Introduction to Basic R

Abhinav Anand

Setup

The following discussion assumes we have donwloaded R and RStudio. The package `gapminder` needs to be installed prior to running the commands below.

1. For downloading R, visit https://cran.r-project.org/
2. For downloading RStudio visit https://www.rstudio.com/
3. To install `gapminder`, type `install.packages("gapminder")` in the RStudio console.

```
library(tidyverse)
library(rmarkdown)
library(knitr)
library(xml2)
library(gapminder)

knitr::opts_chunk$set(echo = T,
                      warning = T,
                      message = F,
                      eval = T,
                      include = T
                      )
```

The most basic way in which one can use R is as a calculator:

```
(t <- sin(pi/4)) #enclosing in parentheses prints output
```

```
## [1] 0.7071068
```

```
(2.342929*1.19483)/4.9802244
```

```
## [1] 0.5621036
```

```r
x <- 3*10.4293939
round(x, digits = 4)
```

```
## [1] 31.2882
```

Remarks

Note the use of `<-` as opposed to `=`. In other programming languages, one uses the equality sign for assignment but in R the "assignment operator" is different. This is so because `=` is reserved for use in function arguments. (Example: `round(x, digits = 4)`. Note however, that even if `=` is used in lieu of `<-` all codes work okay. Confirm by computing, say `(x = 3*10.4293939)`)

Some other useful operations:

```r
1:10
```

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

```r
seq(1:10) #the sequence generator function
```

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

R has several packages with in-built functions. In order to use them, one needs to declare their use. For our purposes, we will extensively use the suite of packages `tidyverse` written primarily by Hadley Wickham.

```r
library(tidyverse)
```

Pipes

Consider the following problem: given a vector, we need to take its sine, then take the mean, then square it and then if the resulting number is more than 1, print "more than 1", else print "less than 1".

We can do this in the following way:

```r
x <- 1:100 #initial vector

x_1 <- sin(x) #note vectorized function

x_2 <- (mean(x_1))^2 #take mean, square

if(x_2 >= 1) #note the syntax for the if statement
{
  print("more than 1")
} else
{
  print("less than 1")
}
```

```
## [1] "less than 1"
```

```r
# or more succinctly in one line

if(mean(sin(x))^2 >= 1)
{
  print("more than 1")
} else
{
  print("less than 1")
}
```

```
## [1] "less than 1"
```

However, there is another way to do this with pipes which can be read as the "then" operator. In fact, whenever we encounter the pipe symbol %>%, without loss of generality, we can replace it with the word "then".

```r
x_3 <- x %>% sin(.) %>% mean(.) %>% .^2
# read as: take x, then take sine
# then take mean, then square
```

```r
if(x_3 >= 1)
{
  print("more than 10")
} else
{
  print("less than 1")
}
```

```
## [1] "less than 1"
```

Pipes are very powerful when expressing a sequence of operations. The pipe, `%>%` comes from the `magrittr` package but tidyverse loads it automatically.

Data-types in R: Some Examples

Vectors

There are two main types of vectors in R: atomic and non-atomic. Atomic vectors have components of the same type, say double, or integer or logical etc. These are also referred to as "numeric" vectors. By default all vectors in R are considered column-vectors.

Non-atomic vectors (including "lists") can contain heterogenous components.

Recycling

There are no built-in scalars in R. Scalars are implemented as vectors of length 1. Most functions in R, hence, are vectorized—they take vector arguments and operate on it component-wise. For example, for the vector `x <- 1:100`, the function `sin(x)` produces each component's sine, i.e., `sin(1):sin(100)`.

When we mix scalars and vectors, the scalars are automatically replicated to be the same size as the vector. For example:

```
1 + 1:10
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```
#is the same operation as

rep(1, 10) + 1:10 #note the highly useful rep() function
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

A related question: what happens if we add two vectors of different lengths?

```
1:3 + 1:15
```

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

```
#is the same as

rep(1:3, 5) + 1:15
```

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

This is "recycling": R replicates the shorter vector to the same length as the longer vector and then adds the two together. While such usage is uncommon for other programming languages, it has undeniable utility, though one needs to be cautious when implementing this idea.

Note

1. While recycling works for vectors, it doesn't do so for matrices or other rectangular data-types.

The `c()` operator

The `c()` operator stands for "concatenate" or according to some writers, "combine".

```
y <- 1:4

(y_2 <- c(5:10, y)) #concatenate
```

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

```
(y_3 <- y_2[c(3:7)] )#note the square brackets
```

```
## [1]  7  8  9 10  1
```

```
(y_4 <- y_2[-c(1, 4)]) #note the - sign
```

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

Arrays and Matrices

Matrices are two-dimensional data-types with columns and rows as the two dimensions. Arrays are data-types that can contain more than two dimensions as well (height, say in addition to rows and columns). These may or may not have special named attributes such as column names or row names.

Matrices can be constructed from component vectors by the usage of commands `cbind()` and `rbind()`

```
c_1 <- c(4, 9, 10, 12)
c_2 <- c(10, 3, 1, 10)

(mat_c <- cbind(c_1, c_2)) #column-bind
```

```
##      c_1 c_2
## [1,]   4  10
## [2,]   9   3
## [3,]  10   1
## [4,]  12  10
```

```
(mat_r <- rbind(c_1, c_2)) #row-bind
```

```
##     [,1] [,2] [,3] [,4]
## c_1    4    9   10   12
## c_2   10    3    1   10
```

Dataframes

Dataframes are data-types with a collection of vectors of the same length which may be of different types. Each column (variable) has a column-name that can be used to access the whole vector. Additionally, the column can be extracted by appending to the dataframe, the `$` sign followed by the column-name. The full set of names can be extracted by the command `names()`

```
df <- data.frame(a = runif(1:10),
                 b = rnorm(10, 0, 1),
                 c = 11:20
                 )

df$a %>% head()
```

```
## [1] 0.07334647 0.30922432 0.52732537 0.08402270 0.02421654 0.42104029
```

```
names(df)
```

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

Subsetting Dataframes

If we need only part of a dataframe, we can refer to the relevant indices in square brackets appended to the name of the dataframe. Leaving an index blank includes all entries.

```
df[, 1] %>% head() #column 1
```

```
## [1] 0.07334647 0.30922432 0.52732537 0.08402270 0.02421654 0.42104029
```

```
df[3, ] %>% head() #row 3
```

```
##           a          b  c
## 3 0.5273254 -0.5366784 13
```

```
df[c(1,2), "c"]
```

```
## [1] 11 12
```

Subsetting with Logical Indices

```
df[df$b < 0.5, ] #only those rows whose column b has values < 0.5
```

```
##            a          b  c
## 1 0.07334647  0.2891909 11
## 2 0.30922432 -0.1983550 12
## 3 0.52732537 -0.5366784 13
## 4 0.08402270 -1.8201355 14
## 5 0.02421654 -0.6866562 15
## 6 0.42104029 -1.1016686 16
## 8 0.79314242 -0.1150933 18
## 9 0.11660511  0.4335739 19
```

Subsetting and Assignment

```
df[df$c >= 14, "a"] <- NA


df
```

```
##             a          b  c
## 1  0.07334647  0.2891909 11
## 2  0.30922432 -0.1983550 12
## 3  0.52732537 -0.5366784 13
## 4           NA -1.8201355 14
## 5           NA -0.6866562 15
## 6           NA -1.1016686 16
## 7           NA  0.8073319 17
## 8           NA -0.1150933 18
## 9           NA  0.4335739 19
## 10          NA  1.5956745 20
```

Functions

Functions automate common tasks succinctly. According to Hadley Wickham, a general rule of thumb is that if we need to copy-paste more than thrice, it's time to

write a function. Writing functions has many uses: its name betrays its intentions; if changes are needed only one edit is enough; and the chance of errors while copy-pasting decreases drastically when working with functions.

Suppose there are normal random variables and we wish to transform them to standard normal.

```
t_1 <- rnorm(100, 5, 10) #number of points, mean, sigma
t_2 <- rnorm(100, 10, 100)
t_3 <- rnorm(100, 20, 200)

t_1_s <- (t_1-5)/10
t_2_s <- (t_1-10)/100
t_3_s <- (t_1-20)/200
```

However, this could be automated if we write a function

```
norm_std <- function(t, mu, sigma) #note the syntax
{
  t_std <- (t - mu)/sigma

  return(t_std) #note the return function
}

norm_std(t_1, 5, 10) %>% head(.) #what does this mean?
```

```
## [1] -0.7241235 -1.9404914 -0.4513705  1.3123735  1.4099941 -0.2813271
```

```
norm_std(t_2, 10, 100) %>% head(.)
```

```
## [1]  0.38316849 -0.04957717  1.01885476 -2.28554036 -0.11806042 -0.28770394
```

```
norm_std(t_3, 20, 200) %>% head(.)
```

```
## [1]  1.5747761 -0.7079810 -0.2185073  2.6911674 -1.4895251  0.4935895
```

Directory Management

If we save something in R or if we wish to access some file in R, we need to ensure that they are in the same folder (directory) as our code. This gives rise to the notion of the working directory which is displayed at the top of the console. We can explicitly find the name of the working directory by the command `getwd()`.

To pinpoint addresses of folders, Mac and Linux use slashes (say `plots/plot_1.pdf`) but Windows uses backslashes (`plots\plot_1.pdf`). However, backslashes are reserved in R—in that they cannot be used as they are. Hence to include each backslash, we need to prefix another backslash, which may be confusing. Hence many writers suggest using the Linux/Mac style addresses.

RStudio Projects

It is good practice to keep all files related to a project together in one folder. This may include script files (with .R extension), plots, data files etc. RStudio uses "projects" with .Rproj extension for this special purpose.

In general, it is considered good practice to keep one project for each data analysis exercise and put all data files, scripts, outputs (such as plots) in that folder. It is also recommended that relative paths be used and not absolute paths.