montana  MT          West     989415    12  1.2128379
## 28             Nebraska  NE North Central    1826341    32  1.7521372
## 29               Nevada  NV          West    2700551    84  3.1104763
## 30        New Hampshire  NH     Northeast    1316470     5  0.3798036
## 31           New Jersey  NJ     Northeast    8791894   246  2.7980319
## 32           New Mexico  NM          West    2059179    67  3.2537239
## 33             New York  NY     Northeast   19378102   517  2.6679599
## 34       North Carolina  NC         South    9535483   286  2.9993237
## 35         North Dakota  ND North Central     672591     4  0.5947151
## 36                 Ohio  OH North Central   11536504   310  2.6871225
## 37             Oklahoma  OK         South    3751351   111  2.9589340
## 38               Oregon  OR          West    3831074    36  0.9396843
```

```
## 39         Pennsylvania  PA     Northeast   12702379   457  3.5977513
## 40         Rhode Island  RI     Northeast    1052567    16  1.5200933
## 41       South Carolina  SC         South    4625364   207  4.4753235
## 42         South Dakota  SD North Central     814180     8  0.9825837
## 43            Tennessee  TN         South    6346105   219  3.4509357
## 44                Texas  TX         South   25145561   805  3.2013603
## 45                 Utah  UT          West    2763885    22  0.7959810
## 46              Vermont  VT     Northeast     625741     2  0.3196211
## 47             Virginia  VA         South    8001024   250  3.1246001
## 48           Washington  WA          West    6724540    93  1.3829942
## 49        West Virginia  WV         South    1852994    27  1.4571013
## 50            Wisconsin  WI North Central    5686986    97  1.7056487
## 51              Wyoming  WY          West     563626     5  0.8871131
```

```
class(murders)
```

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

d

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)

murders_tibble
```

```
## # A tibble: 51 x 6
##    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
```

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

```
region_murders <- murders %>%
  group_by(region)

class(region_murders)
```

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

```
exp(mean(log(murders$population)))
```

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

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.

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

compute_s_n_2 <- function(n){
  x <- 1:n
  tibble(s_n_2 = sum(x))
}

tib_1 <-tibble(n = 1: 100,
               map_df(1:100, compute_s_n),
               map_df(1:100, compute_s_n_2))

tib_1
```

```
## # 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
```