My first R Markdown

Rudy Rupp

21/08/2020

Week 1

What is Data Science? 0:43 - 0:56 They state that a data scientist is broadly defined as someone who combines the skills of software programmer, statistician, and storyteller/artists to extract the nuggets of gold hidden under mountains of data.

Week 2

R Studio tour

```
matrix(c(1, 2, 3, 4, 5, 6, 7, 8), nrow = 4, ncol = 2)
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8
```

R Packages

Installing from CRAN
If you are installing from the CRAN repository, use the *install.packages()* function, with the name of the package you want to install in quotes between the parentheses (note: you can use either single or double quotes). For example:

install.packages("ggplot2")

To install multiple packages at once, you can do so by using a character vector:
install.packages(c("ggplot2", "devtools", "lme4"))

Installing from Bioconductor
The BioConductor repository uses their own method to install packages. First, to get the basic functions required to install through BioConductor, use:

source("https://bioconductor.org/biocLite.R")

This makes the main install function of BioConductor, *biocLite()*, available to you. Following this, you call the package you want to install in quotes, between the parentheses of the biocLite command, like so:

biocLite("GenomicFeatures")

Installing from GitHub
You first must find the package you want on GitHub and take note of both the package name AND the author of the package. Check out this guide for installing from GitHub:

- *install.packages("devtools")* - only run this if you don't already have devtools installed.
- *library(devtools)* - loads the devtools package
- *install_github("author/package")* replacing "author" and "package" with their GitHub username and the name of the package.

Loading packages
Installing a package does not make its functions immediately available to you. First you must load the package into R; to do so, use the *library()* function. For example:

```
library(ggplot2)
```

NOTE: Do not put the package name in quotes! Unlike when you are installing the packages, the *library()* command does not accept package names in quotes!

Checking what packages you have installed
installed.packages() or *library()* with nothing between the parentheses to check.

Updating packages
old.packages() identify packages that have been updated since last updated.
To update all packages, use update.packages(). If you only want to update a specific package, use install.packages("packagename")

Unloading packages
To unload a given package you can use the detach() function. For example, *detach("package:ggplot2", unload=TRUE)*

Uninstalling packages
If you no longer want to have a package installed, you can simply uninstall it using the function remove.packages(). For example, *remove.packages("ggplot2")*

How do you know what version of R you have?
Type *version* into the console and it will output information on the R version you are running.
sessionInfo() - will tell you what version of R you are running along with a listing of all of the packages you have loaded. The output of this command is a great detail to include when posting a question to forums - it tells potential helpers a lot of information about your OS, R, and the packages that you are using.

What commands are included in a package?
You can use the *help()* function to access a package's help files.

```
help(package = "ggplot2")
```

If you still have questions about what functions within a package are right for you or how to use them, many packages include "vignettes." These are extended help files, that include an overview of the package and its functions, but often they go the extra mile and include detailed examples of how to use the functions in plain words that you can follow along with to see how to use the package. To see the vignettes included in a package, you can use the *browseVignettes()* function.
Eg: *browseVignettes("ggplot2")*

You should see that there are two included vignettes: "Extending ggplot2" and "Aesthetic specifications." Exploring the Aesthetic specifications vignette is a great example of how vignettes can be helpful, clear instructions on how to use the included functions.

Week 3:

Version control (Git & GitHub) Vocabulary

- **Repository** online folder that contains the whole, shared project Commit to save your changes locally, and add explanations of what has been changed.
- **Push** upload changes made locally to the repository, and merge them.
- **Pull** download the most recent changes from the repository and merge with your local folder.
- **Staging** to commit files separately and add separate explanations to each section edited, in order to simplify looking at the change log.
- **Branch** your local files and edits, before you push them.
- **Merge** independent branches are joined together into a single file. May need manual control if the same section of the same file was edited differently in both branches.
- **Conflict** when branches have been edited in the same section and file, making automatic merging hard.
- **Clone** make a local copy of a whole Git repository's files and all the change history.
- **Fork** make a personal copy of a repository that doesn't affect the original.

Best Practices:

- Purposeful, single issue commits
- Informative commit messages
- Pull and Push often
- Git username Rodrupp
- Git email rodrupp@gmail.com
- Git password: automatically generated on firefox.

Linking an Existing Project with GitHub Open Git Bash or Terminal and navigate to the directory containing your project files. Move around directories by typing cd ~/dir/name/of/path/to/file
Once here, type *git init* followed by *git add .* -> this initializes (init) this directory as a git repository and adds all of the files in the directory (.) to your local repository. Commit these changes to the git repository using *git commit -m "Initial commit"*

Go to GitHub.com, and again, create a new repository:

- 1) Make sure the name is the exact same as your R project;
- 2) Do NOT initialize a README file, .gitignore, or license.

Find *"...push an existing repository from the command line"* and copy to clipboard. Paste these into the Terminal, then reload the repository page.

Week 4

R Markdown Example

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

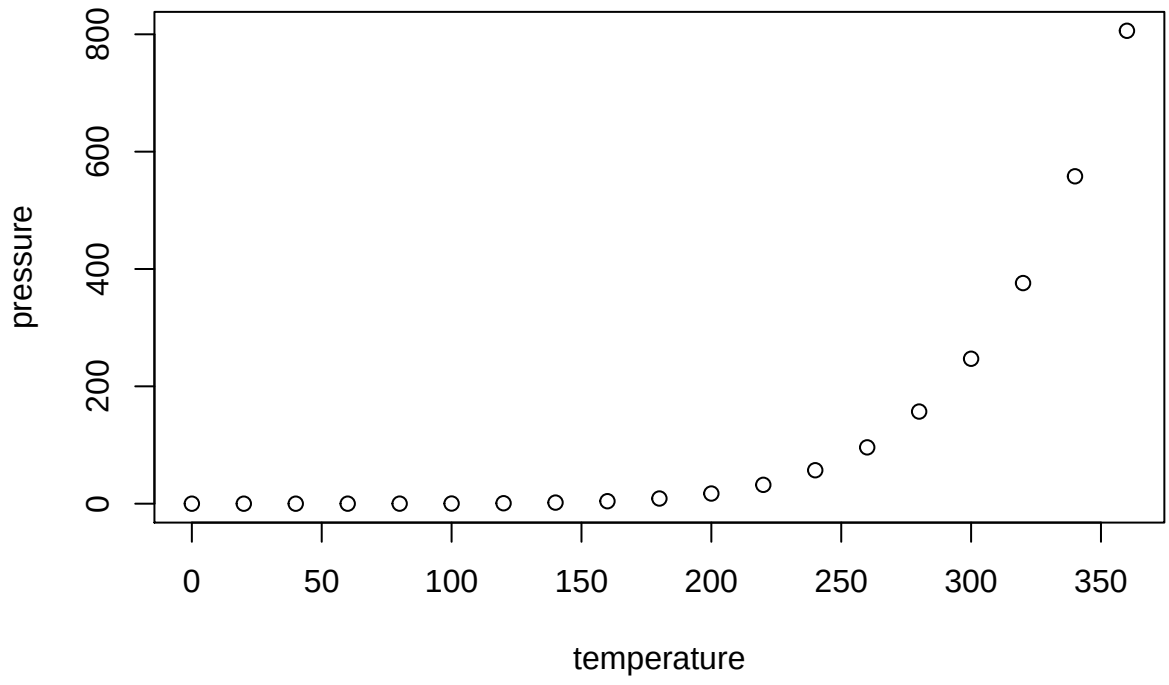
When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

```
##      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
```

Including Plots

You can also embed plots, for example:



Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.

R Markdown fromatting

To bold text use **bold**.

to make it italics use *italics*.

to strikethrough text use ~~strikethrough~~

Header biggest

Second biggest

third heading.

```
print("Testing some code here.")
```

```
## [1] "Testing some code here."
```

```
print("testing the keyboard shortcut ctrl+alt+i")
```

```
## [1] "testing the keyboard shortcut ctrl+alt+i"
```

```
print("testing the insert button on RStudio")
```

```
## [1] "testing the insert button on RStudio"
```

```
print("testing multiple lines of code.")
```

```
## [1] "testing multiple lines of code."
```

```
print("and here is the third line")
```

```
## [1] "and here is the third line"
```

Use Ctrl+enter to run a line of code in RStudio without having to knit the document.
Use Ctrl+shift+enter to run all lines in a chunk.

Making bulletted lists:

- bullets must finish with two spaces
- this is an example
- i wonder what happens if i don't include two spaces?
- ok, here goes:
- did it work?

This is a link to google

Types of Data Analysis

1) Descriptive
2) Exploratory
3) Inferential
4) Predictive
5) Causal
6) Mechanistic

Descriptive Describes or summarises a set of data. Often the first analysis to perform. eg mean, standard deviation, etc.

Exploratory Analysis Examine data and find new relationships. Help formulate hypotheses and drive future studies.

Inferential Analysis Use a small sample of data do infer something about a larger population provides estimate and uncertainty. Depends on your sampling.

Predictive Analysis Use current data and historical data to make predictions. Dependent on measuring the right variables. More data and simple model is usually better.

Causal Analysis See what happens to one variable when we manipulate another. Gold standard in data analysis. Often applied to randomized studies designed to identify causation. Often challenging to obtain data.

Mechanistic Analysis Understant the exact changes in variables that lead to exact changes in other variables. Less common and more complex. Applied to simple situation or those that are nicely modeled by deterministic equations (eg physics or engineering).

Experimental Design

Experimental design is organizing an experiment, so that you have the correct data and enough of it to clearly and effectively answer your data science question.

In this order:

1) Formulate your question before collecting any data
2) Design the experiment.
3) Identify problems.
4) Collect data.

And then a bunch of stuff I already know...

Big Data

Volume, Velocity, Variety, Unstructured

Examples: Text files and documents, Websites and applications, Sensor data, Image Files, Audio files, Video files, Email Data, Social Media data.

Before you can start looking for answers, you need to turn your unstructured data into a format that you can analyze.

R Console Input

Setting values

```
x <- 1  ##sets the value of x to 1
x  ## prints the value of x
```

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

```
msg <- "hello"  ##gives msg the text "hello"
msg ## prints the text set to msg
```

```
## [1] "hello"
```

You can also set calculations into the value

```
x <- x+1 ##Increases the value of x by one.
y <- 5
z <- x+y
print(z)
```

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

The result of the expression may be auto printed.

```
x <- 5 ## nothing pinter
x  ## auto-printing occurs
```

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

```
print(x)  ## explicit printing
```

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

A vector may also be set for a variable

```
x <- 1:20  ## creates an integer sequence from 1 to 20
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Calculations can also be applied on vectors

```r
x <- 1:20
y <- x*3
y
```

```
##  [1]  3  6  9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60
```

Objects and Attributes

R has 5 basic or "atomic"classes of objects
- character (text)
- numeric (real numbers, ie decimal numbers)
- integer
- complex
- logical (true/false)

The most basic object is a vector.
- Vectors can contain only objects of the same class
- except *lists*

Empty vectors can be created with the vector() function. It has 2 arguments: the class of the object and the length of the vector.

Numbers

Typically treated as numeric objects (Even if "1" is entered, it is understood as "1.0000000000"). You can enter 1L to explicitly make it an integer.

Inf can be used as a number for infinity. *-Inf* also possible
NaN can be used as a missing or undefined value, e.g 0/0

Attributes

Objects can have attributes such as:
- names, dimnames
- dimensions (eg matrices and arrays)
- class
- length (the length of a vector is the number of elements in the vector)
- other, user-defined attributes

Attributes can be accessed using the *attributes()* function.

Vectors and Lists

The c() function can be used to create vectors. c can stand for concatonate.

```r
x <- c(0.5, 0.6)  ## numeric
y <- c(T, F)   ## logical
z <- c("a", "b", "c")  ## character
w <- 9:29  ## integer
v <- c(1+0i, 2+4i)  ## complex

print(c(x,y,z,w,v))
```

```
##  [1] "0.5"   "0.6"   "TRUE"  "FALSE" "a"     "b"     "c"     "9"     "10"
## [10] "11"    "12"    "13"    "14"    "15"    "16"    "17"    "18"    "19"
## [19] "20"    "21"    "22"    "23"    "24"    "25"    "26"    "27"    "28"
## [28] "29"    "1+0i"  "2+4i"
```

Or use vector function:

```
x <- vector("numeric", length = 10)
x
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

By default for numeric vectors, the value is 0.

Mixing Objects

Objects can be forced into a different class. You won't get an error message.

```
x <- c(1.7, "a")  ## all will be treated as characters / text
x
```

```
## [1] "1.7" "a"
```

```
y <- c(TRUE, 2)  ## all will be treated as numeric. T = 1, F = 0
y
```

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

```
z <- c("a", TRUE)  ## all will be treated as characters / text
z
```

```
## [1] "a"    "TRUE"
```

You can explicitly coerce objects into another class.

```
x <- 0:6
class(x)
```

```
## [1] "integer"
```

```
as.numeric(x)
```

```
## [1] 0 1 2 3 4 5 6
```

```
as.logical(x)
```

```
## [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

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

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

```r
as.complex(x)
```

```
## [1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```

Nonsensical coercion results in NAs

```r
x <- c("a", "b", "c")
as.numeric(x)
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA NA NA
```

```r
as.logical(x)
```

```
## [1] NA NA NA
```

```r
as.complex(x)
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA NA NA
```

Lists

Can contain objects of different classes. Elements are encased in double brackets.

```r
x <- list(1, "a", TRUE, 1+4i)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

Matrices

Are vectors with a *dimension* attribute.

```r
m <- matrix(1:6, nrow = 2, ncol = 3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```r
dim(m)    ## gives dimensions of matrix, rows and columns
```

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

```r
attributes(m)   ## shows attribute dim
```

```
## $dim
## [1] 2 3
```

Matrices get constructed column-wise, top to bottom.

Matrices can be made from normal vectors

```r
m <- 1:10
m
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
dim(m) <- c(2,5)  ## adds the dimension attribute to it, while assigning a value to
 ## the dim attribute of m: the number of rows and columns.
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

Making matrices by column binding or row binding:

```r
x <- 1:3
y <- 10:12
cbind(x, y)
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```r
rbind(x, y)
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

Factors

Used to represent categorical data. Ordered or unordered. You could think of a factor as an integer vector where each integer has a label. Eg: 1 - high; 2 - medium; 3 - low. They are treated specially by modeling functions like *lm()* and *glm()*. They are better than using integers because they are self-descriptive.

```
x <- factor(c("yes", "yes", "no", "yes", "no"))
x
```

```
## [1] yes yes no  yes no
## Levels: no yes
```

```
table(x)  ## counts how many of each level
```

```
## x
##  no yes
##   2   3
```

```
unclass(x)  ## removes the class of the vector, shows integer values
```

```
## [1] 2 2 1 2 1
## attr(,"levels")
## [1] "no"  "yes"
```

The order of the levels of the factor is set in alphabetical order. between "yes" and "no", No will be the baseline level. You can explicitly change this with the *level()* function.

```
x <- factor(c("yes", "yes", "no", "yes", "no"),
            levels = c("yes", "no"))
x
```

```
## [1] yes yes no  yes no
## Levels: yes no
```

Missing Values

Denoted by NA (or NaN for undefined mathematical operations).
- *is.na()* and *is.nan()* used to test objects.
- NA values can have classes also, eg Integer NA, character NA, etc.
- NaN value is also NA, but converse is not true

```
x <- c(1, 2, NA, 10, 3)
is.na(x)
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE
```

```
is.nan(x)
```

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

```r
x <- c(1, 2, NaN, NA, 4)
is.na(x)
```

```
## [1] FALSE FALSE  TRUE  TRUE FALSE
```

```r
is.nan(x)
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE
```

Data Frames

Used to store tabular data. - represented as a special type of list, but each element of the list has to have the same length.
- each element of the list is a column, and the length of each element is the number of rows.
- they can store different classes of objects in each column, which makes them different from matrices.
- special attribute called *row.names*
- usually created with *read.table()* or *read.csv()* or *data.frame()*
- can be converted into matrices with *data.matrix()* (will coerce objects into being the same class)

```r
x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
x
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

```r
nrow(x)
```

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

```r
ncol(x)
```

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

Names Attribute

Important for writing readable code and self describing objects.

```r
x <- 1:3
names(x)
```

```
## NULL
```

```r
names(x) <- c("foo", "bar", "norf")
x
```

```
##  foo  bar norf
##    1    2    3
```

```
names(x)
```

```
## [1] "foo"  "bar"  "norf"
```

List names and matrix names:

```
x <- list(a = 1, b = 2, c = 3)
x
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
```

```
m <- matrix(c("BB", "Bb", "Bb", "bb"), nrow = 2, ncol = 2)
dimnames(m) <- list(c("B", "b"), c("B", "b"))  ##row names, then column names
m
```

```
##   B    b
## B "BB" "Bb"
## b "Bb" "bb"
```

Reading Tabular Data

Principal functions
- *read.table()*, *read.csv()* read text files in rows and columns and return a data frame in R
- *readLines()* reads lines of a text file, gives a character vector
- *source()* read R code files
- *dget()* read R files: objects that are "deparsed"
- *load()* reading binary objects into R
- *unserialize()* also for binary objects

Analogous functions for writing data, in order:
- *write.table()*
- *writeLines()*
- *dump()*
- *dput()*
- *save()*
- *serialize()*

Reading files with read.table()

Common arguments:
- *file()* name of file / path to file
- *header()* indicates if the first line is a header or not
- *sep()* indicates how columns are separated

- *colClasses()* character vector of the same length as number of columns. Indicates class of each column (can be different for each column).
- *nrows()* number of rows (not required)
- *comment.char()* indicates what character indicates comments
- *skip()* skips a number of lines
- *stringsAsFactors()* default=true, are character variables encoded as factors?

for moderately sized datasets, you can call read.table without any arguments other than *file()*. R will automatically:
- skip lines that begin with #
- find number of rows
- classify variables

However, giving R these arguments will make it runs more smoothly and predictably. *read.csv()* is an identical command except that the default separator is a comma instead of a space. CSV files are common outputs from excel files and many other programs.

Reading Larger Datasets with read.table

Things to do to facilitate reading.
- Read the Help page carefully for read.table - Make a rough calculation of the memory required to store your dataset. Is it larger than the RAM on your computer? If so don't proceed.
- Set *comment.char = " "* if there are no commented lines in your file.
- use the colClasses argument. Specifying this option instead of using the default. R won't need to try to figure it out. If all classes are the same class you can set them all with *colClasses = "numeric"* for example. Alternatively, use the first 100 rows and let read.table find the classes, then apply them to reading the whole text:

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)  ## S apply calls the class of each column
tabAll <- read.table("datatable.txt", colClasses = classes)
```

- set *nrows* to help reduce memory usage. a mild overestimate is okay.

It's useful to know a few things:
- How much memry is available?
- what other applications are in use?
- are other users using resources of the computer?
- what operating system are you using? is it 32 or 64 bit?

Calculating Memory Requirements

Consider a data frame with 1,500,00 rows and 120 columns, all of which are numeric data. How much memory is required?

each numeric object required 8 bytes, stored in 64 bits of data.

1,500,00 x 120 x 8 bytes/numeric = 1,440,000,000 bytes
1,440,000,000 / (2^20 bytes / MB) = 1,373.29 MB
= 1.34 GB

This is the RAW data. A bit more is required to read the data: rule of thumb is you require double the memory as the raw data.