Data Wrangling & Presentation in R

An introduction to R and Markdown

Course Instructors

- Alex Chubaty **B7250** achubaty@sfu.ca
- Corey Phillis **B6226** cphillis@sfu.ca
- Franz Simon **B7250** fsimon@sfu.ca
- Sean Anderson **B6226** scanders@sfu.ca

Before we start

- Install R version 3.0.0 or higher from CRAN
- Install R Studio from rstudio.org
- make sure the `knitr` package is installed

```
install.packages("knitr")
library(knitr)
```

Intro to R

The purpose

The purpose of these companion notes is to act as a supplement to lecture 1 of Data Wrangling and Visualization in R. Therefore, these notes will cover three things

1. Basics of good programming practices
2. Introduction to basic R syntax
3. Introduction to functions that allow you to analyze your data.

At most points these notes will mention common pitfalls and good techniques for working with your data. As you will soon discover there are a million ways to skin a cat; however, using a knife is more effective than a photo copier.

What is R?

R is a free software environment for statistical computing and graphics. To see more details see:

www.r-project.org > free software environment for statistical computing and graphics

Cross-platform (Windows, OSX, Linux)

Why use R?

A more interesting question is "Why use R?" There are several reasons:

1. Customization

- R allows you more control over graphing and statistics than most GUI platforms like JMP.

2. Complexity

- R is open source and is supported by a large community. This means that new statistcal techniques when created are quicker to be implented than using SAS since you don't have to wait for a new release of your software

3. It's *free*

- This is one of its biggest selling points. As graduate students you are two things: poor and transient. Which means that unless your supervisor has paid a lot of money for a license you will not be able to afford buying proprietary software. The more important thing is you will only be at graduate school for 2-6 years. This means that after you leave university you will no longer be able to work on your statistics anymore if you use a proprietary product.

Using R

- R is a command line tool
    - commands can be entered one line at a time

For example if you open up R console and type:

```
{r} 2 + 2
```

Then press enter you should get a line saying 4.

Your code will have been executed. - pressing the up arrow in the R terminal will recall the previous command

Entering things in line by line is usually inefficent and does not take advantage of one of the big pros of R which is reproducibility. Reprocuibility allows you to perform the same action again in the exact same way. This allows you to run the same analysis on different data, or be able to address reviewer mistakes quicker because you will have saved the commands saying exactly what it is you have done.

- You can instead execute code by writing a script and executing the script.

This approach has the advantage of reproduciblity. Also it allows for you to make far bigger and more complex programs since you do not have to re-enter in each command. To do this you will need a code editor. We will be using R-Studio for this course.

Introduction to R-Studio

When you open R studio you will notice four windows. Today you will only focus on the two on the right. The top one is the editor and the bottom window is the R-Console that you used previously.

Now you can create a new R script by typing `Ctrl + Shift + N` or going to the File tab and clicking on new R script.

You will notice that this script works exactly like notepad. You can write and delete text. For example type:

```
2 + 2
```

```
## [1] 4
```

again and press enter. Nothing will happen. In order to actually run your code highlight `2 + 2` and press `Ctrl + Enter`. You will see in the bottom right panel that your code will have executed.

We have now learned how to run a script in R. A whole new world has been opened to you! And it is a dazzling place to be.

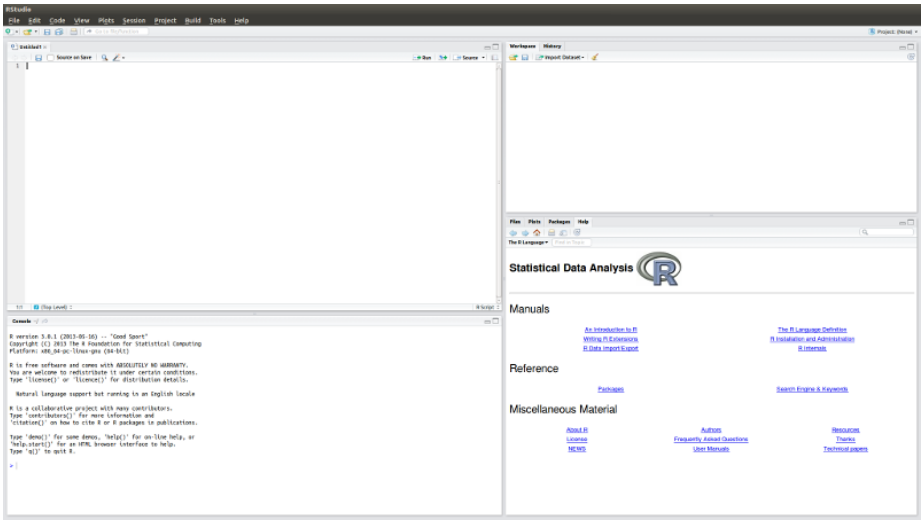
Getting help with R

The first and most important thing to know how to do in R is how to get help. You know you are on your way to becoming a guRu when you can figure out how to do new things all on your lonesome.

- R has to built in ways to get help. First is `?`. `?` will give you a help file. For example if you want to know how to use the `summary()` function you can type `?summary()` and a help file will appear on the bottom right panel of R-Studio. This file will tell you different things about the function and will also often give you examples.
- The other built in help operator is `??` this is for when you don't know the name of the function or package you want. For example, if you want to do some bootstrapping but don't know the name of the function type `??bootstrap` and it will produce a list of packages that are related to bootstrapping.

- Google is your friend. Google is a great help in finding solutions to any problems you may have. The only difficulty is asking the write question. As you become more familiar with R this will become easier.
- R Stack Exchange is an extremely good resource. It is a help forum for R. I suggest searching for similiar questions before posting anything. People who are active on this site are a little prickly when you ask a question that has been asked before. Do not be intimidated though it is a great resource!!!

R studio



Basic R Syntax

Syntax are the symbols you are going to be using to create scripts. R only uses a couple.

`?`

See above.

`#`

This is the most important symbol in all of R! It is used to create comments. Comments are sections of code that the computer is told not to read when it is executing your script, which isn't as useless as it sounds. Comments are the

way you can write notes to yourself about what your code is actually doing. Just think how foreign R looks right now just wait until you don't use it for a year and you come back... Therefore, as a seasoned programmer I give you this advice. Comment!!! and then comment some more. I have never heard a programmer complain that they didn't comment enough.

`<-`

This is the assign symbol. It says take what you have on the right and make whatever is on the left of it have that value, for example:

```
x <- 2
### Assigns the variable x the value of 2. You can now do fun things like
y <- x + 5  ### Add the 5 to the value of x.
print(y)  ###Display the value of y.
```

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

`=`

Is the equality symbol. We will learn more about this when we get to control operators. It can be used for

`()`

Round braces are used in a bunch of ways in R from containing the arguments of functions to their mathematical use. How `()` works depends on the context in which they are used.

`{}`

Curly braces are used for denoting chunks of code that are part of functions and iteriation operators.

`[]`

Used for indicating the index of a (location of a value stored in a) vector, matrix, array, or list, etc..

`:`

`;`

`""`

Used for denoting characters.

Basic Commands

Assigning variables

Assigning variables is a key of programming. It is a little different from the world of math and algebra. For example, we can do this.

```
x <- 3
print(x) ## x has the value of 3.
x <- 2
x <- 7
print(x) ## x now has the value of 7. The assign is not an equality sign it will rewrite the
```

The fact that you will rewrite over symbols if you assign something to them is foundational. This is often very useful but you can run into big problems when you have bigger scripts. To avoid overridding a value when you don't want to I suggest using informative names. Maybe the variable `x` was my mass when I was born in kg. So instead call it `massBirthkg` or `mass.birth.kg`. You can use any name you want as long as it don't start with a number. You can use numbers in variable names just not as the first letter.

Assigning vectors

Vectors are an ordered list of numbers. Thing of variables as a single column of mailboxes at a post office. Each mail box (element in computer speak) contains your bills(a value) however they also have an address on them. Letting you refer to them as the 1st mail box, the 2nd, etc...

There are mutiple ways to create vectors, for example:

```
a <- c(11.2, 33.4, 16.7, 13.2, 7.8, -23.5)
b <- c(1:10)
d <- seq(from=0, to=1, by=0.1)
```

In order to access the value of the 3rd element of `a` you write `a[3]`. Vectors in R are listed starting from 1.

Assigning variables

R is very flexible and pretty smart. It will allow you to assign more than numeric values to variables. For example:

```
name <- "Alex"
names <- c("Alex", "Brian", "Conan", "Dave", "Erin", "Franz")
colr <- list("red", "green", "blue")
```

This flexibility can lead to trouble down the road however because functions will require specific variable types. For example if you try to add all the elements of `names` using `sum(names)` you will get an error.

Arithmetic

You can do arithmetic in R and thankfully the notation is familiar

```
+ ### Add
- ### Subtract
* ### Multiply
/ ### Divide
```

These will follow BEDMAS.

You can do fun things like

```
x <- 7
y <- 13.5
z <- 2.5

x <- ((x * y + z) * 5 - 25)
print(x)
```

```
## [1] 460
```

Powers

```
exp()
^
log()
log10()
```

Mathematical functions

What is a function? Think of a function as a machine. You put an input (an argument) in and you get an output. Here are some examples of mathematic functions.

```
sum()
abs()
mean()
sd()
sqrt()
```

Data Types

- R treats different data in different ways
- R will automatically select what it thinks is the appropriate data type
- However, you may need/want to specify data types yourself
- R uses numeric and string variables, lists, arrays, matrices, dataframes

```
is()

# A family of functions. is.numeric() checks whether an object is of
# type numeric, for example.

is.na()

# Identifies elements in the data set that are NA. NA means no value in R.
# Often an important step before data analysis when you need to clean up your
# data.

as()

# This is a family of functions that allow you to coerce an object to another
# type. For example, you can convert a dataframe to a matrix but take a dataframe
# such as trees using the following code:

matrix.trees <- as.matrix(trees)
```

An example

```
# sample data from the trees dataset
summary(trees)
?trees # tells us about the dataset
# Height is in feet
# Girth (diameter) is in inches measured at 4'6"
# Volume is in cubic feet
radii <- trees$Girth / 2
radii.ft <- radii / 12
volumes <- pi * (radii.ft)^2 * trees$Height
trees$Volume
```

Working with data in R

Importing and viewing data

```
setwd("path/to/data/directory")
```

```
# Quotations are important for setwd(). You should set your working directory
# at the beginning of each script. You should have a different folder for each
# project or group of related projects.

data <- read.csv("filename.csv", header=TRUE )
# Imports your data from a csv into a dataframe. The header argument lets you
# import the first row of your spreadsheet as header for your dataframe.

summary(data)
# Summarizes the data giving basic summary statistics depending on datatype.

head(data)
# Prints the first five lines and the header of a dataframe.

tail(data)
# Prints the last five lines and the header of a dataframe.

names(data)
# Prints only the headers of the dataframe.

which()
# Is used with truth operators to find the position of elements of which the
# statement is true. For example,

d <- c(3, 4, 1, 9, 34)

which(d > 7)
# which will print values at indices 4 & 5.
```

Using additional packages

```
install.packages("plyr")
require(plyr)
```

Sample R script

```
# change working directory
setwd("path/to/working/directory")

# install/load additional packages
if (!require(lme4)) {
    install.packages("lme4"); library(lme4)
}
```

```
# import data
data <- read.csv("filename.csv, header="TRUE")

# first-look at the data
head(data)
summary(data)

# clean up data
...

# visualize the data
...

# analyse the data
...
```

Control operators

Conditional statements

```
if (condition1) {
    doThisThing
} else {
    doAnotherThing
}
```

Evaluation of conditions

```
<
>
== ### You must use two since = means assign
<=
<=
!= ### Does not equal
&& ### And
|| ### Or (in programming "or" is inclusive)
```

This means in programming if you say "Bob is a tall or a boy" It is true for all of these conditions:

- Bob is tall, and not boy,
- Bob is tall, and a boy,
- Bob is short and a boy

The only thing it is not true for is Bob is short and not a boy.

Loops

- `for` loops are used when the number of iterations in *known*
  - An example of where you would use this kind of loop is if you wanted to know how a population of animals grew over the next 50 years. You will want to stop at 50 years so you know how many loops you need.
- `while` loops are used when the number of iterations is *unknown*
  - An example would be instead of wanting to know the population size at 50 years if you wanted to know how long until a population went extinct. You would keep running the simulation until the population hit zero. For this kind of task you won't know for how long you will have to run the simulation.
- Most of the time you should use `for` loops since it is possible when using a `while` loop to great an infinite loop that never finishes.

```
pies <- numeric(100)
for (i in 1:length(pies)) {
    pies[i] <- pi * i
}
```

```
x <- 0; done=FALSE
while (!done) {
    if (x > 23) {
        done = TRUE
    } else {
        x <- x + pi
    }
    print(x, digits=4)
}
```