Basic Operations in R

Code

Instructor: Soumya Mukherjee

Content credit: Dr. Matthew Beckman and Olivia Beck

July 6, 2023

R Programming Basics

To run a line within a chunk bring the prompt/cursor at the beginning of the line and press “Ctrl + enter” on a Windows machine or press “Cmnd + return” on a Mac. To run the whole chunk, just press the green play button at the top of each chunk.

The Basics

- R works just like a calculator

Hide

```
5 + 2
```

```
[1] 7
```

Hide

```
10 * (2 - 9)
```

```
[1] -70
```

Hide

```
3^2 / 9
```

```
[1] 1
```

Use : to sequential list

- list all numbers from 1 to 10

Hide

```
1:10
```

- list all numbers from 10 to 1

Hide

```
10:1
```

- arithmetic progression from 2 to 10 in steps of 2

Hide

```
seq(from = 2,to = 10,by = 2)
```

- Use " " or ' ' for strings

Hide

```
"Hello World"
'This is also a string'
"1, 2, 3" #this is a string, NOT a vector
```

- To write a vector of items we use c()

Hide

```
c(1, 2, 3)
```

- we can also put strings in vectors

Hide

```
c("A", "B", "C")
```

- R has some basic functions already loaded in

Hide

```
mean(c(1, 2, 3, 4, 5)) #mean of 1, 2, 3, 4, 5
factorial(4) # calculate 4!
rnorm(10, mean = 5, sd = 2) #generate 10 normal random variables with mean 5 and standard deviation 2
```

- Whenever we want more information about a function we type - ?function_name . This will prompt the help files to pop up in the bottom right pane of RStudio. Help files provide information about what the function does, the arguments is takes in, and examples. We can directly search in the help pane as well.

Hide

```
?mean
?rnorm
?c
```

Assignment operator

- in R we use <- (assign) to assign variable and function values
- we use = (equals) inside functions to give arguments values

Hide

```
my.number <- 1
dice <- 1:6
random.numbers <- rnorm(10, mean = 5, sd = 2)
```

##Indexing

- WARNING: R starts indexing a 1 (most other coding languages start indexing at 0)
- In R we use [ ] to reference items at an index in a vector

Hide

```
my.vector <- 10:15
my.vector[1] # get the 1st element in my.vector
my.vector[2:4] # get the 2nd, 3rd, 4th elements in my.vector
```

- Be careful that you do not try to index something that does not exist

Hide

```
my.vector[7] # this will say NA because there is nothing in the 7th element of my.vector
```

- We can use the “-” sign to indicate we want all elements “except” the ones listed

Hide

```
# output all elements except the first one
my.vector[ -1 ]

#output all elements except the the first and third ones
my.vector[ -c(1, 3) ]
```

- To create a matrix, we use the matrix function

Hide

```
my.matrix <- matrix(1:10, nrow = 2, ncol=5)
my.matrix
```

- When indexing matrices, the format is [row_number, column_number]

Hide

```
my.matrix[1, 2]
```

- If we leave the row_number blank, it will output every element in that column

Hide

```
my.matrix[ , 5] #outputs all number in column 5
```

- Similar for columns

Hide

```
my.matrix[1, ] #outputs all number in row 1
```

Vectorization

- Unlike most other coding languages, R is vectorized. What does this mean??? The default for R is to do component wise operations and not traditional matrix multiplication.
- In R, we can multiply a constant by a vector. This works just like traditional mathematics would

Hide

```
x <- c(1, 2, 3)
2 * x # = c(2, 4, 6)
```

- However, that is where the similarities between R and traditional mathematics end. In traditional mathematics if I multiply [2 4 6] * [5, 10, 15] i would get [2 4 6] * [5, 10, 15] = (2 * 5) + (4 * 10) + (6 * 15) = 140
- This is not what happen in R

Hide

```
v1 <- c(2, 4, 6)
v2 <- c(5, 10, 15)

v1 * v2 #outputs c(10, 40, 90). This is not 140. What happened?
```

- R did element wise computations. R saw that you have 2 vectors, each of length 3. It multiplied the first element in v1 and the first element in v2, then multiplied the second element in v1 and the second element in v2, then multiplied the third element in v1 and the third element in v2, and outputs one vector of length 3.
- Similar things happen for +, -, /, and ^

Hide

```
v1 + v2
v1 - v2
v1 / v2
v2^v1
```

- If we want to do traditional matrix multiplication, we use `%*%` . The output will be a matrix, not a vector

Hide

```
v1 %*% v2 # = 140
```

- Note, if we ever want to calculate the inverse of a matrix, we use solve()

Hide

```
mat <- matrix(c(1,2,3,4), nrow = 2, ncol = 2)

?solve
solve(mat) #inverse of mat

mat %*% solve(mat) # identity matrix
```

Data Frames

- R stores data in objects called data frames
- Lets create a data frame with 2 columns, Age (in years) and Height (in cm)

Hide

```
my.data <- data.frame( "Age" = c(23, 12, 90, 41, 87),
                       "Height" = c(162.3, 121.0, 174.8, 185.2, 168.9))

# This will display your data frame in the console
my.data
```

- This is will display your data in a new window

Hide

```
View(my.data)
```

- Equivalently, you can also click on the “my.data” row in the Environment pane
- We can reference the information in “Age” column in a few different ways. All of the following are equivalent

Hide

```
my.data[ , 1]
my.data[ , "Age"]
my.data$Age
```

Packages + Tidyverse

- You only need to install a package once (on every machine you’re on)

Hide

```
install.packages("tidyverse",repos = "http://cran.us.r-project.org")

## every time you want to use a function inside a package, you need to load the library
library(tidyverse) #now we can use the tidyverse functions!
```

- Before we get into functions in the tidyverse, we will introduce one more special operator in R
- The tidyverse package contains the `%>%` (pipe) operator
- Pipe takes what ever is before the `%>%` and inputs as the first element into whatever is after the `%>%` .

An example using the mean function

Hide

```
?mean
numbers <- c(1,5,6,8,23,45,67)
```

-The following are equivalent - Method 1

Hide

```r
mean(numbers)
```

- Method 2

Hide

```r
numbers %>%
  mean()
```

- Pipe is usually used when we have a series of operations we want to compute
- Lets use the iris data set as an example

Hide

```r
iris
```

- Say I want to arrange the rows in this data set by the Sepal.Length.
- The following are equivalent
- Method 1

Hide

```r
arrange(iris, Sepal.Length)
```

- Method 2

Hide

```r
iris %>%
  arrange(Sepal.Length)
```

- But now say we only want to keep rows with Sepal.Width > 3 AND arrange the table by Sepal.Length
- The following are equivalent
- Method 1

Hide

```
temp1 <- arrange(iris, Sepal.Length)
result <- filter(temp1, Sepal.Width > 3)
result
```

- Method 2

Hide

```
iris %>%
  arrange(Sepal.Length) %>%
  filter(Sepal.Width > 3)
```

- Both methods technically work, but method 2 is must easier to read and took a lot less typing!
- (advanced) Method 2 takes up significantly less memory on your machine than Method 1. This is one of the primary reasons we use %>%

Additional resources

- Learning base R operations (https://www.w3schools.com/r/ (https://www.w3schools.com/r/)).
- Learning tidyverse (specifically dplyr) operations (https://dplyr.tidyverse.org/articles/dplyr.html (https://dplyr.tidyverse.org/articles/dplyr.html)).