Hand On with R

Task 1: Basic building

R can be used as an interactive calculator

```
> 5 + 7
[1] 12
```

Similarly

```
x <- 5 + 7
```

Now let's create a vector

```
z <- c(1.1, 9, 3.4)
```

Any time you can get help using

```
> ?c
```

combine vectors to make a new vector

```
c(z, 555, z)
[1] 1.10   9.00   3.14 555.00   1.10   9.00   3.14
```

Numeric vectors can be used in arithmetic expressions

```
> z * 2 + 100
[1] 102.20 118.00 106.28
```

Other common arithmetic operators are `+`, `-`, `/`, and `^` (where x^2 means 'x squared'). To take the square root, use the sqrt() function and to take the absolute value, use the abs() function.

For example :

```
> my_sqrt <- sqrt(z - 1)
```

Will return a vector of length 3

```
> my_sqrt
[1] 0.3162278 2.8284271 1.4628739
```

Recycling (broadcasting in python) in R

```
> c(1, 2, 3, 4) + c(0, 10)
[1]  1 12  3 14
```

However,

```
> c(1, 2, 3, 4) + c(0, 10, 100)
[1]   1  12 103   4
```

This will give you a warning message

Warning message:

In c(1, 2, 3, 4) + c(0, 10, 100) :

longer object length is not a multiple of shorter object length

Tip:

The up arrow will cycle through previous commands.

Task 2: Workspace and Files

In this task, you'll learn how to examine your local workspace in R and begin to explore the relationship between your workspace and the file system of your machine.

> getwd()

[1] "C:/Users/Nitin PC/Documents"

Will give you directory your R session is using as its current working directory using getwd().

List all the objects in your local workspace using ls().

> ls()

[1] "x",”y”,”z”,”my_sqrt”

List all the files in your working directory using list.files() or dir()

Try it yourself

> list.files()

What is the output?

Let’s Play

> old.dir <- getwd()

Use dir.create() to create a directory in the current working directory called "testdir".

> dir.create("testdir")

Set your working directory to "testdir" with the setwd() command.

> setwd("testdir")

You can create a file in your working directory called "mytest.R" using the file.create() function.

> file.create("mytest.R")

[1] TRUE

Let’s validate:

> list.files()

[1] "mytest.R"

Or

> file.exists("mytest.R")

[1] TRUE

Now when an object or a file is created it has certain attributes which can be accessed by

> file.info("mytest.R")

What is the output ?

You can use the $ operator --- e.g., file.info("mytest.R")$mode --- to grab specific items.

Renaming a file

> file.rename("mytest.R", "mytest2.R")

[1] TRUE

Coping a file

> file.copy("mytest2.R", "mytest3.R")

[1] TRUE

Go back to your original working directory using setwd().

> setwd(old.dir)

Task 3: Sequences of Numbers

The simplest way to create a sequence of numbers in R is by using the `:` operator.

> 1:20

 [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Try

> pi:10

[1] 3.141593 4.141593 5.141593 6.141593 7.141593 8.141593 9.141593

Try

> 15:1

 [1] 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

It counted backwards in increments of 1!

Pull up the documentation for `:` now.

> ?`:`

You could also use

```
> seq(0, 10, by=0.5)
 [1]  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5
[17]  8.0  8.5  9.0  9.5 10.0
> seq(5, 10, length=30)
 [1]  5.000000  5.172414  5.344828  5.517241  5.689655  5.862069  6.034483  6.206897
 [9]  6.379310  6.551724  6.724138  6.896552  7.068966  7.241379  7.413793  7.586207
[17]  7.758621  7.931034  8.103448  8.275862  8.448276  8.620690  8.793103  8.965517
[25]  9.137931  9.310345  9.482759  9.655172  9.827586 10.000000
```

If we're interested in creating a vector that contains 40 zeros, we can use rep(0, times = 40). Try it out.

```
> rep(0, times=40)
 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Give another try instead type

```
> rep(c(0, 1, 2), times = 10)
 [1] 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2
```

Task 4: Vectors

The simplest and most common data structure in R is the vector(Array in C , C++ , JAVA)

Two types:

1) Atomic (exactly one data type)
2) List (multiple data types)

Types of atomic vectors include logical, character, integer, and complex.

```
> num_vect <- c(0.5, 55, -10, 6)
```

Try

```
> num_vect < 1
```

Will return a vector with 4 logical values

```
[1]  TRUE FALSE  TRUE FALSE
```

Logical operators

The `<` and `>=` symbols in these examples are called 'logical operators'. Other logical operators include `>`, `<=`, `==` for exact equality, and `!=` for inequality.

Other logical operators are

If we have two logical expressions, A and B, we can ask whether at least one is TRUE with A | B (logical 'or' a.k.a. 'union') or whether they are both TRUE with A & B (logical 'and' a.k.a. 'intersection'). Lastly, !A is the negation of A and is TRUE when A is FALSE and vice versa.

((111 >= 111) | !(TRUE)) & ((4 + 1) == 5)

Will return TRUE

Create a character vector that contains the following words: "Programming", "with", "R".

> my_char <- c("Programming", "with", "R")

> my_char

[1] "Programming", "with", "R"

Type paste(my_char, collapse = " ") now. Make sure there's a space between the double quotes in the `collapse` argument.

> paste(my_char, collapse = " ")

[1] "Programming with R"

The `collapse` argument to the paste() function tells R that when we join together the elements of the my_char character vector, we'd like to separate them with single spaces.

Vector recycling! Try paste(LETTERS, 1:4, sep = "-"), where LETTERS is a predefined variable in R containing a character vector of all 26 letters in the English alphabet.

> paste(LETTERS, 1:4, sep = "-")

[1] "A-1" "B-2" "C-3" "D-4" "E-1" "F-2" "G-3" "H-4" "I-1" "J-2" "K-3" "L-4" "M-1" "N-2"

[15] "O-3" "P-4" "Q-1" "R-2" "S-3" "T-4" "U-1" "V-2" "W-3" "X-4" "Y-1" "Z-2"

Task 5: Missing Values

Missing values play an important role in statistics and data analysis. Often, missing values must not be ignored, but rather they should be carefully studied to see if there's an underlying pattern or cause for their missingness.

In R, NA is used to represent any value that is 'not available' or 'missing' (in the statistical sense).

> x <- c(44, NA, 5, NA)

> x * 3

[1] 132 NA 15 NA

Next, let's create a vector containing 1000 NAs with z <- rep(NA, 1000).

> z <- rep(NA, 1000)

Finally, let's select 100 elements at random from these 2000 values (combining y and z) such that we don't know how many NAs we'll wind up with or what positions they'll occupy in our final vector -- my_data <- sample(c(y, z), 100).

> my_data <- sample(c(y, z), 100)

> my_na <- is.na(my_data)

```
> my_na
  [1]  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE
 [14] FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE
 [27]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [40] FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE
 [53]  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
 [66]  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE
 [79] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
 [92] FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE
```

Let's give that a try here. Call the sum() function on my_na to count the total number of TRUEs in my_na, and thus the total number of NAs in my_data. Don't assign the result to a new variable.

```
> sum(my_na)
[1] 52
```

Let's look at a second type of missing value -- NaN, which stands for 'not a number'. To generate NaN, try dividing (using a forward slash) 0 by 0 now.

```
> 0 / 0
[1] NaN
```

Task 6 : Subsetting Vectors

Extract elements from a vector based on some conditions that we specify.

Try x[1:10] to view the first ten elements of x.

```
> x[1:10]
 [1]           NA  0.5299247274           NA           NA -0.5399988732 -0.1191854427
 [7]           NA -0.0005441663  0.9714911664           NA
```

Recall that `!` gives us the negation of a logical expression, so !is.na(x) can be read as 'is not NA'. Therefore, if we want to create a vector called y that contains all of the non-NA values from x, we can use y <- x[!is.na(x)]. Give it a try.

```
> y <- x[!is.na(x)]
> y
 [1]  0.5299247274 -0.5399988732 -0.1191854427 -0.0005441663  0.9714911664 -1.0325497198
 [7]  0.3798177474 -0.2863554538 -0.3914337301 -0.5790076778  0.4994686757 -0.3207320342
[13] -0.2557526850  1.7753843380 -0.7728033356  0.5825444776 -0.6870112749  1.4382822638
```

```
[19] -1.7819660619  1.2992963698
```

Now type y[y > 0] to see that we get all of the positive elements of y, which are also the positive elements of our original vector x.

```
> y[y > 0]

[1] 0.5299247 0.9714912 0.3798177 0.4994687 1.7753843 0.5825445 1.4382823 1.2992964

> x[x > 0]

 [1]        NA 0.5299247        NA        NA        NA 0.9714912        NA        NA

 [9]        NA        NA 0.3798177        NA        NA        NA        NA 0.4994687

[17]        NA        NA        NA        NA        NA 1.7753843 0.5825445        NA

[25] 1.4382823 1.2992964        NA        NA
```

Since NA is not a value, but rather a placeholder for an unknown quantity, the expression NA > 0 evaluates to NA. Hence we get a bunch of NAs mixed in with our positive numbers when we do this.

We could do this:

```
> x[!is.na(x) & x > 0]

[1] 0.5299247 0.9714912 0.3798177 0.4994687 1.7753843 0.5825445 1.4382823 1.2992964
```

Create a vector of indexes with c(3, 5, 7), then put that inside of the square brackets.

```
> x[c(3, 5, 7)]

[1]        NA -0.5399989        NA
```

Try x[c(-2, -10)] gives us all elements of x EXCEPT for the 2nd and 10 elements.

```
> x[c(-2, -10)]

 [1]           NA           NA           NA -0.5399988732 -0.1191854427           NA

 [7] -0.0005441663  0.9714911664           NA           NA           NA -1.0325497198

[13]  0.3798177474           NA -0.2863554538           NA           NA -0.3914337301

[19]           NA -0.5790076778  0.4994686757           NA           NA           NA

[25] -0.3207320342 -0.2557526850           NA           NA  1.7753843380 -0.7728033356

[31]  0.5825444776           NA -0.6870112749  1.4382822638 -1.7819660619  1.2992963698

[37]           NA           NA
```

Task 7: Matrices and Data Frames

Create a vector

```
> my_vector <- 1:20
```

The dim() function tells us the 'dimensions' of an object.

> dim(my_vector)

NULL

> length(my_vector)

[1] 20

We can assign dimension of a vector

> dim(my_vector) <- c(4, 5)

Another way to see dimensions is by calling the attributes() function on my_vector. Try it now.

> attributes(my_vector)

$dim

[1] 4 5

With more than one dimension it is no more a vector rather it is a Matrix

> my_vector

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

You can verify the same

> class(my_vector)

[1] "matrix"

Another way of defining a matrix is

> my_matrix2 <- as.matrix(1:20)

> my_matrix2

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

```
[12,]   12
[13,]   13
[14,]   14
[15,]   15
[16,]   16
[17,]   17
[18,]   18
[19,]   19
[20,]   20
```

```
> matrix(1:20, nrow = 4, ncol = 5, byrow = FALSE)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

Let's start by creating a character vector containing the names of our patients -- Bill, Gina, Kelly, and Sean. Remember that double quotes tell R that something is a character string. Store the result in a variable called patients.

```
> patients <- c("Bill", "Gina", "Kelly", "Sean")
```

Now we'll use the cbind() function to 'combine columns'. Don't worry about storing the result in a new variable. Just call cbind() with two arguments -- the patients vector and my_matrix.

```
> cbind(patients, my_matrix)

     patients

[1,] "Bill"   "1" "5" "9"  "13" "17"
[2,] "Gina"   "2" "6" "10" "14" "18"
[3,] "Kelly"  "3" "7" "11" "15" "19"
[4,] "Sean"   "4" "8" "12" "16" "20"
```

All the data is in quotes or as strings. So in order to have more than one data type we have data frames

```
> my_data <- data.frame(patients, my_matrix)

> my_data

  patients X1 X2 X3 X4 X5

1     Bill  1  5  9 13 17

2     Gina  2  6 10 14 18

3    Kelly  3  7 11 15 19

4     Sean  4  8 12 16 20
```

Verify

```
Class(my_data)

[1] "data.frame"
```