Introduction to R

Stacey Borrego

5/08/2018

Purpose

The purpose of this document is to give a very brief and digestable introduction to R. There are so many details in every programming language and certainly more thorough explanations are needed than I will provide here. My goal with this document is to expose you to R and hopefully pique your interest to learn more.

Resources used for this document

- *Bioinformatics Data Skills: Reproducible and Robust Research with Open Source Tools* by Vince Buffalo (O'Reilly 2015)
- R Markdown Cheat Sheet

Some Notes about RStudio and R

RStudio

- RStudio provides an enviroment with helpful tools for you to use and build scripts in R. Your basic enviroment will have at least your Console and Source (basically a text editor for you to write your script/file).
- You can change the layout of your RStudio enviroment by going to Tools > Global Options > Pane Layout.
- Learn more about RStudio features.

R

- Some programming languages are very picky about spacing, but R is usually not. That being said, there are style conventions that are recommended. These conventions extend from spacing to naming and everything in between. Here is a Style Guide to get you started.

R Basics

Simple Math

Let's get started by writing some simple math expressions for R to evaluate.

You can run expressions in one of two ways, or both if you are curious!

1. Copy or type each expression into the RStudio Console. Hit Enter/Return.
2. Open a new R script (File > New File > R script).
    - Copy the expression into the new file.
    - Place your cursor on the line you want to run, or select everything.
    - Hit the run button on the top right. This will run the expression on the current line your cursor is on or whatever is selected.

- Keyboard Shortcut to run your script/expression: Command-Enter (OS X) or Control-Enter (Windows, Linux)

```
4 + 3
```

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

```
4 - 3
```

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

```
4 * 3
```

```
## [1] 12
```

```
4 / 3
```

```
## [1] 1.333333
```

To indicate the order of evaluation, use parentheses to indicate which expression should be evaluated first.

```
4 + 3/2
```

```
## [1] 5.5
```

```
(4 + 3)/2
```

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

We can also use functions to perform mathematical operations. Functions are written with the function name followed by parentheses, no spaces! A function takes zero or more arguments, evaluates the input, and outputs a return value.

To find the square root of a number, we can use the function sqrt().

```
sqrt(4)
```

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

```
sqrt(2*2)
```

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

Getting Help

To learn more about the arguments for a function you can use the function args(), just provide the function name as the agrument.

```
args(args)
```

```
## function (name)
## NULL
```

```
args(sqrt)
```

```
## function (x)
## NULL
```

```
args(plot)
```

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

If you want to read the documentation for a function, you can pull up a special help window in Rstudio.

```
help(sqrt)
?sqrt
```

Variables

Variables are a great way to save a value for future use. We can assign a value to a symbol using the <- assignment operator.

```
x <- 4
```

When we want to see what value our variable is assigned we can just type in the variable symbol into the console. You can also look at the Environment window in RStudio and it will show you all the variables you have assigned and their values.

```
x
```

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

Once our variable has a value assigned, we can use it in functions.

```
sqrt(x)
```

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

Variables can also be assigned the ouput of an expression.

```
square_result <- sqrt(x)
square_result
```

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

Vectors

R is most known for its use of vectors and vectorization. Everything is stored in a vector; a single value is stored in a vector of 1. You can see the length of a vector using the function length().

```
length(x)
```

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

We can create longer vectors using c(), which stands for concatenate.

```
y <- c(1, 5, 7)
y
```

```
## [1] 1 5 7
```

```
length(y)
```

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

Vectorization is the process by which a process is applied to a whole array instead of a single element. This means if the variable **y** were multiplied by 2, each element will be muliplied by 2 but the vector length will remain at 3.

```
y * 2
```

```
## [1]  2 10 14
```

```
length(y * 2)
```

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

Vectorization allows us to perform arithmetic operations on two vectors, each operation occurring elementwise (e.g. a1 + b1, a2 + b2, a3 + b3 . . .)

```
a <- 1:3
a
```

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

```
b <- 4:6
b
```

```
## [1] 4 5 6
```

```
a + b
```

```
## [1] 5 7 9
```

If the two vectors are not the same length, R will recycle the values of the shorter vector (e.g. c1 + d1, c2 + d2, c3 + d1, c4 + d2)

```
c <- 1:4
c
```

```
## [1] 1 2 3 4
```

```
d <- 1:2
d
```

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

```
c + d
```

```
## [1] 2 4 4 6
```

A warning will pop up if the longer vector length is not a multiple of the shorter vector length. The operation will still be performed but R thought it should say something.

```
e <- 1:5
e
```

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

```
f <- 1:2
f
```

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

```
e + f
```

```
## Warning in e + f: longer object length is not a multiple of shorter object
## length
```

```
## [1] 2 4 4 6 6
```

We can also provide vectors as an argument to a function. Depending on what the function does, your output will look differently. For example, sqrt() only takes the square root of one number at a time, so it returns a value for each number in the vector **y** – sqrt(1), sqrt(5), sqrt(7). However, mean() computes the average of a vector of numbers and thus returns a single value.

```
y
```

```
## [1] 1 5 7
```

```
sqrt(y)
```

```
## [1] 1.000000 2.236068 2.645751
```

```
mean(y)
```

```
## [1] 4.333333
```

Note: When you look up the arguments for a function, you will often see **x** given as the first argument. The documentation will provide more information as to what **x** is. For example when you look up the documentation for square root (?sqrt), **x** is defined as "a numeric or complex array". When you look at the documentation for mean (?mean), **x** is defined as "typically a vector-like object". Just don't confuse this with a variable that you may have set, it is just a placeholder.

Vector Types

Vectors in R must contain elements of the same type. We can check the vector type by using the function class() or typeof(). It is important to note that R will coerce vectors of different types to the type that leads to no information loss. For example, a vector containing both characters and numeric data will be coerced into character values.

Numeric (aka double)

- Any real number
- example: 4, -4, 4.4, -4.04
- *Test function*: is.numeric()
  - returns TRUE or FALSE value
- *Coercion function*: as.numeric()
  - coerces values to be numeric

```
num_example <- c(4, -4, 4.4, -4.04)
class(num_example)
```

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

Integer

- Any whole number
- example: 4, -4, 44
- By default, integer values are assigned as numeric. It must be explicitly indicated that the values should be treated as integers.
- *Test function*: is.integer()
- *Coercion function*: as.integer()
- Will sometimes be indicated with an L after the number.

```
int_example <- c(4, -4, 44)
class(int_example)
```

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

```
int_example <- as.integer(int_example)
class(int_example)
```

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

```
int_example2 <- 5
is.integer(int_example2)
```

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

```
int_example2 <- as.integer(5)
is.integer(int_example2)
```

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

```
int_example3 <- as.integer(5.9999)
int_example3
```

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

Character

- Character data represent text, which are called strings. Text data that is enclosed in either double or single quotes is interpreted as a string.
- example: "i am a string", "ABCD"
- *Test function*: is.character()
- *Coercion function*: as.character()

```
char_example <- c("a", "b", "c")
class(char_example)
```

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

```
num_example
```

```
## [1]  4.00 -4.00  4.40 -4.04
```

```
is.character(num_example)
```

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

```
char_example2 <- as.character(num_example)
char_example2
```

```
## [1] "4"     "-4"    "4.4"   "-4.04"
```

```
is.character(char_example2)
```

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

Logical

- Logical values represent Boolean values, a binary value having only two options. In R, Boolean values are TRUE or FALSE
- *Test function*: is.logical()
- *Coercion function*: as.logical()

```
log_example <- c(TRUE, FALSE)
is.logical(log_example)
```

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

```
num_example
```

```
## [1]  4.00 -4.00  4.40 -4.04
```

```
is.logical(num_example)
```

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

```
log_example2 <- as.logical(num_example)
log_example2
```

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

```
is.logical(log_example2)
```

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

Special Values

There are four special values in R that may cause problems in your analysis, whether you are aware of it or not. These values are NA, NULL, Inf/-Inf, and NaN.

NA

- "Not Available"
- NA represents missing data and any function performed on NA will result in NA
- A few ways to handle them:
  - is.na()
  - na.omit()
  - complete.cases()
  - na.rm = TRUE (this is an argument for a function)
  - sort(x, na.last = TRUE)

NULL

- "No value, none"
- Null represents not having a value, which is different than the missing data of NA
- A few ways to handle them:
  - is.null()

Inf/-Inf

- "Positive infinity, negative infinity"
- A few ways to handle them:
  - is.infite()
  - is.finite()

NaN

- "Not a Number"
- Values that are not numbers include: 0/0, infinity, negative infinity
- A few ways to handle them:
  - is.nan()
  - coerce all NaN values to NA using is.na()

Working with Data

Where am I?

You can find your current working directory by using getwd() and setting the working directory with setwd(). You will want to replace my path with the path of your choosing.

Using RStudio, in the Files tab you can navigate to a directory, select More, and then click Set as Working Directory.

```
getwd()
```

```
## [1] "/Users/stacey/Data/GitHub/Introduction-to-R"
```

```
setwd("/Users/stacey/Data/GitHub/Introduction-to-R")
```

Reading Data into R

The following uses the combined_out.txt file that was generated in the Read Counts meeting. Following the steps from the tutorial should generate the same file I have posted on the BioinformaticsSG GitHub page. To get this file just follow the link to the GitHub page, click on the green **Clone or download** button in the upper right corner, and choose **Download ZIP**.

There are two common ways to read data files into R, depending on the type of file they are. For comma separated value (CSV) files use read.csv() and for tab-delimited files use read.table(). There are options for reading in data of other types and code from a script but we will focus on these today for our data.

For the read.table() command below, replace the path I have provided with your path to the file. If you have already set the working directory and it contains your file, you can just provide the file name.

Note: the path and file names are strings and must be enclosed in quotes.

```
read.table("/Users/stacey/Data/GitHub/Introduction-to-R/combined_out.txt")
```

When you run the read.table() command, you will see the data file printed to your console. It would be nice if it could be referred to without having to type out the read.table() command, so let's assign a variable to the data table.

```
my_data <- read.table("/Users/stacey/Data/GitHub/Introduction-to-R/combined_out.txt")
```

Now it will be easier to access and use the data table. You can now look at the top of the data using head(), at the end using tail(), determine the number of columns using ncol() or length(), the number of rows using nrow(), and the dimensions of the data dim().

```
head(my_data)
```

```
##                   V1 V2 V3 V4
## 1 ENSG00000277248.1  0  0  0
## 2 ENSG00000274237.1  0  0  0
## 3 ENSG00000280363.1  0  0  0
## 4 ENSG00000279973.1  0  0  0
## 5 ENSG00000226444.2  0  0  0
## 6 ENSG00000276871.1  0  0  0
```

```
tail(my_data)
```

```
##               V1  V2   V3   V4
## 1458 ERCC-00163   5    7   13
## 1459 ERCC-00164   0    1    0
## 1460 ERCC-00165  33   64   41
## 1461 ERCC-00168   2    1    1
## 1462 ERCC-00170   1    2    3
## 1463 ERCC-00171 916 1253 1056
```

```
ncol(my_data)
```

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

```
length(my_data)
```

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

```
nrow(my_data)
```

```
## [1] 1463
```

```
dim(my_data)
```

```
## [1] 1463    4
```

Changing Column Names

As a default, read.table() assumes the data file does not have a first row with the names of each column. Although this is true for our data, it will not always be. If your data does have a header, just set the header argument to TRUE like so - read.table("file.txt", header=TRUE). You will notice, the header that R made for us when header=FALSE is V1, V2, V3, V4.

Let's give our column names something more intuitive. First, look at a list of the column names using colnames().

```
colnames(my_data)
```

```
## [1] "V1" "V2" "V3" "V4"
```

There are four columns that need new names. I am going to make a vector of strings that will represent each column's new name. Then I will assign the column names to be the names in the vector.

```
new_names <- c("GeneID", "Sample_1", "Sample_2", "Sample_3")
colnames(my_data) <- new_names
colnames(my_data)
```

```
## [1] "GeneID"   "Sample_1" "Sample_2" "Sample_3"
```

```
head(my_data)
```

```
##              GeneID Sample_1 Sample_2 Sample_3
## 1 ENSG00000277248.1        0        0        0
## 2 ENSG00000274237.1        0        0        0
## 3 ENSG00000280363.1        0        0        0
## 4 ENSG00000279973.1        0        0        0
## 5 ENSG00000226444.2        0        0        0
## 6 ENSG00000276871.1        0        0        0
```

Indexing and Subsetting

When you read tabular data into R using read.csv() or read.table() it is stored as a dataframe. Just as any data table, dataframes have rows for observations and columns for each variable of the dataset. Each column of a dataframe is a vector and contains the same type of data as previously described. A dataframe, however, contains vectors of different types and that is why they exist.

More often then not, we want to select certain columns or rows to work with, which is possible with an understanding of indexing and subsetting.

First, let's take a look at indexing. R vectors are 1-indexed, which means that the first element in a vector is index 1. We can select the element(s) we want by calling the appropriate index/indices in brackets (example: vector[index]). Here **alphabet** is a vector of letters that you can select elements from.

```
alphabet <- LETTERS
alphabet
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```
alphabet[1]
```

```
## [1] "A"
```

```
alphabet[10]
```

```
## [1] "J"
```

```
alphabet[1:4]
```

```
## [1] "A" "B" "C" "D"
```

```
alphabet[c(8, 5, 12, 12, 15)]
```

```
## [1] "H" "E" "L" "L" "O"
```

You can also remove elements in the same manner.

```
alphabet[-1]
```

```
##  [1] "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R"
## [18] "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```
alphabet[-(1:13)]
```

```
##  [1] "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```
alphabet[-c(1, 26)]
```

```
##  [1] "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R"
## [18] "S" "T" "U" "V" "W" "X" "Y"
```

You can select elements of a dataframe as well but a little differently. A common way to access columns of a dataframe is by using the $ operator and the name of the column. I am not showing the output but you should see a list printed on the console window.

```
my_data$GeneID
my_data$Sample_1
```

Alternatively, you can use brackets just as we did in the vector example. Since dataframes have two dimensions (rows and columns) the bracket operator can take two indexes separated by a comma (e.g. [row, column]). Omitting the row index will return all the rows (e.g. [, column]) and ommitting the column index will return all the columns (e.g. [row,]).

```
my_data[1, ]
```

```
##              GeneID Sample_1 Sample_2 Sample_3
## 1 ENSG00000277248.1        0        0        0
```

```
my_data[1:4, ]
```

```
##              GeneID Sample_1 Sample_2 Sample_3
## 1 ENSG00000277248.1        0        0        0
## 2 ENSG00000274237.1        0        0        0
## 3 ENSG00000280363.1        0        0        0
## 4 ENSG00000279973.1        0        0        0
```

```
my_data[1:4, 1]
```

```
## [1] ENSG00000277248.1 ENSG00000274237.1 ENSG00000280363.1 ENSG00000279973.1
## 1463 Levels: ENSG00000008735.13 ENSG00000015475.18 ... ERCC-00171
```

```
my_data[1:4, 1:2]
```

```
##              GeneID Sample_1
## 1 ENSG00000277248.1        0
## 2 ENSG00000274237.1        0
## 3 ENSG00000280363.1        0
## 4 ENSG00000279973.1        0
```

```
my_data[1:4, c(1, 3)]
```

```
##              GeneID Sample_2
## 1 ENSG00000277248.1        0
## 2 ENSG00000274237.1        0
## 3 ENSG00000280363.1        0
## 4 ENSG00000279973.1        0
```

```
my_data[1:4, "GeneID"]
```

```
## [1] ENSG00000277248.1 ENSG00000274237.1 ENSG00000280363.1 ENSG00000279973.1
## 1463 Levels: ENSG00000008735.13 ENSG00000015475.18 ... ERCC-00171
```

```
my_data[1:4, c("GeneID", "Sample_1")]
```

```
##              GeneID Sample_1
## 1 ENSG00000277248.1        0
## 2 ENSG00000274237.1        0
## 3 ENSG00000280363.1        0
## 4 ENSG00000279973.1        0
```

```
my_data[1:4, c("GeneID", "Sample_2")]
```

```
##              GeneID Sample_2
## 1 ENSG00000277248.1        0
## 2 ENSG00000274237.1        0
## 3 ENSG00000280363.1        0
## 4 ENSG00000279973.1        0
```

Changing Row Names

When you look a the data table, you will notice the left most column are numbers, these are your row names. Some programs require that your row names are set to your gene names and expect the gene name column to be excluded.

We can do this by using row.names() and subsetting the column containing the gene names. We will then exclude the gene name column and re-assign the my_data variable so it will contain the modifications we just created.

Note: Changing the column names or row names modifies the dataframe and does not require a re-assignment of the variable that the dataset is saved as.

```
head(my_data)
```

```
##              GeneID Sample_1 Sample_2 Sample_3
## 1 ENSG00000277248.1        0        0        0
## 2 ENSG00000274237.1        0        0        0
## 3 ENSG00000280363.1        0        0        0
## 4 ENSG00000279973.1        0        0        0
```

```
## 5 ENSG00000226444.2        0        0        0
## 6 ENSG00000276871.1        0        0        0
```

```
row.names(my_data)[1:10]
```

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

```
row.names(my_data) <- my_data$GeneID
head(my_data)
```

```
##                              GeneID Sample_1 Sample_2 Sample_3
## ENSG00000277248.1 ENSG00000277248.1        0        0        0
## ENSG00000274237.1 ENSG00000274237.1        0        0        0
## ENSG00000280363.1 ENSG00000280363.1        0        0        0
## ENSG00000279973.1 ENSG00000279973.1        0        0        0
## ENSG00000226444.2 ENSG00000226444.2        0        0        0
## ENSG00000276871.1 ENSG00000276871.1        0        0        0
```

```
my_data <- my_data[, -1]
head(my_data)
```

```
##                   Sample_1 Sample_2 Sample_3
## ENSG00000277248.1        0        0        0
## ENSG00000274237.1        0        0        0
## ENSG00000280363.1        0        0        0
## ENSG00000279973.1        0        0        0
## ENSG00000226444.2        0        0        0
## ENSG00000276871.1        0        0        0
```

```
row.names(my_data)[1:10]
```

```
##  [1] "ENSG00000277248.1" "ENSG00000274237.1" "ENSG00000280363.1"
##  [4] "ENSG00000279973.1" "ENSG00000226444.2" "ENSG00000276871.1"
##  [7] "ENSG00000277683.1" "ENSG00000277821.1" "ENSG00000273663.1"
## [10] "ENSG00000278534.1"
```

If you want a shortcut to renaming your rows, you can assign the column you want to be your row names when you first read your data in R. You know that index 1 is the "GeneID" column so you can pass 1 into your row.names argument.

The column names are untouched and will still need to be modified as described above.

```
named_data <- read.table("/Users/stacey/Data/GitHub/Introduction-to-R/combined_out.txt",
                         row.names=1)
head(named_data)
```

```
##                   V2 V3 V4
## ENSG00000277248.1  0  0  0
## ENSG00000274237.1  0  0  0
## ENSG00000280363.1  0  0  0
## ENSG00000279973.1  0  0  0
## ENSG00000226444.2  0  0  0
## ENSG00000276871.1  0  0  0
```

```
rownames(named_data)[1:10]
```

```
##  [1] "ENSG00000277248.1" "ENSG00000274237.1" "ENSG00000280363.1"
##  [4] "ENSG00000279973.1" "ENSG00000226444.2" "ENSG00000276871.1"
##  [7] "ENSG00000277683.1" "ENSG00000277821.1" "ENSG00000273663.1"
```

```
## [10] "ENSG00000278534.1"
```

Packages

Packages are collections of R functions, data, and code that help beef up R's capabilities. It is very likely that someone has written a package that will help you achieve whatever your ultimate goal may be. Base R comes with a set of packages but for every new package you want to add you will have to install it manually – don't worry, it is easy!

CRAN - install.packages()

- CRAN = Comprehensive R Archive Network
- This option downloads and installs packages from CRAN-like repositories or from local files and is the most common way to get packages
- Find out more about this using ?install.packages

```
install.packages("dplyr")
```

Bioconductor

- R packages for genomic data analysis are usually found on the Bioconductor website
- In an upcoming meeting, we will use DESeq2. HERE is the Bioconductor page with installation instructions, documentation, and other helpful information.

```
source("https://bioconductor.org/biocLite.R")
biocLite("DESeq2")
```

RStudio - Packages

Lastly, RStudio has a feature to help you install packages in a point and click fashion.

Choose the Packages tab > click Install > choose Repository(CRAN) > type the package name > hit Install.

This the same as the install.packages() option so you may need to check Bioconductor if you don't find the package you are looking for.