Basics of Programming in R

Karthik Durvasula

August 28, 2019

1 Preliminaries

You will need to download **R** and **R Studio** for this workshop. **R** is the real programming engine, and by itself is sufficient to do the stuff we will do today. **R Studio** is just a nice "front-end" (IDE = Integrated Development Environment) that allows you to look at everything you need in one "integrated" window. Installation locations:

1. Install R.
2. Install R Studio. You can download FREE version of **RStudio Desktop**.

2 How to print stuff

There are actually a few ways in which you can do this in **R**, but I will discuss only one way that is probably easy to remember. As you become more proficient in R, you can figure out the other ways :).
Note: Anything after "#" on a line is ignored. So, if it very useful for commenting.

```
#Printing something
#Don't forget the quotes, if it is a string.
print("Hello World!")

## [1] "Hello World!"
```

3 Assignment in R

There are at least two ways in which assignment is done in R.

1. The traditional way (the prescriptivists love it)
   x <- 1
2. The other way (the prescriptivists hate it; but I love it)
   x = 1

I will use the second way throughout this pre-workshop. But, this comes down to individual preference. So, feel free to use the other one if you are more used to it. One important thing: be consistent about your choice so that you don't confuse yourself.

Note: For most purposes, both work the same way. You can read about some subtle difference about the scope of operation amongst other things on Stack Overflow.

4 Some useful datatypes in R

4.1 `vectors` (or Arrays)

A `vector` is a collection of similar elements (integers, decimal point numbers, characters, factors, ...). Below I have created different `vectors` with just a single value each.

```
#Integer vector
x = c(4)
print(x)    #Note: No need for quotes if it is a variable or if it is not a string.

## [1] 4

#Character vector
y = c("a")
print(y)

## [1] "a"
```

Now, we can see more complex `vectors` with multiple elements:

```
#Integer vector
x = c(5,6,7,8)
print(x)

## [1] 5 6 7 8

#Character vector
y = c("a","b","c","d")
print(y)

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

4.1.1 Understanding `vector` math (and `vector` operations more generally)

R works off vectors, such that almost all of the functions naturally work with vectors. Let's work with addition and subtraction.

Create two new integer `vectors`:

```
#Integer vectors
x = c(5,6,7,8)
y = c(1,2,3,4)  #Also try, y = c(1:4)
```

Now, try to add the `vectors`:

```
x + y

## [1]  6  8 10 12
```

If the `vectors` are of different lengths, then R throws a "warning". But, it automatically cycles through the shorter `vector` so that they are the same length.

```
#Integer vectors
x = c(5,6,7)
y = c(1,2,3,4)
x + y
```

```
## Warning in x + y: longer object length is not a multiple of shorter object length

## [1]  6  8 10  9
```

4.2 data.frame

A `data.frame` is a collection of `vectors`, where each column is a different `vector`. For example, we can take the above `vectors` and make them a `data.frame`.

```
#Integer vectors
x = c(5,6,7,8)
y = c(1,2,3,4)

#Making a data.frame
data = data.frame(x, y)
print(data)

##   x y
## 1 5 1
## 2 6 2
## 3 7 3
## 4 8 4
```

You can also create the `data.frame` directly, i.e., without creating the `vectors` first:

```
#Making a data.frame directly
#You can put the two vectors on the same line too, but it is nicer to read this way.
data = data.frame(x = c(5,6,7,8),
                  y = c(1,2,3,4))
print(data)

##   x y
## 1 5 1
## 2 6 2
## 3 7 3
## 4 8 4
```

But, note the different `vectors` of the `data.frame` *have* to be of the same length, otherwise you will get an error. Try the following:

```
#Making a data.frame
#Two different types of vectors in the same data.frame
data = data.frame(x = c(5,6,7),
                  y = c(1,2,3,4))

## Error in data.frame(x = c(5, 6, 7), y = c(1, 2, 3, 4)): arguments imply differing
number of rows: 3, 4
```

You can create a `data.frame` with diffent types of `vectors`, as long as all the elements within a `vector` are the same type of element:

```
#Making a data.frame directly
data2 = data.frame(x = c("a","b","c","d"),
                   y = c(1,2,3,4))
print(data2)
```

```
##   x y
## 1 a 1
## 2 b 2
## 3 c 3
## 4 d 4
```

5 Subsetting with data types

By "subsetting", I mean getting a part of the data from that data type.

5.1 Subsetting from a `vector`

For `vectors`, you can mention the position of the element or a range of elements within square brackets to identify the corresponding element.

```
#Making a vector
x = c("a","b","c","d")

#For the 3rd element
x[3]

## [1] "c"

#For all the elements from 3rd-4th positions
#Note, the colon allows you to get a sequence of numbers
x[3:4]

## [1] "c" "d"

#For the elements from a random list of positions
x[c(1,3,4)]

## [1] "a" "c" "d"
```

5.2 Subsetting from a `data.frame`

For a `data.frame`, there are a few ways in which you can subset. I will teach you one way that I think will be useful in the long run. Use the "$" to isolate the column/`vector` in the `data.frame`, and then use the bracket notation to isolate the values of interest.

```
#Making a data.frame
data3 = data.frame(x = c("a","b","c","d","e","f"),
                   y = c(6:11))

#For the 3rd element from the "x" column
data3$x[3]

## [1] c
## Levels: a b c d e f

#For all the elements from 3rd-4th positions from the "x" column
data3$x[3:4]
```

```
## [1] c d
## Levels: a b c d e f

#For the elements from a list of positions from the "x" column
data3$x[c(1,3,4)]

## [1] a c d
## Levels: a b c d e f
```

What if you didn't know the name of the column, but knew the column number?

Advice: Though it is logically possible that you don't know the name of some column, and I show you how to access information in such a case. Honestly speaking, if you don't know the column names, then you should look at your data more carefully. There is no point trying to do data analysis if you don't know what you are analysing. For this reason the "$" notation is going to be almost always the preferred way of referring to columns.

```
#If you don't know the column name
#For the 4th element from the 1st column
data3[4,1]

## [1] d
## Levels: a b c d e f
```

6 Useful programming elements

6.1 Conditionals or `if…else` statements

These are useful if you want to do something only if something else is TRUE/FALSE. For example, print "Hello World", if the 3rd value of a vector is equal to 6.

```
#Creating vector
x = c(4,7,6,1,2,10)

#Basic conditional
#For conditional "equal to", there need to be two "=".
if(x[3] == 6){
  print("Hello World")
}

## [1] "Hello World"

#For "greater than or equal to"
if(x[3] >= 6){
  print("Hello World")
}

## [1] "Hello World"

#For "greater than"
if(x[3] > 6){
  print("Hello World")
}
```

```r
#For "less than or equal to"
if(x[3] <= 6){
  print("Hello World")
}

## [1] "Hello World"

#For "less than or equal to"
if(x[3] < 6){
  print("Hello World")
}
```

Conditionals can also have an "else" part that is done if the conditional is FALSE.

```r
#Creating vector
x = c(4,7,6,1,2,10)

#Basic conditional
# For "equal to", there need to be two "=".
if(x[3] == 7){
  print("Hello, World!")
}else{
  print("Hi, fool!")
}

## [1] "Hi, fool!"
```

6.2 Loops or `for` statements

These are useful if you want to do something multiple times. There are many types of loops in R (`for` loops, `while` loops, `repeat` loops, ...), but we will only work with `for` loops as these are likely to work for most looping issues you might face. `for` loops require a counter of some sort within them, as they execute a set of commands a prespecified number of times. Below, I use the counter or variable "i", but you could use any name/word that you think is clear.

```r
#Basic for loop
for(i in c(1:6)){
  print("Hello")
}

## [1] "Hello"
## [1] "Hello"
## [1] "Hello"
## [1] "Hello"
## [1] "Hello"
## [1] "Hello"

#Using "if statements" within "for loops" affords a lot more power

#Create vector
x = c(1:6)

#Looping thru each value and checking if the value matches "4"
for(i in x){
```

```
  if(i == 4){
    print("Yippie!")
  }else{
    print(":(")
  }
}
```

```
## [1] ":("
## [1] ":("
## [1] ":("
## [1] "Yippie!"
## [1] ":("
## [1] ":("
```

7 Working with pre-collected data using `tidyverse`

The functions below are part of the `dplyr` library, which is loaded when you load `tidyverse`.

7.1 The right packages/libraries need to be installed and used

Packages/libraries are repositories of other functions that might be useful for us. There are literally thousands of such libraries for R. We are going to use a collection of packages called the `tidyverse` - when we install and load this library, we will be able to use the many libraries and functions developed or inspired by the work of **Hadley Wickham**.

The following command installs packages/libraries. [**Note:** the quotes are necessary.]

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

It is not enough to install a library in R; you also have to "load" it in every session you want to use it. When you run the command, you will get a set of lines that look like the following, don't worry, it's just loading the relevant libraries associated with `tidyverse`. [**Note:** No quotes here.]

```
library(tidyverse)
```

```
## Registered S3 methods overwritten by 'ggplot2':
##   method         from
##   [.quosures     rlang
##   c.quosures     rlang
##   print.quosures rlang
## -- Attaching packages ---------------------------------------------------------------------------
tidyverse 1.2.1 --
## v ggplot2 3.1.1     v purrr   0.3.2
## v tibble  2.1.1     v dplyr   0.8.1
## v tidyr   0.8.3     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
## -- Conflicts ------------------------------------------------------------------------------------
tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Advice: Put all the `install.packages` and `library` commands at the beginning of your script. This way, you will know what all libraries are used in your script.

7.2 Reading in data

Advice: You should try to store your data in a **csv** file (comma separated values), if possible. The file format is easy to work with it, and can really be opened by a variety of different programs.

To open a **csv** file that is in a particular directory, you will need to "set the working directory" with the function `setwd()`, and then use the `read_csv()` file to open the data. If you do this correctly, you will see a `data.frame` names **measurements** in your environment list.

Note: There are other functions like `read.csv()` that can be used too. I am just teaching you one of them.

```
#setting the working directory
#Replace content with the relevant directory address
setwd("/.../.../.../")

#Opening the csv file and storing it to a (data.frame) variable
#The variable can be called anything, I am just calling it ''measurements''
measurements = read_csv("Measurements.csv")
```

```
## Parsed with column specification:
## cols(
##   Subject = col_double(),
##   Vowel = col_character(),
##   Speed = col_character(),
##   Consonant1Duration = col_double(),
##   Consonant2Duration = col_double()
## )
```

Above, I encouraged you to use the **csv** file type, but every now-and-then, it is easier to work with other file types (**xlsx, spss data, ...**). In those cases, there are other libraries or functions that you can use. If you want to open an **xlsx** file, then you need to use the following function:

```
#Loading the relevant library (Note: this is installed with the tidyverse,
#but still has to be loaded separately.)
library(readxl)

#Setting the working directory
setwd("/.../.../.../")

#Opening the first sheet in an excel file and storing it to a (data.frame) variable
measurements = read_excel("Measurements.xlsx",sheet=1)
```

7.3 Getting information about your `data.frame`

There are many ways you can view or get information about your `data.frame`

```
#Viewing only the first "n" rows of your data
head(measurements,10)
```

```
## # A tibble: 10 x 5
##    Subject Vowel Speed  Consonant1Duration Consonant2Duration
##      <dbl> <chr> <chr>               <dbl>              <dbl>
##  1       1 ae    FAST                 60.0               52.4
```

```
##  2       1 ae    NORMAL               65.6                64.0
##  3       1 ae    SLOW                 64.6                56.2
##  4       1 o     FAST                 69.8                62.2
##  5       1 o     NORMAL               74.0                69.8
##  6       1 o     SLOW                 76.0                69.4
##  7       3 ae    FAST                 70.2                62.2
##  8       3 ae    NORMAL               70.6                58.2
##  9       3 ae    SLOW                 65.3                59.5
## 10       3 o     FAST                 65.7                67.3
```

```
#The default is 6 rows
head(measurements)
```

```
## # A tibble: 6 x 5
##   Subject Vowel Speed  Consonant1Duration Consonant2Duration
##     <dbl> <chr> <chr>                <dbl>              <dbl>
## 1       1 ae    FAST                  60.0               52.4
## 2       1 ae    NORMAL                65.6               64.0
## 3       1 ae    SLOW                  64.6               56.2
## 4       1 o     FAST                  69.8               62.2
## 5       1 o     NORMAL                74.0               69.8
## 6       1 o     SLOW                  76.0               69.4
```

```
#If you want to see some general information about each column
summary(measurements)
```

```
##     Subject     Vowel              Speed           Consonant1Duration
##  Min.   :1   Length:36          Length:36          Min.   :56.52
##  1st Qu.:3   Class :character   Class :character   1st Qu.:63.07
##  Median :5   Mode  :character   Mode  :character   Median :65.68
##  Mean   :5                                         Mean   :66.34
##  3rd Qu.:7                                         3rd Qu.:69.92
##  Max.   :9                                         Max.   :77.08
##  Consonant2Duration
##  Min.   :52.39
##  1st Qu.:58.94
##  Median :62.09
##  Mean   :62.27
##  3rd Qu.:66.83
##  Max.   :73.50
```

If you want to view **all** the data in a separate tab in **RStudio**, then use `View(name of data.frame)`.

7.4 Subsetting to only some rows of your `data.frame`

Sometimes, you want to remove some of the data because it is not relevant to the analysis you are performing. Let's say that in our dataset, I want to get only the "SLOW" values. Then, we do the following. First, we select the relevant `data.frame`, and then **pipe/chain** a `filter()` function to it with the piping function %>%.

```
#Selects only the "SLOW" values
measurements2 = measurements %>%
  filter(Speed == "SLOW")
```

If you want everything but the "SLOW" values, then:

```
#Selects only the "SLOW" values
measurements2 = measurements %>%
  filter(Speed != "SLOW")
```

7.5 Selecting only some columns of your `data.frame`

Let's say you have a gigantic `data.frame`, with lots of columns, but you are interested only in some columns, then it makes sense to remove everything else for current purposes. It would be a terrible idea to delete it from the original **csv** file, as we might lose the data forever; it is better to do it in **R**, so that the elimination is just temporary.

```
#Selects only the relevant columns
measurements2 = measurements %>%
  select(Subject,Vowel,Speed,Consonant1Duration)
```

If you want to both filter rows and select columns, then use the pipe twice:

```
#Filter and then select only the relevant columns
measurements2 = measurements %>%
  filter(Speed != "SLOW") %>%
  select(Subject,Vowel,Speed,Consonant1Duration)


#If all the selected columns are adjacent to one another,
#you can use the ":" notation
measurements2 = measurements %>%
  filter(Speed != "SLOW") %>%
  select(Subject:Consonant1Duration)
```

7.6 Arranging your data in descending or ascending order according to some column of your `data.frame`

To view the data, sometimes it makes sense to arrange it in descending or ascending order according to some column(s) in your `data.frame`. In which case, we can use `arrange`.

```
#Arranges in ascending order according to the column "Speed"
measurements3 = measurements2 %>%
  arrange(Speed)
head(measurements3)
```

```
## # A tibble: 6 x 4
##   Subject Vowel Speed Consonant1Duration
##     <dbl> <chr> <chr>              <dbl>
## 1       1 ae    FAST                60.0
## 2       1 o     FAST                69.8
## 3       3 ae    FAST                70.2
## 4       3 o     FAST                65.7
## 5       4 ae    FAST                64.2
## 6       4 o     FAST                66.7
```

```
#Arranges in descending order according to the column "Speed"
measurements3 = measurements2 %>%
  arrange(desc(Speed))
head(measurements3)
```

```
## # A tibble: 6 x 4
##   Subject Vowel Speed  Consonant1Duration
##     <dbl> <chr> <chr>                <dbl>
## 1       1 ae    NORMAL                65.6
## 2       1 o     NORMAL                74.0
## 3       3 ae    NORMAL                70.6
## 4       3 o     NORMAL                66.6
## 5       4 ae    NORMAL                67.3
## 6       4 o     NORMAL                77.1
```

```
#You can do more complex arrangements
#descending (alphabetic) order for "Speed", and then ascending order for "Consonant1Duration"
#Note: the order matters
measurements3 = measurements2 %>%
  arrange(desc(Speed),Consonant1Duration)
head(measurements3)
```

```
## # A tibble: 6 x 4
##   Subject Vowel Speed  Consonant1Duration
##     <dbl> <chr> <chr>                <dbl>
## 1       9 ae    NORMAL                56.5
## 2       9 o     NORMAL                57.7
## 3       7 o     NORMAL                61.5
## 4       7 ae    NORMAL                63.6
## 5       6 ae    NORMAL                64.4
## 6       1 ae    NORMAL                65.6
```

7.7 Creating a new column in your `data.frame`

Sometimes, it is useful to create a new column in your `data.frame()`: maybe you want to create some new column, or you want to keep track of some information in the data.

```
#Creates a new column with the same value
measurements3 = measurements2 %>%
  mutate(NewValue = 1)
head(measurements3)
```

```
## # A tibble: 6 x 5
##   Subject Vowel Speed  Consonant1Duration NewValue
##     <dbl> <chr> <chr>                <dbl>    <dbl>
## 1       1 ae    FAST                  60.0        1
## 2       1 ae    NORMAL                65.6        1
## 3       1 o     FAST                  69.8        1
## 4       1 o     NORMAL                74.0        1
## 5       3 ae    FAST                  70.2        1
## 6       3 ae    NORMAL                70.6        1
```

```
#Creates a new column where the value depends on another column.
#In this case, using the function "ifelse()" inside mutate is super useful in the long run.
measurements3 = measurements2 %>%
  mutate(NewValue = ifelse(Consonant1Duration<65, yes="Low", no="High"))
head(measurements3)
```

```
## # A tibble: 6 x 5
```

```
##   Subject Vowel Speed  Consonant1Duration NewValue
##     <dbl> <chr> <chr>               <dbl> <chr>
## 1       1 ae    FAST                 60.0 Low
## 2       1 ae    NORMAL               65.6 High
## 3       1 o     FAST                 69.8 High
## 4       1 o     NORMAL               74.0 High
## 5       3 ae    FAST                 70.2 High
## 6       3 ae    NORMAL               70.6 High
```

7.8 Summarising your `data.frame`

This is extremely useful, if you want to get average values for participants or some combination of column values. It requires the use of two functions `group_by()` and `summarise()/summarize()`.

```
#Summarising the data with the mean value for each participant
measurements3 = measurements2 %>%
  group_by(Subject) %>%
  summarise(MeanSubjectValue = mean(Consonant1Duration))
head(measurements3)

## # A tibble: 6 x 2
##   Subject MeanSubjectValue
##     <dbl>            <dbl>
## 1       1             67.4
## 2       3             68.3
## 3       4             68.8
## 4       6             69.9
## 5       7             61.1
## 6       9             59.3

#Summarising the data with the mean value for each participant for each speed
measurements3 = measurements2 %>%
  group_by(Subject,Speed) %>%
  summarise(MeanSubjectValue = mean(Consonant1Duration))
head(measurements3)

## # A tibble: 6 x 3
## # Groups:   Subject [3]
##   Subject Speed  MeanSubjectValue
##     <dbl> <chr>             <dbl>
## 1       1 FAST               64.9
## 2       1 NORMAL             69.8
## 3       3 FAST               68.0
## 4       3 NORMAL             68.6
## 5       4 FAST               65.4
## 6       4 NORMAL             72.2
```

8 Plotting your data

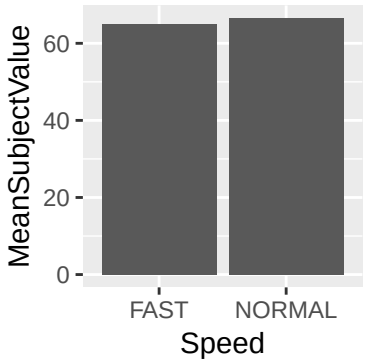
This is where **R** really shines. The plots you get are really pretty, and super scalable (extendable to more complex cases/plots). The functions below are part of the `ggplot2` library, which is loaded when you load

`tidyverse`.

Say, you wanted to get a bar-plot for the mean value of "Consonant1Duration" at each "Speed":

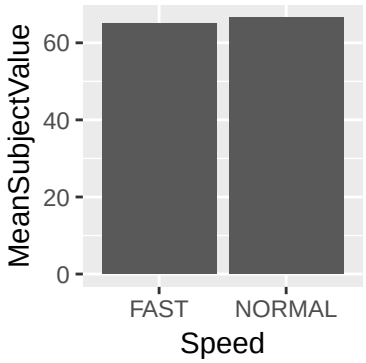
```
#First get the mean value of Consonant1Duration at each speed
measurements3 = measurements2 %>%
  group_by(Speed) %>%
  summarise(MeanSubjectValue = mean(Consonant1Duration))

#Then plot the values
ggplot(measurements3, aes(x=Speed,y=MeanSubjectValue))+
  geom_bar(stat="identity")
```



You can infact combine the data analysis steps and the plotting steps. But, remember, the pipe for data analysis is %>%, and for plotting using `ggplot2` functions is +.

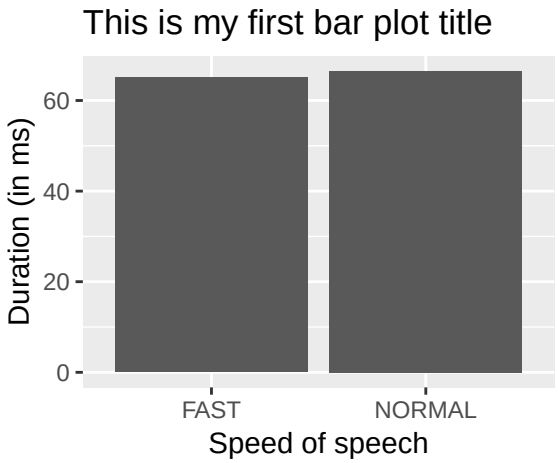
```
#Bar-plot for the mean value of Consonant1Duration at each speed
measurements2 %>%
  group_by(Speed) %>%
  summarise(MeanSubjectValue = mean(Consonant1Duration)) %>%
  ggplot(aes(x=Speed,y=MeanSubjectValue)) +
  geom_bar(stat="identity")
```



What if you want to give the plot a nice title and nice x-axis and y-axis labels?

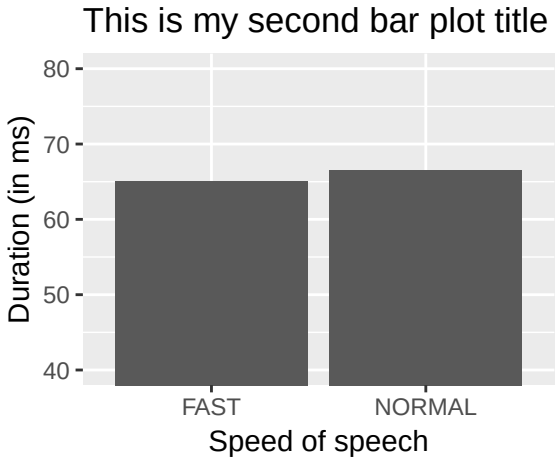
```
#Plotting
ggplot(measurements3, aes(x=Speed,y=MeanSubjectValue))+
  geom_bar(stat="identity") +
  ggtitle("This is my first bar plot title") +
  xlab("Speed of speech") +
```

```
  ylab("Duration (in ms)")
```



What if you want to restrict the range of y-values?

```
#Plotting
ggplot(measurements3, aes(x=Speed,y=MeanSubjectValue))+
  geom_bar(stat="identity") +
  ggtitle("This is my second bar plot title") +
  xlab("Speed of speech") +
  ylab("Duration (in ms)") +
  coord_cartesian(ylim=c(40,80))
```



Ok, you are done with the plotting, and you want to save your plot. You can do it manually using the options in the plotting window, but you could also automate it:

```
#If you want to save the plot as a file, first it has to be saved to a variable.
#Again, the variable can be called anything. I will call it "plot".
plot = ggplot(measurements3, aes(x=Speed,y=MeanSubjectValue))+
  geom_bar(stat="identity") +
  ggtitle("This is my second bar plot title") +
  xlab("Speed of speech") +
  ylab("Duration (in ms)") +
  coord_cartesian(ylim=c(40,80))
```

```
#Now, set the directory where you want to save it
setwd("/.../.../.../")

#Now you can save it
#Note: There are a variety of plot filetypes - png, jpeg, tiff,... I suggest png.
ggsave("Plot.png",plot)

#You can also specify the dimensions of the saved plot
ggsave("Plot.png",plot,width=4,height=4,units="in")
```

```
## Saving 3 x 3 in image
```

9 Some useful references before the RBootcamp

Every MSU student/faculty can get a free DataCamp account from here (http://msudatascience.com/datacamp/). If you need more practice, or want to go over the material at a different pace with an online instructor, then the beginning R course on DataCamp is VERY useful.

A lot of doubts/questions you will have are very likely already answered online. Search for whatever your doubt is and include "Stack Overflow" or "Cross Validated" to your search string on Google. It will most probably show you links to the solutions to the problem/doubt on the online community **Stack Overflow** or **Cross Validated**, which are both fantastic online communities for programming or statistics related info, respectively.

Make sure to read the first four chapters of Stefan Gries's book [**Statistics for Linguistics with R (2nd edition)**]. The first four chapters of the book deal with the basics in statistics and R; they are **required reading** for successful participation in the bootcamp. Links to access the book for free (with an MSU id) are available on the RBootcamp webpage (http://rbootcamp.web.cal.msu.edu/pre-workshop/statistics-for-linguistics-with-r/).