

Introduction to Basic R

Abhinav Anand

Setup

The following discussion assumes you have downloaded 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, type `install.packages("gapminder")` in the RStudio console.

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
```

```
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:

```
1:10
```

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

```
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.

```
library(tidyverse)
```

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. 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.

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
```

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:

```
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"
```

```
# 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”.

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

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 the tidyverse loads it automatically.

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_2-10)/100
t_3_s <- (t_3-20)/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] -0.1940707 -0.2795063 -0.4637535 -0.5994307  1.0164079  0.8276887
```

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

```
## [1] -0.2217160 -1.2597919 -0.2057303  1.1201297 -1.7149043  2.1334339
```

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

```
## [1]  0.1422560 -0.2089479  0.7960424  0.3972417  1.0497377  0.9905420
```

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.