Getting Started with R

Econ 440 - Introduction to Econometrics

Patrick Toche

20 February 2022

Contents

Introduction

This document is work in progress. It will be updated as we proceed through the course, in response to questions that arise for particular tasks. It is designed to be permanently incomplete and permanently revised. Also, please use with care: Always back up your data!

Know your data types

We are going to be dealing with the following data types: numbers, strings, booleans, factors, and containers filled with these, including: vectors/lists, named lists, matrices, dataframes, tibbles.

Start a new session, type the letter a into the console: you get an error!

```
a
```

Now type the letter c into the console: you get something!

```
c
#> function (...)  .Primitive("c")
```

That's because in R the letter `c` is the name of a function `c()`. That's unfortunate, but R was designed in the 1970s when very few people had a need of symbols beyond a and b. Lol.

Know your data types

Other than `c()`, other names that are used by R, and are therefore best avoided as names for other objects, include `df` and `data`. But because they are names of functions, these names are actually still available for other objects. For instance,

```
# define c
c = 0

# check the value of c
c
#> [1] 0

# but c() is still available
c(0,1)
#> [1] 0 1
```

It's not too smart, but we do it all the time. Better people use C or DF, because R is case sensitive and the upper-case symbols are available.

Know your data types

The following is definitely stupid:

```r
# define c
c = function(x) NULL

# check c(0,1)
c()
#> NULL

# remove this before it ruins our life
rm(c)
```

Know your data types

So let's investigate some data types:

```r
# floating-point number aka num
a1 = 0
str(a1)
#>  num 0

# string aka character chr
a2 = "0"
str(a2)
#>  chr "0"

# boolean aka logical
a3 = TRUE
str(a3)
#>  logi TRUE
```

In R booleans are called `logical` vectors.

Know your data types

```r
# integer aka int
a4 = 0L
str(a4)
#>  int 0

# factor
a5 = factor(0, level=1)
str(a5)
#>  Factor w/ 1 level "1": NA

# vector of factors
a6 = factor(c(0,1), levels=c(1,2))
str(a6)
#>  Factor w/ 2 levels "1","2": NA 1
```

Know your data containers

We will be dealing mostly with ‘vectors’ and ‘dataframes’ In R lists are vectors. A vector is the most fundamental data container in R. Even a single object is under the hood a vector with a single value.

```r
# a vector/list of numbers:
l1 = c(0, 1)
str(l1)
#>  num [1:2] 0 1

# a vector/list of integers:
l2 = c(0, 1L)
str(l2)
#>  num [1:2] 0 1

# a vector/list of booleans:
l3 = c(FALSE, TRUE)
str(l3)
#>  logi [1:2] FALSE TRUE
```

Know your data containers

In R named lists have a purpose similar to ‘dictionaries’ in `Python`.

```
# a named list:
l4 = c("name1" = 1, "name2" = TRUE)
str(l4)
#>  Named num [1:2] 1 1
#>  - attr(*, "names")= chr [1:2] "name1" "name2"
```

Every object stored in a named list has a name and a value:

```
# get the name and value with []
l4["name1"]
#> name1
#>     1

# get the value with [[]]
l4[["name1"]]
#> [1] 1
```

Know your data containers

The easiest way to create a dataframe is from several vectors.

```
# define band member, the date they joined/quit the band, and their net worth
# data from a quick web search - not meant to be accurate!
member <- c('Brian', 'Mick', 'Keith', 'Stewart', 'Bill', 'Charlie', 'Ronnie')
start <- as.Date(c('1962-07-12', '1962-07-12','1962-07-12','1962-07-12',
                   '1962-12-07','1963-02-02','1976-04-23'))
today <- as.character(Sys.Date())
end <- as.Date(c('1969-06-09', today, today,
                 '1963-05-01', '1992-12-01', '2021-08-24', today))
worth <- c(10, 500, 500, 1, 80, 250, 200)
df <- rolling.stones <- data.frame(member, start, end, worth)
str(rolling.stones)
#> 'data.frame':    7 obs. of  4 variables:
#>  $ member: chr  "Brian" "Mick" "Keith" "Stewart" ...
#>  $ start : Date, format: "1962-07-12" "1962-07-12" ...
#>  $ end   : Date, format: "1969-06-09" "2022-02-20" ...
#>  $ worth : num  10 500 500 1 80 250 200

# add information about the units:
rolling.stones$currency <- "$"  # R is clever that way
```

Know your data containers

We can output the core structure of a dataframe with the `dput()` function.

```
dput(df)
#> structure(list(member = c("Brian", "Mick", "Keith", "Stewart",
#> "Bill", "Charlie", "Ronnie"), start = structure(c(-2730, -2730,
#> -2730, -2730, -2582, -2525, 2304), class = "Date"), end = structure(c(-206,
#> 19043, 19043, -2437, 8370, 18863, 19043), class = "Date"), worth = c(10,
#> 500, 500, 1, 80, 250, 200)), class = "data.frame", row.names = c(NA,
#> -7L))
```

This is useful to share the dataframe without creating a `csv` file or similar. It is the best way to share toy data on websites like stackoverflow.

You can convert the dataframe to a `tibble` using the `as_tibble()` function:

```
dput(as_tibble(df))
#> structure(list(member = c("Brian", "Mick", "Keith", "Stewart",
#> "Bill", "Charlie", "Ronnie"), start = structure(c(-2730, -2730,
#> -2730, -2730, -2582, -2525, 2304), class = "Date"), end = structure(c(-206,
#> 19043, 19043, -2437, 8370, 18863, 19043), class = "Date"), worth = c(10,
#> 500, 500, 1, 80, 250, 200)), class = c("tbl_df", "tbl", "data.frame"
#> ), row.names = c(NA, -7L))
```

which shows that the `tibble` does not contain more information than the `dataframe`.

Convert strings and factors to numeric values

Conversion across types must be done with care, obviously.

Functions such as `as.numeric()`, `as.character()` are 'vectorized', which means they can be applied to a vector/list and also to a single column in a dataframe.

```r
# convert a number ot a string:
as.character(0)
#> [1] "0"

# convert a string to a number:
as.numeric("0")
#> [1] 0

# convert a string to an integer:
as.integer("0")
#> [1] 0
as.integer("0.5")
#> [1] 0

# convert a vector of booleans:
as.numeric(c(TRUE, FALSE))
#> [1] 1 0
```

Convert strings and factors to numeric values

```
# convert a factor to number:
f = factor(c(0,1), levels=c(1,2))
as.character(f)
#> [1] NA  "1"
as.numeric(as.character(f))
#> [1] NA  1

# convert a column of a dataframe and reassign back to dataframe
df$worth <- as.integer(df$worth)
str(df)
#> 'data.frame':    7 obs. of  4 variables:
#>  $ member: chr  "Brian" "Mick" "Keith" "Stewart" ...
#>  $ start : Date, format: "1962-07-12" "1962-07-12" ...
#>  $ end   : Date, format: "1969-06-09" "2022-02-20" ...
#>  $ worth : int  10 500 500 1 80 250 200
```

Select data inside a dataframe

```r
# get a column
df$member
#> [1] "Brian"   "Mick"    "Keith"   "Stewart" "Bill"    "Charlie" "Ronnie"

# get one row
df[2,]
#>   member      start        end worth
#> 2   Mick 1962-07-12 2022-02-20   500

# get several rows
df[2:3,]
#>   member      start        end worth
#> 2   Mick 1962-07-12 2022-02-20   500
#> 3  Keith 1962-07-12 2022-02-20   500

# get a row by criterion
df[df$worth >= 300,]
#>   member      start        end worth
#> 2   Mick 1962-07-12 2022-02-20   500
#> 3  Keith 1962-07-12 2022-02-20   500
```

Add data to a dataframe

We can create categorical variables as follows:

```r
df$poor <- df$worth < 100
df$middle <- df$worth >= 100 & df$worth < 300
df$rich <- df$worth >= 300
```

We can create a duration variable as follows:

```r
df$duration <- round(100*as.integer((df$end-df$start))/as.integer(max(df$end-df$start)))
```

This calculates the interval between the start and end dates, calculates the maximum time interval in the datasets, and for each interval calculates the percentage it represents.

```r
df
#>    member      start        end worth  poor middle  rich duration
#> 1   Brian 1962-07-12 1969-06-09    10  TRUE  FALSE FALSE       12
#> 2    Mick 1962-07-12 2022-02-20   500 FALSE  FALSE  TRUE      100
#> 3   Keith 1962-07-12 2022-02-20   500 FALSE  FALSE  TRUE      100
#> 4 Stewart 1962-07-12 1963-05-01     1  TRUE  FALSE FALSE        1
#> 5    Bill 1962-12-07 1992-12-01    80  TRUE  FALSE FALSE       50
#> 6 Charlie 1963-02-02 2021-08-24   250 FALSE   TRUE FALSE       98
#> 7  Ronnie 1976-04-23 2022-02-20   200 FALSE   TRUE FALSE       77
```

Use data in a dataframe

The correlation between net worth and the duration variable is positive, as expected:

```
cor(df$duration, df$worth)
#> [1] 0.883
```

And significant:

```
cor.test(df$duration, df$worth)
#> 
#>  Pearson's product-moment correlation
#> 
#> data:  df$duration and df$worth
#> t = 4, df = 5, p-value = 0.008
#> alternative hypothesis: true correlation is not equal to 0
#> 95 percent confidence interval:
#>  0.388 0.983
#> sample estimates:
#>   cor 
#> 0.883
```

Import data

The most common data format is `csv` — comma separated values.

The most commonly needed options are `stringsAsFactors=FALSE` to prevent R from converting strings to factors and the option to use or discard header information, with `header=FALSE` or `header=TRUE`:

```
# basic import:
d <- read.csv("Age_HourlyEarnings.csv", stringsAsFactors=FALSE)

# import with more control:
d <- read.csv("Age_HourlyEarnings.csv", stringsAsFactors=FALSE,
              skip=1, na.strings="NA", header=TRUE, strip.white=TRUE)
```

There are many ways to import excel data. One option is to use the `read_xlsx()` function from the `readxl` package.

```
library(readxl)
d <- read_xlsx("Age_HourlyEarnings.xlsx", col_names=TRUE, skip=1, trim_ws=TRUE)
```

Export data

To save a dataframe as a `csv` file:

```
write.csv(df, file = "filename.csv", row.names=FALSE)
```

where you typically do not want to save the row names.

To save a dataframe as a `xlsx` file, among many options, you can use the `write_xlsx()` function from the `writexl` package:

```
library(writexl)
write_xlsx(df, file = "filename.csv", row.names=FALSE)
```

You can also save the data in the native `RData` format. This format will preserve information about data types, information that will otherwise be lost with the `csv` and `xlsx` formats. The only downside of the `RData` format is that it cannot be imported by spreadsheets and may not be easily read by other software, whereas the `csv` format is universal and the `xlsx` format nearly universal.

```
save(df, file = "filename.RData")
```

Export data

It is possible to import data from other formats, including `Stata`, SPSS, SAS. One option, among several, comes from the `Hmisc` and `foreign` packages. Examples:

```
library(foreign)
df.stata <- read.dta("stata-format.dta")

library(Hmisc)
df.spss <- spss.get("spss-format.por", use.value.labels=TRUE)

library(Hmisc)
df.sas <- sasxport.get("sas-format.xpt")
```

Quick sums

R knows how to sum the elements of a vector:

```
x = c(1,2,3,4,5)
sum(x)
#> [1] 15

x = c(TRUE, TRUE)
sum(x)
#> [1] 2
```

Other operations are possible in the same way:

```
x = c(TRUE, FALSE)
mean(x)
#> [1] 0.5
```

To sum across the columns or rows of a dataframe or matrix, the `colSums()` and `rowSums()` are optimized for speed. Similar are `colMeans()` and `rowMeans()`.

```
m = matrix(1:9, nrow = 3)
colSums(m)
#> [1]  6 15 24
rowMeans(m)
#> [1] 4 5 6
```

Split/Apply

To apply a particular calculation to a subset of the data, the `split()` function is convenient.

```
# split the data: poor vs the rest
split(df, df$poor)
#> $`FALSE`
#>    member      start        end worth  poor middle  rich duration
#> 2    Mick 1962-07-12 2022-02-20   500 FALSE  FALSE  TRUE      100
#> 3   Keith 1962-07-12 2022-02-20   500 FALSE  FALSE  TRUE      100
#> 6 Charlie 1963-02-02 2021-08-24   250 FALSE   TRUE FALSE       98
#> 7  Ronnie 1976-04-23 2022-02-20   200 FALSE   TRUE FALSE       77
#>
#> $`TRUE`
#>    member      start        end worth poor middle  rich duration
#> 1   Brian 1962-07-12 1969-06-09    10 TRUE  FALSE FALSE       12
#> 4 Stewart 1962-07-12 1963-05-01     1 TRUE  FALSE FALSE        1
#> 5    Bill 1962-12-07 1992-12-01    80 TRUE  FALSE FALSE       50

# apply a calculation: compute the mean for each group
sapply(split(df$worth, df$poor), mean)
#> FALSE  TRUE
#> 362.5  30.3
```

Functions

It is often convenient to create convenience functions. Ah yes.

A population standard deviation does not exist in base R. For quick check, `?sd`: The Help page for `sd` states “Like `var` this uses denominator n-1.”

Let’s roll our own:

```
std <- function(x) sqrt(sum((x-mean(x))^2)/length(x))
std(df$worth)
#> [1] 196
```

Note that we divide by `length(x)` and not `length(x)-1`

Functions that stretch over several lines use an opening and closing curly brace:

```
weighted.std <- function(x,w) {
    sqrt(sum(w*(x-weighted.mean(x,w))^2)/sum(w))
}
std(df$worth)
#> [1] 196
```

END OF DOCUMENT

This document ends here. Please check back later for updates.