Playing with data I: data manipulation and logical statements

Gustavo Diaz
Lucie Lu
UIUC Political Science Math Camp

August 13, 2019

Objectives

- Quick overview of the project data set and introduction to logical statements
- Basic data manipulation: Other useful operations to manipulate and visualize data

Expectation

- Type the codes along with us even if they are provided on the sheet to familiarize yourself with the syntax in R.
- Do the exercises in class. Participate and ask questions.

Part I: Quick overview of the dataset and logical operations in R

Logical statements are a natural extension of logical data. To the extent that R can be understood as a programming language, we can use logical statements to perform a variety of tasks. We will not cover all the uses today, just the ones that are useful for quantitative social science.

The way R handles logic is very similar to mathematical logic. After all, computers are nothing but fancy calculators, and software is just a tool to interface with it. However, what can be considered a logical statement in R extends a little bit beyond math. Mostly because we are working with data at multiple levels at the same time. So we need an extended vocabulary to communicate those nuances.

Logical operators are a core part of this process. However, before we manipulate our data in any way, the first thing we want to do is take a quick look at what we are working with.

To illustrate this, we will go back to the `Melanoma` data set from the `MASS` package. The first step is to load the package and the data within it:

1. Load the data

```r
# Load package
library(MASS)

# Load data
data("Melanoma")

#Note that the following doesn't work
mydata <- data("Melanoma")
```

```r
class(Melanoma) #data.frame
```

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

```r
class(mydata) #character
```

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

```r
mydata <- Melanoma #save it correctly
class(mydata) #data.frame
```

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

```r
rm(mydata) #drop it

# It may be helpful to look at data documentation too
?Melanoma
```

2. Overview of the dataset

The most elementary function for visualization will simply open a new window to show the data frame as a spreadsheet.

```r
#Double click or use the View function to look at the data

View(Melanoma)
```

This is useful, but will quickly become hard to look at as the number of variables and observations increases. If what you want is a brief description of variables with a few sample values, you may try the following:

```r
head(Melanoma)
```

```
##   time status sex age year thickness ulcer
## 1   10      3   1  76 1972      6.76     1
## 2   30      3   1  56 1968      0.65     0
## 3   35      2   1  41 1977      1.34     0
## 4   99      3   0  71 1968      2.90     0
## 5  185      1   1  52 1965     12.08     1
## 6  204      1   1  28 1971      4.84     1
```

```r
# Show more than 6
head(Melanoma, 10)
```

```
##    time status sex age year thickness ulcer
## 1    10      3   1  76 1972      6.76     1
## 2    30      3   1  56 1968      0.65     0
## 3    35      2   1  41 1977      1.34     0
## 4    99      3   0  71 1968      2.90     0
## 5   185      1   1  52 1965     12.08     1
## 6   204      1   1  28 1971      4.84     1
## 7   210      1   1  77 1972      5.16     1
## 8   232      3   0  60 1974      3.22     1
## 9   232      1   1  49 1968     12.88     1
## 10  279      1   0  68 1971      7.41     1
```

```r
# Also you may want to know about its counterpart
tail(Melanoma)
```

```
##     time status sex age year thickness ulcer
## 200 4479      2   0  19 1965      1.13     1
## 201 4492      2   1  29 1965      7.06     1
## 202 4668      2   0  40 1965      6.12     0
## 203 4688      2   0  42 1965      0.48     0
## 204 4926      2   0  50 1964      2.26     0
## 205 5565      2   0  41 1962      2.90     0
```

Descriptive statistics is a whole world on its own, but a good start is the `summary` function.

```
#Look at all the variables and distribution at one time
summary(Melanoma)
```

```
##       time          status          sex               age       
##  Min.   :  10   Min.   :1.00   Min.   :0.0000   Min.   : 4.00  
##  1st Qu.:1525   1st Qu.:1.00   1st Qu.:0.0000   1st Qu.:42.00  
##  Median :2005   Median :2.00   Median :0.0000   Median :54.00  
##  Mean   :2153   Mean   :1.79   Mean   :0.3854   Mean   :52.46  
##  3rd Qu.:3042   3rd Qu.:2.00   3rd Qu.:1.0000   3rd Qu.:65.00  
##  Max.   :5565   Max.   :3.00   Max.   :1.0000   Max.   :95.00  
##       year        thickness         ulcer      
##  Min.   :1962   Min.   : 0.10   Min.   :0.000  
##  1st Qu.:1968   1st Qu.: 0.97   1st Qu.:0.000  
##  Median :1970   Median : 1.94   Median :0.000  
##  Mean   :1970   Mean   : 2.92   Mean   :0.439  
##  3rd Qu.:1972   3rd Qu.: 3.56   3rd Qu.:1.000  
##  Max.   :1977   Max.   :17.42   Max.   :1.000
```

```
#Look at the names of variables only
names(Melanoma)
```

```
## [1] "time"      "status"    "sex"       "age"       "year"      "thickness"
## [7] "ulcer"
```

```
#Return the numbers of observations
nrow(Melanoma)
```

```
## [1] 205
```

```
#Return the numbers of variables
ncol(Melanoma)
```

```
## [1] 7
```

```
#combine the outpus of ncol and nrow
dim(Melanoma)
```

```
## [1] 205   7
```

This is enough to observe variable names and sample values. If you want a bit more information, we can use the `tibble` format to get more information about the objects we work with.

```
# install.packages("tidyverse")
library(tidyverse)

as.tibble(Melanoma)
```

```
## # A tibble: 205 x 7
##     time status   sex   age  year thickness ulcer
##    <int>  <int> <int> <int> <int>     <dbl> <int>
## 1     10       3     1     76  1972       6.76     1
## 2     30       3     1     56  1968       0.65     0
## 3     35       2     1     41  1977       1.34     0
## 4     99       3     0     71  1968       2.9      0
## 5    185       1     1     52  1965      12.1      1
## 6    204       1     1     28  1971       4.84     1
## 7    210       1     1     77  1972       5.16     1
## 8    232       3     0     60  1974       3.22     1
## 9    232       1     1     49  1968      12.9      1
## 10   279       1     0     68  1971       7.41     1
## # ... with 195 more rows
```

A tibble is a modern approach to data frames, they are part of the tidyverse family of packages developed by the R studio community. Tibbles have many uses, but for now is enough to know that they understand your data better than base R. The first line in the output normally shows the dimensions of your object. That is, the number of rows (observations) and columns (variables). For each variable it will also indicate the type. In this example `<int>` stands for integer, and `<dbl>` stands for double (i.e. numbers that allow decimals).

While good for a description of the structure of the data, tibbles do not say much about the distribution of the variables.

##3. Use "$" operator and bracket to extract information

The `status` variable contains information about melanoma patients (i.e. skin cancer). 1 means they died from melanoma, 2 means they are still alive, and 3 that they died from other causes. To zoom into a specific variable, we can use the basic extraction operator `$`:

```
Melanoma$status
```

```
##   [1] 3 3 2 3 1 1 1 3 1 1 1 3 1 1 1 3 1 1 1 1 1 1 1 1 1 1 3 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 3 1 2 1 2 2 2 1 3 2 1 2 1 2 1 2 1 2 2 2 2 2 2 2 2 1 2
##  [71] 2 1 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2
## [106] 2 2 2 2 2 1 1 2 3 2 1 2 1 2 2 2 2 2 2 2 1 2 2 2 2 1 2 2 2 2 2 1 2 2 2
## [141] 2 2 1 2 2 2 2 2 1 2 2 2 2 1 2 2 2 2 2 3 2 3 2 2 2 2 2 2 2 2 1 2 2 2 2
## [176] 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

The `$` operator helps us extract information from a named list or object. In the case of data frames, because variables have names, we can use extraction to refer to them.

Keep in mind that the output of `summary()` varies a lot depending on the type of object it is applied to. It is also good to know that you can combine this function with the extraction operator to focus on a single variable of interest.

```
summary(Melanoma$status)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    1.00    2.00    1.79    2.00    3.00
```

When we extract a variable from a data frame, the output becomes a **vector**. Vectors interface easily with mathematical operators. Of interest here are logical statements.

Another way of retrieving individual variables is to use indexing inside square brackets[ ], as done for a vector. Since a data frame object is usually two-dimensional, we need two indexes, one for rows and the other for columns. We use [rows, columns] to call specific rows and columns by either calling their numbers or names.

```
#Use bracket to extract the column called "status"
Melanoma[, "status"] #called by name
```

```
##   [1] 3 3 2 3 1 1 1 3 1 1 1 3 1 1 1 3 1 1 1 1 1 1 1 1 1 1 3 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 3 1 2 1 2 2 2 1 3 2 1 2 1 2 1 2 1 2 2 2 2 2 2 2 2 1 2
##  [71] 2 1 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2
## [106] 2 2 2 2 2 1 1 2 3 2 1 2 1 2 2 2 2 2 2 2 1 2 2 2 2 1 2 2 2 2 2 1 2 2 2
## [141] 2 2 1 2 2 2 2 2 1 2 2 2 2 1 2 2 2 2 2 3 2 3 2 2 2 2 2 2 2 2 1 2 2 2 2
## [176] 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

```
Melanoma[, 2] #status is the second column
```

```
##   [1] 3 3 2 3 1 1 1 3 1 1 1 3 1 1 1 3 1 1 1 1 1 1 1 1 1 1 3 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 3 1 2 1 2 2 2 1 3 2 1 2 1 2 1 2 1 2 2 2 2 2 2 2 2 1 2
##  [71] 2 1 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2
## [106] 2 2 2 2 2 1 1 2 3 2 1 2 1 2 2 2 2 2 2 2 1 2 2 2 2 1 2 2 2 2 2 1 2 2 2
## [141] 2 2 1 2 2 2 2 2 1 2 2 2 2 1 2 2 2 2 2 3 2 3 2 2 2 2 2 2 2 2 1 2 2 2 2
## [176] 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

```
Melanoma[1:3, ] #first three rows
```

```
##   time status sex age year thickness ulcer
## 1   10      3   1  76 1972      6.76     1
## 2   30      3   1  56 1968      0.65     0
## 3   35      2   1  41 1977      1.34     0
```

```
Melanoma[1:3, "status"] #first three rows of the "status" column
```

```
## [1] 3 3 2
```

```
Melanoma[c(2,5,10,15), ] #use c() sequencing function to extract any rows you like
```

```
##    time status sex age year thickness ulcer
## 2    30      3   1  56 1968      0.65     0
## 5   185      1   1  52 1965     12.08     1
## 10  279      1   0  68 1971      7.41     1
## 15  469      1   0  14 1969      2.42     1
```

In some circumstances we want to focus on more than one variable at a time. The referencing operator `[]` serves that purpose by letting us specify elements within an object. In the case of data frames, rows and columns.

```
# Show the first six rows, equivalent to head()
Melanoma[1:6, ]
```

```
##   time status sex age year thickness ulcer
## 1   10      3   1  76 1972      6.76     1
## 2   30      3   1  56 1968      0.65     0
## 3   35      2   1  41 1977      1.34     0
## 4   99      3   0  71 1968      2.90     0
## 5  185      1   1  52 1965     12.08     1
## 6  204      1   1  28 1971      4.84     1
```

```
head(Melanoma)
```

```
##   time status sex age year thickness ulcer
## 1   10      3   1  76 1972      6.76     1
## 2   30      3   1  56 1968      0.65     0
## 3   35      2   1  41 1977      1.34     0
## 4   99      3   0  71 1968      2.90     0
## 5  185      1   1  52 1965     12.08     1
## 6  204      1   1  28 1971      4.84     1
```

```
# Select three variables and save them in a new object
vars <- c("status", "sex", "age")
```

```
#Melanoma[vars] #output too long
Melanoma[1:6, vars] #extract the first six rows
```

```
##   status sex age
## 1      3   1  76
## 2      3   1  56
## 3      2   1  41
## 4      3   0  71
## 5      1   1  52
## 6      1   1  28
```

Exercise 1

1. Use head function, and extract the first 10th rows of the data
2. Extract the "age" column from the 10th and 15th rows
3. Use dollar sign to extract access other individual variables that you find interesting
4. Extract the third, fifth and eighth rows of variables "time", "ulcer" and "year".

4. Logical operators

Logical operators evaluate each element of a vector.

```
# less than
Melanoma$status < 2

# greater than
Melanoma$status > 2

# less or equal than
Melanoma$status <= 1

identical(Melanoma$status < 2, Melanoma$status <= 1)
# Why are these two identical?
```

Two things to be cautious of when working with mathematical logic in R is that the `=` operator stands for assignment. Which means it is equivalent to `<-`. Also, there is no keystroke to write the inequality ≠ sign. This is how you do both of them:

```
# equal
equal <- Melanoma$status == 1

# not equal
unequal <- Melanoma$status != 1

# Also, note the behavior of ! in general
identical(equal, unequal)
```

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

```
!identical(equal, unequal)
```

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

```r
identical(equal, !unequal)
```

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

By now, it is apparent that logical data in R can be either `TRUE` or `FALSE`. Also you can use logical operators & ("AND") and | ("OR") respectively.

```r
FALSE & TRUE
```

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

```r
TRUE & TRUE
```

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

```r
FALSE & FALSE
```

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

```r
TRUE | FALSE
```

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

```r
FALSE | FALSE
```

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

But you should also know that missing data is ignored in logical operations. Pay attention to how we handle NAs.

```r
# Vector of fake data, for the sake of exposition
fakedata <- c(1, 2, 3, 4, 5, 6, NA)

fakedata > 3
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE    NA
```

```r
fakedata != 1
```

```
## [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE    NA
```

```r
# use logical conjunction/disjunction of two vectors with logical values

(fakedata > 2) & (fakedata < 4)
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE FALSE    NA
```

```r
(fakedata > 2) | (fakedata < 4)
```

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

```r
# count how many satisfy this condition; number of trues
sum((fakedata > 2) & (fakedata < 4)) #why doesn't work; preview
```

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

```r
sum((fakedata > 2) & (fakedata < 4), na.rm = TRUE)
```

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

One more thing to know about logical data is that, despite the TRUE/FALSE labels, it is actually binary (0 or 1). That means that some operations that are valid for binary data can apply to logical data without any transformations. For example:

```
# How do we interpret this quantity?
mean(Melanoma$status == 1)
```

```
## [1] 0.2780488
```

```
#Verify yourself
sum(Melanoma$status == 1)
```

```
## [1] 57
```

```
length(Melanoma$status == 1)
```

```
## [1] 205
```

```
sum(Melanoma$status == 1)/length(Melanoma$status == 1)
```

```
## [1] 0.2780488
```

Now we know how to use logical operators to extract variables and evaluate logical statements about them. This logic is the basis of many tasks in R. In the new section we put this intuition to practice.

##Exercise 2

1. Create a vector called `fakedata2` that contains the numbers 2, 4, 6, 8, 10.
2. Use logical conjunction to find the number of elements in `fakedata2` that are between 5 and 9.
3. Create a logical vector that finds the proportion of elements in `fakedata2` that are smaller than or equal to 8.

Part II: Basic data manipulation

The idea of cleaning data, more formally know as data munging or data wrangling, comprises all the tasks that come before performing the statistical analysis that are often reported in a research article. They involve loading the data, general exploration of its elements, subsetting to relevant groups, and recoding variables. It may also involve exploratory data analysis (EDA), but we are not covering that here.

Referencing makes subsetting data fairly straightforward. If we want to select specific observations, we use [,] to subset rows or columns. Note that the first element of the square brackets specifies the rows to be retained (under certain logical conditions) and the second element specifies the columns to be kept.

1. Subsetting

```
# Remove people dying from other causes, and save it in a new object
Mel_short <- Melanoma[Melanoma$status < 3,]

# Table is also a nice way to visualize data
table(Melanoma$status)
```

```
##
##   1   2   3
##  57 134  14
```

```
table(Mel_short$status) #check if subset is successful
```

```
##
##   1   2
##  57 134
```

```
head(Mel_short)
```

```
##    time status sex age year thickness ulcer
## 3    35      2   1  41 1977      1.34     0
## 5   185      1   1  52 1965     12.08     1
## 6   204      1   1  28 1971      4.84     1
## 7   210      1   1  77 1972      5.16     1
## 9   232      1   1  49 1968     12.88     1
## 10  279      1   0  68 1971      7.41     1
```

If we want to extract only certain variables, we subset observations:

```
Mel_narrow <- Melanoma[, vars]

head(Mel_narrow)
```

```
##   status sex age
## 1      3   1  76
## 2      3   1  56
## 3      2   1  41
## 4      3   0  71
## 5      1   1  52
## 6      1   1  28
```

An alternative to subsetting is the `subset()` function. It also depends on logical statements, but the syntax is more explicit.

```
Mel_subset_1 <- subset(Melanoma, age >= 50 & sex == 0, select = c(status, sex, age))

as.tibble(Mel_subset_1)
```

```
## # A tibble: 71 x 3
##    status   sex   age
##     <int> <int> <int>
##  1      3     0    71
##  2      3     0    60
##  3      1     0    68
##  4      1     0    53
##  5      3     0    64
##  6      1     0    68
##  7      1     0    89
##  8      1     0    67
##  9      3     0    86
## 10      1     0    56
## # ... with 61 more rows
```

```
# Note how they are different
Mel_subset_2 <- subset(Melanoma, age >=50 | sex == 0, select = c(status, sex, age))

as.tibble(Mel_subset_2)
```

```
## # A tibble: 176 x 3
##    status   sex   age
##     <int> <int> <int>
##  1      3     1    76
##  2      3     1    56
##  3      3     0    71
## 4       1     1    52
## 5       1     1    77
## 6       3     0    60
## 7       1     0    68
## 8       1     0    53
## 9       3     0    64
## 10      1     0    68
## # ... with 166 more rows
```

```r
Mel_subset_male <- subset(Melanoma, sex == 1) #male only
Mel_subset_female <- subset(Melanoma, sex == 0) #female only

mean(Mel_subset_male$age)
```

```
## [1] 53.89873
```

```r
mean(Mel_subset_female$age)
```

```
## [1] 51.56349
```

```r
#What's the average age gap across gender groups?
```

Alternatively, if you do not want to subset the data set, you can use the tapply() function, which applies a function repeatedly within each level of the factor variables.

Here we use the function as in tapply(X, INDEX, FUN), which applies the function indicated by argument FUN to the object X for each group defined by INDEX. In the following example, we apply the `mean()` function to the `age` variable for each category of the `sex` in the `Melanoma` data frame.

```r
tapply(Melanoma$age, Melanoma$sex, mean)
```

```
##        0        1
## 51.56349 53.89873
```

```r
#check yourself if tapply() function as you expect
mean(Melanoma$age[Melanoma$sex==1]) #male
```

```
## [1] 53.89873
```

```r
tapply(Melanoma$age, Melanoma$sex, mean) - mean(Melanoma$age[Melanoma$sex==1])
```

```
##         0         1
## -2.335242  0.000000
```

Exercise 3.1

What is the average age of patients dying from melanoma for males and females respectively? Hint: you may need to explore the documentation of this data set.

2. `ifelse()` and `recode()` function.

What if instead of subsetting our data we want to change some of its values? A different kind of logical statement, more familiar to programmers, is the `ifelse()` function. It performs conditional element selection. Suppose we want to create a new binary variable called `status2`. `status2` is collapsing the three categories in `status` into two. If the patient is alive, it is coded as 0. If the patient is dead for whatever reason, it becomes 1.

The function ifelse(X, Y, Z) contains three elements. For each element in X that is true, the corresponding element in Y is returned. In contrast, For each element in X that is false, the corresponding element in Z is returned.

```
# ?ifelse

Melanoma$status2 <- ifelse(Melanoma$status != 2, 1, 0)


#check the contingency table
table(Melanoma$status2,
      Melanoma$status)
```

```
##
##       1   2   3
##   0   0 134   0
##   1  57   0  14
```

This is useful but may end up producing code that is too convoluted if you want to recede based on more than one condition. It is still doable, but it becomes hard to follow:

```
Melanoma$status3 <- ifelse(Melanoma$status == 1, "dead",
                           ifelse(Melanoma$status == 2, "alive",
                                  ifelse(Melanoma$status == 3, "other", NA)))

#Melanoma$status3
```

And it becomes longer the more conditions you want to include, a more efficient way is to use the `recode()` function in the `dplyr` package, which is also part of the `tidyverse`.

```
Melanoma$status4 <- recode(Melanoma$status, "dead", "alive", "other")

table(Melanoma$status4, Melanoma$status)
```

```
##
##           1   2   3
##   alive   0 134   0
##   dead   57   0   0
##   other   0   0  14
```

Note that `recode()` matches by position. In this case we were lucky that the data was in order. But in some cases we have to pay attention to how variables are arranged.

Exercise 3.2

Create a new vector `age50` that divides patients into two groups based on age under 50 and over or equal to 50. 0: under 50; 1: equal to 50 or above. What's the percentage of patients over 50? Hint: using ifelse() function and make sure to check if your conditional statement is correct.

3. Create a new data frame

Note that `status2`, `status3`, `status4` will be added into the current data frame Melanoma. If you want to avoid messing up with the original data frame, what would you do?

You can work on a new data frame by duplicating the original one, then do the preceding data manipulation in the new data frame. When you subset the original data frame, it is also a good idea to keep track of which subsetting data frame you have been working on. Create names that are easier for you to keep track of, or use in-line comments to remind yourself after you have done the manipulation.

```
Melanoma2 <- Melanoma

#check youself
all.equal(Melanoma2, Melanoma)
```

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

Now let's try to create a data frame ourselves, using the `data.frame()` function.

```
#set.seed(23983)
Newdata <- data.frame("NUM" = 1:6, "Age" = c(21, 15, 18, 35, 50, 16),
                      "Name" = c("John", "Dora", "Emily", "Tonia", "Allison", "Jay"),
                      "Call" = sample(c(0,1), replace=TRUE, size=6))

Newdata
```

```
##   NUM Age    Name Call
## 1   1  21    John    1
## 2   2  15    Dora    0
## 3   3  18   Emily    0
## 4   4  35   Tonia    0
## 5   5  50 Allison    1
## 6   6  16     Jay    0
```

```
class(Newdata)
```

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

Exercise 3.3

Notice how many functions within data.frame() we have talked about, and what do they mean respectively?

Exercise 3.4

What are the variables' names in the `Newdata`? How many observations in this data frame? Can you assess one variable from the data frame, say, the variable `Call`?

Notice that we have talked about using `$` to assess the columns of the data frame. We can also use `[]`, or `[[]]` to do so.

```
Newdata[["Call"]]
```

```
## [1] 1 0 0 0 1 0
```

```
Newdata["Call"]
```

```
##   Call
## 1    1
## 2    0
## 3    0
## 4    0
## 5    1
## 6    0
```

```r
class(Newdata["Call"])
```

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

```r
class(Newdata[["Call"]])
```

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

```r
class(Newdata$Call)
```

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

```r
#Both numerical vector and a data frame with such single vector are computable.
mean(Newdata[["Call"]])
```

```
## [1] 0.3333333
```

```r
mean(Newdata$Call)
```

```
## [1] 0.3333333
```

```r
#Integer is computable as well
class(Newdata$NUM)
```

```
## [1] "integer"
```

```r
sort(Newdata$NUM)
```

```
## [1] 1 2 3 4 5 6
```

```r
mean(Newdata$NUM)
```

```
## [1] 3.5
```

```r
#Notice that "Name" is a factor
class(Newdata$Name)
```

```
## [1] "factor"
```

```r
sort(Newdata$Name)
```

```
## [1] Allison Dora    Emily   Jay     John    Tonia
## Levels: Allison Dora Emily Jay John Tonia
```

```r
mean(Newdata$Name) #return NA
```

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

Exercise 3.5

Create a simple new data frame yourself. Name it as you like.

4. Create new (factor) variables

Let us learn how to create a new factor variable, which is just another name for a categorical variable. In the following example, first, I would like to create a variable `sex` for each person in `Newdata`.

```r
Newdata$sex <- NA
Newdata$sex <- c("Male", "Female", "Female", "Female", "Female", "Male") #Notice I put quotation marks.
#Newdata$sex2 <- c(Male, Female, Female, Female, Female, Male) return error
```

```
class(Newdata$sex)
```

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

It turns out the new variable is a character vector, and so we can use the as.factor() function to turn this vector into a factor variable. A factor variable is easier to work with in functions.

```
#coerce the character variable into a factor variable

Newdata$sex <- as.factor(Newdata$sex)

#check by yourself
class(Newdata$sex)
```

```
## [1] "factor"
```

```
#list all levels of a factor vairble
levels(Newdata$sex)
```

```
## [1] "Female" "Male"
```

What if I don't like the labels "Female" or "Male"? I can use "1" or "0" instead. ("1" = "Male", "0" = "Female").

```
Newdata$sex2 <- ifelse(Newdata$sex=="Male", 1, 0)
class(Newdata$sex2) #numeric
```

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

```
#Demonstrate levels() only applies to factor variable
levels(Newdata$sex2)
```

```
## NULL
```

```
#Note that if you want to coerce numeric variable into factor variable, R may mess up. The trick is to


Newdata$sex2 <- as.character(Newdata$sex2)
Newdata$sex2 <- as.factor(Newdata$sex2)

#What does this tell you?
levels(Newdata$sex2)
```

```
## [1] "0" "1"
```

Exercise 3.6

Create a new factor variable that differentiates the six persons living in urban or not.

Exercise 3.7

Use tapply() function to calculate the mean ages for female and male.

5. Drop and rename variables

What if I create a variable that I would like to drop?

```r
Newdata$NOTWANT <- NA
Newdata <- Newdata[, -7]

#OR
Newdata$NOTWANT <- NA
Newdata$NOTWANT <- NULL
```

```r
#R is case sensitive

#Change the first column's name to Num

colnames(Newdata)[1] <- "Num"
```

Exercise 3.8

Your turn to change the names “sex” and “sex2” to match the case, or to whatever name you like. Hint: use c() function.

Please also convert the factor variable “sex” back to character variable. Hint: Note the name of the variable you just changed.