Chapter Six: Data Manipulation

Dr. Simona Simona

4/12/2020

This chapter will address the following topics:

- Reading in data

- Recoding data

- Cleaning data

Most quantitative social science research is based on secondary data as we have already noted in the previous chapter. Secondary data however, is usually messy and disorganised, it does not come to us in a 'ready to be coooked' format. This is because in most cases, these data are not collected for purposes of research. They may be collected for research but certainly not for our research. If we have to use to answer our research questions, we have to do the work of repurposing and that involves transforming and reorganising it so that it can be suitable to answer our research questions.

It is estimated that 80% of the data analyst's time is spent on data manipulation. That is a large chunk of time, but it only illustrates the importance of data manipulation in the data analysis workflow. There are many other terms that refer to data manipulation including data management, data munging, data transformation, janitor work and more recently data wrangling. Data manipulation is the process of getting data from various sources, cleaning and transforming it for visualisation and modelling purposes. In fact, data can only be useful if the data analyst is able to prepare and clean it for substantive modelling work. This process is even much more so in today's world where data availability has expanded exponentially. In this chapter, I introduce facilities available in R programming that help turn messy and unrefined raw data into clear and actionable bits of information. We shall look at some of the most common data manipulation procedures like reading, recoding, subsetting, selecting, merging and saving data. At the end of the chapter we will do a case study of collecting and manipulating the the worldbank data that we are using in this book. We will go through the process of collecting, saving the csv files, reading them in R and working on it until it is in the format we are using it in this book.

Creating data in R

R is cool but probably one of the things it may not be supper good at is entering data directly. You can still do that with small dataset by creating a few vectors and then combining them to form a data frame as we did in **Chapter Three**. For example we have created three vectors x,y,z separately and then combine them to form a data frame named `birth_weight`.

```r
x <- c("Wednesday","Tuesday","Tuesday","Friday","Thursday","Sunday")
y <- c(2.8, 3.5,3.3,2.6,2.9,3.1)
```

```r
z <- c("TRUE","TRUE","FALSE","TRUE","TRUE","FALSE")
```

```r
birth_weight <- data.frame(x,y,z)
names(birth_weight) <- c("day","weight","present")
```

We have replaced variable names from x,y,z to the more intuitive ones of day, indicating the day of the week the birth was given, birth weight for the weight of the baby at birth and present for whether the baby's father was present when the birth was given.

As you can see, this process can be quite laborious when you have large datasets. It is advisable therefore, that if you are conducting a primary study, you may want to consider to use one of the many dedicated spread sheet packages for data entry and then exporting the data to R afterwards to take advantage of it's data manipulation capabilities explored in this chapter.

Reading in the data

As we can imagine, for us to be analyse any type of data, we first have to bring it into R. When you are a new R user, this process may not be that straightfoward. There are different mathods of reading in data in the R environment and these depend on the file format the original data is saved in and the R package that you are using to read the data in. The most common data file formats that you will encounter would be excel files (.xlsx), comma separated files (.csv), text files (.txt), table files (.tab). You will also reading data saved in other programmes such as SPSS (.sav), Stata (.dta), SAS (.sas) among others. There are many R packages that can be used to read data into R. We will be considering an appropriate package to use for each data file.

We can read data in using RStudio by clicking on the `Import Dataset` tab in the **Environment** window. But this may not be the best method because you can not assign it to an object of your choice. The data may come in with long and complicated names that may make it difficult to deal with. We could also us `file.choose()` function to interactively read in the data as we did in **Chapter Three**. For large datasets though, this may be complicated to do. We shall instead use the **working dictory** in this chapter.

The working directory

The working directory is a folder on your computer that contains the files you want to use in your R Studio session. Whenever you open your RStudio, it is pointed to a working directory on your computer. You can check what it is by using the `getwd()` function, for get working directory. This function returns a path to your working directory for the current R session. If the folder is not your desired working directory, you can change using the `setwd()` function and supplying the path to your working directory as an argument. If you want to find the path of your folder on a Windows machine, you right-click your folder and clicking on the **properties** in the drop down menu. Copy the file path under location and remember to change back slashes into forward slashes compatible with R. For Mac, click on the folder and then **Get info**. The path is under `Where` on the left hand side.

Creating a path to your folder can sometimes cause unnecessary pain. You can avoid that by going to the **Session** tab on the menu in RStudio $\Longrightarrow$ **Set Working Directory** $\Longrightarrow$ **Choose Working Directory**. To avoid repeating the same thing everytime you have a new session, you can copy the working directory and put it in the `setwd()` function accordingly. Whichever method you use, make sure you don't include your file name when you are setting the working directory. That is likely to be mistake you make here.

The data we will use in this chapter is in the `Data` folder and we set the working directory accordingly.

```r
setwd("/Users/simonajsimona/Documents/Quantitative/Data")
```

Reading in Text Files

After we have set the working directory, we can now work with the files that are in it. We will start by reading in text files. These are quite common files because almost any data application has capability to export data to text file formats. Commonly text files are in comma delimited values fomart with `.csv` and `.txt` file extensions. We can read in data using functions from base R or the `readr` package. In base R we use the function `read.csv()`. Supply the file name with a `.csv` in the brackets inclosed in quatation marks. We will read in `world_data.csv` file and put it in the `data` object.

```r
data <- read.csv("world_data.csv")
```

This will be sufficient most of the time but there are many other arguments you can specify depending on the nature of your data. By default `read.csv()` function will have `header = TRUE`, `sep` = ",” and `StringsAsfactors = TRUE`. It also replaces white spaces in the variable names by a "." operator. The `header = TRUE` arguemnt tells R to treat the first row of the original dataset as variable names. `sep = ""`, tells R that the fields or cells in the original dataset are separated by a comma. `stringAsFactors = TRUE` to treat any strings as factor variables. If the default settings are not appropriate for you or if you want to check what other arguments are available to include, you can typing `?read.csv` in your `R script` window or the `console`. If you have read the file in correctly, you must see the `data` object appearing the `Environment` located in the top right-hand window in RStudio.

The `.text` extension files are tab delimited, which means that the data fields are separated by tabs or white space basically. To read them in we use the `read.table()` function. Importantly, the default for this function is `sep = ""`. We will read in the text version World Bank data saved as `world_data.txt` from the same working directory. Again, after successfully reading in the data, you should see the `datawb` object appearing in the **Environment** window in RStudio.

```r
datawb <- read.table("world_data.txt", header = TRUE, sep = "\t")
```

Reading in data using the `readr` package is not very from different from base R other than the fact that you are using specific functions from an external package. Like all other packages, you first need to install it using the `install.packages()` function and the load it using the `library()`function. But since `readr` is part of the `tidyverse` echo system of packages which we will use throughout the book, I would always encourage you to install and load the `tidyverse` package instead, because it contains all the tidyverse group of packages.

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

The functions to use for text files are `read_csv()` and `read_table()` for `.csv` and `.tab` file extensions respectively.

```r
data <- read_csv("world_data.csv")
datawb <- read_table("world_data.tab")
```

There are a few reason why you would want to use `readr` instead of base R especially if you are dealing with large datasets. `readr` function are around 10x faster than the equivalent base R functions. `readr` functions read in column names as they are in the original dataset i.e they do not coerce white spaces into a "." as base R does. The stringAsFactor is set to FALSE by default. Furthermore, `readr` functions offer more options as arguments for adjusting your data as it is being read into R. You can check the details by requesting help `?read_csv`. Please note that for external packages, this functionality can only work if you have already loaded your package in the current R session.

Reading in Excel files

Excel is probably the most popular spreadsheet there is. As such, it is important that there is functionality in R to interact with it so that data can be easily imported into R. In most cases researchers change the format to csv in order to read the data into R using the `read.csv()` function. This may not be too efficient. There are many packages that can be used to import excel files into R including `xlsx`, `gdata` and `RExcel`. But we will use the `readxl` package in this book. It is part of the tidyverse suite of packages I introduced earlier. In addition to being faster than other excel packages, `readxl` can read in both `.xls` and more modern `.xlsx` files. Furthermore, it doesn't have any external dependencies, which means you can use it to read data across platforms. The function we use is `readxl_excel()`. Again as always, we neeed to first install and load the package before we are able to use them. In this case, we will use the tidyverse package again as we did ealier to avoid repeating this process over and over. Like in all other packages, if you have already installed the `readxl` package, you don't have to do it again. You can load the function with library(tidyverse) just to be sure.

```r
install.packages("tidyverse")
library(readxl)
```

We will use the excel version of our `world_data.xlsx` dataset. Note that `readxl_excel()`, recognises that excel has mulple sheets where data can be stored. If your data is not in the first sheet (which is the default detting), you can specify the sheet name or number where your data is stored.

```r
dataxl <- read_excel("world_data.xlsx")
```

The data is stored in the `dataxl` object and you should be able to see it in your RStudion `Environment` window if you have it correctly. If your data starts at a particular row in the original dataset, you can skip some rows using the `skip` argument which allows you to read in data beginning at a particular row number. For example if your data starts in row 8, you can include `skip = 8` argument. You can use functions for specific file extensions directly you know them. You can use `read.xls()` function for `.xls` extensions and `read.xlsx()` functions for `.xlsx` functions. All details can be gotten from the `?readxl` functionality.

Read in data from sofware programmes

There are many programmes that you may want to interact if want to be an effective quantitative methods researcher and data scientist. R allows you to import data from R itself and several other programmes. We will start by reading in data saved in `.Rda` fomart, which is R itself. You don't require any package for this. As long as you are in the right directory, you can use the `load()` function to read in the data. We will read in the `.Rda` version of the African barometer data.

```r
load("data.Rda")
```

For the `load()` function to work, the `.Rda` data should be saved in your working directory. Also, note that you have to know the file name it is saved in. The file nam might be different from data object name that will appear in the work space under the `Environment` window. We shall revist this when we look at writing data in R.

Other programmes that you will be interested in include Stata, SPSS and SAS. I will also include reading in data from JASON files here because we shall encounter data from the web in this book. For Stata, SPSS and SAS, the best package you may use is `haven`. Of course there are other packages like `foreign` that we used read in data from other programmes as we did in **Chapter Three**. However, you may encounter a few problems especially with new versions of the programmes. We will first install the `haven` package and load it in our session as usual.

```r
install.packages("haven")
```

To read in data from Stata, we use the World Bank data for 2015 named `data_2015.dta`. The function we use is `read_dta()` and we will put it in an object `data2015`. The mistake you are likely to make here is to leave out the file extension and to forget to put your data file in invented commas. Again if you have done it correctly you should see your data frame object in the `Environment window`.

```r
data2015 <- read_dta("data_2015.dta")
```

To import data sved in an SPSS format, we use the `read_sav()` and for SAS we use the `read_sas()` function. Everything else is the same. Make sure you specify your file name as it appears in your working directory and not forgeting the appropriate file extenstion. If you need to make more adjustment as you read in the data, you can call for further documentation using the `?` operator before your function.

JSON is short for JavaScript Object Notation. It is a data-interchage language that is readable by human beings. We will discuss more about it in **Chapter Eighteen** where we deal with data from the web. But here we will just show how to read in JSON files. We use the `fromJSON()` function from the `rjson` package.

```r
install.packages("rjson")
library(rjson)
```

We are reading in data from the data.json file in the working directory. We have put it in the object json_data. Again, if you have read it properly, you should see it in the work space.

```r
json_data <- fromJSON(file = "data.json")
```

Working with data from database management systems

As the world is becoming more and more sophisticated, the statistical programming languages gives more flexibility to work with big data which sometimes will be stored in databases. There are many examples of database management systems that R is able to interact with, but we will only ficus on those supporting the structural querry language (SQL) and these include MySQL, PostgreSQL and SQLite. I give links to the documentation of these systems where you can find further information about how R can interact with each of these systems:

MySQL: https://cran.r-project.org/web/packages/RMySQL/index.html
PostgreSQL: https://cran.r-project.org/web/packages/RPostgreSQL/index.html
SQLite: https://cran.r-project.org/web/packages/RSQLite/index.html

Subsetting data

Now that we have finished learning the importation process, the next thing is to read in the actual real world data that we are going to use for this chapter and a few others in this book. I have already stated that we are using the Afrobarometer dataset in this chapter saved as `africa.csv` in our working directory. Afrobarometer data measuring attitudes towards the governance systems in Africa. We read it in using the `read.csv()` function and assign it to the `afro_data` object accordingly.

```r
afro_data <- read.csv("Data/africa.csv")
```

It should indicate in your **Environment** window that we have 53935 observations and 364 variables. This dataset is significantly large. It contains data from 36 African countries[1]. I wanted to start with the whole dataset just like it is downloaded from the website so that we can see how real world data looks like and learn how we can select specific rows and columns from it for purposes of our analysis. The first thing is to take a look at the variable names and we use the `names()` function for that.

```r
names(afro_data)
```

Examining data

After you have read in the data, the next task is to examine it and see if it makes sense before making any decisions about how to transform, manipulate or manage it. The first thing you want to check is if it contains the number of observation and variables that you are expecting. This is also a way of knowing whether you have read the data in correctly. R can be confusing sometimes, you can read the data without error and yet you may find the data is not appearing as expected in the work space. You can know this by using the `dim()` function for dimension and passing the data frame object in the brackets. This should show you the number of observation and variables you have in your `Environment` window.

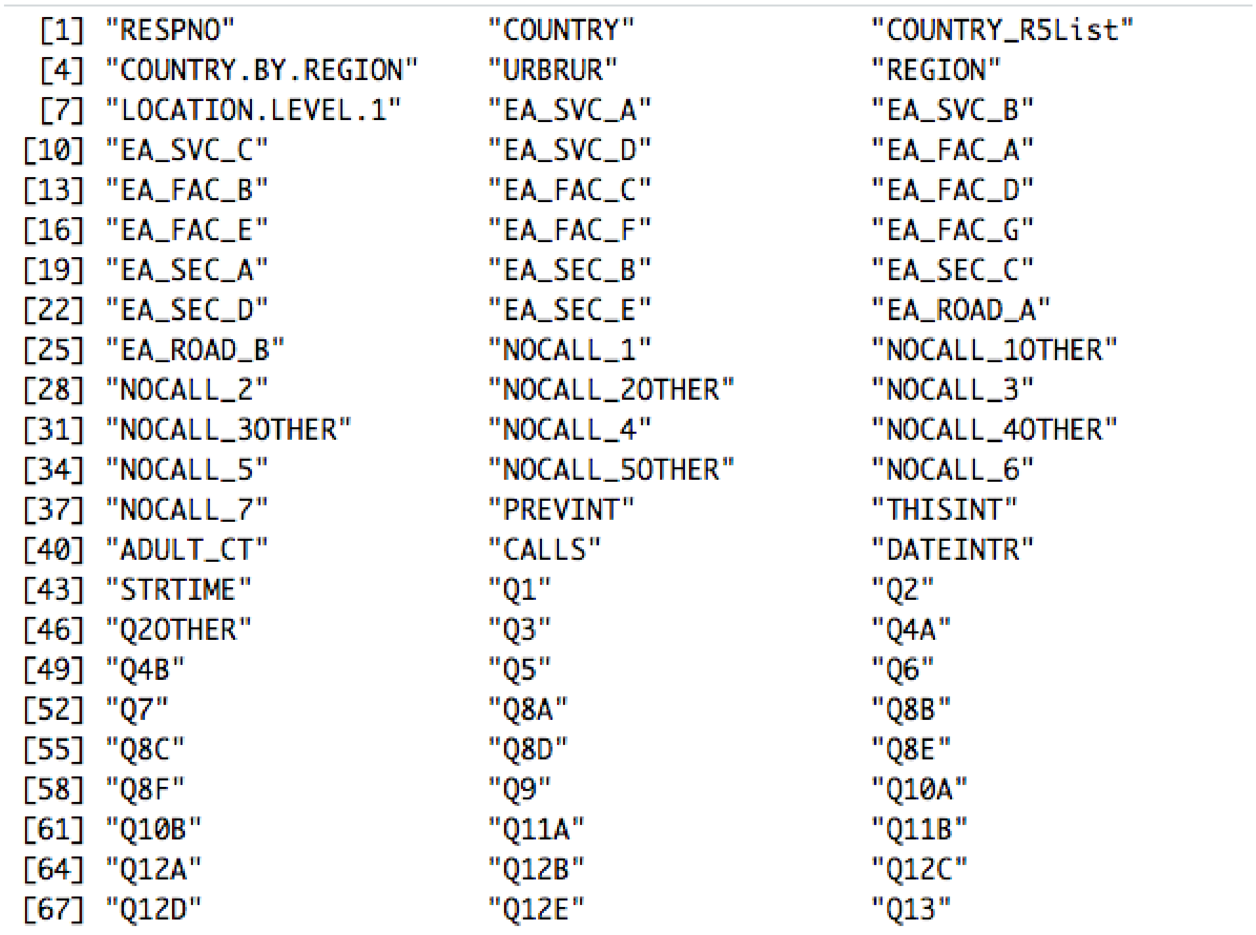
```r
dim(afro_data)
```

```
## [1] 53935   365
```

Using the `dim()` function on the `afro_data` data frame gives us **** observations and 364 variables. You will notice that this function actually gives us what we are aready seeing in the `Environment`. But it is still important because we can use indexing (i discuss this below) and check specific rows or colums in the data frame.

The next this we might want to do is to look at the whole dataset just like we are used to in terms of the spread sheet. We use `View()` function for that. Take note that this function contains the upper case `V`.

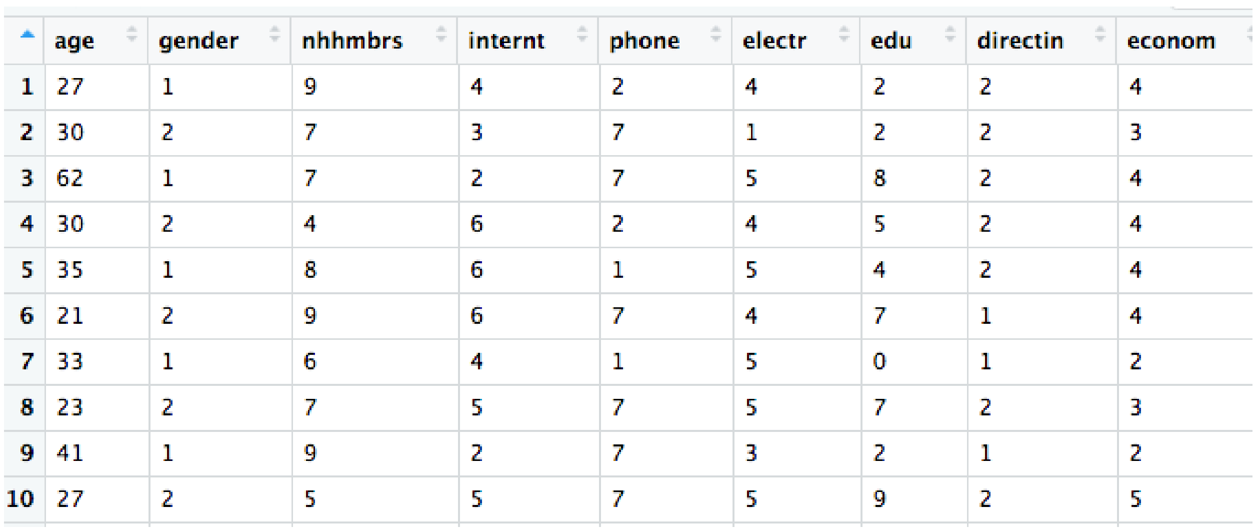
[1] 1=Algeria, 2=Benin, 3=Botswana, 4=Burkina Faso, 5=Burundi, 6=Cameroon, 7=Cape Verde, 8=Cote d'Ivoire, 9=Egypt, 10=Gabon, 11=Ghana, 12=Guinea, 13=Kenya, 14=Lesotho, 15=Liberia, 16=Madagascar, 17=Malawi, 18=Mali, 19=Mauritius, 20=Morocco, 21=Mozambique, 22=Namibia, 23=Niger, 24=Nigeria, 25=São Tomé and Príncipe, 26=Senegal, 27=Sierra Leone, 28=South Africa, 29=Sudan, 30=Swaziland, 31=Tanzania, 32=Tog, 33=Tunisia, 34= Uganda, 35=Zambia, 36=Zimbabwe



```
 [1] "RESPNO"            "COUNTRY"           "COUNTRY_R5List"
 [4] "COUNTRY.BY.REGION" "URBRUR"            "REGION"
 [7] "LOCATION.LEVEL.1"  "EA_SVC_A"          "EA_SVC_B"
[10] "EA_SVC_C"          "EA_SVC_D"          "EA_FAC_A"
[13] "EA_FAC_B"          "EA_FAC_C"          "EA_FAC_D"
[16] "EA_FAC_E"          "EA_FAC_F"          "EA_FAC_G"
[19] "EA_SEC_A"          "EA_SEC_B"          "EA_SEC_C"
[22] "EA_SEC_D"          "EA_SEC_E"          "EA_ROAD_A"
[25] "EA_ROAD_B"         "NOCALL_1"          "NOCALL_1OTHER"
[28] "NOCALL_2"          "NOCALL_2OTHER"     "NOCALL_3"
[31] "NOCALL_3OTHER"     "NOCALL_4"          "NOCALL_4OTHER"
[34] "NOCALL_5"          "NOCALL_5OTHER"     "NOCALL_6"
[37] "NOCALL_7"          "PREVINT"           "THISINT"
[40] "ADULT_CT"          "CALLS"             "DATEINTR"
[43] "STRTIME"           "Q1"                "Q2"
[46] "Q2OTHER"           "Q3"                "Q4A"
[49] "Q4B"               "Q5"                "Q6"
[52] "Q7"                "Q8A"               "Q8B"
[55] "Q8C"               "Q8D"               "Q8E"
[58] "Q8F"               "Q9"                "Q10A"
[61] "Q10B"              "Q11A"              "Q11B"
[64] "Q12A"              "Q12B"              "Q12C"
[67] "Q12D"              "Q12E"              "Q13"
```

Figure 1: View data data using the View() function

```
View(afro_data)
```



	age	gender	nhhmbrs	internt	phone	electr	edu	directin	econom
1	27	1	9	4	2	4	2	2	4
2	30	2	7	3	7	1	2	2	3
3	62	1	7	2	7	5	8	2	4
4	30	2	4	6	2	4	5	2	4
5	35	1	8	6	1	5	4	2	4
6	21	2	9	6	7	4	7	1	4
7	33	1	6	4	1	5	0	1	2
8	23	2	7	5	7	5	7	2	3
9	41	1	9	2	7	3	2	1	2
10	27	2	5	5	7	5	9	2	5

Figure 2: View data data using the View() function

The `names()` function gives us all the names of variables in our data frame. When we apply this function on our example data frame, we get all the variable names in that data frame.

```
names(afro_data)
```

We use the `str()` to get an overall picture of the structure of the data in terms of the dimensions, variable names and data types. In our `afro_data` data, we have the same dimensions (53935 obs and 27 variables) and variable names we have alread met above. But here we have an additional component of data type. The `int` in the output shows that all the data has the integer class. Which means the data points are represented by numbers. This is likely to change when we start manipulating our data and transforming it accordingly.

```
str(afro_data)
```

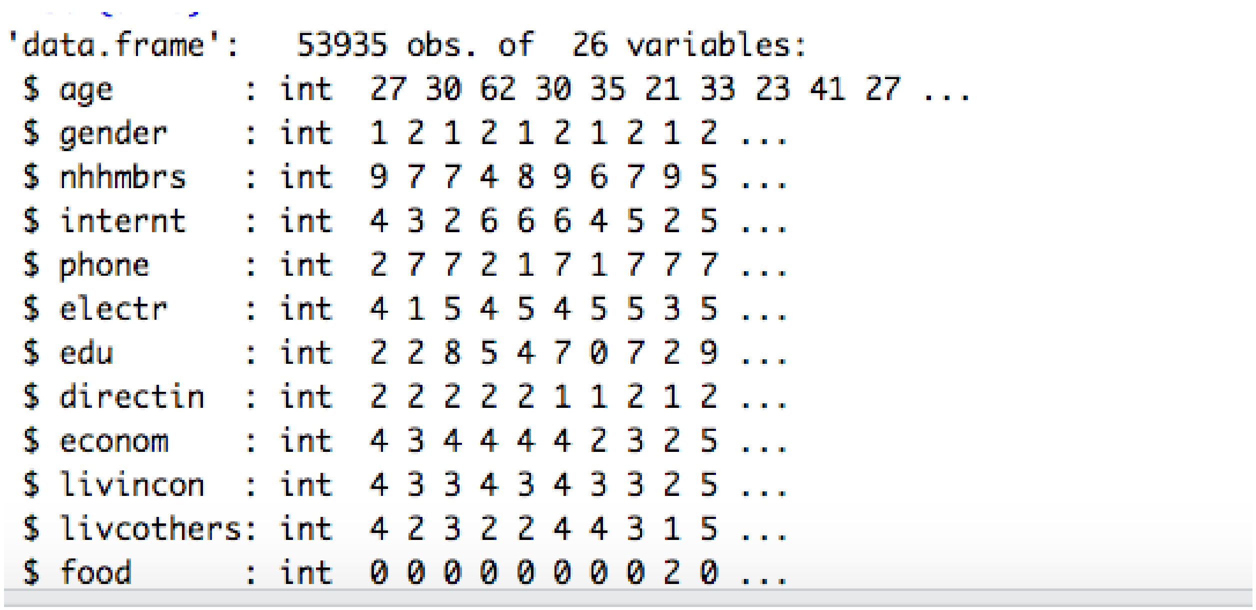
We can also get a glimse of the dataset by getting the first six rows of our data. For this, we use the `head()` and for the last 6 rows we use the `tail()` function. These functions are important especilly when you are working large dataset.

```
head(afro_data)
```

```
tail(afro_data)
```

Missing values

Mising values are notorious in any kind of analysis ie you will always find them. The problem with missing values is that they can distort the representativeness of the sample and subsequently bias parameter estimates. There are different types of missing values, some more dangerous than others. Dealing with missing values depends on the nature of missingness and as such, we will briefly the



```
'data.frame':   53935 obs. of  26 variables:
 $ age       : int  27 30 62 30 35 21 33 23 41 27 ...
 $ gender    : int  1 2 1 2 1 2 1 2 1 2 ...
 $ nhhmbrs   : int  9 7 7 4 8 9 6 7 9 5 ...
 $ interrnt  : int  4 3 2 6 6 6 4 5 2 5 ...
 $ phone     : int  2 7 7 2 1 7 1 7 7 7 ...
 $ electr    : int  4 1 5 4 5 4 5 5 3 5 ...
 $ edu       : int  2 2 8 5 4 7 0 7 2 9 ...
 $ directin  : int  2 2 2 2 2 1 1 2 1 2 ...
 $ econom    : int  4 3 4 4 4 4 2 3 2 5 ...
 $ livincon  : int  4 3 3 4 3 4 3 3 2 5 ...
 $ livcothers: int  4 2 3 2 2 4 4 3 1 5 ...
 $ food      : int  0 0 0 0 0 0 0 0 2 0 ...
```

Figure 3: Using the str() function

three types of missingness before we get to dealing with missing values in R[2].

There are basically three types of missing data including missing completely at random (MCAR), missing at random (MAR) and missing not at random (MNAR). MCAR occurs when the probability of missingness is the same for all cases. This means that the reasons for missingness are completely unrelated to the values of the other variables or the non-missing data elements in the variable for the missing data (Harrell Jr, 2015; Gelman and Hill, 2007). This includes cases where a respondent would omit a response to a question for reasons unrelated to a response she would have given or to any of her characteristics. Thus, MCAR does not apply when people from certain demographic elect not to answer a certain question because then, missingness will be correlated with that demographic characteristics. Where MCAR applies, deleting missing values does not bias parameter estimates (Gelman and Hill, 2007).

MAR is said to be a more general, realistic and a much broader class than MCAR. In this case, the probability that a value is missing depends on values of variables that were actually observed (Harrell Jr, 2015). In other words, given the values of other available variables, respondents having missing values are only randomly different from other respondents. The key element about MAR is that the values of the missing data can somehow be predicted from some of the other variables being studied and this is why MAR and MCAR are often called ignorable non-responses (Harrell Jr, 2015; Gelman and Hill, 2007)

MNAR is the most problematic of all because it means that the probability of missingness varies for reasons that are unknown. The tendency for values to be missing in this case are a function of data that are not recorded, and this missing information also predicts the missing values. MNAR occurs for instance, when respondents from a lower income bracket of the sample are less likely to answer a question on income. This kind of missingness is no longer at random and it is also often called nonignorable non-response. (Harrell Jr, 2015).

[2]The subject of missing values is very broad and will not do justice to it here. We are just looking and the different types, and signposting how we may deal with each type

With ignorable non-responses, and the data doesn't reduce significantly

Variable classes

In the previous chapter we studied levels of measurements and we saw that variables are measured at the nominal, ordinal and interval/ratio levels. We also stated that variables measured a the nominal and ordinal levels can called categorical variables while those measured at the interval and ration levels are called continuous variables. In R, categorical variables are classified as **factors** for nominal variables and **ordered factors** for ordinal level variables. Continuous variables are classified as **numeric** variables. We are going to add another type of variable here which we haven't met before and that is a **character** string. Characters are basicallys strings of words and sentences which are common in text files. Before doing any analysis, you have to ensure that your variables reflect correct variable types. Otherwise this can significantly complicate your data visualisation and anlysis.

In addition to the `str()` which we use to check the nature of variables in the dataset, we can use the `class()` function to check types of one variable in the dataset. After checking the type of variable we can change it if it is not recorded as expected in R. Let's use our Kenya barometer dataset to check the `gender`, `freedom` and `age` variables and see if we would need to change them.

```
class(kenya_data$gender)
```

```
## [1] "haven_labelled"
```

```
class(kenya_data$freedom)
```

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

```
class(kenya_data$age)
```

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

We can see that in all of our variables, `gender` is the only one that has come out as desirable (factor). The`freedom` variable is constructed out of a question that asked people whether in their country they are free to say whatever they want. The responses given are categorised into 4 levels of `not at all free`, `not very free`, `somewhat free`, and `completely free`. Since these categories have an order to them, we would want them to be ordered factors. We want the `age` variable to be numeric. We change the variable types using the `as.numeric()` function if we want to change in a numeric variable and `as.factor()` function if we want to change into a factor.

```
kenya_data$freedom2 <- factor(kenya_data$freedom, ordered = TRUE)
class(kenya_data$freedom2)
```

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

I have added a few things in the above code. First of all, I have created a new variable `freedom2` based on the original `freedom` variable. It is always important to create new variables whenever you want to transform the original variable so that you can leave the original dataset intact. Secondly, I have used the argument `ordered = TRUE` because i want the new variable to be an ordered factor corresponding to the nature of the original variable. This argument wouldn't be necessary if we has an ordinal level variable. You may not need to do this because sometimes ordered factors give problems in regression analysis, you should be fine by leaving them as a factor varables if you encounter problems.

```
kenya_data$age <- as.numeric(kenya_data$age)
class(kenya_data$age)
```

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

The `age` variable is pretty straightforward, we have just changed from being a character variable to a numeric variable. If we wanted to change a to a character variable we would use `as.character()` function accordingly.

Recoding variables

Recoding of variables is a very important data management strategy. R rarely brings in data with it' labels, you will want to add the labels once your data has been read in R. Additionally, you may want to collapse numeric variables into factor variables. It is important to note though that reducing variables is not very desirable because by doing so, we lose richness of our data. However, there are times when reducing the values of a numeric variable makes practical sense. For example, it might be useful for us to collapse continuous variables if we want to tabulate the results to make more readable and easy to interprate. R allows us to do that. Furthermore, we can manipulate factor variables to reduce their categories. Before doing any recoding, however, you will need to to check the frequency distribution of the concerned variable using the `table()` function. We will look at the `age` variable and see if we can reduce it to a few categories.

```
table(kenya_data$age)
```

```
##
##   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20
##   6  38  61  53  77  77  53  85  88  92  94 105  86  93  73 106  55  59  91  49
##  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40
##  57  71  45  74  50  57  41  27  41  28  36  24  18  32  19  34  15  22  18  23
##  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60
##  19  18   9  26  13  16  10  12  14   8   3   9   4  11   7   2   2   3   6   3
##  61  62  63  64  65  66  67  68  69  70  71  72
##   5   5   2   4   2   2   1   2   2   1   1   2
```

We can see that the `age` variable has got values from 1 up 72. To collupse the data into categories, we use the `cut()` function. The function divides the variable into a number of categories specificied by a single number supplied as an argument giving the number of intervals into which the variable is to be cut. We will first cut the `age` variable into 4 equal categories by supplying single number as an argument and saving the results into a new variable called `age3`.

```
kenya_data$age3 <- cut(kenya_data$age, 4)
table(kenya_data$age3)
```

```
##
## (0.929,18.8]  (18.8,36.5]  (36.5,54.2]  (54.2,72.1]
##         1301          794          250           52
```

The `cut()` ensures that all the values are allocated an interval that's why the intervals begin with the lower bound of 0.929 in the first interval and end with the upper bound of 72.1. Note that we didn't have any control in what the interval will be. We just let R divide the values in to 4 equal intervals. We can in fact have control over the bounds of our intervals in case we want the bounds to retain some kind of meaning. For example, we might want to devide `age` into chidren (1 to 16), youths (17 to 35) and middle age (36 to 72). We can do that by supplying a `breaks` argument in the `cut()` function.

```
kenya_data$age4 <- cut(kenya_data$age, breaks = c(0.5, 16, 35, 72))
table(kenya_data$age4)
```

```
##
## (0.5,16]  (16,35]  (35,72]
##     1187      874      336
```

We have created a new variable `age4` breaking the original `age` variable. Notice that we don't start with the first value number 1, which is the first value item in the variable `age` but instead, we start with 0.5[3] but we end with number 72, which is the upper bound value of the variable. This is because the `breaks` argument does not include the lower bound value into the category but the upper bound value is included.

We can also add labels of `children`, `youths` and `adults` to our intervals so that they can be more meaningful. We will create a new variable `age5` from the original `age` variable and supply our labels with the argument `labels`.

```
kenya_data$age5 <- cut(kenya_data$age, breaks = c(0.5, 16, 35, 72), labels = c("children", "you
table(kenya_data$age5)
```

```
##
## children   youths   adults
##     1187      874      336
```

After adding labels it has become much easier for us to understand the information we are getting. There are children, youths and adults in our datatset.

To reduce the number of categories, we need to recode our variables by combining some values according to what we want to achieve. For example, lets look at the distribution of the education attainment variable `edu` below using the `table()` function. We have added the argument `exclude = NULL` so that we include all categories including missing values.

```
table(kenya_data$edu,exclude = TRUE)
```

```
##
##   0   1   2   3   4   5   6   7   8   9  99
## 157  43 430 439 307 566 324  38  73  19   1
```

We can see that the education variable in our dataset has several values. The codebook gives us what the values we are seeing represent and they are 0 = no formal schooling, 1 = informal schooling only (including Koranic schooling), 2 = some primary schooling, 3 = primary school completed, 4=Intermediate school or some secondary school / high school, 5 = secondary school / high school completed , 6 = post-secondary qualifications, other than university e.g. a diploma or degree from a polytechnic or college, 7 = some university, 8 = university completed, 9 = post-graduate, 99 = don't know, 98 = refused to answer, -1 = missing.

[3]We could have started with any value that is less than 1

Some categories however, may not be too useful. We may do well by condensing this variable so that it can retain only a few meaninful categories. There are many options we can choose in recoding the variable depending on the nature of our study. Here, we don't want to complicate our lives, we will recode it into only three categories of `no education`, which combines 0 and 1, `primary education`, which groups 2 and 3 and `secondary or higher education` which groups categories between 4 and 9. We don't have missing values within this variable, we will make 99 missing values NA.

There are many functions we can use for recoding variables, but we will use the `Recode()` function from the `car` package, short for **companion to applied regression**. Note that we use the uppercase R for this function. The lower case `recode()` function also works but sometimes it gives a few problems because there is another package that uses it. As usual, we need to first install the package and then load it.

```r
install.packages("car")
```

```r
library(car)
```

```
## Loading required package: carData

##
## Attaching package: 'car'

## The following object is masked from 'package:dplyr':
##
##     recode

## The following object is masked from 'package:purrr':
##
##     some
```

```r
kenya_data$edu2 <- Recode(kenya_data$edu, "0:1 = 0; 2:3 = 1; 4:9 =2; 99 = NA")
table(kenya_data$edu2)
```

```
##
##    0    1    2
##  200  869 1327
```

Now we need to label our categories and we can do easily using the base R function `levels()`. But this function can only work on factor variables and we need to check that first using `class()` function and change it if it is not a factor variable.

```r
class(kenya_data$edu2)
```

```
## [1] "haven_labelled"
```

As we can see, `have_labelled` which is certainly not a factor class. We would have to change it using `as.factor()` function to change the variable type. After doing that, we can use the `levels()`function to assign labels to the `edu2` variable.

```r
kenya_data$edu2 <- as.factor(kenya_data$edu2)
class(kenya_data$edu2)
```

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

```r
levels(kenya_data$edu2) <- c("no education", "primary", "secondary or higher")
table(kenya_data$edu2)
```

```
##
##        no education             primary secondary or higher
##                 200                 869                1327
```

Now we can interpret the variable by saying there are 200 people with no education, 869 people with primary education and 1327 people with secondary education or higher.

Merging data

Saving data