R Basics

Instructor: Eric Dunford

Email: edunford@gmail.com

Overview

The main goal today is to gain a familiarity with the R envirnment. This initial session will be foundational in getting situated with the R programming environment.

We will cover:

- **Understanding R & R Studio**
- **Objects/Data Structures**
- **Packages**
- **Data Basics** (importing/exporting data, viewing data, subsetting/appending)

R in a Nut Shell

`R` is a statistical and graphical programming language that is based off a much older language called `S`. It's source code is written in C, Fortran, and R. And it's completely **free** under a GNU General Public License.

What this means for us:

- **No Barriers to Entry**: easy to acquire, easy to contribute
- **Active Community**: if you can think it, there is likely a package out there that does it.
- **Powerful and Adaptive**: build an estimator from scratch, scrape a web-site, automate the coding of a dataset. All is within one's reach.

Why use R?

`R` offers a powerful way to

- analyze data
- clean excel spreadsheets
- migrate projects across platforms
- format and clean text
- manage *any* data source
- produce compelling graphics and maps

R Studio

R Studio is a graphical user interface (GUI) for the R programming language. The software makes `R` more user-friendly adding some point and click functionality along with a complete integration of graphs, the data environment, and the coding script.

Think of it like this: `R` is the engine that runs all our commands, and R Studio is the leather seats and steering wheel. One does the work, the other eases how that work is done.

Installing R and R Studio

To install **R**, download R from CRAN via the following:

- Windows: https://cran.r-project.org/bin/windows/base/
- Mac: https://cran.r-project.org/bin/macosx/

To install **R Studio**, download from the following:

- https://www.rstudio.com/products/rstudio/download/

Getting Familiar with R Studio

R Studio can be arranged and customized to the users preference. The two main features of the program are the script and the console.

The Console

The console is where all the action happens. This is "`R`". All commands are processed through the console directly (that is, one can type commands directly into it) or via a **script**.

Scripts

A script is a `.R` text file where we write and run code our code. When we write a line of code, we can run it in the console by highlighting the text and...

- clicking **`run`**
- pressing **`command + enter`** (mac)
- pressing **`control + enter`** (windows)

Everything in a script will be treated as **code** – that is if you run it, the line will be processed through the console.

However, we can leave comments and notes to ourselves by **commenting out** sections of the script using a **`#`**

Objects

R uses a specific set of rules to goven how it looks up values in the environment.

We manage data by assigning it a name, and referencing that name when we need to use the information again.

Officially, this is called `lexical scoping`, which comes from the computer science term "lexing". Lexing is the process by which text represents meaningful pieces of information that the programming langauge understands.

Assigning an Object

In simple terms, an `object` is a bit of text that represents a specific value.

```
x <- 3
x
```

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

Here we've assigned the value `3` to the letter `x`. Whenever we type `x`, `R` understands that we really mean `3`.

There are three standard assignment operators:

- `<-`
- `=`
- `assign()`

"Best practice" is to use the `<-` assignment operator.

```
x1 <- 3
x2 = 3
assign("x3",3)
cat(x1, x2, x3)
```

```
## 3 3 3
```

Note that lexical scoping is flexible. Objects can be written and re-written when necessary. Down the road it will help to give objects meaningful names!

```
object <- 5
object
```

```
## [1] 5
```

```
object <- "A Very Vibrant Shade of Purple"
object
```

```
## [1] "A Very Vibrant Shade of Purple"
```

One can see all the objects in the environment by either looking at the user interface in RStudio (specifically, the **environment tab**)... or by typing `ls()` in the console.

```
ls()
```

```
## [1] "object" "x"      "x1"     "x2"     "x3"
```

Object Classes

Once assigned, an object has a **class**. A class describes the properties of the **data type** or **data structure** assigned to an object.

We can use the function `class()` to find out what kind of data type or structure our object is.

```
class(x)
```

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

The object `x` is of class `numeric`, i.e. a number.

There are many classes that an object can take.

```
obj1 <- "This is a sentence"
obj2 <- TRUE
obj3 <- factor("This is a sentence")
c(class(obj1),class(obj2),class(obj3))
```

```
## [1] "character" "logical"   "factor"
```

Understanding what class of object one is dealing with is important — as it will determine what kind of manipulations one can do or what functions an object will work with.

As noted, there are many different **data types** in R. We will primarily run into the following types: Integers (7), Numeric (4.56), Character ("Hello!"), Logical (TRUE), Factor ("cat" (1)).

Object Coersion

When need be, an object can be **coerced** to be a different class.

```
x
```

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

```
as.character(x)
```

```
## [1] "3"
```

Here we transformed x – which was an object containing the value 3 – into a character. x is now a string with the text "3".

Objects: So what's the point?

Objects offer a way to **reference different data**. This means that we can play around with *a lot* of different data type simultaneously.

This makes it easier to: + **manage** and use multiple datasets at the same time + **extract** and manipulate single variables + work with little bits of data at a time to make sure your calculations work.

> Note that the only way to hold onto information is to assign it as an object! Else the information is printed but instantly forgotten by R

Functions

What are functions?

A **function** is a type of object in R that can perform a specific task. Unlike objects that hold data, functions take **arguments** and return the output of some manipulation.

A function is specified first with the object name and then parentheses. For example, the function log() calulates the natural log of any number placed inside the parentheses.

```
log(4)
```

```
## [1] 1.386294
```

Functions operate in the **background**.

There are a number of functions in R, known as **base functions**, that are always running when you turn R on.

When we need to do things that are **not** a part of the base functionality, we can import new functions by installing **packages**. But more on this later.

We've already come a across a few functions, and we'll learn a lot more moving forward. Just keep in mind that whenever something is wrapped in parentheses (), it's a function.

All functions in R contain rich documentation regarding how a function works, the inputs it requires, and example code. We can access this documentation by using ? in front of the function.

```
?c()
```

Data Structures

There are also many ways data can be **organized** in R.

The same object can be organized in different ways depending on the needs to the user. Some commonly used data structures include:

- vector
- matrix
- data.frame
- list
- array

Vector

```
X <- c(1, 2.3, 4, 5, 6.78, 6, 10)
X
```

```
## [1]  1.00  2.30  4.00  5.00  6.78  6.00 10.00
```

```
class(X)
```

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

```
length(X)
```

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

Data Frame

```
data.frame(X)
```

```
##       X
## 1  1.00
## 2  2.30
## 3  4.00
## 4  5.00
## 5  6.78
## 6  6.00
## 7 10.00
```

Matrix

```
matrix(X)
```

```
##       [,1]
## [1,]  1.00
## [2,]  2.30
## [3,]  4.00
```

```
## [4,]  5.00
## [5,]  6.78
## [6,]  6.00
## [7,] 10.00
```

List

```
list(X)
```

```
## [[1]]
## [1]  1.00  2.30  4.00  5.00  6.78  6.00 10.00
```

Array

```
array(X,dim = c(2,2,2))
```

```
## , , 1
##
##      [,1] [,2]
## [1,]  1.0    4
## [2,]  2.3    5
##
## , , 2
##
##      [,1] [,2]
## [1,] 6.78   10
## [2,] 6.00    1
```

The point is that we need to understand the *structure* of a data object to understand how to *access* the information inside.

There are many ways to organize the same piece of information in `R`, and different data structures afford us different advantages and bring with them different limitations.

Throughout this short course, **data frames** will be the dominate data structure that we use; however, as you become more acquainted with `R`, you'll see and use other types of data structures more often.

Accessing a Data Object

One must understand the **structure of an object** in order to systematically access the material contained within it.

Let's use a dataset inherent to `R` called `cars`. There are a number of datasets that are built into `R`. These are for demonstration purposes.

Note that these data will not appear in the environment **until we assign them to an object**.

```
data <- cars
class(data)
```

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

An easy way to see what's inside a data object is to just `print()` it. `R` prints objects automatically in the console.

```
data
```

```
##    speed dist
## 1      4    2
## 2      4   10
## 3      7    4
## 4      7   22
## 5      8   16
## 6      9   10
## 7     10   18
## 8     10   26
## 9     10   34
## 10    11   17
## 11    11   28
## 12    12   14
## 13    12   20
## 14    12   24
## 15    12   28
## 16    13   26
## 17    13   34
## 18    13   34
## 19    13   46
## 20    14   26
## 21    14   36
## 22    14   60
## 23    14   80
## 24    15   20
## 25    15   26
## 26    15   54
## 27    16   32
## 28    16   40
## 29    17   32
## 30    17   40
## 31    17   50
## 32    18   42
## 33    18   56
## 34    18   76
## 35    18   84
## 36    19   36
## 37    19   46
## 38    19   68
## 39    20   32
## 40    20   48
## 41    20   52
## 42    20   56
## 43    20   64
## 44    22   66
## 45    23   54
## 46    24   70
## 47    24   92
## 48    24   93
## 49    24  120
## 50    25   85
```

We can look at the **structure** of a data object by using the `str()` function.

```
str(data)
```

```
## 'data.frame':    50 obs. of  2 variables:
##  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
##  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

We can leverage what we know about the dimensionality of the data to extract parts of it.

We do this by using brackets `[]` alongside the data object. We then can access the **dimensions** in the data by specifying the row and column

`data[row,column]`

The function `dim()` can tell use about the dimensions of a data object.

```
dim(data)
```

```
## [1] 50  2
```

We now know that the object `data` has **50 rows** and **2 columns**.

```
data[,2] # Access the entire 2nd column
```

```
##  [1]   2  10   4  22  16  10  18  26  34  17  28  14  20  24  28  26  34
## [18]  34  46  26  36  60  80  20  26  54  32  40  32  40  50  42  56  76
## [35]  84  36  46  68  32  48  52  56  64  66  54  70  92  93 120  85
```

```
data[49,] # Access just the 49th row
```

```
##    speed dist
## 49    24  120
```

```
data[1,2] # Access just a cell
```

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

The key is to keep in mind the dimensions. We can't access data that isn't there.

```
data[51,]
```

```
##    speed dist
## NA    NA   NA
```

Most data objects can be accessed using `$` call sign.

`$` acts as a **key** by which we can extract a specific variable or data feature.

If we hit **Tab** after specifying the `$` after our data object, R Studio will offer a list of all available variables.

Here we call the `speed` variable from our dataset, using the handle.

```
data$speed
```

```
##  [1]  4  4  7  7  8  9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14 14
## [24] 15 15 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22 23 24
## [47] 24 24 24 25
```

There are many functions designed to help us understand the **dimensions** of a data structure.

```
dim(data) # Dimensions
```

```
## [1] 50  2
```

```r
nrow(data) # Number of Rows
```

```
## [1] 50
```

```r
ncol(data) # Number of Columns
```

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

There are also some useful functions built into R to view portions of a data structure.

```r
head(data,3) # Reports the 3 first entries
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
```

```r
tail(data,3) # Reports the 3 last entries
```

```
##    speed dist
## 48    24   93
## 49    24  120
## 50    25   85
```

`summary()` allows for one to quickly summarize the distributions across a set of variables

```r
summary(data)
```

```
##      speed           dist
##  Min.   : 4.0   Min.   :  2.00
##  1st Qu.:12.0   1st Qu.: 26.00
##  Median :15.0   Median : 36.00
##  Mean   :15.4   Mean   : 42.98
##  3rd Qu.:19.0   3rd Qu.: 56.00
##  Max.   :25.0   Max.   :120.00
```

Removing objects from the Environment

We often want to get rid of objects after creating them. To **delete** (or drop) an object from the working directory, use the function `rm()` – which stands for "remove".

```r
ls()
```

```
##  [1] "data"   "obj1"   "obj2"   "obj3"   "object" "x"      "X"
##  [8] "x1"     "x2"     "x3"
```

```r
rm(x,x1,x2,x3,X)
ls()
```

```
## [1] "data"   "obj1"   "obj2"   "obj3"   "object"
```

Clearing the Environment

We can also remove ***all*** objects from the environment at once by typing the following command.

```r
rm(list=ls(all=T))
```

Or we can do so from **R Studio** by clicking on the `broom icon`.

R Packages

There are a number of `packages` that are supplied with the R distribution. These are known as "base packages" and they are in the background the second one starts a session in R.

A `package` is a set of functions and programs that perform specific tasks. By installing packages, **we introduce new forms of functionality to the R environment**. To use the content in a package, one first needs to **install it**. One can do this by utilizing the following function: `install.packages()`. By inserting the name of a specific package, we can connect to an R "mirror" and download the binary of the package.

```
install.packages("ggplot2")
```

The version of that package is then saved on your computer and can be called at any time (on or offline).

Once installed, it's on the system for good. You can then reference or load the package any time you wish to use a function from it.

There are two functions we can use to load a package: `library()` and `require()`.

```
library(ggplot2)

# or

require(ggplot2)
```

You must *load* the package before you can use any function in it.

`R Studio` also offers us a way to install packages through the interface.

If we click on the `Packages` tab and then click `Install`, we can download a package by typing its name.

We then can **load** the package from R Studio by clicking the check box beside the packages name.

Sometimes one has *a lot* of packages running simultaneously.

No problem: we can see what packages are up and running by typign `sessionInfo()` into the console.

This will tell us everything about the version of R and the packages we are using to run our analysis.

```
sessionInfo()
```

```
## R version 3.3.2 (2016-10-31)
## Platform: x86_64-apple-darwin13.4.0 (64-bit)
## Running under: OS X El Capitan 10.11.3
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## loaded via a namespace (and not attached):
##  [1] backports_1.0.5 magrittr_1.5    rprojroot_1.2   tools_3.3.2
##  [5] htmltools_0.3.5 yaml_2.1.14     Rcpp_0.12.7     stringi_1.1.2
##  [9] rmarkdown_1.3   knitr_1.15      stringr_1.1.0   digest_0.6.12
## [13] evaluate_0.10
```

Importing/Exporting Data

R allows you to import a large variety of datasets into the environment. However, R's base packages only support a few data types.

No Fear: there is usually always an **external package** that can do the job!

We are going to focus on four packages to import different data types:

- `foreign` & `readstata13` — for .dta (stata) file formats.
- `readxl` — for excel spreadsheets
- `haven` — for SPSS and SAS

First, we need to **install** these packages onto our computer.

```
install.packages("foreign") # For .dta (STATA 12 or lower)

install.packages("readstata13") # For .dta (STATA 13 or higher)

install.packages("readxl") # For excel spreadsheets

install.packages("haven") # For SPSS and SAS
```

And then **load** them into our current R Session.

```
require(foreign)

require(readstata13)

require(readxl)

require(haven)
```

But where is my data exactly?

R doesn't intuitively know where your data is. If the data is in a special folder entitled "`important_research`", we have to tell R how to get there.

We can do this three ways:

1. Set the **working directory** to that folder
2. Set the directory via a point-and-click option in R Studio
3. Establish the **path** to that directly to the folder

Setting the Working Directory

Every time R boots up, it does so in the same place, unless we tell it to go somewhere else.

We can find out which directory we are in by using the `getwd()` function.

```
getwd() # Get the current working directory
```

```
/Users/username/
```

Every time R boots up, it does so in the same place, unless we tell it to go somewhere else.

We can then set a new working director by establishing the path to the folder we want to work in as a **string** in the function `setwd()`

```
setwd("/Users/username/Desktop/important_research")
getwd()
```

`/Users/username/Desktop/important_research/`

Setting the working directory via R Studio

R Studio also makes setting the working directory really easy.

Click: `Session/Set Working Directory/Choose Directory...`

This will allow you to set the working directly quickly. The downside is that you have to do it manually every time you return to this project. By writing a script for everything you do, it is easier to replicate (and for others to replicate) your work.

Establishing the Path

Finally, we can also just point directly to the data by outlining the specific path.

Here we are assigning a sting containing our **path** to the object `path.to.data`.

```
path.to.data <- "/Users/edunford/Desktop/important_research/data.csv"
```

We then load the data using that path.

```
read.csv(path.to.data)
```

Importing data

Here we will review how to import five seperate data types: - `.dta` — STATA file format - `.csv` — comma seperated file format - `.sav` — SPSS file format - `.xlsx` — standard Excel file format - `.Rdata` — R's file format

.dta

For STATA 12 <=

```
require(foreign)
data <- read.dta(file = "data.dta")
```

For STATA 13 >=

```
require(readstata13)
data <- read.dta13(file = "data.dta")
```

.csv

`read.csv()` and `read.table()` are both **base functions** in `R`.

```
data = read.csv(file = "data.csv",
                stringsAsFactors = F)
# Or

data = read.table(file = "data.csv",
```

```
                  header = T,
                  sep=",",
                  stringsAsFactors = F)
```

These functions have specific **arguments** that we are referencing:

- `stringsAsFactors` means that we don't want all `character` vectors in the `data.frame` to be converted to `factors`.
- `header` means the first row of the data are column names.
- `sep` means that entries are seperated by commas.

.sav

For `SPSS` and `SAS` file formats, the `haven` packages offers a simple way of reading in data.

```
require(haven)
read_sav(file = "data.sav") # SPSS
```

.xlsx

```
require(readxl)
read_excel(file="data.xlsx")
```

.Rdata

`.Rdata` is the data source inherent to `R`. It saves and loads `objects`.

```
load(file='data.Rdata')
```

Importing Data Using R Studio

There is also a point-and-click option for importing and exporting data in R.

If we go into the `Environments` tab and then click `Import Dataset`

Exporting data

Exporing data is the same process in reverse. Instead of **reading** the data, we want to **write** a new version of it.

There are a series of functions (each provided by their respective packages) that allow us to do just that.

Though the arguments for each function differ somewhat, each require that you input the **data** that you're looking to export and specify the **file name** and **paths** to tell the computer where the file is going.

.dta

Assuming we have the `foreign` and `readstata13` packages already loaded, we can create a new `.dta` file, using the `write.dta()` and `save.dta13()` functions.

```
write.dta(data,file="data.dta") # stata <= 12

save.dta13(data,file="data.dta") # stata >= 13
```

.csv

`write.csv()` and `write.table()` are both **base functions** in `R`.

```
write.csv(data,file="data.csv",row.names = F)

write.table(data,file="data.csv",sep = ",")
```

.sav

```
write_sav(data,file="data.sav") # SPSS
```

.Rdata

`.Rdata` offers two options to save data. We can either save a single data object, or save the entire workspace

```
# Save just an object
save(data, file="data.Rdata")


# Save the entire workspace
save.image(file="workspace.Rdata")
```