Getting Started with R

William Ou

5/23/2020

Why use R?

1. Clean and fast way of doing repetitive things
2. Reproducible (anyone with your script can do exactly what you did)
3. Easy calculations
4. Data wrangling: cleaning and organizing data
5. Data visualization
6. Tons of libraries for you to use - libraries are essentially functions that people have written already that you can use
7. A friendly community where everyone helps everyone!

Any thing else?

What is R Studio?

- an open-source Integrated Development Environment (IDE) for R
- helps you keep track of things

1. Code Editor - Basically your note pad or sketch pad for code
2. R Console - This is what R itself actually is – where your executed code goes to and where the output of those executed code shows, too
3. Workspace & History- Keeps track of all the variables you have assigned, data you have loaded, and code that you executed
4. Plots and files - The graphical device of your plots and shows where your files are (not too useful IMO)

R Studio interface

R syntax and defintions

- "<-" is used to assign a value or object to a variable

```r
a <- 3 # The value 3 is assigned to the variable a
a
```

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

```r
a + 1
```

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

```r
A #R is case sensitive!
```

```
## Error in eval(expr, envir, enclos): object 'A' not found
```

1. Operators

1a. Arithmetic: +, -, /, *

```
1+1
```

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

```
1-5
```

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

```
1+5/2 #Order of operations still holds!
```

```
## [1] 3.5
```

```
2*4-2
```

```
## [1] 6
```

1b. Logical or Boolean: >,<,<=,>=,==, !=

```
1>3 #greater than
```

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

```
1>=0.1 #greater than or equal to
```

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

```
1<=4 #less than or equal to
```

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

```
112==2 #equal to
```

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

```
15!=2 #not equal to
```

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

2. Class (or the "type" of object)

- This is usually the cause of all problems!
- Most functions can only accept a certain "class" of object as input. When it is not, you will get an error message
- Whenever you get an error message, check if your objects are the right class!

2a. Numeric - outputs of arithmetic expressions are numeric

```
class(5+1)
```

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

2b. Logical - output of logical/boolean expressions

- These are essentially binary outputs (ie. TRUE = 1 and FALSE = 0)
- In R, you can apply arithmetic operators to TRUE and FALSE

```r
class(5>2)
```

```
## [1] "logical"
```

```r
(5>2)+(1==1)
```

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

```r
class((5>2)+(1==1)) #
```

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

- this one is an integer, which generally works the same as numeric. The main difference has something to do with how the memory is stored in the machine… TLDR not important but watch out

2c. Character or "string"

- non-numerical "values" like words (but numerical values can be stored as charcaters too!)
- these values are specified with quotation marks "" or ''
- letters or words without "" marks are read by R as variables
- ***NOTE: characters are sometimes automatically converted to another class known as "factors", like categorical variables in an ANOVA***

```r
sentence <- "Hello, mortals."
print(sentence)
```

```
## [1] "Hello, mortals."
```

```r
class(sentence)
```

```
## [1] "character"
```

```r
print('sentence') #by adding '', the letters in senetence is a charcater object
```

```
## [1] "sentence"
```

```r
#and not the variable that I have previously assigned
class('5193')
```

```
## [1] "character"
```

```r
class(5193)
```

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

```r
'5123'+3 #common mistake/error !!
```

```
## Error in "5123" + 3: non-numeric argument to binary operator
```

2d. Factors

- factors are almost identical to characters, in fact, they are interchangeable sometimes
- are more commonly used in terms of data analysis
- differ from characters in that they have "levels", or unique entries in the vector of factor elements

```r
char <- c('a','a','a','b','d','d','d') #a character vector
char
```

```
## [1] "a" "a" "a" "b" "d" "d" "d"
```

```r
factor(char)
```

```
## [1] a a a b d d d
## Levels: a b d
```

- By default, factor levels will be sorted alphabetically (or numerically if the factors are numbers)
- This can be changed by including an additional argument in the ***factor()*** function:

```
new_fact <- factor(char)
factor(new_fact,levels=c('d','a','b'))
```

```
## [1] a a a b d d d
## Levels: d a b
```

Converting between classes

- you can use built-in function in R to convert between classes
- it usually looks something like as.*class to convert to* ()

```
v <- c(1,2,4,2)
as.factor(v) #to factor
```

```
## [1] 1 2 4 2
## Levels: 1 2 4
```

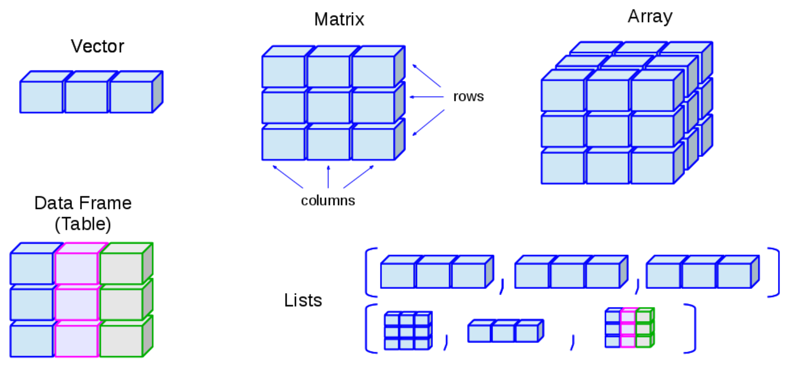
```
as.character(v)
```

```
## [1] "1" "2" "4" "2"
```

```
as.integer(v)
```

```
## [1] 1 2 4 2
```

Data structures



1. Vectors

- A series of n-elements in which all elements are of the same class
- Key here is that everything is of the same class!
- Becuase all elements are of the same class, the class() function returns the class of the objects and not a "vector" class (ie. there is no "vector" class)
- to create a vector, you can use c()

Indexing: ***vec[i]*** (ie. the *ith* element of vector *vec*)

```r
c(1,2,5,2) # vector of integers/numerical values
```

```
## [1] 1 2 5 2
```

```r
class(c(1,2,5,2))
```

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

```r
class(c(2,2,4,'bb')) #numerical values are forced as characters
```

```
## [1] "character"
```

```r
length(c(1,2,4,5,14)) #length() is a function that tells you the number of elements of your object
```

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

BONUS!! A cool thing about vectors: It makes things faster!

- You might hear people say that a solution or algorithm is "vectorized", and this is what they are talking about.

Say we have a numeric vector, *vec*:

```r
vec <- c(1,2,4,2,5)
vec
```

```
## [1] 1 2 4 2 5
```

Imagine we want to add 1 to all of the elements of *vec*. One way to do it is to iterate through each of the vector elements and add 1 to each iteration, like so:

```r
# for loop
system.time(for (i in 1:length(vec)){
  vec[i] <- vec[i]+1
  })
```

```
##    user  system elapsed
##   0.016   0.001   0.018
```

```r
vec
```

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

OR Option 2, we could just simply +1 to the whole vector

```r
system.time(vec <- vec+1)
```

```
##    user  system elapsed
##       0       0       0
```

```r
vec
```

```
## [1] 3 4 6 4 7
```

Option 2 is so fast that the decimals were not enough to show the time! Although Option 1 was pretty fast too (0.002s), these kind of computations will become extremely long to compute when you have a LARGE data set (eg. 10000+). Moreover, Option 2 is also just a more elegant piece of code!

2. Data frames

- This is probably what you will work with the most, so get familiar with it!
- The best way to describe a data frame is that it is a list of vectors, in which each column is a vector and the rows are the elements of the vector

- Following this description, elements within a column of a dataframe should always be of the same class while elements between columns do not necessarily have to!
- This brings up another important point: Data should **ALWAYS** be organized in the long format! Ie. Each column should represent a single variable and rows should be represent 1 some sample of that/those variable/s

Indexing: - Data frames are indexed by the notation *df[row,column]* (eg. row 1 column 3 of df is df[1,3]) - $ notation can be also used to index the columns of data frames

```
df <- data.frame(x=c(2,4,1,5,9),y=c('b','b','b','a','a'))
df
```

```
##   x y
## 1 2 b
## 2 4 b
## 3 1 b
## 4 5 a
## 5 9 a
```

```
df$x #outputs the vector x of data frame df
```

```
## [1] 2 4 1 5 9
```

```
class(df$x)
```

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

```
class(df$y)
```

```
## [1] "factor"
```

```
df[2,] #outputs row 2 of every column; empty column means all columns are selected
```

```
##   x y
## 2 4 b
```

```
df[,1] #same as df$x but using indices instead of the variable name
```

```
## [1] 2 4 1 5 9
```

What's wrong with this?

```
baddata
```

```
##                      site  Glasgow Guanacaste  Cape Town   Okinawa     Pasoh
## 1 population size (10^6) 366.4400  266.28000  319.20000 279.79000 374.35000
## 2            area (km^2) 954.0000 1040.00000 1040.00000 958.00000 935.00000
## 3             GDP (10^3)  40.6833   42.44315   39.12455  40.48914  38.83771
```

3. Matrices

- Are essentially multidimensional vectors
- Just like vectors, they can only contain values of one ***class*** at a time
- m x n * n x p = m x p matrix

```
set.seed(12)
mn <- matrix(rnorm(n=12,mean=4),nrow=4) #4*3 matrix
np <- matrix(rnorm(n=9,mean=10,sd=3),nrow=3) #3*3 matrix
```

```
mn
```

```
##          [,1]     [,2]     [,3]
## [1,] 2.519432 2.002358 3.893536
## [2,] 5.577169 3.727704 4.428015
## [3,] 3.043256 3.684651 3.222280
## [4,] 3.079995 3.371745 2.706118
```

```
np
```

```
##           [,1]      [,2]      [,3]
## [1,]  7.661300  7.889607 11.520905
## [2,] 10.035855 13.566637  9.120085
## [3,]  9.542751 11.021537 10.670924
```

```
mn%*%np #matrix multiplication
```

```
##           [,1]      [,2]      [,3]
## [1,]  76.55255  89.95535  88.83544
## [2,] 122.39451 143.37761 145.50202
## [3,]  91.04334 109.51290 103.05010
## [4,]  83.25892  99.86876  95.11170
```

4. Lists

- Lists are useful for storing ***groups*** of similar objects together
- These objects *CAN* be of different classes *(LIKE VERY DIFFERENT)*
- Can become very useful when you want to apply different functions to different sections of your data

```
vec <- c('how','now','brown','cow')
protime <- system.time(1+1)
mat <- matrix(rnorm(10),nrow=5)
df <- data.frame(color=c('red','blue','gree'),integers=c(2,5,1))

demolist <- list(vec,protime,mat,df)
demolist
```

```
## [[1]]
## [1] "how"   "now"   "brown" "cow"
```

```
##
## [[2]]
##    user  system elapsed
##       0       0       0
##
## [[3]]
##             [,1]       [,2]
## [1,]  2.0072015 -0.1991057
## [2,]  1.0119791  0.1311226
## [3,] -0.3024592  0.1457999
## [4,] -1.0252448  0.3620647
## [5,] -0.2673848  0.6739812
##
## [[4]]
##   color integers
## 1   red        2
## 2  blue        5
## 3  gree        1
```

Exercises

Excercise 1: Create a new vector from an existing one that has its elements shifted to the right (1st becomes 2nd and last becomes the first)

- if the original vector is c(5,3,1,2), return the vector c(2,5,3,1)
- There are multiple ways to do it but the BEST answer is one that is generalized and works for any vector
- HINT: try using length() and seq(); use ?seq (or ?length) to see what it does

Excercise 2: Use logical operators to check how many values in this numeric vector is *below* the value 10

```
set.seed(123)
num_vec <- rnorm(5000,30,10)
```

- the answer should be 117

Excercise 3: Multiply the values of vec that are >9 and <12 by 1000

```
answer
```

```
##   [1] 10333.828  9467.528  9857.895  9477.772 10072.515  9476.630  9292.489
##   [8] 10584.816 11444.283 10570.436 10614.953 10382.924  9215.107 10563.491
##  [15]  9623.175 10417.948 11136.748 11473.832 11397.724 10748.547 11515.272
##  [22]  9945.422  9813.113 11751.024 10655.559 10084.618 10059.212 11363.330
##  [29] 10313.384 10074.142 11483.083  9051.854 11185.685 10600.598  9063.606
##  [36]  9555.229 11650.800 11183.213 10350.177 10008.226 11931.070  9362.857
##  [43] 10329.123 10512.129  9091.081 10170.461 11771.127  9874.802 10810.482
##  [50] 11436.394 11394.124  9871.313 10541.479  9220.076 10599.308 10432.152
##  [57]  9938.800  9418.586 11564.275 10539.866  9767.689 10839.818  9701.429
##  [64] 11695.539 11645.633  9956.524 10965.536 11326.544 10592.945 11160.279
##  [71]  9409.146 10969.230 11019.601  9188.359 11755.714 10267.387 10514.730
##  [78] 10268.895 11852.929 11991.508  9821.807  9738.904  9763.730 11745.046
##  [85] 10036.728  9257.934 10023.727 10269.863 10791.150 11548.092  9251.438
##  [92] 10956.422 10148.711 10743.062 11194.260 11922.199 11172.455 10800.089
##  [99] 10336.615 11479.026 11751.691 11874.853 10530.505
```

Loading data

- There are numerous ways to load data, usually it depends on the format (.csv, .txt, .xlsx etc.) of data you are loading
- As much as possible, don't use excel because of formatting issues
- tab delimited files are the most stable
- you can directly load data by its name if you are working directory is in the same as where the file is
- or you can load it by providing the path of the file

```
dat <- read.csv('zooplankton_clean.csv')

dat <- read.csv('/Users/jiaangou/Desktop/RStudyGroup/zooplankton_clean.csv')

dat <- read.csv("https://www.dropbox.com/s/k3xvoi2vxg9p64k/zooplankton_clean.csv?dl=1")
```

Data manipulation with *tidyverse* package

- *tidyverse* is a series of packages which was created to make data science as "tidy" as possible
- Useful for quick and clean manipulation and visualization (ggplot) of complex data
- It has its own unique syntax, which is often more intuitive than base R (IMO)
- ***Problem***: Too many different packages and too many different functions that sometimes do the similar things. Also a lot of error
- Thankfully, there are a lot of cheatsheets and people are helpful

Install and loading library

```
install.packages('tidyverse')
library(tidyverse)
```

Practice data set: Zooplankton body sizes across 4 different lakes in BC

Load directly from dropbox:

```
zooplankton <- read.csv("https://www.dropbox.com/s/500av3zullmiti7/zooplankton.csv?dl=1")
```

- Inspect the data set first. Some helpful functions to help you do so: View(), head(), str()
- Make some easy plots: hist(), boxplot(y~x)
- If you inspect the genus variable more closely you might find some errors