Lab One: Introduction and Basics

Alex Davis
Cody Adams

This lab serves as a primer to R by introducing the basics. It is advised to follow the lab via the Rmd file within RStudio rather than solely the compiled PDF. This way, students can experiment with code within the "code blocks" provided. **Note:** References to R code are exclusively italicized.

Part I: R, as a Calculator

The first thing to know about R is that it is essentially a large calculator capable of performing arithmetic:

```
1+1
```

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

```
8*8
```

```
## [1] 64
```

```
2^8 # exponent
```

```
## [1] 256
```

```
(5+2)^4
```

```
## [1] 2401
```

```
5+2^4
```

```
## [1] 21
```

R also supports elementary and algebraic functions such as log and square root.

```
log(100)
```

```
## [1] 4.60517
```

```
sqrt(49)
```

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

Order of Operations

R solves equations according to the order of operations, "PEMDAS":

1. Parentheses
2. Exponents
3. Multiplication
4. Division
5. Addition
6. Subtraction

The following video provides a refresher on the order of operations: https://www.youtube.com/watch?v=94yAmf7GyHw

Try this! Using R, solve: (4+6)^2/(3-2)+8

Part II: Objects

R is an "object oriented" programming language. Put simply, R uses objects to store attributes. Objects can be used to store attributes. Objects are created by assigning an attibute to them via the <- operation. You can always view the attribute of an object by typing the object name.

```
object1 <- 10 + 10
object1
```

```
## [1] 20
```

Try this! Objects can be named almost anything!

R includes various functions for managing created objects. The ls() function lists all existing objects.

```
ls()
```

```
## [1] "object1"
```

The rm() function removes existing objects.

```
rm(object1)
```

There is no strict convention for naming objects; however, there are best practices:

1. Avoid spaces, use *underscores*, periods, or CamelCase (or camelCase) for long object names.
    - e.g., This_is_a_long_name This.Is.A.Long.Name, thisIsALongName
2. Avoid names of existing functions or reserved R objects.
3. Be descriptive but keep it short (less than 10 characters).
4. Avoid special characters.
    - e.g., ? $ % ^ &

Important: Object names are case sensitive. - object.One and Object.One refer to two separate objects.

```
object.One <- 10+10
Object.One <- 5+5

object.One
```

```
## [1] 20
```

```
Object.One
```

```
## [1] 10
```

Part III: Functions

In addition to elementary and algebraic functions, R also include functions to simplify statistical analysis. For example, the *mean()* functione exists to calculate mean. **Note:** Sometimes *na.rm=TRUE* is necessary within the paranetheses to instruct R to ignore missing data.

```
mean(cars$dist)
```

```
## [1] 42.98
```

R includes the *cars* data set, consisting of various information about cars. This data set is provided to help demonstrate and/or experiment with R functions.

A note about syntax: the dollar sign, $, is used to indicate the variable of interest relative to a data set. This is important in the case of multiple data sets that contain variables of the same name. In the previous code, R calculated the mean using the *dist* variable within the *cars* data set by specifying *cars$dist*

To ignore missing data when calculating the mean of *dist*, include the *na.rm=TRUE* argument within the paranetheses as follows.

```
mean(cars$dist, na.rm=TRUE)
```

```
## [1] 42.98
```

Note: Given the mean is exactly the same, it is unlikely there is any missing data in the *dist* variable.

Object Types

Object types are improtant in R and the type of an object is contingent on the attribute stored by the object. Put simply, an object storing characters (e.g., "blue") has a different type than an object storing a number (e.g., 1). Use of R functions is contingent on the type of objects. For example, functions like *mean()* work only for objects containing numbers. The R *str()* function describes the structure of objects and functions.

```
str(mean)
```

```
## function (x, ...)
```

```
str(object.One)
```

```
##  num 20
```

```
str(cars)
```

```
## '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 ...
```

```
str(cars$dist)
```

```
##  num [1:50] 2 10 4 22 16 10 18 26 34 17 ...
```

The *str()* function described *mean* as a function, *object.One* as a numeric object, the *cars* data is a data frame, etc.

Previously, objects were introduced as a method of storing single attributes, either a specified value or the result of arithmetic. In addition, objects can contain a collection of data via a vector or list. In mathematics and physics, a vector is defined as a quantity of both direction and magnitude. In R, vectors are defined a collection of data of the same type. The *c()* function creates a vector.

```
vectorObject <- c(1,2,3)
vectorObject
```

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

```
str(vectorObject)
```

```
##  num [1:3] 1 2 3
```

Further, a list is defined as a collection of multiple data types.

```
listObject <- list("your name", 1, F)
listObject
```

```
## [[1]]
## [1] "your name"
##
## [[2]]
## [1] 1
```

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

```
str(listObject)
```

```
## List of 3
##  $ : chr "your name"
##  $ : num 1
##  $ : logi FALSE
```

Note: The structure of the list object consists of a character, a number, and a logic (True/False).

Part IV: Packages

Packages expand R to include additional functions important to statistical analysis. The default R functions may not be inadequate or complicated to conducting certain analysis.

Installing Packages

Installing packages in R is performed via *install.packages("package_name")*, whereby the name of the desired package must be within quotation marks.

Note: Occasionally a package may require dependencies for installation. The dependencies can be automatically installed along with the package by including the *dependencies=TRUE* argument within the *install.packages()* function.

Try this! In R, install: In your own code chunk, include the following (without the numbers):

1. *car*
2. *psych*
3. *memisc*
4. *Rcpp*

Loading Packages

Installed packages must be loaded to use their functions within R. Packages are loaded via *library(package_name)*, whereby the name of the desired package must be within parantheses. **Note:** Unlike the *install.packages()* function, the *library()* function does not use quotation marks around the package name.

```
## Loading required package: carData
```

```
##
## Attaching package: 'psych'
```

```
## The following object is masked from 'package:car':
##
##     logit
```

```
## Loading required package: lattice
```

```
## Loading required package: MASS
```

```
##
## Attaching package: 'memisc'
## The following object is masked from 'package:car':
##
##     recode

## The following objects are masked from 'package:stats':
##
##     contr.sum, contr.treatment, contrasts

## The following object is masked from 'package:base':
##
##     as.array
```

Note: The *memisc* package contains object/package conflicts with the *car* package for the *recode* object. A conflict occurs when two packages contain objects (e.g., functions) of the same name. A conflict will not prevent loading packages; however, use of a specific package's object requires an explicit call to the desired parent package. For example, to use the *recode()* function from *car*, the *car::recode(variable)* statement will explicitly call *recode()* from the *car* package. Vice versa, *memisc::recode()* will explicitly call *recode()* from the *memisc* package.

Updating Packages

Most packages are regularly updated. The *old.packages()* function compare installed packages to their latest versions online.

The *update.packages()* function updates out of date packages. **Note:** Updating packages requires consent. The *ask=FALSE* argument will skip the additional consent step to save time.

The *library()* function lists currently loaded packages.

```
library()
```

As previously demonstrated, occasionally conflicts exist between packages. The *conflicts()* lists conflicts between loaded packages.

```
conflicts()
```

```
##  [1] "initialize"      "show"            "logit"
##  [4] "recode"          "contr.sum"       "contr.treatment"
##  [7] "contrasts"       "contrasts<-"     "prompt"
## [10] "npk"             "Arith"           "coerce"
## [13] "Compare"         "initialize"      "Math"
## [16] "Math2"           "show"            "Summary"
## [19] "%in%"            "as.array"        "as.factor"
## [22] "as.ordered"      "body<-"          "formals<-"
## [25] "format"          "kronecker"       "labels"
## [28] "merge"           "print"           "row.names"
## [31] "sample"          "subset"          "summary"
## [34] "unique"          "within"
```

The *detach()* function detaches packages and is an alternative method to resolve conflicts. Supplying *unload=TRUE* argument within the *detach()* function will unload the package. For example, to resolve the *recode()* function conflict between *car* and *memisc*, the memisc package can be detached and unloaded as follows: *detach(package:memisc, unload=TRUE).*

Part V: R Help

R includes a help function to assist with functions, accessible by including a *?* prior to the function name.

```
? mean
```

Note: The help documentation will display in the bottom right quadrant of RStudio. Alternatively, typing the function name into the help search bar will yield a similar result.

To search all of R documentation for help about a function, use *??*.

```
?? mean
```

Note: Google is a valuable tool for finding help. Large communities like StackExchange consist of answers and explanations to common issues in R. At times a particular problem may seem unique, but someone else has almost certainly had the same problem and the solution likely can be found online.

Part VI: Setting a Working Directory

The working directory is the location where files accessed and saved within a R session. Normally, the working directory is set at the beginning of every R file. The working directory should be set and the class data set at the beginning of each lab.

There are two methods of setting the working directory. First, the *setwd()* function can be used with the directory path. For example, *setwd("C:/Directory_to_folder/")*. **Note:** Forward slashes are used in place of backward slashes for directory paths.

Second, within RStudio, the "Session" tab will set the working directory. The following steps provide guidance to the "Session" tab functionality:

1. Click the "Session" tab.
2. Select "Set Working Directory."
3. Select "Choose Directory."
4. Select the working directory.

The *getwd()* function returns the set working directory.

```
getwd()
```

```
## [1] "/Users/lolevo/Documents/GitHub/qrmlabs"
```

Part VII: Importing Your Data

Data sets should be downloaded from the GitHub repository, http://www.need_url.com. **Note:** The data set should be saved within the working directory.

Data sets are commonly saved with the comma separated value file extension (e.g., .csv). CSV files stores data similar to the Excel spreadsheet data structure, and therefore can be viewed easily within Excel.

Note: Be careful when accessing CSV files within Excel. Excel removes leading zeroes from values, which may impact the integrity of your data if you save the file after viewing in Excel.

Loading a data set from a CSV file into R requires loading the data set with a function, and saving the contents of that data set to an object. For example, *data_set_name <- read.csv("file_name.csv", header=TRUE)*. The *header=TRUE* argument specifies that the "first row" of the CSV file contains the variable names for that data set's columns. If the CSV file does not contain the variable names in the first row, you can change *TRUE* to *FALSE*.

For labs, the following code should look like this:

ds <- read.csv("Class Data Set Factored.csv", header = TRUE)

Try it! Load a data set.

```r
ds <- read.csv("Class Data Set Factored.csv", header = TRUE)
```

The name of a data set object can be anything. The best practice is to use an acronym, such as *ds*.

Part VIII: Connecting to Your Data

Imported data can be accessed various ways. First, variables can be directly accessed by attaching the data set via the *attach()* function. For example, *attach(ds)*.

Attached data can be detached via a similar function, *detact()*. For example, *detach(ds)*.

However, the *attch()* function is not the recommended method. Instead, variables can be accessed by joining the data set object name and the variable name via the dollar sign, $.Forexample,_d svariable$

The following exemplifies this method using the *dist* variable within the *car* package, within the *mean()* function.

```r
mean(cars$dist, na.rm = TRUE)
```

```
## [1] 42.98
```

Using the class data set stored in the *ds* object, there exists an *age* variable. The *mean()* function can provide the mean age of this data set.

```r
mean(ds$age, na.rm = TRUE)
```

```
## [1] 60.36749
```

Additionally, there is a variable *income* within *ds*. **Try it!** Find the mean of the *income* variable:

Note: The *na.rm=TRUE* argument is required within the *mean()* function to avoid a NA result.

Part IX: Exporting a CSV

Saving a CSV data set can be performed via the *write.csv()* function by specifying the object to save and the file name to write. For example, *write.csv(data_set_name, file = "filename.csv")*.

To demonstrate, the *ds* data set is saved as newds.csv within the working directory.

```r
write.csv(ds, file = "newds.csv")
```