MTH 2520 R Notes 6

5 Lists

5.1 Creating and Examining Lists

- The so-called ***atomic*** vectors we've been working with up to now must be *homogeneous* (elements all the same type).
- ***Lists*** (also called ***recursive*** vectors) are vectors that can be *heterogeneous* (elements not all the same type). In fact, their elements can be *any* R *objects*. Lists are used to bundle objects together under one name.
- Lists are created and examined using:

```
list()            # Create a list
length()          # Returns the number of elements in a list
is.list()         # Indicates whether or not an object is a list
str()             # Describes the structure of a list
```

- Here's a simple example:

```
> y <- list(3, "a", TRUE)
> y

[[1]]
[1] 3

[[2]]
[1] "a"

[[3]]
[1] TRUE

> is.list(y)

[1] TRUE
```

  Notice that the indices of list elements are in double square brackets `[[ ]]`.

- Here's a list with three elements, two of which are vectors and the third a matrix:

```
> x <- list(x1 = c(1, 2, 3),
+           x2 = c("a", "b", "c"),
+           x3 = matrix(1:4, nrow = 2, ncol = 2))
> x

$x1
[1] 1 2 3

$x2
[1] "a" "b" "c"

$x3
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Notice that when list elements have names (like `x1`, `x2`, and `x3` above), R prints the names preceded by a dollar sign `$`.

- `length()` tells how many elements are contained in a list:

```
> length(x)

[1] 3
```

- `str()` tells us the "structure" of the list:

```
> str(x)

List of 3
 $ x1: num [1:3] 1 2 3
 $ x2: chr [1:3] "a" "b" "c"
 $ x3: int [1:2, 1:2] 1 2 3 4
```

The output above says that `x` is a list with three elements: a numeric vector `x1`, a `"character"` vector `x2`, and a 2×2 `"integer"` matrix `x3`.

Section 5.1 Exercises

Exercise 1

a) Write commands that create a list named `Employees` containing three elements:

- `Name`, a `"character"` vector containing the names of four employees,
  `"Joe", "Kim", "Ann", "Bob"`
- `Salary`, a numeric vector containing their salaries,
  `56000, 67000, 60000, 55000`
- `Union`, a `"logical"` vector indicating whether or not they belong to a union,

```
TRUE, TRUE, FALSE, FALSE
```

b) Use `str()` to look at the structure of the list. Report the results.

c) Use `length()` to find the number of elements in the list. Report the result.

5.2 General List Operations

5.2.1 Accessing List Elements

- We access elements of a list using double square brackets `[[ ]]`, the dollar sign `$`, or single square brackets `[ ]`:

```
[[ ]]          # Access a list element via its index or name
$              # Access a list element via its name
[ ]            # Access a list element via its index or name, re-
               # turning a list (rarely used)
```

- We *almost always* use either `[[ ]]` or `$`. The main distinctions are:
  - `[[ ]]` and `$` return the actual *object* from the list, e.g. if the list element being extracted is a *vector*, they return a *vector*.
  - `[ ]` returns a *list* containing the object, e.g. if the list element being extracted is a *vector*, it returns *list* containing that vector.
  - `[[ ]]` can be used with numerical indices or names of list elements (in quotation marks).
  - `$` can only be used with names of list elements (without quotation marks).
  - `[ ]` can extract more than one list element, but `[[ ]]` and `$` extract only a single element.

- Here's an example using `[[ ]]` (and the list `x` from above):

```
> x[[2]]                          # We could also use x[["x2"]]

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

  Note that a `"character"` vector is returned. We could also have used `x[["x2"]]`.

- Here's an example using `$`:

```
> x$x2

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

  A `"character"` vector is returned here too.

- Now watch what happens when we use single brackets `[ ]`:

```
> x[2]

$x2
[1] "a" "b" "c"
```

This time a *list* with one element (the vector `x2`) is returned. This is usually *not what we want*.

5.2.2 Adding and Deleting List Elements

- The operators `[[ ]]`, `$`, and `[ ]` can also be used to add or delete a list element.
- Below, we add the value 16 as the fourth element of `x` and give it the name `x4`:

```
> x$x4 <- 16                        # We could also use x[["x4"]] <- 16
> x

$x1
[1] 1 2 3

$x2
[1] "a" "b" "c"

$x3
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$x4
[1] 16
```

Above, we could also have used `x[["x4"]] <- 16`.

- To delete a list element, we set its value to `NULL`:

```
> x$x4 <- NULL
> x

$x1
[1] 1 2 3

$x2
[1] "a" "b" "c"

$x3
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

- If we want to add multiple elements to a list all at once, we need to use `[ ]`:

```
> x[4:5] <- c(16, 17)          # We could also use x[4:5]<-list(16, 17)
> x

$x1
[1] 1 2 3

$x2
[1] "a" "b" "c"

$x3
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[4]]
[1] 16

[[5]]
[1] 17
```

  but then we have to add the element names (using `names()`, as described later).

Section 5.2 Exercises

Exercise 2 Recreate the `Employees` list from Exercise 1.

a) Will the following command return a vector or a list containing the vector?

```
> Employees[[2]]
```

Will the following command return a vector or a list containing the vector?

```
> Employees$Salary
```

Will the following command return a vector or a list containing the vector?

```
> Employees[2]
```

b) Write a command involving `[[ ]]` that returns the `"logical"` vector `Union` from the `Employees` list.

c) Now write a command involving `$` that returns the `"logical"` vector `Union` from the `Employees` list.

d) Guess what the result of the following command will be, then check your answer:

```
> Employees$Salary[2]
```

Guess what the result of the following command will be, then check your answer:

```
> Employees[[2]][2]
```

e) Write a command that returns `TRUE` or `FALSE` depending on whether the 2nd employee earns more than $60,000.

f) Write a command involving either `[[ ]]` or `$` that adds a fourth element to the `Employees` list, a numeric vector named `Experience` containing their years of experience,

```
14, 6, 9, 12
```

5.3 Named List Elements

- We can get or assign list element names using:

```
names()       # Examine or change the names of list elements
```

- As an example, consider again the list `x`:

```
> x

$x1
[1] 1 2 3

$x2
[1] "a" "b" "c"

$x3
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

To get the list element names, we type:

```
> names(x)

[1] "x1" "x2" "x3"
```

and to change them to, say, `y1`, `y2`, and `y3`, we type:

```
> names(x) <- c("y1", "y2", "y3")
> names(x)

[1] "y1" "y2" "y3"
```

- To remove the names, we'd type:

```
> names(x) <- NULL
```

Section 5.3 Exercises

Exercise 3 Here's a simple list `x`:

```
> x <- list(x1 = c(1, 2), x2 = c("a", "b"), x3 = 12)
```

a) What will the following command return? Check your answer.

```
> names(x)
```

b) Write a command that will change the names of the elements of `x` to `y1`, `y2`, and `y3`. Make sure to check your answer.

5.4 Applying a Function to a List Using `lapply()` and `sapply()`

- We can apply a function separately to each element of a *list* using the *list* versions of the `apply()` function:

```
lapply()      # Apply a function separately to each element of a list,
              # returning a list
sapply()      # Apply a function separately to each element of a list,
              # returning a vector
```

- Both `lapply()` and `sapply()` take two main arguments, `X`, a list, and `FUN`, a function to be applied to each element of the list.

  The difference between the two is that `lapply()` returns a *list* whereas `sapply()` returns a *vector*.

  (The name `lapply()` means "list apply", and `sapply()` is short for "simplified apply".)

- As an example, consider the list `HtWtAge` containing heights, weights, and ages of five people:

```
> HtWtAge

$Height
[1] 65 68 70 60 61

$Weight
[1] 160 171 158 148 215
```

```
$Age
[1] 23 20 37 40 44
```

- To compute the average of each variable using `lapply()`, we type:

```
> lapply(X = HtWtAge, FUN = mean)

$Height
[1] 64.8

$Weight
[1] 170.4

$Age
[1] 32.8
```

Note that `lapply()` returns a *list*.

- Usually, we'd prefer the results to be in a *vector*. To compute the averages and get the results in a *vector* we use `sapply()`:

```
> sapply(X = HtWtAge, FUN = mean)

Height Weight    Age
  64.8  170.4   32.8
```

Notice that `sapply()` returns a *vector* (with named elements).

- Additional arguments to `FUN` are specified after `FUN` in the call to `lapply()` or `sapply()`.

Section 5.4 Exercises

Exercise 4 Here's the `HtWtAge` list again:

```
> HtWtAge <- list(Height = c(65, 68, 70, 60, 61),
                  Weight = c(160, 171, 158, 148, 215),
                  Age = c(23, 20, 37, 40, 44))
```

a) Write a command using `lapply()`, with `FUN = max`, that returns a *list* containing the maximum value of each variable (height, weight, and age).

b) Now write a command using `sapply()` that does the same thing, but returns a *vector*.

6 Data Frames

6.1 Creating and Viewing Data Frames

- ***Data frames*** are two-dimensional, like matrices, but they can be *heterogeneous* (columns not all the same type).
- Each each row corresponds to an individual (person, place, thing, etc.), and each column contains the value of a ***variable*** observed (or measured) on that individual. An entire row is called an ***observation***.

  Alternatively, sometimes rows are called ***records*** and columns ***fields***.
- For example, the following data set (from a study of eight mice) could be stored as a *data frame* in R:

Mice Data Set

Color	Weight	Length
white	23	3.8
grey	21	3.7
black	18	3.0
brown	26	3.4
black	25	3.4
white	22	3.1
black	26	3.5
white	19	3.2

Each row corresponds to a mouse, and each column a variable observed or measured on the mouse. Note that the data set has a mix of `"character"` and numerical columns, and therefore *couldn't* be stored as a matrix.

6.1.1 Creating Data Frames Using `data.frame()`

- We can create a data frame on the R command line using:

```
data.frame()       # Create a data frame from a set of vectors of the
                   # same length
```

(We'll see later how to create a data frame by reading the data from a file.)

- In addition, several functions let us view various aspects of a data frame:

```
head(); tail()   # Prints the first (or last) six rows of a data frame
nrow(); ncol()   # Indicates the number of rows (or columns)
dim()            # Gives the dimensions (number of rows and columns)
str()            # Gives the structure of a data frame
is.data.frame()  # Indicates whether or not an object is a data frame
```

- Consider again the **mice data set** from above. After creating vectors containing the data:

```
> col <- c("white", "grey", "black", "brown", "black", "white", "black", "white")
> wt <- c(23, 21, 12, 26, 25, 22, 26, 19)
> len <- c(3.8, 3.7, 3.0, 3.4, 3.4, 3.1, 3.5, 3.2)
```

  we create a data data frame named `mice.data` by typing:

```
> mice.data <- data.frame(Color = col, Weight = wt, Length = len,
+                         stringsAsFactors = FALSE)
```

  We can check that the data frame was created properly by typing its name:

```
> mice.data
```

```
  Color Weight Length
1 white     23    3.8
2  grey     21    3.7
3 black     12    3.0
4 brown     26    3.4
5 black     25    3.4
6 white     22    3.1
7 black     26    3.5
8 white     19    3.2
```

  and we can make sure its a data frame by typing:

```
> is.data.frame(mice.data)
```

```
[1] TRUE
```

  Specifying `stringsAsFactors = FALSE` in `data.frame()` indicates that we *don't* want the `"character"` vector `col` to be converted to a so-called *factor* when the data frame is created. (The default for the argument `stringsAsFactors` is `TRUE`.) We'll discuss *factors* later.

- The functions `head()` and `tail()` print just the first and last six rows, respectively:

```
> head(mice.data)
```

```
  Color Weight Length
1 white     23    3.8
2  grey     21    3.7
3 black     12    3.0
4 brown     26    3.4
5 black     25    3.4
6 white     22    3.1
```

  They're more useful with large data sets.

- To find out how many rows a data frame has, type:

```
> nrow(mice.data)

[1] 8
```

To find out how many columns it has, use `ncol()`. We can also use `dim()`.

- To look at the "structure" of `mice.data`, type:

```
> str(mice.data)

'data.frame':        8 obs. of  3 variables:
 $ Color : chr  "white" "grey" "black" "brown" ...
 $ Weight: num  23 21 12 26 25 22 26 19
 $ Length: num  3.8 3.7 3 3.4 3.4 3.1 3.5 3.2
```

This indicates that there are eight observations of three variables, `Color`, a `"character"` vector, and `Weight` and `Length`, both numeric vectors.

6.1.2 Creating Data Frames by Reading Data from a File Using `read.table()`

- We can create a data frame by reading the data from a file using:

```
read.table()        # Read data from a text (.txt) file into a data
                    # frame
read.csv()          # Read data from a 'comma separated value' (.csv)
                    # file into a data frame
```

- Suppose we have a text (.txt) file, **C:\Users\MyName\Documents\mice.txt**, that contains the **mice data set**. The contents of the text file would look like this:

Color	Weight	Length
white	23	3.8
grey	21	3.7
black	18	3.0
brown	26	3.4
black	25	3.4
white	22	3.1
black	26	3.5
white	19	3.2

- We read it into a data frame named `mice` by typing:

```
> # We could also use read.csv():
> mice.data <- read.table('C:/Users/MyName/Documents/mice.txt',
                          header = TRUE, stringsAsFactors = FALSE)
```

It's always a good idea to check that the data were read in correctly:

```
> mice.data
```

```
  Color Weight Length
1 white     23    3.8
2  grey     21    3.7
3 black     18    3.0
4 brown     26    3.4
5 black     25    3.4
6 white     22    3.1
7 black     26    3.5
8 white     19    3.2
```

Above, we could also have used `read.csv()` instead of `read.table()`.

`read.table()` and `read.csv()` are the *same function*, but have different default settings for some of the arguments.

- Here are some comments about using `read.table()` and `read.csv()`:
  - The usual back slashes (\) are written as forward slashes (/) in `read.table()` and `read.csv()`.
  - Specifying `header = TRUE` is used to indicate that the first row of the text file contains the variable names.
  - You can use either single (') or double (") quotations when specifying the location and name of the file.
  - By default, R recognizes one or more white spaces, tabs, or newline characters as separators of data values in the input file. Other separators can be specified via the `sep` argument to `read.table()` and `read.csv()`.
  - As before, specifying `stringsAsFactors = FALSE` indicates that we *don't* want the character column (Color) to be converted to a *factor*.
  - To read data from an **Excel** file, you can either:
    * Save the Excel file as a tab delimited text (.txt) file, and then use `read.table()`, or
    * Save the Excel file as a 'comma separated value' (.csv) file, and then read it into R using `read.csv()`.

    (There are also more specialized functions for reading in Excel files, but we won't cover them.)

Section 6.1 Exercises

Exercise 5

a) Here's the **mice data set** as three vectors:

```
> col <- c("white", "grey", "black", "brown", "black", "white", "black",
           "white")
> wt <- c(23, 21, 12, 26, 25, 22, 26, 19)
> len <- c(3.8, 3.7, 3.0, 3.4, 3.4, 3.1, 3.5, 3.2)
```

After creating the vectors, write a command involving `data.frame()` that creates a data frame containing the data. Make sure the column names are `Color`, `Weight`, and `Length`.

Check that you created the data frame correctly by typing its name on the command line, for example:

```
> mice.data
```

b) Before proceeding, remove the data frame you just created from your Workspace, for example by typing:

```
> rm(mice.data)
```

and double check that it's no longer there by typing

```
> ls()                    # Or you could use objects().
```

The file **mice.txt** contains the **mice data set**. After saving the file onto your computer, write a command involving `read.table()` or `read.csv()` that reads the data from the .txt file into a data frame.

Make sure to check that the data were read in correctly, for example by typing:

```
> mice.data
```

6.2 Accessing and Replacing Elements, Rows, or Columns of a Data Frame

- Data frames have features of both *matrices and lists*. To extract a specific value, row, or column, we use:

```
[ , ]      # Access data frame elements, rows, or columns via their
           # row and column indices (separated by a comma)
[[ ]]      # Access a data frame variable (column) by specifying its
           # index or name
$          # Access a data frame variable (column) by specifying its
           # name
```

6.2.1 Accessing Rows and Columns Using `[ ]`

- As with matrices, we can access a specific element, row, or column of a data frame using single square brackets `[ ]`. For example, the value in the 3rd row and 2nd column of `mice` is:

```
> mice.data[3, 2]

[1] 18
```

The entire 2nd column is accessed via:

```
> mice.data[, 2]

[1] 23 21 18 26 25 22 26 19
```

and the entire 3rd row would be accessed via `mice.data[3, ]`.

- We can also use square brackets and the assignment operator to *replace* a value in a data frame, for example:

```
> mice.data[3, 2] <- 12
```

6.2.2 Accessing and Replacing Columns Using `[[ ]]` and `$`

- In fact, **data frames are lists**:

```
> is.data.frame(mice.data)

[1] TRUE

> is.list(mice.data)

[1] TRUE
```

The list elements are the columns of the data frame.

- Therefore, the list operators `$` and `[[ ]]` can also be used to access columns of a data frame:

```
> mice.data$Color           # Access a variable by name

[1] "white" "grey"  "black" "brown" "black" "white" "black" "white"

> mice.data[[1]]            # Access a variable by list index (i.e. column number).
                            # We could also use mice[["Color"]].

[1] "white" "grey"  "black" "brown" "black" "white" "black" "white"
```

Above, we could also access the `Color` column using `mice.data[["Color"]]`.

- We *replace* a column using the assignment operator. For example:

```
> mice.data$Color <- c("blue", "white", "orange", "pink", "magenta", "blue", "orange", "blue
```

Above, we could also have replaced the `Color` column using any of the following:

- `mice.data[, 1] <-`
- `mice.data[[1]] <-`
- `mice.data[["Color"]] <-`

In the first two of these, R chooses a name for the variable in the data frame. In the third one, it sticks with `Color`.

6.2.3 Adding a New Column Using `cbind()`, `[ ]`, `[[ ]]`, or `$`

- We can add another column to a data frame using `cbind()`:

```
> Bodyfat <- c(2.5, 2.1, 3.1, 3.0, 2.7, 2.6, 1.8, 2.0)
> cbind(mice.data, Bodyfat)
```

```
  Color Weight Length Bodyfat
1 white     23    3.8     2.5
2  grey     21    3.7     2.1
3 black     12    3.0     3.1
4 brown     26    3.4     3.0
5 black     25    3.4     2.7
6 white     22    3.1     2.6
7 black     26    3.5     1.8
8 white     19    3.2     2.0
```

Be aware that `cbind()` will convert a `"character"` vector to a so-called *factor* unless you specify `stringsAsFactors = FALSE` in the call to `cbind()`.

- Alternatively, we can add a column to a data frame using `[ ]`, `[[ ]]`, or `$` and the assignment operator. For example, any of the following will work:

```
> mice.data$Fat <- Bodyfat          # The variable in mice.data will be named Fat
> mice.data[["Fat"]] <- Bodyfat     # The variable in mice.data will be named Fat
> mice.data[[4]] <- Bodyfat         # R will choose a name for the variable in mice.data
> mice.data[ , 4] <- Bodyfat        # R will choose a name for the variable in mice.data
```

As per the comments, only the first two methods above allow you to choose a name for the new column. You'd have to add a name later (using `names()`) if you use the third or fourth ones.

Section 6.2 Exercises

Exercise 6 Consider the following data on nine people set:

Status	Age	Education
Married	36	HS Diploma
Single	33	Bachelor of Arts
Single	21	Bachelor of Science
Married	29	Bachelor of Science
Single	19	HS Diploma
Married	35	Bachelor of Arts
Married	39	Master of Science
Single	28	HS Diploma
Single	21	HS Diploma

Here are vectors containing the data:

```
> status <- c("Married", "Single", "Single", "Married", "Single",
              "Married", "Married", "Single", "Single")
> age <- c(36, 33, 21, 29, 19, 35, 39, 28, 21)
> educ <- c("HS Diploma", "Bachelor of Arts", "Bachelor of Science",
            "Bachelor of Science", "HS Diploma", "Bachelor of Arts",
            "Master of Science", "HS Diploma", "HS Diploma")
```

and here's a command that will create a data frame containing the data:

```
> my.data <- data.frame(Status = status, Age = age, Education = educ)
```

a) Write a command involving square brackets `[ ]` that returns the age of the person in the 6th row.

b) Write a command involving square brackets `[ ]` and the assignment operator `<-` that replaces the age of the person in the 6th row by 36.

c) Write three different commands that return the entire 2nd column of the data frame:

- Using single square brackets `[ ]`.
- Using the dollar sign operator `$`.
- Using double square brackets `[[ ]]`.

d) The nine people have each aged one year since the data were collected. Here's a vector containing their current ages:

```
> age2 <- <- c(37, 34, 22, 30, 20, 36, 40, 29, 22)
```

Write three different commands that replace the entire 2nd column of the data frame by the `age2` vector:

- Using single square brackets `[ ]`.
- Using the dollar sign operator `$`.
- Using double square brackets `[[ ]]`.

e) Here's another vector:

```
> ageofspouse <- c(39, NA, NA, 34, NA, 27, 30, NA, NA)
```

Write three different commands that add the `ageofspouse` data as a new column in the data frame:

- Using single square brackets `[ ]`.
- Using the dollar sign operator `$`.
- Using double square brackets `[[ ]]`.

6.3 Viewing and Changing Variable Names in a Data Frame

- We can get or change the names of the columns of a data frame using:

```
names()          # Get or assign the names of the variables in a
                 # data frame
```

- For example (using the `mice.data` data frame from above):

```
> names(mice.data)

[1] "Color"  "Weight" "Length"

> names(mice.data) <- c("Col", "Wt", "Len")

> names(mice.data)

[1] "Col" "Wt"  "Len"
```

- To change just one of the column names, such as the 3rd one, we could use:

```
> names(mice.data)[3] <- "Len"
```

Section 6.3 Exercises

Exercise 7 Create the following data frame:

```
> x <- data.frame(A = 1:5, B = 6:10, C = c("a", "b", "c", "d", "e"))
```

a) Guess what each of the following commands will return, then check your answers:

```
> names(x)

> is.vector(names(x))
```

```
> typeof(names(x))
```

b) Guess what the following will do, then check your answer:

```
> names(x) <- c("AA", "BB","CC")
```

c) Now write a command that changes the name of the 3rd column of `x` to `"DD"`.

6.4 Rearranging the Rows or Columns of a Data Frame

6.4.1 Rearranging the Rows or Columns

- We rearrange (permute) the rows (or columns) of a data frame just as we did for a matrix, using a vector of indices with the desired permutation before (or after) a comma in square brackets `[ ]`.

  For example here are the **mice data** again:

```
> mice.data
```

```
    Col Wt Len
1 white 23 3.8
2  grey 21 3.7
3 black 12 3.0
4 brown 26 3.4
5 black 25 3.4
6 white 22 3.1
7 black 26 3.5
8 white 19 3.2
```

```
> mice.data[ , c(2, 3, 1)]                # Rearranges the columns
```

```
  Wt Len   Col
1 23 3.8 white
2 21 3.7  grey
3 12 3.0 black
4 26 3.4 brown
5 25 3.4 black
6 22 3.1 white
7 26 3.5 black
8 19 3.2 white
```

```
> mice.data[8:1, ]                  # Rearranges the rows
```

```
    Col Wt Len
8 white 19 3.2
7 black 26 3.5
6 white 22 3.1
5 black 25 3.4
4 brown 26 3.4
3 black 12 3.0
2  grey 21 3.7
1 white 23 3.8
```

6.4.2 Sorting Rows by the Values of One Variable

- We can sort the rows of a data frame according to the values of one of its variables (columns) using square brackets `[ ]` and the `order()` function.
- `order()` takes a vector argument `x`, and returns a set of indices that will permute the elements of `x` in increasing order:

```
> x <- c(7, 9, 5)
> order(x)                    # Returns the indices that will sort x

[1] 3 1 2
```

This says that the 3rd element of `x` is the smallest, the 1st element is the second smallest, and the 2nd element is the largest. Thus to sort `x`, we'd type:

```
> x[c(3, 1, 2)]

[1] 5 7 9
```

or just:

```
> x[order(x)]                 # This does the same thing as sort(x)

[1] 5 7 9
```

- As another example, here's the set of indices that will order the weights of the mice from `mice.data` from smallest to largest:

```
> order(mice.data$Wt)

[1] 3 8 2 6 1 5 4 7
```

If we want to order the *entire set of rows* of `mice.data` by weight, we type:

```
> mice.data[order(mice.data$Wt), ]
```

```
    Col Wt Len
3 black 12 3.0
8 white 19 3.2
2  grey 21 3.7
6 white 22 3.1
1 white 23 3.8
5 black 25 3.4
4 brown 26 3.4
7 black 26 3.5
```

Section 6.4 Exercises

Exercise 8 Here's a small data frame:

```
> x <- data.frame(x1 = c("a", "b", "c"), x2 = c(1, 2, 3))
> x
```

```
  x1 x2
1  a  1
2  b  2
3  c  3
```

Write a command involving indices inside square brackets `[ ]` that rearranges the rows of `x` so that they're in the following order:

```
  x1 x2
3  c  3
1  a  1
2  b  2
```

Exercise 9 Here's another small data frame:

```
> x <- data.frame(x1 = c(6, 4, 5), x2 = c("h", "g", "f"),
                  stringsAsFactors = FALSE)
> x
```

```
  x1 x2
1  6  h
2  4  g
3  5  f
```

a) Guess what the following command will do, then check your answer:

```
> x <- x[order(x$x1), ]
```

b) What do you think will happen if you try to sort the rows of `x`, as in part *a*, but using the `"character"` variable `x2`? Try it.

6.5 Filtering on Data Frames

- Recall that **filtering** means extracting a subset of rows that satisfy some condition. We can filter rows from a data frame using either square brackets `[ ]` or `subset()`, just as we did with matrices.

6.5.1 Filtering Rows Using Square Brackets `[ ]`

- For example, to extract from `mice.data` the rows corresponding to *black mice* using square brackets `[ ]`, we type:

```
> mice.data[mice.data$Col == "black", ]    # mice.data$Col=="black" is a "logical" vector
```

```
    Col Wt Len
3 black 12 3.0
5 black 25 3.4
7 black 26 3.5
```

Note that `mice.data$Col=="black"` is a `"logical"` vector.

6.5.2 Filtering Rows Using `subset()`

- `subset()` takes two main arguments, a data frame, `x`, and a condition to be met, `subset`, and then extracts from `x` the rows for which the condition is met.
- For example, to extract the rows of `mice.data` the rows corresponding to *black mice*, type:

```
> subset(mice.data, subset = Col == "black")
```

```
    Col Wt Len
3 black 12 3.0
5 black 25 3.4
7 black 26 3.5
```

Notice that there's no need to use the dollar sign operator `$` with `Col` when `subset()` is used.

Section 6.5 Exercises

Exercise 10 The built-in R data frame `warpbreaks` contains data from a study of the strength of yarn used in weaving. There are three variables in the data set:

`breaks`	The number of breaks per loom, where a loom corresponds to a fixed length of yarn.
`wool`	The type of wool (A or B)
`tension`	The level of tension (L, M, H)

To look at the data, type:

```
> warpbreaks
```

If you want to read about it, look at its help file by typing:

```
> ? warpbreaks
```

a) Write a command involving square brackets `[ ]` that returns just the rows of `warpbreaks` corresponding to observations made at the `M` level of `tension`.

b) Now write a command that uses `subset()` to do the same thing as in part *a*.

6.6 More on the Treatment of NAs

- If our data set contains `NA`s, we may want to exclude from the data analysis any observations (rows) for which one or more values are `NA`. For this, the following function may be useful:

```
complete.cases()        # Checks whether or not each row of a data frame
                        # is complete (i.e. doesn't contain NAs), and
                        # returns a "logical" vector
```

- `complete.cases()` takes a data frame as its main argument, and returns a `"logical"` vector whose elements are `TRUE` if the corresponding row of the data frame is "complete" (doesn't contain any `NA`s) and `FALSE` otherwise.

  Consider, for example, the following data frame:

```
> cars
```

```
         Make  Model Year CityMPG HighwayMPG
1        Ford  Focus 2012      17         25
2      Toyota  Prius 2013      51         NA
3        Ford Fusion 2014      22         31
4   Chevrolet   Volt 2015      NA         NA
5       Honda Accord 2011      23         33
6  Volkswagen Beetle 2012      22         31
7   Chevrolet Impala 2015      22         31
8       Tesla ModelS 2014      NA         NA
9       Honda  Civic 2011      26         34
10      Honda  S2000 2005      17         23
11     Toyota  Camry 2013      25         35
```

  To determine which rows are "complete", type:

```
> complete.cases(cars)
```

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

This says that all of the rows *except* the 2nd, 4th, and 8th are "complete".

We use the `"logical"` vector returned by `complete.cases()` to extract a data frame's "complete" rows as follows:

```
> cars[complete.cases(cars), ]
```

```
         Make  Model Year CityMPG HighwayMPG
1        Ford  Focus 2012      17         25
3        Ford Fusion 2014      22         31
5       Honda Accord 2011      23         33
6  Volkswagen Beetle 2012      22         31
7   Chevrolet Impala 2015      22         31
9       Honda  Civic 2011      26         34
10      Honda  S2000 2005      17         23
11     Toyota  Camry 2013      25         35
```

Section 6.6 Exercises

Exercise 11 Here's a data frame:

```
> x <- data.frame(x1 = c(8, 2, 6, NA, 9, 4, 3),
                  x2 = c("u", "r", "s", "v", "c", "t", "w"),
                  x3 = c(2, NA, 4, 7, 7, NA, 1))
> x
```

```
  x1 x2 x3
1  8  u  2
2  2  r NA
3  6  s  4
4 NA  v  7
5  9  c  7
6  4  t NA
7  3  w  1
```

a) Guess what the following command will return, then check your answer:

```
> complete.cases(x)
```

b) Now write a command that returns just the rows of `x` that are "complete" (i.e. that don't contain `NA`s).

6.7 Merging Data Frames

- ***Merging*** two data frames means combining them together, either by placing the rows of

one at bottom of the other, or by tacking the columns of one on to the right side of the other.

6.7.1 Merging the Rows of Two Data Frames

To merge the *rows* of two data frames, we can use `rbind()`:

```
rbind()       # Create a new data frame by "binding" the rows of one
              # data frame to those of another
```

- For example, we can combine the rows of the following two data frames:

```
> SomeNamesAndAges

      Name Age
1     John  23
2    Karen  27
3 Margaret  19
4      Ann  36
5     Karl  32
6     Eric  24

> MoreNamesAndAges

    Name Age
1 Justin  22
2  Janet  28
```

by typing:

```
> NamesAndAges <- rbind(SomeNamesAndAges, MoreNamesAndAges)
```

The result is:

```
> NamesAndAges

      Name Age
1     John  23
2    Karen  27
3 Margaret  19
4      Ann  36
5     Karl  32
6     Eric  24
7   Justin  22
8    Janet  28
```

6.7.2 Merging the Columns of Two Data Frames

- To merge the columns of two data frames, some useful functions are:

```
cbind()       # Create a new data frame by "binding" the columns of one
              # data frame to those of another
merge()       # Merge two data frames that share one or more variables
              # in common
```

- For example, suppose we want to combine the following data frame with `NamesAndAges` (from above):

```
> NamesAndWeights

      Name Weight
1     John    155
2    Karen    170
3 Margaret    147
4      Ann    159
5     Karl    201
6     Eric    184
7   Justin    161
8    Janet    154
```

If we combine them using `cbind()`, we get:

```
> cbind(NamesAndAges, NamesAndWeights)

      Name Age     Name Weight
1     John  23     John    155
2    Karen  27    Karen    170
3 Margaret  19 Margaret    147
4      Ann  36      Ann    159
5     Karl  32     Karl    201
6     Eric  24     Eric    184
7   Justin  22   Justin    161
8    Janet  28    Janet    154
```

Notice that the `Name` variable is common to both data frames, and `cbind()` leaves them as two separate columns. If the people were listed in different orders, `cbind()` would make no attempt to put them in the same order:

```
> JumbledNamesAndWts

      Name Weight
6     Eric    184
4      Ann    159
2    Karen    170
7   Justin    161
3 Margaret    147
1     John    155
8    Janet    154
5     Karl    201
```

```
> cbind(NamesAndAges, JumbledNamesAndWts)
```

```
      Name Age     Name Weight
6     John  23     Eric    184
4    Karen  27      Ann    159
2 Margaret  19    Karen    170
7      Ann  36   Justin    161
3     Karl  32 Margaret    147
1     Eric  24     John    155
8   Justin  22    Janet    154
5    Janet  28     Karl    201
```

- If we want the names to be matched when we merge `NamesAndAges` with `JumbledNamesAndWts`, we use `merge()`, specifying `by = "Name"`:

```
> merge(NamesAndAges, JumbledNamesAndWts, by = "Name")
```

```
      Name Age Weight
1      Ann  36    159
2     Eric  24    184
3    Janet  28    154
4     John  23    155
5   Justin  22    161
6    Karen  27    170
7     Karl  32    201
8 Margaret  19    147
```

Notice that `merge()` sorted the rows of both data frames according to the alphabetical ordering of `Name` before merging them.

Notice also that `merge()` didn't include a redundant `Name` column in the result.

- Sometimes the values in *two* columns are needed to distinguish between individuals in a data set.

For example, suppose some `Names` were *duplicated* (e.g. there are two Johns and three Anns below), but we had another column `City` that could be used to distinguish between them:

```
> NamesAndAges
```

```
      Name       City Age
1     John     Denver  23
2     John  Longmont  42
3    Karen     Salida  27
4 Margaret     Denver  19
5      Ann    Boulder  36
6      Ann     Denver  29
7      Ann Leadville  45
8     Karl     Denver  32
9     Eric    Boulder  24
```

```
> JumbledNamesAndWts
```

```
      Name      City Weight
7      Ann Leadville    164
4 Margaret    Denver    147
2     John  Longmont    203
5      Ann   Boulder    159
3    Karen    Salida    170
1     John    Denver    155
6      Ann    Denver    161
9     Eric   Boulder    184
8     Karl    Denver    201
```

In this case, to properly merge the two data frames, it needs to be done by the values in *both* columns.

To to do this, we specify by both `Name` *and* `City` in a `"character` vector passed to `merge()` via the `by` argument:

```
> merge(NamesAndAges, JumbledNamesAndWts, by = c("Name", "City"))
```

```
      Name      City Age Weight
1      Ann   Boulder  36    159
2      Ann    Denver  29    161
3      Ann Leadville  45    164
4     Eric   Boulder  24    184
5     John    Denver  23    155
6     John  Longmont  42    203
7    Karen    Salida  27    170
8     Karl    Denver  32    201
9 Margaret    Denver  19    147
```

In fact, by default `merge()` merges data frames by whatever column names they have in common, so in the examples above it actually wasn't necessary to specify anything via the argument `by`.

Section 6.7 Exercises

Exercise 12 Here are two data frames, `dfA` and `dfB`, containing responses to a survey question:

```
> dfA <- data.frame(IDNumber = c(1000, 1001, 1002, 1003),
                    Response = c(55, 62, 39, 45))
> dfA
```

```
  IDNumber Response
1     1000       55
2     1001       62
```

```
3     1002       39
4     1003       45


> dfB <- data.frame(IDNumber = c(1004, 1005, 1006),
                    Response = c(70, 77, 56))
> dfB


  IDNumber Response
1     1004       70
2     1005       77
3     1006       56
```

Write a command to merge the rows of these two data frames. You should end up with this:

```
  IDNumber Response
1     1000       55
2     1001       62
3     1002       39
4     1003       45
5     1004       70
6     1005       77
7     1006       56
```

Exercise 13 Here are two data frames containing responses to two survey questions:

```
> dfA <- data.frame(IDNumber = c(1000, 1001, 1002, 1003, 1004, 1005, 1006),
                    Response1 = c(55, 62, 39, 45, 70, 77, 56))
> dfA


  IDNumber Response1
1     1000        55
2     1001        62
3     1002        39
4     1003        45
5     1004        70
6     1005        77
7     1006        56


> dfB <- data.frame(IDNumber = c(1003, 1002, 1000, 1004, 1006, 1001, 1005),
                    Response2 = c(12, 17, 23, 24, 19, 30, 20))
> dfB
```

```
  IDNumber Response2
1     1003        12
2     1002        17
3     1000        23
4     1004        24
5     1006        19
6     1001        30
7     1005        20
```

a) Note that the `IDNumbers` are the same, but in different orders. Why would `cbind()` *not* be a good way to merge the *columns* of the two data frames?

b) Write a command involving `merge()` that merges the columns of the two data frames so that they're in the correct order. You should end up with this:

```
  IDNumber Response1 Response2
1     1000        55        23
2     1001        62        30
3     1002        39        17
4     1003        45        12
5     1004        70        24
6     1005        77        20
7     1006        56        19
```

Exercise 14 Here are two data frames:

```
> dfA <- data.frame(LastName = c("Smith", "Smith", "Jones", "Smith",
                                 "Olsen", "Taylor", "Olsen"),
                    FirstName = c("John", "Kim", "John", "Marge", "Bill",
                                  "Bill", "Erin"),
                    Gender = c("M", "F", "M", "F", "M", "M", "F"),
                    ExamScore = c(75, 80, 64, 78, 90, 89, 79))
> dfA
```

```
  LastName FirstName Gender ExamScore
1    Smith      John      M        75
2    Smith       Kim      F        80
3    Jones      John      M        64
4    Smith     Marge      F        78
5    Olsen      Bill      M        90
6   Taylor      Bill      M        89
7    Olsen      Erin      F        79
```

```
> dfB <- data.frame(LastName = c("Olsen", "Jones", "Taylor", "Smith",
                                 "Olsen", "Smith", "Smith"),
                    FirstName = c("Bill", "John", "Bill", "Kim", "Erin",
```

```
                                    "John", "Marge"),
                    Gender = c("M", "M", "M", "F", "F", "M", "F"),
                    Grade = c("A", "D", "B", "B", "C", "C", "C"))
> dfB


  LastName FirstName Gender Grade
1    Olsen      Bill      M     A
2    Jones      John      M     D
3   Taylor      Bill      M     B
4    Smith       Kim      F     B
5    Olsen      Erin      F     C
6    Smith      John      M     C
7    Smith     Marge      F     C
```

Notice that the two data frames are composed of the same seven people, but in different orders. Notice also that both the `LastName` *and* `FirstName` are needed to uniquely identify the people.

Write a command involving `merge()` that merges the columns of the two data frames by person. You should end up with this:

```
  LastName FirstName Gender ExamScore Grade
1    Jones      John      M        64     D
2    Olsen      Bill      M        90     A
3    Olsen      Erin      F        79     C
4    Smith      John      M        75     C
5    Smith       Kim      F        80     B
6    Smith     Marge      F        78     C
7   Taylor      Bill      M        89     B
```

6.8 Stacking and Unstacking Columns of a Data Frame

- Sometimes data are arranged in separate columns representing, say, different *groups*, but we'd prefer them to be in a single column with an adjacent column indicating the group.

  For example, the data might look like this:

  ```
  > unstacked.data


    Grp1 Grp2 Grp3
  1   23   19   31
  2   11   26   28
  3   14   24   34
  4   16   29   25
  ```

  but we'd prefer them to look like this:

```
> stacked.data
```

```
   Response Group
1        23  Grp1
2        11  Grp1
3        14  Grp1
4        16  Grp1
5        19  Grp2
6        26  Grp2
7        24  Grp2
8        29  Grp2
9        31  Grp3
10       28  Grp3
11       34  Grp3
12       25  Grp3
```

- Other times we need to do the opposite – we want to take a single column, that has an adjacent *group* identifier column, and turn it into multiple columns, each of which corresponds to a group.
- The functions below are useful for such tasks.

```
stack()           # "Stack" columns in a data frame
unstack()         # "Unstack" a column in a data frame
```

- For example, to get from the `unstacked.data` above to the `stacked.data`, we'd type:

```
> stacked.data <- stack(unstacked.data)
```

- To go in the opposite direction, we need indicate which column we want to "unstack" and which one is the group indicator column. We do this by passing, via the argument `form` in `unstack()`, a so-called ***formula*** whose left side is the name of the column to be unstacked and whose right side is name of the group indicator column. The left and right sides of the *formula* are separated by a tilde `~`:

```
> unstacked.data <- unstack(stacked.data, form = Response ~ Group)
> unstacked.data
```

```
  Grp1 Grp2 Grp3
1   23   19   31
2   11   26   28
3   14   24   34
4   16   29   25
```

Section 6.8 Exercises

Exercise 15 Here's a data frame containing responses for four individuals in each of three treatment groups in an experiment:

```
> x <- data.frame(GrpA = c(1, 4, 2, 3),
                  GrpB = c(7, 5, 8, 6),
                  GrpC = c(9, 9, 8, 7))
> x
```

```
  GrpA GrpB GrpC
1    1    7    9
2    4    5    9
3    2    8    8
4    3    6    7
```

Write a command involving `stack()` that "stacks" the columns of `x`. You should end up with something like this:

```
   values  ind
1       1 GrpA
2       4 GrpA
3       2 GrpA
4       3 GrpA
5       7 GrpB
6       5 GrpB
7       8 GrpB
8       6 GrpB
9       9 GrpC
10      9 GrpC
11      8 GrpC
12      7 GrpC
```

Exercise 16 Here's a data frame containing data from an experiment involving a *treatment group* and a *control group*:

```
> x <- data.frame(Group = c("Trt", "Trt", "Trt", "Trt", "Trt", "Ctrl",
                            "Ctrl", "Ctrl", "Ctrl", "Ctrl"),
                  Y = c(22, 45, 32, 45, 30, 60, 44, 24, 56, 59))
> x
```

```
   Group  Y
1    Trt 22
2    Trt 45
3    Trt 32
4    Trt 45
5    Trt 30
6   Ctrl 60
```

```
7   Ctrl 44
8   Ctrl 24
9   Ctrl 56
10  Ctrl 59
```

Write a command involving `unstack()` that "unstacks" the `Y` column in `x`. You should end up with this:

```
  Ctrl Trt
1   60  22
2   44  45
3   24  32
4   56  45
5   59  30
```

6.9 Applying Functions to Columns of a Data Frame

- Recall that *data frames* are *lists* (whose *elements* are the *columns* of the data frame).

  Thus we can use the *list* versions of the `apply()` function to apply a function separately to each *column* of a data frame:

```
lapply()      # Apply a function separately to each column of a data
              # frame, returning a list
sapply()      # Apply a function separately to each column of a data
              # frame, returning a vector
```

- When applied to data frames, `lapply()` and `sapply()` both take arguments `X`, a data frame, and `FUN`, the function to be applied to each column of the data frame. Recall that the difference between `lapply()` and `sapply()` is that:
  - `lapply()` returns a *list*. This usually *isn't* what we want.
  - `sapply()` returns a *vector*. This is usually what we want.
- Here are two examples in which `sapply()` is used, with `FUN` set to `mean()` and `sd()`, to compute the mean and standard deviation for each column of the data frame `unstacked.data` from above:

```
> unstacked.data

  Grp1 Grp2 Grp3
1   23   19   31
2   11   26   28
3   14   24   34
4   16   29   25
```

```
> sapply(X = unstacked.data, FUN = mean)

Grp1 Grp2 Grp3
16.0 24.5 29.5

> sapply(X = unstacked.data, FUN = sd)

    Grp1     Grp2     Grp3
5.099020 4.203173 3.872983
```

Note that `sapply()` returns a vector in each case.

To do the same thing using `lapply()`, we'd type:

```
> unstacked.data

  Grp1 Grp2 Grp3
1   23   19   31
2   11   26   28
3   14   24   34
4   16   29   25

> lapply(X = unstacked.data, FUN = mean)

$Grp1
[1] 16

$Grp2
[1] 24.5

$Grp3
[1] 29.5

> lapply(X = unstacked.data, FUN = sd)

$Grp1
[1] 5.09902

$Grp2
[1] 4.203173

$Grp3
[1] 3.872983
```

Note that `lapply()` returns a list, which usually isn't what we want.

- If we just want the mean or sum of each column, it's sometimes easier to use:

```
colMeans()    # Compute the mean separately for each column of a data
              # frame
colSums()     # Compute the sum separately for each column of a data
              # frame
```

- Here's an example:

```
> colMeans(unstacked.data)

Grp1 Grp2 Grp3
16.0 24.5 29.5
```

Section 6.9 Exercises

Exercise 17 Here are commands that will create the `unstacked.data` from above:

```
> Group1 <- c(23,11,14,16)
> Group2 <- c(19, 26, 24, 29)
> Group3 <- c(31, 28, 34, 25)
> unstacked.data <- data.frame(Grp1 = Group1, Grp2 = Group2, Grp3 = Group3)
> unstacked.data


  Grp1 Grp2 Grp3
1   23   19   31
2   11   26   28
3   14   24   34
4   16   29   25
```

a) Write commands involving `sapply()` that do the following:

- Find the minimum value in each column of `unstacked.data`. **Hint**: Use `FUN = min` in `sapply()`.
- Find the index (position) of the minimum value in each column. **Hint**: Use `FUN = which.min` in `sapply()`.

b) Now use `colMeans()` and `colSums()` to compute the mean and sum separately for each column of `unstacked.data`.

Exercise 18 Here's the data from the experiment involving a treatment group and a control group:

```
> x <- data.frame(Group = c("Trt", "Trt", "Trt", "Trt", "Trt", "Ctrl",
                            "Ctrl", "Ctrl", "Ctrl", "Ctrl"),
                  Y = c(22, 45, 32, 45, 30, 60, 44, 24, 56, 59))
> x
```

```
   Group  Y
1    Trt 22
2    Trt 45
3    Trt 32
4    Trt 45
5    Trt 30
6   Ctrl 60
7   Ctrl 44
8   Ctrl 24
9   Ctrl 56
10  Ctrl 59
```

How would you reorganize the data so that you could use `colMeans()` to compute the means of the treatment and control groups separately? Give your R command(s).