June Workshop - Most Useful R Functions + Minimal Examples

R-Ladies Coding Club (London)

15 June '16

> Use this document to follow along during the session, published version on RPubs: http://rpubs.com/crt34/june_workshop (http://rpubs.com/crt34/june_workshop)

> This is *NOT* intended to be fully comprehensive list of every useful R function that exists, but is a practical demonstration of selected relevant examples presented in user-friendly format, all available in base R. For a wider collection to work through, this Reference Card is recommended: https://cran.r-project.org/doc/contrib/Baggott-refcard-v2.pdf (https://cran.r-project.org/doc/contrib/Baggott-refcard-v2.pdf)

> Additional CRAN reference cards and R guides (including non-English documentation) found here: https://cran.r-project.org/other-docs.html (https://cran.r-project.org/other-docs.html)

Schedule

6.30pm | A. Essentials

- 1. `getwd()` , `setwd()`
- 2. `?foo` , `help(foo)` , `example(foo)`
- 3. `install.packages("foo")` , `library("foo")`
- 4. `devtools::install_github("username/packagename")`
- 5. `data("foo")`
- 6. `read.csv` , `read.table`
- 7. `write.table()`
- 8. `save()` , `load()`

6.50pm | B. Basics

- 9. `c()` , `cbind()` , `rbind()` , `matrix()`
- 10. `length()` , `dim()`
- 11. `sort()` , `'vector'[]` , `'matrix'[]`
- 12. `data.frame()` , `class()` , `names()` , `str()` , `summary()` , `View()` , `head()` , `tail()` , `as.data.frame()`

7:15pm | Break

7:30pm | C. Core

- 13. `df[order(),]`
- 14. `df[,c()]` , `df[which(),]`
- 15. `table()`
- 16. `mean()` , `median()` , `sd()` , `var()` , `sum()` , `min()` , `max()` , `range()`
- 17. `apply()`
- 18. `lapply()` using `list()`
- 19. `tapply()`

7:50pm | D. Common

- 20. `if` statement, `if...else` statement
- 21. `for` loop
- 22. `function()...`

8:15pm | Close

"There's more than one way to skin a cat."

REMEMBER: KEY R LANGUAGE SYNTAX

- **Case Sensitivity**: as per most UNIX based packages, R is case sensitive, hence `X` and `x` are different symbols and would refer to different variables.
- **Expressions vs Assignments**: and expression, like `3 + 5` can be given as a command which will be evaluated and the value immediately printed, but not stored. An assignment however, like `sum <- 3 + 5` using the assignment operator `<-` also evaluates the expression `3 + 5` , but instead of printing and not storing, it stores the value in the object `sum` but doesn't print the result. The object `sum` would need to be called to print the result.
- **Reserved Words**: choice for naming objects is almost entirely free, except for these reserved words: https://stat.ethz.ch/R-manual/R-devel/library/base/html/Reserved.html (https://stat.ethz.ch/R-manual/R-devel/library/base/html/Reserved.html)
- **Spacing**: outside of the function structure, spaces don't matter, e.g. `3+5` is the same as `3+        5` is the same as `3 + 5` . For more best-practices for R code Hadley Wickham's Style Guide is a useful reference: http://adv-r.had.co.nz/Style.html (http://adv-r.had.co.nz/Style.html)
- **Comments**: add comments within your code using a hastag, `#` . R will ignore everything to the right of the hashtag within that line

---

A. SOME ESSENTIAL FUNCTIONS

1. Working Directory management - `getwd()` , `setwd()`

R/RStudio is always pointed at a specific directory on your computer, so it's important to be able to check what's the current directory using `getwd()` , and to be able to change and specify a different directory to work in using `setwd()` .

```
#check the directory R is currently pointed at
getwd()

#specify a directory for R to point at - MAC
setwd("/Users/User Name/Documents/FOLDER") #from the root directory
setwd("~/Documents/FOLDER") #or from R's default working directory where "~/" refe
rs to "/Users/User Name"

#specify a directory for R to point at - Windows
setwd("C:/Users/User Name/Documents/FOLDER") #from the root directory
setwd("~/FOLDER") #or from R's default working directory where "~/" refers to "C:/
Users/User Name/Documents"
```

2. Bring up help documentation & examples - `?foo` , `help(foo)` , `example(foo)`

```
?boxplot
help(boxplot)
example(boxplot)
```

3. Load & Call CRAN Packages - `install.packages("foo")` , `library("foo")`

Packages are add-on functionality built for R but not pre-installed (base R), hence you need to install/load the packages you want yourself. The majority of packages you'd want have been submitted to and are available via CRAN. At time of writing, the CRAN package repository featured 8,592 available packages.

```
#AIM: load & call the "wordcloud" package from CRAN
install.packages("wordcloud")

library("wordcloud") #best-practice loading
require("wordcloud") #alternative function to 'library', but don't use it (http://
yihui.name/en/2014/07/library-vs-require/)
```

4. Load & Call Packages from GitHub - `devtools::install_github("username/packagename")`

e.g. to install the `shinyapps` package available from RStudio's GitHub account

```
#AIM: load & call the "shinyapps" package from RStudio's GitHub
install.packages("devtools") #pre-requisite for `devtools...` function

devtools::install_github("rstudio/shinyapps") #install specific package from speci
fic GitHub account

library("shinyapps") #Call package
```

5. Load datasets from base R & Loaded Packages - `data("foo")`

```
#AIM: show available datasets
data()

#AIM: load an available dataset
data("iris")
```

6. I/O Loading Existing Local Data - `read.csv`, `read.table`

(a) I/O When already in the working directory where the data is

Import a local **csv** file (i.e. where data is separated by **commas**), saving it as an object:

```
#this will create a data frame called "object"
#the header argument is defaulted to TRUE, i.e. read.csv assumes your file has a h
eader row and will take the first row of your csv to be the column names
object <- read.csv("xxx.csv")

#if your csv does not have a header row, add header = FALSE to the command
#in this call default column headers will be assigned which can be changed
object <- read.csv("xxx.csv", header = FALSE)
```

Import a local tab delimited file (i.e. where data is separated by **tabs**), saving is as an object:

```
#this will create a data frame called "object"
#the header argument is defaulted to TRUE, i.e. read.csv assumes your file has a h
eader row and will take the first row of your csv to be the column names
object <- read.table("xxx.txt", sep = "\t")

#if your csv does not have a header row, add header = FALSE to the command
#in this call default column headers will be assigned which can be changed
object <- read.table("xxx.txt", sep = "\t", header = FALSE)
```

(b) I/O When NOT in the working directory where the data is

For example to import and save a local **csv** file from a different working directory you either need to specify the file path (operating system specific), e.g.:

```
#on a mac
object <- read.csv("~/Desktop/R/data.csv")

#on windows
object <- read.csv("C:/Desktop/R/data.csv")
```

OR

You can use the file.choose() command which will interactively open up the file dialog box for you to browse and select the local file, e.g.:

```
object <- read.csv(file.choose())
```

(c) I/O Copying & Pasting Data

For relatively small amounts of data you can do an equivalent copy paste (operating system specific):

```
#on a mac
object <- read.table(pipe("pbpaste"))

#on windows
object <- read.table(file = "clipboard")
```

(d) I/O Loading Non-Numerical Data - character strings

Be careful when loading text data! R may assume character strings are statistical factor variables, e.g. “low”, “medium”, “high”, when are just individual labels like names. To specify text data NOT to be converted into factor variables, add `stringsAsFactor = FALSE` to your `read.csv/read.table` command:

```
object <- read.table("xxx.txt", stringsAsFactors = FALSE)
```

(e) I/O Downloading Remote Data

For accessing files from the web you can use the same `read.csv/read.table` commands. However, the file being downloaded does need to be in an R-friendly format (maximum of 1 header row, subsequent rows are the equivalent of one data record per row, no extraneous footnotes etc.). Here is an example downloading an online csv file of coffee harvest data used in a Nature study:

```
object <- read.csv("http://sumsar.net/files/posts/2014-02-04-bayesian-first-aid-on
e-sample-t-test/roubik_2002_coffe_yield.csv")
```

7. I/O Exporting Data Frame - `write.table()`

Navigate to the working directory you want to save the data table into, then run the command (in this case creating a tab delimited file):

```
write.table(object, "xxx.txt", sep = "\t")
```

8. I/O Saving Down & Loading Objects - `save()` , `load()`

These two commands allow you to save a named R object to a file and restore that object again. Navigate to the working directory you want to save the object in then run the command:

```
save(object, file = "xxx.rda")

#reload the object
load("xxx.rda")
```

B. SOME BASIC FUNCTIONS

9. Vector & Matrix Construction - `c()` , `cbind()` , `rbind()` , `matrix()`

Vectors (lists) & Matrices (two-dimensional arrays) are very common R data structures.

```
#use c() to construct a vector by concatenating data
foo <- c(1, 2, 3, 4) #example of a numeric vector
oof <- c("A", "B", "C", "D") #example of a character vector
ofo <- c(TRUE, FALSE, TRUE, TRUE) #example of a logical vector

#use cbind() & rbind() to construct matrices
coof <- cbind(foo, oof) #bind vectors in column concatenation to make a matrix
roof <- rbind(foo, oof) #bind vectors in row concatenation to make a matrix

#use matrix() to construct matrices
moof <- matrix(data = 1:12, nrow=3, ncol=4) #creates matrix by specifying set of v
alues, no. of rows & no. of columns
```

10. Vector & Matrix Explore - `length()` , `dim()`

```
length(foo) #length of vector

dim(coof) #returns dimensions (no. of rows & columns) of vector/matrix/dataframe
```

11. Vector & Matrix Sort & Select - `sort()` , `'vector'[]` , `'matrix'[]`

```
#create another numeric vector
jumble <- c(4, 1, 2, 3)
sort(jumble) #sorts a numeric vector in ascending order (default)
sort(jumble, decreasing = TRUE) #specify the decreasing arg to reverse default ord
er

#create another character vector
mumble <- c( "D", "B", "C", "A")
sort(mumble) #sorts a character vector in alphabetical order (default)
sort(mumble, decreasing = TRUE) #specify the decreasing arg to reverse default ord
er

jumble[1] #selects first value in our jumble vector
tail(jumble, n=1) #selects last value
jumble[c(1,3)] #selects the 1st & 3rd values
jumble[-c(1,3)] #selects everything except the 1st & 3rd values

coof[1,] #selects the 1st row of our coof matrix
coof[,1] #selects the 1st column
coof[2,1] #selects the value in the 2nd row, 1st column
coof[,"oof"] #selects the column named "oof"
coof[1:3,] #selects columns 1 to 3 inclusive
coof[c(1,2,3),] #selects the 1st, 2nd & 3rd rows (same as previous)
```

12. Create & Explore Data Frames - `data.frame()` , `class()` , `names()` , `str()` , `summary()` , `View()` , `head()` , `tail()` , `as.data.frame()`

A data frame is a matrix-like data structure made up of lists of variables with the same number of rows, which can be of differing data types (numeric, character, factor etc.) - matrices must have columns all of the same data type.

```
#create a data frame with 3 columns with 4 rows each
doof <- data.frame("V1"=1:4, "V2"=c("A","B","C","D"), "V3"=5:8)

class(doof) #check data frame object class
names(doof) # returns column names
str(doof) #see structure of data frame
summary(doof) #returns basic summary stats
View(doof) #invokes spreadsheet-style viewer
head(doof, n=2) #shows first parts of object, here requesting the first 2 rows
tail(doof, n=2) #shows last parts of object, here requesting the last 2 rows

convert <- as.data.frame(coof) #convert a non-data frame object into a data frame
```

C. SOME CORE FUNCTIONS

13. Data Frame Sort - `df[order(),]`

```
#use 'painters' data frame
library("MASS") #call package with the required data
data("painters") #load required data
View(painters) #scan dataset

#syntax for using a specific variable: df=data frame, '$', V1=variable name
df$V1

#AIM: print the 'School' variable column
painters$School

#syntax for df[order(),]
df[order(df$V1, df$V2...),] #function arguments: df=data frame, in square brackets
specify within the order() the columns with which to sort the ROWS by, where defau
lt ordering is Ascending, the tailing comma specifies returning all the columns in
the df. If only certain columns are wanted this can be specified after the comma.

#AIM: order the dataset rows based on the painters' Composition Score column, in A
scending order
painters[order(painters$Composition),] #Composition is the sorting variable

#AIM: order the dataset rows based on the painters' Composition Score column, in D
escending order
painters[order(-painters$Composition),] #append a minus sign in front of the varia
ble you want to sort by in Descending order

#AIM: order the dataset rows based on the painters' Composition Score column, in D
escending order but return just the first 3 columns
painters[order(-painters$Composition), c(1:3)]
```

14. Data Frame Select & Deselect - `df[,c()]`, `df[which(),]`

```
#use 'painters' data frame

#syntax for select & deselect based on column variables
df[, c("V1", "V2"...)] #function arguments: df=data frame, in square brackets spec
ify columns to select or deselect. The comma specifies returning all the rows. If
certain rows are wanted this can be specified before the comma.

#AIM: select the Composition & Drawing variables based on their column name
painters[, c("Composition", "Drawing")] #subset the df, selecting just the named c
olumns (and all the rows)

#AIM: select the Composition & Drawing variables based on their column positions i
n the painters data frame
painters[, c(1,2)] #subset the df, selecting just the 1st & 2nd columns (and all t
he rows)

#AIM: drop the Expression variable based on it's column position in the painters d
ata frame and return just the first 5 rows
painters[c(1:5), -4] #returns the subsetted df having deselected the 4th column, E
xpression and the first 5 rows


#syntax for select & deselect based on row variable values
df[which(),] #df=data frame, specify the variable value within the `which()` to su
bset the df on. Again, the tailing comma specifies returning all the columns. If c
ertain columns are wanted this can be specified after the comma.

#AIM: select all rows where the painters' School is the 'A' category
painters[which(painters$School == "A"),] #returns the subsetted df where equality
holds true, i.e. row value in School variable column is 'A'

#AIM: deselect all rows where the painters' School is the 'A' category, i.e. retur
n df subset without 'A' values, AND also only select rows where Colour score > 10
painters[which(painters$School != "A" & painters$Colour > 10),] #returns the subse
tted df where equality holds true, i.e. row value in School variable column is 'no
t A', AND the Colour score filter is also true.
```

15. Data Frame Frequency Calculations - `table()`

```
#create new data frame
flavour <- c("choc", "strawberry", "vanilla", "choc", "strawberry", "strawberry")
gender <- c("F", "F", "M", "M", "F", "M")
icecream <- data.frame(flavour, gender) #icecream df made up of 2 factor variables
, flavour & gender, with 3 & 2 levels respectively (choc/strawberry/vanilla & F/M)

#AIM: create a frequency distribution table which shows the count of each gender i
n the df
table(icecream$gender)

#AIM: create a frequency distribution table which shows the count of each flavour
in the df
table(icecream$flavour)

#AIM: create Contingency/2-Way Table showing the counts for each combination of fl
avour & gender level
table(icecream$flavour, icecream$gender)
```

16. Descriptive/Summary Stats Functions - `mean()` , `median()` , `sd()` , `var()` , `sum()` , `min()` , `max()` , `range()`

```
#re-using the jumble vector from before
jumble <- c(4, 1, 2, 3)

mean(jumble)
median(jumble)
sd(jumble)
var(jumble)
sum(jumble)
min(jumble)
max(jumble)
range(jumble)
```

17. Apply Functions - `apply()`

`apply()` returns a vector, array or list of values where a specified function has been applied to the 'margins' (rows/cols combo) of the original vector/array/list.

```
#re-using the moof matrix from before
moof <- matrix(data = 1:12, nrow=3, ncol=4)

#apply syntax
apply(X, MARGIN, FUN,...) #function arguments: X=an array, MARGIN=1 to apply to ro
ws/2 to apply to cols, FUN=function to apply

#AIM: using the moof matrix, apply the sum function to the rows
apply(moof, 1, sum)

#AIM: using the moof matrix, apply the sum function to the columns
apply(moof, 2, sum)
```

18. Apply Functions - `lapply()` using `list()`

A list, a common data structure, is a generic vector containing objects of any types. `lapply()` returns a list where each element returned is the result of applying a specified function to the objects in the list.

```
#create list of various vectors and matrices
bundle <- list(moof, jumble, foo)

#lapply syntax
lapply(X, FUN,...) #function arguments: X=a list, FUN=function to apply

#AIM: using the bundle list, apply the mean function to each object in the list
lapply(bundle, mean)
```

19. Apply Functions - `tapply()`

`tapply()` applies a specified function to specified groups/subsets of a factor variable.

```
#tapply syntax
tapply(X, INDEX, FUN,...) #function arguments: X=an atomic object, INDEX=list of 1
+ factors of X length, FUN=function to apply

#AIM: calculate the mean Drawing Score of the painters, but grouped by School cate
gory
tapply(painters$Drawing, painters$School, mean) #grouping the data by the 8 differ
ent Schools, apply the mean function to the Drawing Score variable to return the 8
mean scores
```

D. SOME COMMON FUNCTIONS

20. Programming Tools - `if` statement, `if...else` statement

An `if` statement is used when certain computations are conditional and only execute when a specific condition is met - if the condition is not met, nothing executes. The `if...else` statement extends the `if` statement by adding on a computation to execute when the condition is not met, i.e. the ‘else’ part of the statement.

```
#if-statement syntax
if ('test expression')
    {
    'statement'
    }

#if...else statement
if ('test expression')
    {
    'statement'
    }else{
    'another statement'
    }

#AIM: here we want to test if the object, 'condition_to_test' is smaller than 10.
If it is smaller, another object, 'result_after_test' is assigned the value 'small
er'. Otherwise, the 'result_after_test' object is assigned the value 'bigger'

#specify the 'test expression'
condition_to_test <- 7

#write your 'if...else' function based on a 'statement' or 'another statement' dep
endent on the 'condition_to_test'.
if (condition_to_test > 5)
    {
    result_after_test = 'Above Average'
    }else{
    result_after_test = 'Below Average'
    }

#call the resulting 'statement' as per the instruction of the 'if...else' statemen
t
result_after_test
```

21. Programming Tools - `for` loop

A `for` loop is an automation method for repeating (looping) a specific set of instructions for each element in a vector.

```
#for loop syntax requires a counter, often called 'i' to denote an index
for ('counter' in 'looping vector')
    {
    'instructions'
    }

#AIM: here we want to print the phrase "In the Year yyyy" 6x, once for each year b
etween 2010 to 2015.
#this for loop executes the code chunk 'print(past("In the Year", i)) for each of
the 'i' index values
for (i in 2010:2015)
    {
    print(paste("In the Year", i))
    }

#AIM: create an object which contains 10 items, namely each number between 1 and 1
0 squared
#to store rather than just print results, an empty storage container needs to be c
reated prior to running the loop, here called container
container <- NULL
for (i in 1:10)
    {
    container[i] = i^2
    }

container #check results: the loop is instructed to square every element of the lo
oping vector, 1:10. The ith element returned is therefore the value of i^2, e.g. t
he 3rd element is 3^2.
```

22. Programming Tools - `function()...`

User-programmed functions allow you to specify customised arguments and returned values.

```
#AIM: to create a simplified take-home pay calculator (single-band), called 'takeh
ome_pay'. Our function therefore uses two arguments, a 'tax_rate', and an 'income'
level. The code in the curly braces {} instructs what the 'takehome_pay' function
should do when it is called, namely, calculate the tax owed in an object 'tax', an
d then return the result of the 'income' object minus the 'tax' object
takehome_pay <- function(tax_rate, income)
    {
    tax = tax_rate * income
    return(income - tax)
    }

takehome_pay(tax_rate = 0.2, income = 25000) #call our function to calculate 'take
home_pay' on a 'tax_rate' of 20% and an 'income' of 25k
```

E. FURTHER FUNCTIONS

23. Strings - `grep()`, `tolower()`, `nchar()`

24. Further Data Selection - `quantile()` , `cut()` , `which()` , `na.omit()` , `complete.cases()` , `sample()`

25. Further Data Creation - `seq()` , `rep()`

26. Other Apply-related functions - `split()` , `sapply()` , `aggregate()`

27. More Loops - `while` loop, `repeat` loop

…..Ad Infinitum!!

Happy Coding!!
