R Notebook

Code

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.

Hide

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

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

Hide

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

average <dbl>	standard_deviation <dbl>
64.93942	3.760656

1 row

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 $:

Hide

```
s$average
```

```
[1] 64.93942
```

Hide

```
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:

Hide

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

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

Hide

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

mean(rate) <dbl>
2.779125

1 row

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:

Hide

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

rate <dbl>
3.034555

1 row

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:

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:

Hide

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

median_min_max <dbl>
64.98031
51.00000
79.00000

3 rows

Hide

```
NA
```

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:

Hide

```
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 <dbl>	minimum <dbl>	maximum <dbl>
64.98031	51	79

1 row

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:

Hide

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

sex <fctr>	height <dbl>
Male	75.00000
Male	70.00000
Male	68.00000
Male	74.00000
Male	61.00000

sex <fctr>	height <dbl>
Female	65.00000
Female	66.00000
Female	62.00000
Female	66.00000
Male	67.00000

1-10 of 1,050 rows Previous **1** 2 3 4 5 6 ... 100 Next

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:

Hide

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

sex <fctr>	average <dbl>	standard_deviation <dbl>
Female	64.93942	3.760656
Male	69.31475	3.611024

2 rows

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:

Hide

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

region <fctr>	median <dbl>	minimum <dbl>	maximum <dbl>
Northeast	1.802179	0.3196211	3.597751
South	3.398069	1.4571013	16.452753
North Central	1.971105	0.5947151	5.359892
West	1.292453	0.5145920	3.629527

4 rows

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

Hide

```
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:

Hide

```
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:

Hide

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

us_murder_rate
```

```
[1] 3.034555
```

which is now a numeric:

Hide

```
class(us_murder_rate)
```

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

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:

Hide

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

	state <chr>	a… <chr>	region <fctr>	population <dbl>	total <dbl>	rate <dbl>
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

6 rows

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:

Hide

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

	state <chr>	abb <chr>	region <fctr>	population <dbl>	total <dbl>	rate <dbl>
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

6 rows

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:

Hide

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

state <chr>	a… <chr>	region <fctr>	population <dbl>	total <dbl>	rate <dbl>
District of Columbia	DC	South	601723	99	16.4527532
Louisiana	LA	South	4533372	351	7.7425810
Missouri	MO	North Central	5988927	321	5.3598917
Maryland	MD	South	5773552	293	5.0748655

state <chr>	a… <chr>	region <fctr>	population <dbl>	total <dbl>	rate <dbl>
South Carolina	SC	South	4625364	207	4.4753235
Delaware	DE	South	897934	38	4.2319369
Michigan	MI	North Central	9883640	413	4.1786225
Mississippi	MS	South	2967297	120	4.0440846
Georgia	GA	South	9920000	376	3.7903226
Arizona	AZ	West	6392017	232	3.6295273

1-10 of 51 rows Previous **1** 2 3 4 5 6 Next

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:

Hide

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

	state <chr>	abb <chr>	region <fctr>	population <dbl>	total <dbl>	rate <dbl>
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

6 rows

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:

Hide

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

state <chr>	a… <chr>	region <fctr>	population <dbl>	total <dbl>	rate <dbl>
District of Columbia	DC	South	601723	99	16.452753

state <chr>	a… <chr>	region <fctr>	population <dbl>	total <dbl>	rate <dbl>
Louisiana	LA	South	4533372	351	7.742581
Maryland	MD	South	5773552	293	5.074866
Missouri	MO	North Central	5988927	321	5.359892
South Carolina	SC	South	4625364	207	4.475323

5 rows

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:

Hide

```
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:

Hide

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

```
[1] NA
```

Hide

```
sd(na_example)
```

```
[1] NA
```

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

Hide

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

```
[1] 2.301754
```

Hide

```
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.

Hide

```
ref <- NHANES %>%
  filter(AgeDecade == " 20-29") %>%
  summarize(mean(BPSysAve, na.rm = TRUE) , sd(BPSysAve, na.rm = TRUE))

ref
```

mean(BPSysAve, na.rm = TRUE) <dbl>	sd(BPSysAve, na.rm = TRUE) <dbl>
113.1583	11.71519

1 row

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

Hide

```
ref_avg <- NHANES %>%
  filter(AgeDecade == " 20-29") %>%
  summarize(avg = mean(BPSysAve, na.rm = TRUE)) %>%
  pull(avg)

ref_avg
```

```
[1] 113.1583
```

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

Hide

```
ref_avg <- NHANES %>%
  filter(AgeDecade == " 20-29") %>%
  summarize(min_max = quantile(BPSysAve, c(0,1), na.rm = TRUE))

ref_avg
```

min_max <dbl>
84
179

2 rows

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

Hide

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

AgeDecade <fctr>	mean(BPSysAve, na.rm = TRUE) <dbl>	sd(BPSysAve, na.rm = TRUE) <dbl>
0-9	99.95041	9.071798
10-19	104.27466	9.461431
20-29	108.42243	10.146681
30-39	111.25512	12.314790
40-49	115.49385	14.530054
50-59	121.84245	16.179333
60-69	127.17787	17.125713
70+	133.51652	19.841781
NA	141.54839	22.908521

9 rows

5. Repeat exercise 4 for males.

Hide

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

AgeDecade <fctr>	mean(BPSysAve, na.rm = TRUE) <dbl>	sd(BPSysAve, na.rm = TRUE) <dbl>
0-9	97.41912	8.317367
10-19	109.59789	11.227769
20-29	117.85084	11.274795
30-39	119.40063	12.306656
40-49	120.78390	13.968338
50-59	125.75000	17.760536
60-69	126.88578	17.478117

AgeDecade <fctr>	mean(BPSysAve, na.rm = TRUE) <dbl>	sd(BPSysAve, na.rm = TRUE) <dbl>
70+	130.20172	18.657475
NA	136.40000	23.534731

9 rows

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).

Hide

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

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

AgeDecade <fctr>	Gen… <fctr>	mean(BPSysAve, na.rm = TRUE) <dbl>	sd(BPSysAve, na.rm = TRUE) <dbl>
0-9	female	99.95041	9.071798
0-9	male	97.41912	8.317367
10-19	female	104.27466	9.461431
10-19	male	109.59789	11.227769
20-29	female	108.42243	10.146681
20-29	male	117.85084	11.274795
30-39	female	111.25512	12.314790
30-39	male	119.40063	12.306656
40-49	female	115.49385	14.530054
40-49	male	120.78390	13.968338

1-10 of 18 rows Previous **1** 2 Next

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.

Hide

```
race_4 <- NHANES %>%
  filter(AgeDecade == " 40-49") %>%
  group_by(Race1) %>%
  summarize(avg = mean(BPSysAve, na.rm = TRUE))

arrange(race_4, avg)
```

Race1 <fctr>	avg <dbl>
Other	114.8438
Hispanic	117.2326
White	117.5360
Mexican	120.3897
Black	124.5403

5 rows

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?

Hide

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

state <chr>	a… <chr>	region <fctr>	population <dbl>	total <dbl>	rate <dbl>
Alabama	AL	South	4779736	135	2.8244238
Alaska	AK	West	710231	19	2.6751860
Arizona	AZ	West	6392017	232	3.6295273
Arkansas	AR	South	2915918	93	3.1893901
California	CA	West	37253956	1257	3.3741383
Colorado	CO	West	5029196	65	1.2924531
Connecticut	CT	Northeast	3574097	97	2.7139722
Delaware	DE	South	897934	38	4.2319369
District of Columbia	DC	South	601723	99	16.4527532
Florida	FL	South	19687653	669	3.3980688

1-10 of 51 rows Previous **1** 2 3 4 5 6 Next

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:

Hide

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

```
[1] "grouped_df" "tbl_df"     "tbl"
[4] "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:

Hide

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

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

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

Hide

```
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 $:

Hide

```
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:

Hide

```
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:

Hide

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

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

```
NULL
```

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:

Hide

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

id <dbl>	func <list>
1	<fun>
2	<fun>
3	<fun>

3 rows

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.

Hide

```
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.

Hide

```
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.

Hide

```
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:

Hide

```
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:

Hide

```
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

Hide

```
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:

Hide

```
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:

Hide

```
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.

Hide

```
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:

Hide

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

sum <int>
1
3
6
10
15
21
28
36
45
55

1-10 of 25 rows Previous **1** 2 3 Next

We need to change the function to make this work:

Hide

```
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:

Hide

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

```
[1] "Negative" "Negative" "Zero"
[4] "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:

Hide

```
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)
```

group <chr>	rate <dbl>
New England	1.723796
Other	2.708144
South	3.626558
West Coast	2.899001

4 rows

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

Hide

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

```
Error: object 'b' not found
```

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

Hide

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

```
Error: `left` must be length 1
Run `rlang::last_error()` to see where the error occurred.
```

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

murders is in tidy format and is stored in a data frame.

Hide

```
murders
```

state <chr>	**a…** <chr>	**region** <fctr>	**population** <dbl>	**total** <dbl>	**rate** <dbl>
Alabama	AL	South	4779736	135	2.8244238
Alaska	AK	West	710231	19	2.6751860
Arizona	AZ	West	6392017	232	3.6295273
Arkansas	AR	South	2915918	93	3.1893901
California	CA	West	37253956	1257	3.3741383
Colorado	CO	West	5029196	65	1.2924531
Connecticut	CT	Northeast	3574097	97	2.7139722
Delaware	DE	South	897934	38	4.2319369
District of Columbia	DC	South	601723	99	16.4527532
Florida	FL	South	19687653	669	3.3980688

1-10 of 51 rows Previous **1** 2 3 4 5 6 Next

Hide

```
class(murders)
```

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

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

Hide

```
library(dplyr)

murders_tibble <- as_tibble(murders)
```

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

Hide

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

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

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

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 %>%.

Hide

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

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

Hide

```
library(purrr)
m_tib <- murders %>%
  .$population %>%
  map_dbl(log) %>%
  mean() %>%
  exp()

m_tib
```

```
[1] 3675209
```

Hide

```
class(m_tib)
```

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

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.

Hide

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

s_n <- map_df(n, compute_s_n)
s_n_2 <- map_df(n, compute_s_n)

n
```

```
  [1]   1   2   3   4   5   6
  [7]   7   8   9  10  11  12
 [13]  13  14  15  16  17  18
 [19]  19  20  21  22  23  24
 [25]  25  26  27  28  29  30
 [31]  31  32  33  34  35  36
 [37]  37  38  39  40  41  42
 [43]  43  44  45  46  47  48
 [49]  49  50  51  52  53  54
 [55]  55  56  57  58  59  60
 [61]  61  62  63  64  65  66
 [67]  67  68  69  70  71  72
 [73]  73  74  75  76  77  78
 [79]  79  80  81  82  83  84
 [85]  85  86  87  88  89  90
 [91]  91  92  93  94  95  96
 [97]  97  98  99 100
```

Hide

```
s_n
```

sum <int>
1
3
6
10
15
21
28
36

sum <int>
45
55

1-10 of 100 rows Previous **1** 2 3 4 5 6 ... 10 Next

Hide

```
s_n_2
```

sum <int>
1
3
6
10
15
21
28
36
45
55

1-10 of 100 rows Previous **1** 2 3 4 5 6 ... 10 Next

Chapter 5 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.

Hide

```
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.

Hide

```
library(tidyverse)
```

```
Registered S3 methods overwritten by 'dbplyr':
  method         from
  print.tbl_lazy
  print.tbl_sql
-- Attaching packages ---------
√ ggplot2 3.3.5     √ readr   2.0.1
√ tibble  3.1.4     √ stringr 1.4.0
√ tidyr   1.1.3     √ forcats 0.5.1
-- Conflicts ------------------
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

Hide

```
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:

Hide

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

```
[1] "C:/Users/rhdqn/Documents/R/win-library/4.1/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.

Hide

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

```
 [1] "data"
 [2] "DESCRIPTION"
 [3] "extdata"
 [4] "help"
 [5] "html"
 [6] "INDEX"
 [7] "MD5"
 [8] "Meta"
 [9] "NAMESPACE"
[10] "R"
[11] "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.

Hide

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

If you need to change your working directory, you can use the function setwd or you can change it through RStudio by clicking on "Session." 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:

Hide

```
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:

Hide

```
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:

Hide

```
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.

Hide

```
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.

Hide

```
file.copy(fullpath, "murders.csv")
```

```
[1] FALSE
```

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:

Hide

```
list.files()
```

```
 [1] "2010_bigfive_regents.xls"
 [2] "bsms222_259_son.Rproj"
 [3] "bsms222_259_son_0913.nb.html"
 [4] "bsms222_259_son_0913.Rmd"
 [5] "carbon_emissions.csv"
 [6] "extdata"
 [7] "fertility-two-countries-example.csv"
 [8] "HRlist2.txt"
 [9] "life-expectancy-and-fertility-two-countries-example.csv"
[10] "murders.csv"
[11] "olive.csv"
[12] "quiz_0915.html"
[13] "quiz_0915.Rmd"
[14] "RD-Mortality-Report_2015-18-180531.pdf"
[15] "README.md"
[16] "ssa-death-probability.csv"
```

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:

Hide

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

```
[1] FALSE
```

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:

Hide

```
library(readr)
```

The following functions are available to read-in spreadsheets:

Function Format Typical suffix read_table white space separated values txt read_csv comma separated values csv read_csv2 semicolon separated values csv read_tsv tab delimited separated values tsv 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:

Hide

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

```
[1] "state,abb,region,population,total"
[2] "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:

Hide

```
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:

Hide

```
View(dat)
```

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

Hide

```
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

Hide

```
library(readxl)
```

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

Function Format Typical suffix read_excel auto detect the format xls, xlsx read_xls original format xls 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:

Hide

```
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"
```

Hide

```
read_csv("2010_bigfive_regents.xls")
```

```
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.
```

state <chr>	abb <chr>	region <chr>	population <dbl>	total <dbl>
Alabama	AL	South	4779736	135
Alaska	AK	West	710231	19
Arizona	AZ	West	6392017	232
Arkansas	AR	South	2915918	93
California	CA	West	37253956	1257
Colorado	CO	West	5029196	65
Connecticut	CT	Northeast	3574097	97
Delaware	DE	South	897934	38
District of Columbia	DC	South	601723	99
Florida	FL	South	19687653	669

1-10 of 51 rows Previous **1** 2 3 4 5 6 Next

Hide

```
read_csv("carbon_emissions.csv")
```

```
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.
```

state <chr>	abb <chr>	region <chr>	population <dbl>	total <dbl>
Alabama	AL	South	4779736	135
Alaska	AK	West	710231	19
Arizona	AZ	West	6392017	232
Arkansas	AR	South	2915918	93
California	CA	West	37253956	1257
Colorado	CO	West	5029196	65
Connecticut	CT	Northeast	3574097	97
Delaware	DE	South	897934	38
District of Columbia	DC	South	601723	99
Florida	FL	South	19687653	669

1-10 of 51 rows Previous **1** 2 3 4 5 6 Next

Hide

```
read_csv("fertility-two-countries-example.csv")
```

```
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.
```

state <chr>	abb <chr>	region <chr>	population <dbl>	total <dbl>
Alabama	AL	South	4779736	135

state <chr>	abb <chr>	region <chr>	population <dbl>	total <dbl>
Alaska	AK	West	710231	19
Arizona	AZ	West	6392017	232
Arkansas	AR	South	2915918	93
California	CA	West	37253956	1257
Colorado	CO	West	5029196	65
Connecticut	CT	Northeast	3574097	97
Delaware	DE	South	897934	38
District of Columbia	DC	South	601723	99
Florida	FL	South	19687653	669

1-10 of 51 rows Previous **1** 2 3 4 5 6 Next

Hide

```
read_csv("HRlist2.txt")
```

```
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.
```

state <chr>	abb <chr>	region <chr>	population <dbl>	total <dbl>
Alabama	AL	South	4779736	135
Alaska	AK	West	710231	19
Arizona	AZ	West	6392017	232
Arkansas	AR	South	2915918	93
California	CA	West	37253956	1257
Colorado	CO	West	5029196	65
Connecticut	CT	Northeast	3574097	97
Delaware	DE	South	897934	38
District of Columbia	DC	South	601723	99
Florida	FL	South	19687653	669

1-10 of 51 rows Previous **1** 2 3 4 5 6 Next

Hide

```
read_csv("life-expectancy-and-fertility-two-countries-example.csv")
```

```
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.
```

state <chr>	abb <chr>	region <chr>	population <dbl>	total <dbl>
Alabama	AL	South	4779736	135
Alaska	AK	West	710231	19
Arizona	AZ	West	6392017	232
Arkansas	AR	South	2915918	93
California	CA	West	37253956	1257
Colorado	CO	West	5029196	65
Connecticut	CT	Northeast	3574097	97
Delaware	DE	South	897934	38
District of Columbia	DC	South	601723	99
Florida	FL	South	19687653	669

1-10 of 51 rows Previous **1** 2 3 4 5 6 Next

Hide

```
read_csv("murders.csv")
```

```
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.
```

state <chr>	abb <chr>	region <chr>	population <dbl>	total <dbl>
Alabama	AL	South	4779736	135
Alaska	AK	West	710231	19

state <chr>	abb <chr>	region <chr>	population <dbl>	total <dbl>
Arizona	AZ	West	6392017	232
Arkansas	AR	South	2915918	93
California	CA	West	37253956	1257
Colorado	CO	West	5029196	65
Connecticut	CT	Northeast	3574097	97
Delaware	DE	South	897934	38
District of Columbia	DC	South	601723	99
Florida	FL	South	19687653	669

1-10 of 51 rows Previous **1** 2 3 4 5 6 Next

Hide

```
read_csv("olive.csv")
```

```
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.
```

state <chr>	abb <chr>	region <chr>	population <dbl>	total <dbl>
Alabama	AL	South	4779736	135
Alaska	AK	West	710231	19
Arizona	AZ	West	6392017	232
Arkansas	AR	South	2915918	93
California	CA	West	37253956	1257
Colorado	CO	West	5029196	65
Connecticut	CT	Northeast	3574097	97
Delaware	DE	South	897934	38
District of Columbia	DC	South	601723	99
Florida	FL	South	19687653	669

1-10 of 51 rows Previous **1** 2 3 4 5 6 Next

Hide

```
read_csv("RD-Mortality-Report_2015-18-180531.pdf")
```

```
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.
```

state <chr>	abb <chr>	region <chr>	population <dbl>	total <dbl>
Alabama	AL	South	4779736	135
Alaska	AK	West	710231	19
Arizona	AZ	West	6392017	232
Arkansas	AR	South	2915918	93
California	CA	West	37253956	1257
Colorado	CO	West	5029196	65
Connecticut	CT	Northeast	3574097	97
Delaware	DE	South	897934	38
District of Columbia	DC	South	601723	99
Florida	FL	South	19687653	669

1-10 of 51 rows Previous **1** 2 3 4 5 6 Next

Hide

```
read_csv("ssa-death-probability.csv")
```

```
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.
```

state <chr>	abb <chr>	region <chr>	population <dbl>	total <dbl>
Alabama	AL	South	4779736	135
Alaska	AK	West	710231	19
Arizona	AZ	West	6392017	232

state <chr>	abb <chr>	region <chr>	population <dbl>	total <dbl>
Arkansas	AR	South	2915918	93
California	CA	West	37253956	1257
Colorado	CO	West	5029196	65
Connecticut	CT	Northeast	3574097	97
Delaware	DE	South	897934	38
District of Columbia	DC	South	601723	99
Florida	FL	South	19687653	669

1-10 of 51 rows Previous **1** 2 3 4 5 6 Next

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.

Hide

```
dat <- read_csv("olive.csv")
```

```
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.
```

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

Hide

```
names(dat)
```

```
[1] "state"      "abb"
[3] "region"     "population"
[5] "total"
```

Hide

```
read_lines(dat)
```

```
Warning in if (is_url(path)) { :
  the condition has length > 1 and only the first element will be used
Warning in if (file.exists(path)) return(normalizePath(path, "/", mustWork = FALSE)) :
  the condition has length > 1 and only the first element will be used
Warning in if (!is_absolute_path(path)) { :
  the condition has length > 1 and only the first element will be used
Error: 'AlabamaAlaskaArizonaArkansasCaliforniaColoradoConnecticutDelawareDistrict of ColumbiaFl
oridaGeorgiaHawaiiIdahoIllinoisIndianaIowaKansasKentuckyLouisianaMaineMarylandMassachusettsMich
iganMinnesotaMississippiMissouriMontanaNebraskaNevadaNew HampshireNew JerseyNew MexicoNew YorkN
orth CarolinaNorth DakotaOhioOklahomaOregonPennsylvaniaRhode IslandSouth CarolinaSouth DakotaTe
nnesseeTexasUtahVermontVirginiaWashingtonWest VirginiaWisconsinWyoming' does not exist in curre
nt working directory ('C:/bsms222_259_son').
```

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:

Hide

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

The read_csv file can read these files directly:

Hide

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

```
Rows: 1 Columns: 1
-- Column specification ----------------
Delimiter: ","
chr (1): https://raw.githubuserconte...

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:

Hide

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

```
trying URL 'https://raw.githubusercontent.com/rafalab/dslabs/master/inst/
extdata/murders.csv'
Content type 'text/plain; charset=utf-8' length 1684 bytes
downloaded 1684 bytes
```

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:

Hide

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

```
trying URL 'https://raw.githubusercontent.com/rafalab/dslabs/master/inst/
extdata/murders.csv'
Content type 'text/plain; charset=utf-8' length 1684 bytes
downloaded 1684 bytes
```

Hide

```
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.
```

Hide

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

```
Warning in file.remove(tmp_filename) :
  cannot remove file 'C:₩Users₩rhdqn₩AppData₩Local₩Temp₩RtmpMbqCYu₩file45a430eb5af4', reason 'P
ermission denied'
```

```
[1] FALSE
```

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:

Hide

```
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/ (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:

Hide

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

```
Read 260 items
```

Hide

```
x[1:10]
```

```
 [1] "state"      "abb"
 [3] "region"     "population"
 [5] "total"      "Alabama"
 [7] "AL"         "South"
 [9] "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 Unicode 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 (https://stackoverflow.com/questions/18789330/r-on-windows-character-encoding-hell). 5.8 Organizing 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.

1. Introduction 1.1. Background SCN2A is a voltage-gated sodium channel gene that encodes the neuronal sodium channel NaV1.2 and plays a critical role in action potential initiation during early neurodevelopment. The latest study demonstrated that it is loss of function mutations that in SCN2A that lead to autism spectrum disorders (ASD), in contrast to gain of function, which leads to infantile seizures (Ben-Shalom 2018). In this tutorial, we will handle genetic data for SCN2A mutations identified in latest genomic studies, and then explore the data format to describe genetic mutations using R basic functions. Our tutorial will utilize the summary data from Sanders et al. (2018).

1.2. Aims What we will do with this dataset, Understand the dataset from a scientific journal Apply some functions you have learnt from the Chapter 2 and 3 2. Explore your data 2.1. Unboxing your dataset Here we obtain the list of mutations in the Supplementary Table 1 from Sanders et al. (2018). Using the rio package, reading the excel file from the file link into your workspace. If you don't have the rio package in your system, please install as following:

Hide

```
install.packages('rio')
```

```
WARNING: Rtools is required to build R packages but is not currently installed. Please download
and install the appropriate version of Rtools before proceeding:

https://cran.rstudio.com/bin/windows/Rtools/
 ‘C:/Users/rhdqn/Documents/R/win-library/4.1’ 의 위치에 패키지(들)을 설치합니다.
(왜냐하면 ‘lib’ 가 지정되지 않았기 때문입니다)
trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.1/rio_0.5.27.zip'
Content type 'application/zip' length 536720 bytes (524 KB)
downloaded 524 KB
```

```
package ‘rio’ successfully unpacked and MD5 sums checked

The downloaded binary packages are in
    C:\Users\rhdqn\AppData\Local\Temp\RtmpMbqCYu\downloaded_packages
```

Now you can read the file from the website. This will create the d object.

Hide

```
d = rio::import('https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6015533/bin/NIHMS957592-supplemen
t-1.xlsx')
```

```
Registered S3 method overwritten by 'data.table':
  method           from
  print.data.table
```

Let’s explore the object you just loaded. How would you check the class of the object d?

Hide

```
class(d)
```

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

It shows that the d object is data.frame. Then, let’s overview the data frame. We will use head function to print out first few lines.

Hide

```
head(d)
```

	PatientID <chr>	PatientSex <chr>	PatientAgeAtAssessment <chr>	… <dbl>	Pos_hg19 <chr>	Ref <chr>	Alt <chr>
1	.	M	7y	2	154370122	166384278	12Mb Duplic
2	.	F	3y	2	163706754	166506754	2.8Mb Deleti
3	Patient2	F	18m	2	163875903	166478766	2.6Mb Duplic
4	Case1,280269	F	4y	2	165798270	166304847	507kb Duplic
5	.	M	3y	2	166019786	166249879	230kb Deleti
6	.	F	25y	2	166060054	166153823	94kb Deletio

6 rows | 1-9 of 27 columns

When you execute code in a notebook chunk, an output will be visible immediately beneath the input. From this, you can see several rows and columns in the data frame. Let’s look at the first column PatientID and check which class it is.

Hide

```
head(d$PatientID)
```

```
[1] "."             "."
[3] "Patient2"      "Case1,280269"
[5] "."             "."
```

Hide

```
class(d$PatientID)
```

```
[1] "character"
```

Cool. Now you can see the TrueRecurrence column. What is the class of the column TrueRecurrence?

Hide

```
class(d$TrueRecurrence)
```

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

To check the class of columns, you don’t need to type an individual column. We can overview the summary of the dataset using summary function. Which column has the character class?

Hide

```
summary(d)
```

```
  PatientID          PatientSex       
 Length:293         Length:293        
 Class :character   Class :character  
 Mode  :character   Mode  :character  
                                      
                                      
                                      
 PatientAgeAtAssessment      Chr   
 Length:293             Min.   :2  
 Class :character       1st Qu.:2  
 Mode  :character       Median :2  
                        Mean   :2  
                        3rd Qu.:2  
                        Max.   :2  
   Pos_hg19             Ref           
 Length:293         Length:293        
 Class :character   Class :character  
 Mode  :character   Mode  :character  
                                      
                                      
                                      
     Alt                Type          
 Length:293         Length:293        
 Class :character   Class :character  
 Mode  :character   Mode  :character  
                                      
                                      
                                      
    Effect             c.DNA          
 Length:293         Length:293        
 Class :character   Class :character  
 Mode  :character   Mode  :character  
                                      
                                      
                                      
  p.Protein          Inheritence       
 Length:293         Length:293        
 Class :character   Class :character  
 Mode  :character   Mode  :character  
                                      
                                      
                                      
    Source           SourcePMID       
 Length:293         Length:293        
 Class :character   Class :character  
 Mode  :character   Mode  :character  
                                      
                                      
                                      
 Ben-Shalom2017      Wolff2017        
 Length:293         Length:293        
 Class :character   Class :character  
 Mode  :character   Mode  :character  
                                      
                                      
                                      
 AnyRecurrence    UniqueSample/Family
 Min.   : 1.000   Length:293        
 1st Qu.: 1.000   Class :character  
 Median : 1.000   Mode  :character  
 Mean   : 2.119                     
 3rd Qu.: 2.000                     
 Max.   :10.000                     
 TrueRecurrence    Seizures        
 Min.   :1.000   Length:293        
 1st Qu.:1.000   Class :character  
 Median :1.000   Mode  :character  
 Mean   :1.768                     
 3rd Qu.:1.000                     
 Max.   :7.000                     
 SeizureOnsetDays   SeizureType       
 Length:293         Length:293        
 Class :character   Class :character  
 Mode  :character   Mode  :character  




     ASD               DD/ID          
 Length:293         Length:293        
 Class :character   Class :character  
 Mode  :character   Mode  :character  




 DD/ID severity     OtherFeatures     
 Length:293         Length:293        
 Class :character   Class :character  
 Mode  :character   Mode  :character  




 Classification    
 Length:293        
 Class :character  
 Mode  :character  
```

2.2. Difference between data frame and matrix Here we will convert the data frame into a matrix, and compare which part will be different in this. To convert a data frame into a matrix, you can use the command called as.matrix.

Hide

```
m = as.matrix(d)
```

Let’s overview the matrix object. Can you tell difference with data frame?

Hide

```
head(m[,'TrueRecurrence'])
```

```
  1   2   3   4   5   6 
"1" "1" "1" "1" "1" "1" 
```

2.3. Subset and Sort Some patients who have the SCN2A mutation (hereafter called “SCN2A patient”) often have seizures. So we want to know when the seizure occurs in development. Let’s check the class first.

Hide

```
class(d$SeizureOnsetDays)
```

```
[1] "character"
```

Why this column contains character? Let’s head the first few lines.

Hide

```
head(d$SeizureOnsetDays)
```

```
[1] "90"  "."   "3"   "3"   "."   "365"
```

It seems that some rows contain samples who do not have seizure or unknown information. It’s represented by “.”, and also recorded in another column called Seizures.

Hide

```
head(d$Seizures)
```

```
[1] "Y" "N" "Y" "Y" "N" "Y"
```

So we want to subset rows where the seizure phenotype is available.

Hide

```
d1 = d[d$Seizures=='Y',]
```

Let’s see how many samples have seizure phenotypes? Then, you can ask when is the earliest days for the representation of seizure phenotype? How can we check this? The fisrt, as seen previously the SeizureOnsetDays column is character so we cannot apply functions for numeric.

Hide

```
head(d1$SeizureOnsetDays)
```

```
[1] "90"   "3"    "3"    "365"  "30"  
[6] "1825"
```

So we have to convert this into numeric first.

Hide

```
d1$SeizureOnsetDays2 <- as.numeric(d1$SeizureOnsetDays)
```

```
Warning: NAs introduced by coercion
```

Hmm. There’s an warning for NA introduction. This is because some rows do not have character that we can properly convert from character to numeric. So possible solutions are either you can bear with this in your downstream analyses or 2) convert character into an appropriate form of numeric conversion. Then, the question is how can we find the rows with NA? We will ask whether the rows contains NA or not using is.na function. This will return boolean as to NA presence.

Hide

```
head(is.na(d1$SeizureOnsetDays2))
```

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

See we can find some rows with NA. One of them is the 13th row. Let’s see how it looks like.

Hide

```
d1$SeizureOnsetDays[13]
```

```
[1] "<365"
```

Here you have < (angle bracket) in the character so it won’t properly converted to numeric information. Did you find more of these cases?

Hide

```
d1[is.na(d1$SeizureOnsetDays2),]$SeizureOnsetDays
```

```
 [1] "<365" "<365" "<365" "<28"  "<30"
 [6] "<365" "<365" "<365" "<365" "<365"
```

Our NAs all contains <, which prevent converting a character into a numeric. We would fix for downstream analyses. For example, we can convert <365 into 365. One function we can try is gsub. This replace your string into a format that you may not get NA. For example,

Hide

```
# gsub('pattern in your character', 'new character you want to replace', vectors for your chara
cter)
d1$SeizureOnsetDays3 <- gsub('<', '', d1$SeizureOnsetDays)
head(d1$SeizureOnsetDays3)
```

```
[1] "90"   "3"    "3"    "365"  "30"
[6] "1825"
```

Let’s convert them into numerics.

Hide

```
d1$SeizureOnsetDays3 <- as.numeric(d1$SeizureOnsetDays3)
```

Did you get warning for this? Now we can ask our initial question. When is the earliest day for having seizure?

Hide

```
min(d1$SeizureOnsetDays3)
```

```
[1] 0
```

3. Exercise The dataset contains more details for genetic mutations in SCN2A patients. From this information, what can we analyze further? Here I list up few questions you can examine further. Finding the position of the genetic mutations within SCN2A. Which information you would use? If you are not familiar with positional information on genetic variants (or mutations), please find the Figure 1 or the slides for Mutation (BSMS205 Session 3-1). Counting the recurrent mutations at the same protein position (in other words, the same mutations seen across different patients), and examine whether the patients have similar phenotype. Finding the position where different consequences mutations occur. Please note that “consequences” are loss-of-function (Nonsense, Frameshift) or missense. Sketch a plot to visualize your analysis. Figure 1. Standard mutation nomenclature (Ogino et al. 2007). According to this guideline, genetic mutations can be represented for coding DNA reference sequences (c. prefix) and protein-level amino acid sequences (p. prefix).

3.1. For-loops and Vectorization Here we examine more details of genetic mutations as to their functional consequence and position of SCN2A mutations. In the dataset includes, there are two columns called c.DNA and p.Protein, containing the cDNA or protein position for the genetic mutations. During these exercises, we will look at the concept of for-loop and vectorization, which you learn from the Chapter 3.4. Let’s look at the column p.Protein. It contains protein positions from each patient. What would you check at the first place? Task 1 Task 2 Task 3 …. Task N Let’s write down your code to explore this column.

Hide

```
head(d$p.Protein)
```

```
[1] "." "." "." "." "." "."
```

If you need more information on SCN2A, please visit the Uniprot description for SCN2A. The Uniprot database contains description for protein domains. Then, I would remove the characters from the string so we can have only numerics for positions. Here I use gsub function to extract numbers from string. Let’s remove non-numeric characters from the string.

Hide

```
gsub('[^0-9]', '', 'p.R102X')
```

```
[1] "102"
```

In the dataset, we have many rows for protein positions. One way we might try is to set up for-loop to process each row.

Hide

```
for (i in 1:293) {
  a <- gsub('[^0-9]', '', d[i, 'p.Protein'] )
}
```

Do you think this is an effective approach? As we have done in your assignment, for-loop is not a good choice to process vectors because R can do vectorization for this process with a shorter and clearer code. So this mean you can apply gsub on vector and return your output to another column (could be new assign).

Hide

```
a
```

```
[1] "1974"
```

3.2. Counting the recurrent mutations Recurrent mutations are the ones that the same genetic mutations occur in multiple individuals. Recurrent mutations can be common 1) when the mutation does not affect on natural selection, 2) when the mutation is beneficial, 3) in the hotspot for a disease or strongly associated with trait. However, given we are dealing with the genetic mutations from rare disorders, the mutations in the dataset are supposed to be uniquely present in general population. Otherwise, the recurrent mutations can indicate strong association with the phenotype. To assess the recurrent mutations, the first thing we can try is to examine whether the same mutations occur in multiple individuals. Since the dataset contains individual patients for each row, we can simply check the frequency using:

Hide

```
c <- as.data.frame(table(d$p.Protein))
```

or we can check the number of unique variants in the dataset by:

Hide

```
d
```

	PatientID <chr>	PatientSex <chr>	PatientAgeAtAssessment <chr>	… <d
1	.	M	7y	2
2	.	F	3y	2
3	Patient2	F	18m	2
4	Case1,280269	F	4y	2
5	.	M	3y	2
6	.	F	25y	2
7	11	F	6y	2
8	Case4,259404	M	14y	2
9	14525.p1	M	11y	2
10	2-1291-03	M	>8y	2

1-10 of 293 rows | 1-5 of 27 columns Previous **1** 2 3 4 5 6 ... 30 Next

How many unique variants you can find? and which variants are occurred in multiple times? Then, you can use other columns to check frequency for different groups. Which columns you would use for more grouping? If you find that, please check the recurrent mutations for each group.

Hide

```
d
```

	PatientID <chr>	PatientSex <chr>	PatientAgeAtAssessment <chr>	… <d

	PatientID <chr>	PatientSex <chr>	PatientAgeAtAssessment <chr>	… <d
1	.	M	7y	2
2	.	F	3y	2
3	Patient2	F	18m	2
4	Case1,280269	F	4y	2
5	.	M	3y	2
6	.	F	25y	2
7	11	F	6y	2
8	Case4,259404	M	14y	2
9	14525.p1	M	11y	2
10	2-1291-03	M	>8y	2

1-10 of 293 rows | 1-5 of 27 columns Previous **1** 2 3 4 5 6 ... 30 Next

3.3. What is the proportion of diagnosis for SCN2A patient? SCN2A mutation can have multiple different consequences for disease phenotypes. It can cause ASD but also other neurodevelopmental conditions. In total cases, how many phenotypes occur in SCN2A patients. Then, calculate the proportion of the phenotypes among total cases.

Hide

```
d$ASD
```

```
  [1] "N"           "Y"
  [3] "Unknown"     "N"
  [5] "Y"           "N"
  [7] "N"           "Unknown"
  [9] "Y"           "Y"
 [11] "N"           "N"
 [13] "Unknown"     "Y"
 [15] "Y"           "Y"
 [17] "Unknown"     "Y"
 [19] "Y"           "Unknown"
 [21] "Unknown"     "Unknown"
 [23] "Y"           "Y"
 [25] "Unknown"     "Unknown"
 [27] "Unknown"     "N"
 [29] "Unknown"     "N"
 [31] "Unknown"     "Y"
 [33] "Unknown"     "N"
 [35] "Unknown"     "Unknown"
 [37] "Unknown"     "Unknown"
 [39] "Unknown"     "Unknown"
 [41] "Unknown"     "N"
 [43] "Unknown"     "Unknown"
 [45] "Unknown"     "Unknown"
 [47] "N"           "N"
 [49] "N"           "Unknown"
 [51] "N"           "N"
 [53] "Unknown"     "N"
 [55] "Unknown"     "Unknown"
 [57] "Unknown"     "Unknown"
 [59] "Unknown"     "Unknown"
 [61] "Unknown"     "Unknown"
 [63] "Unknown"     "Unknown"
 [65] "Unknown"     "Unknown"
 [67] "Y"           "Y"
 [69] "Y"           "Unknown"
 [71] "Unknown"     "Unknown"
 [73] "Unknown"     "N"
 [75] "Y"           "Unknown"
 [77] "Unknown"     "Y"
 [79] "Y"           "Unknown"
 [81] "Y"           "Y"
 [83] "Unknown"     "Y"
 [85] "Y"           "Y"
 [87] "Unknown"     "Unknown"
 [89] "Unknown"     "Y"
 [91] "Y"           "Y"
 [93] "Unknown"     "Y"
 [95] "Unknown"     "Unknown"
 [97] "Unknown"     "Y"
 [99] "Unknown"     "Unknown"
[101] "Y"           "Unknown"
[103] "Unknown"     "Unknown"
[105] "Unknown"     "Unknown"
[107] "Unknown"     "Unknown"
[109] "Unknown"     "Unknown"
[111] "Unknown"     "Unknown"
[113] "Unknown"     "Y"
```

```
[115] "Unknown"       "Unknown"
[117] "Unknown"       "Unknown"
[119] "Unknown"       "Unknown"
[121] "N"             "Unknown"
[123] "Unknown"       "Unknown"
[125] "Y"             "N"
[127] "Unknown"       "Unknown"
[129] "Y"             "Unknown"
[131] "Unknown"       "Unknown"
[133] "Unknown"       "Unknown"
[135] "Y"             "Y"
[137] "Y"             "Y"
[139] "Y"             "Y"
[141] "Y"             "Unknown"
[143] "Unknown"       "Unknown"
[145] "Unknown"       "Unknown"
[147] "Unknown"       "Unknown"
[149] "Unknown"       "Unknown"
[151] "Y"             "Unknown"
[153] "N"             "N"
[155] "Y"             "Unknown"
[157] "Y"             "Unknown"
[159] "Unknown"       "Unknown"
[161] "Unknown"       "Y"
[163] "Unknown"       "Y"
[165] "Unknown"       "Unknown"
[167] "Y"             "Unknown"
[169] "Y"             "Unknown"
[171] "Unknown"       "Unknown"
[173] "Unknown"       "Unknown"
[175] "Y"             "Unknown"
[177] "Y"             "N"
[179] "N"             "N"
[181] "Unknown"       "Y"
[183] "N"             "Y"
[185] "Unknown"       "N"
[187] "Unknown"       "Unknown"
[189] "Unknown"       "Unknown"
[191] "Unknown"       "N"
[193] "Unknown"       "Unknown"
[195] "Unknown"       "Y"
[197] "Unknown"       "Unknown"
[199] "Unknown"       "Unknown"
[201] "Y"             "Y"
[203] "Maybe"         "Unknown"
[205] "Unknown"       "Y"
[207] "Y"             "Y"
[209] "Unknown"       "Y"
[211] "Y"             "Y"
[213] "Unknown"       "Unknown"
[215] "Unknown"       "N"
[217] "Y"             "Y"
[219] "Y"             "Unknown"
[221] "Unknown"       "Probably not"
[223] "Y"             "N"
[225] "N"             "Unknown"
[227] "Unknown"       "Unknown"
[229] "Y"             "N"
```

```
[231] "N"            "N"
[233] "Y"            "Unknown"
[235] "Unknown"      "N"
[237] "Unknown"      "Unknown"
[239] "Y"            "Y"
[241] "Y"            "Unknown"
[243] "Y"            "Unknown"
[245] "Unknown"      "Unknown"
[247] "Unknown"      "Unknown"
[249] "Y"            "Y"
[251] "Unknown"      "Y"
[253] "Unknown"      "N"
[255] "Unknown"      "Unknown"
[257] "Y"            "Y"
[259] "Unknown"      "Unknown"
[261] "Y"            "Unknown"
[263] "Unknown"      "Y"
[265] "Y"            "Y"
[267] "Unknown"      "Y"
[269] "Unknown"      "Y"
[271] "Unknown"      "Unknown"
[273] "Unknown"      "Unknown"
[275] "Unknown"      "Unknown"
[277] "Unknown"      "Unknown"
[279] "Unknown"      "Unknown"
[281] "Unknown"      "Unknown"
[283] "Unknown"      "Unknown"
[285] "Y"            "Y"
[287] "N"            "Y"
[289] "Unknown"      "Y"
[291] "N"            "Y"
[293] "Y"
```

Then, you might be intrigued to whether females and males have different occurrence in each disorder. Let's check it.

Hide

```
d$ASD %>%
  group_by("PatientSex")
```

```
Error in UseMethod("group_by") :
  no applicable method for 'group_by' applied to an object of class "character"
```

Another question you can ask is whether different mutation consequences occur in each phenotype. Let's find out how many mutation consequences are observed in each phenotype.

3.4. Find the position where different consequences of mutations occur If you checked the recurrent mutations, you might want to find a locus where two or more variants occur. Such loci might indicate functionally important position of the gene and you might find some insight as to a cause of disease.

3.5. Sketch a plot to visualize your analysis When you examine the dataset, you would draw something to show your output. Though we haven't learnt how to plot data yet, we can have a quick sketch for the dataset. There's no restriction on your suggestion. Please submit your hand-drawing for the plot you would like to show from this dataset.