More about R Programming

Dr Ebin Deni Raj

22/01/2022

Conditionals And Control Flow

Last week we have just touched base few concepts like Data type, data objects in R. Let's see what are relational operators in R. We will take a plunge on the wonderful world of Relational operators in R.

Equality and Inequality operator

Relational operators or comparators are operators which help us to see how one R object relates to another. For example you can check whether two objects are equal.You can do this by using a double equal sign. Let's try this:

```
TRUE==TRUE
```

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

```
TRUE==FALSE
```

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

The result of First query is a logical value , in this case TRUE because TRUE equals TRUE.

On the contrary, TRUE==FALSE will give us FALSE. Apart from logical variables we can also check the equality of objects. We can also compare strings and numbers.

```
"hello"=="goodbye"
```

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

```
3==2
```

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

The opposite of equality operator is the inequality operator, written as an exclamation mark followed by an equals sign.

```
"hello" !="goodbye"
```

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

< and > operator

Now let's try to check if an R object is greater than or less than another R object

```
3<5
```

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

```
3>5
```

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

Apart from equality operators, Filip also introduced the less than and greater than operators: $<$ and $>$. You can also add an equal sign to express less than or equal to or greater than or equal to, respectively. Have a look at the following R expressions, that all evaluate to FALSE:

$(1 + 2) > 4$ "dog" $<$ "Cats" TRUE $<=$ FALSE For numerics the above code makes sense, how do we deal with strings? Is "hello" greater than "goodbye"

```
"Hello" > "Goodbye"
```

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

Apparently it has evaluated to TRUE. The reason is that R has used alphabet to sort character strings. Since "H" comes after "G" in the alphabet , "Hello" is considered greater than Goodbye. Now, how about logical values:

```
#TRUE coerces to 1
#FALSE coerces to 0
TRUE<FALSE
```

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

Since 1 is not less than 0, hence the result is FALSE.

Relational operators and Vectors

Kindly recall that we have seen the concept of vector in last session. We know that R is pretty bgood with vectors. We will see how R's comparators handle vectors. Let's try that: Here we will store the number of views you had on linked in profile in a week. Now if you want to find out, on how many days the views exceeded 10, we can directly use the greater than sign.

```
linkedin <- c(16,9,13,5,2,17,14)
linkedin
```

```
## [1] 16  9 13  5  2 17 14
```

```
#let's try a compartor

linkedin > 10
```

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

We will create another vector *Facebook* and compare it with *Linkedin*

```
facebook <- c(17,7,5,16,8,13,14)
facebook
```

```
## [1] 17  7  5 16  8 13 14
```

```
#When are the number of Facebook views less than or equal to the number of Linked in views
facebook<= linkedin
```

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

In this case, the comparison is done for every element of the vector, one by one. For example , in the third day , the number of Facebook views is 5 and the number of Linked in views is 13. The comparison evalutes to TRUE as shown in the result.

Excercise 1

Write R expressions to check whether:

- Write R code to see if TRUE equals FALSE.
- Write R code to check if -6 * 14 is not equal to 17 - 101.
- Write R code for comparison of character strings. Ask R whether the strings "useR" and "user" are equal.
- Write R code to find out what happens if you compare logicals to numerics: are TRUE and 1 equal?
- Write R code to check -6 * 5 + 2 is greater than or equal to -10 + 1.
- Write R code to check "raining" is less than or equal to "raining dogs".
- Write R code to check TRUE is greater than FALSE.

Make sure to have a look at the console output to see if R returns the results you expected.

Compare Vectors

You are already aware that R is very good with vectors. Without having to change anything about the syntax, R's relational operators also work on vectors.

Let's go back to the example that was started in the lecture/handout. You want to figure out whether your activity on social media platforms have paid off and decide to look at your results for LinkedIn and Facebook. Each of the vectors contains the number of profile views your LinkedIn and Facebook profiles had over the last seven days.

Using relational operators, find a logical answer, i.e. TRUE or FALSE, for the following questions:

- On which days did the number of LinkedIn profile views exceed 15?
- When was your LinkedIn profile viewed only 5 times or fewer?
- When was your LinkedIn profile visited more often than your Facebook profile?

```
###Compare Vectors
# The linkedin and facebook vectors have already been created for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)
facebook <- c(17, 7, 5, 16, 8, 13, 14)
```

Using relational operators, find a logical answer, i.e. TRUE or FALSE, for the following questions:

- On which days did the number of LinkedIn profile views exceed 15?
- When was your LinkedIn profile viewed only 5 times or fewer?
- When was your LinkedIn profile visited more often than your Facebook profile?

Have a look at the console output. Your LinkedIn profile was pretty popular on the sixth day, but less so on the fourth and fifth day.

R's ability to deal with different data structures for comparisons does not stop at vectors. Matrices and relational operators also work together seamlessly!

Instead of in vectors (as in the previous exercise), the LinkedIn and Facebook data is now stored in a matrix called views. The first row contains the LinkedIn information; the second row the Facebook information. The original vectors facebook and linkedin are still available as well.

Using the relational operators you've learned so far, try to discover the following:

```
# The social data has been created for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)
facebook <- c(17, 7, 5, 16, 8, 13, 14)
views <- matrix(c(linkedin, facebook), nrow = 2, byrow = TRUE)
```

- When were the views exactly equal to 13? Use the views matrix to return a logical matrix.
- For which days were the number of views less than or equal to 14? Again, have R return a logical matrix.

This exercise concludes the part on comparators. Now that you know how to query the relation between R objects, the next step will be to use the results to alter the behavior of your programs

Logical Operators

Now since you have learned to use relational operators, lets see how to deal with logical operators. Logical operators will help in change or combine the results of comparison. R does this using AND , the OR, and the NOT operator. Let's look at each one of them .

```
TRUE & TRUE
```

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

```
TRUE & FALSE
```

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

```
FALSE & TRUE
```

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

```
FALSE & FALSE
```

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

```
x<-12

x > 5  & x < 15
```

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

```
x<-17

x > 5 & x < 15
```

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

```
# OR operator
 TRUE | TRUE
```

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

```
 FALSE | TRUE
```

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

```
 TRUE | FALSE
```

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

```
FALSE | FALSE
```

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

& and |

Before you work your way through the next exercises, have a look at the following R expressions. All of them will evaluate to TRUE:

```
TRUE & TRUE
FALSE | TRUE
5 <= 5 & 2 < 3
3 < 4 | 7 < 6
```

Watch out: $3 < x < 7$ to check if x is between 3 and 7 will not work; you'll need $3 < x$ & $x < 7$ for that.

In this exercise, you'll be working with the last variable. This variable equals the last value of the linkedin vector that you've worked with previously. The linkedin vector represents the number of LinkedIn views your profile had in the last seven days.

```
#The linkedin and last variable are already defined for you

linkedin <- c(16, 9, 13, 5, 2, 17, 14)
last <- tail(linkedin, 1)
```

Write R expressions to solve the following questions concerning the variable last:

- Is last under 5 or above 10?
- Is last between 15 and 20, excluding 15 but including 20?

Like relational operators, logical operators work perfectly fine with vectors and matrices.

Both the vectors linkedin and facebook are available again. Also a matrix - views - has been defined; its first and second row correspond to the linkedin and facebook vectors, respectively.

- When did LinkedIn views exceed 10 and did Facebook views fail to reach 10 for a particular day? Use the linkedin and facebook vectors.
- When were one or both of your LinkedIn and Facebook profiles visited at least 12 times?
- When is the views matrix equal to a number between 11 and 14, excluding 11 and including 14?

You'll have noticed how easy it is to use logical operators to vectors and matrices. What do these results tell us? The third day of the recordings was the only day where your LinkedIn profile was visited more than 10 times, while your Facebook profile wasn't. Can you draw similar conclusions for the other results?

Reverse the result: !

On top of the & and | operators, you also learned about the ! operator, which negates a logical value. To refresh your memory, here are some R expressions that use !. They all evaluate to FALSE:

```
!TRUE
```

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

```
!(5 > 3)
```

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

```
!!FALSE
```

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

What would the following set of R expressions return?

```
x <- 5
y <- 7
!(!(x < 4) & !!!(y > 12))
```

Conditional Statements

The if statement

Before diving into some exercises on the if statement, have another look at its syntax:

```
if (condition) {
  expr
}
```

Remember your vectors with social profile views? Let's look at it from another angle. The medium variable gives information about the social website; the num_views variable denotes the actual number of views that particular medium had on the last day of your recordings.

```
# Variables related to your last day of recordings
medium <- "LinkedIn"
num_views <- 14

# Examine the if statement for medium
if (medium == "LinkedIn") {
  print("Showing LinkedIn information")
}
```

```
## [1] "Showing LinkedIn information"
```

- Code an if statement that prints "You are popular!" to the console if the num_views variable exceeds 15.

Try to see what happens if you change the medium and num_views variables and run your code again.

Add an else

You can only use an *else* statement in combination with an *if* statement. The *else* statement does not require a condition; its corresponding code is simply run if all of the preceding conditions in the control structure are *FALSE*. Here's a recipe for its usage:

```
if (condition) {
  expr1
} else {
  expr2
}
```

It's important that the *else* keyword comes on the same line as the closing bracket of the *if* part!

Add an else statement to both control structures, such that

"Unknown medium" gets printed out to the console when the if-condition on medium does not hold. R prints out "Try to be more visible!" when the if-condition on num_views is not met.

```
###Add An Else
# Variables related to your last day of recordings
medium <- "LinkedIn"
num_views <- 14

# Control structure for medium
```

```
if (medium == "LinkedIn") {
  print("Showing LinkedIn information")
} else {
  print("Unknown medium")
}
```

The *else if* statement allows you to further customize your control structure. You can add as many *else if* statements as you like. Keep in mind that R ignores the remainder of the control structure once a condition has been found that is TRUE and the corresponding expressions have been executed.

```
# Variables related to your last day of recordings
medium <- "LinkedIn"
num_views <- 14

# Control structure for medium
if (medium == "LinkedIn") {
  print("Showing LinkedIn information")
} else if (medium == "Facebook") {
  # Add code to print correct string when condition is TRUE

} else {
  print("Unknown medium")
}

# Control structure for num_views
if (num_views > 15) {
  print("You're popular!")
} else if (num_views <= 15 & num_views > 10) {
  # Add code to print correct string when condition is TRUE

} else {
  print("Try to be more visible!")
}
```

Add code to both control structures such that:

- R prints out "Showing Facebook information" if medium is equal to "Facebook". Remember that R is case sensitive!
- "Your number of views is average" is printed if num_views is between 15 (inclusive) and 10 (exclusive). Feel free to change the variables medium and num_views to see how the control structure respond. In both cases, the existing code should be extended in the else if statement.

Loops

Loops can come in handy on numerous occasions. While loops are like repeated if statements, the for loop is designed to iterate over all elements in a sequence. The while loop is somewhat similar to if statement because it executes the code inside if the condition is true.

When starting to use R, most people use loops whenever they need to iterate over elements of a vector, list or data.frame. While it is natural to do this in other languages, with R we generally want to use vectorization. That said, sometimes loops are unavoidable, so R offers both for and while loops.

Write a while loop

Let's get you started with building a while loop from the ground up. Have another look at its recipe:

```
while (condition) {
  expr
}
```

Remember that the condition part of this recipe should become FALSE at some point during the execution. Otherwise, the while loop will go on indefinitely.

If your session expires when you run your code, check the body of your *while* loop carefully.

Have a look at the code on the below; it initializes the speed variables and already provides a while loop template to get you started.

```
###Write A While Loop
# Initialize the speed variable
speed <- 64

# Code the while loop
while (speed>30) {
  print("Slow down!")
  speed <- speed - 7
}
```

```
## [1] "Slow down!"
## [1] "Slow down!"
## [1] "Slow down!"
## [1] "Slow down!"
## [1] "Slow down!"
```

```
# Print out the speed variable
speed
```

```
## [1] 29
```

In the previous exercise, you simulated the interaction between a driver and a driver's assistant: When the speed was too high, "Slow down!" got printed out to the console, resulting in a decrease of your speed by 7 units.

There are several ways in which you could make your driver's assistant more advanced. For example, the assistant could give you different messages based on your speed or provide you with a current speed at a given moment.

A while loop similar to the one you've coded in the previous exercise is already available in the editor. It prints out your current speed, but there's no code that decreases the speed variable yet, which is pretty dangerous.

- If the speed is greater than 48, have R print out "Slow down big time!", and decrease the speed by 11. Otherwise, have R simply print out "Slow down!", and decrease the speed by 6.
- If the session keeps timing out and throwing an error, you are probably stuck in an infinite loop! Check the body of your while loop and make sure you are assigning new values to speed.

```
###Throw In More Conditionals
# Initialize the speed variable
speed <- 64

# Extend/adapt the while loop
```

```
while (speed > 30) {
  print(paste("Your speed is",speed))
  if (speed > 48) {
    print("Slow down big time!")
    speed <- speed - 11
  } else {
    print("Slow down!")
    speed <- speed - 6
  }
}
```

```
## [1] "Your speed is 64"
## [1] "Slow down big time!"
## [1] "Your speed is 53"
## [1] "Slow down big time!"
## [1] "Your speed is 42"
## [1] "Slow down!"
## [1] "Your speed is 36"
## [1] "Slow down!"
```

For Loop

Loop over a vector

```
primes <- c(2, 3, 5, 7, 11, 13)

# loop version 1
for (p in primes) {
  print(p)
}

# loop version 2
for (i in 1:length(primes)) {
  print(primes[i])
}
```

Remember our linkedin vector? It's a vector that contains the number of views your LinkedIn profile had in the last seven days. The linkedin vector has already been defined in the editor on the right so that you can fully focus on the instruction

- Write a for loop that iterates over all the elements of linkedin and prints out every element separately. Do this in two ways: using the loop version 1 and the loop version 2 in the example code above.

```
###Loop Over A Vector
# The linkedin vector has already been defined for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)

# Loop version 1
for (p in linkedin) {
  print(p)
}
```

```
## [1] 16
## [1] 9
## [1] 13
```

```
## [1] 5
## [1] 2
## [1] 17
## [1] 14
```

```
# Loop version 2
for (i in 1:length(linkedin)) {
  print(linkedin[i])
}
```

```
## [1] 16
## [1] 9
## [1] 13
## [1] 5
## [1] 2
## [1] 17
## [1] 14
```

Loop over a list Looping over a list is just as easy and convenient as looping over a vector. There are again two different approaches here:

```
primes_list <- list(2, 3, 5, 7, 11, 13)

# loop version 1
for (p in primes_list) {
  print(p)
}

# loop version 2
for (i in 1:length(primes_list)) {
  print(primes_list[[i]])
}
```

Notice that you need double square brackets - [[ ]] - to select the list elements in loop version 2.

Suppose you have a list of all sorts of information on New York City: its population size, the names of the boroughs, and whether it is the capital of the United States.

```
###Loop Over A List
# The nyc list is already specified
nyc <- list(pop = 8405837,
            boroughs = c("Manhattan", "Bronx", "Brooklyn", "Queens", "Staten Island"),
            capital = FALSE)

# Loop version 1
for (info in nyc) {
  print(info)
}
```

```
## [1] 8405837
## [1] "Manhattan"     "Bronx"         "Brooklyn"      "Queens"
## [5] "Staten Island"
## [1] FALSE
```

```
# Loop version 2
for (i in 1:length(nyc)) {
  print(nyc[[i]])
}
```

```
## [1] 8405837
## [1] "Manhattan"     "Bronx"         "Brooklyn"      "Queens"
## [5] "Staten Island"
## [1] FALSE
```

Let's return to the LinkedIn profile views data, stored in a vector linkedin. In the first exercise on for loops you already did a simple printout of each element in this vector. A little more in-depth interpretation of this data wouldn't hurt, right? Time to throw in some conditionals! As with the while loop, you can use the if and else statements inside the for loop.

- Add code to the for loop that loops over the elements of the linkedin vector:
- If the vector element's value exceeds 10, print out "You're popular!".
- If the vector element's value does not exceed 10, print out "Be more visible!"
- If the vector element's value exceeds 16, print out "This is ridiculous, I'm outta here!" and have R abandon the for loop (break).
- If the value is lower than 5, print out "This is too embarrassing!" and fast-forward to the next iteration (next).

```
###Next, You Break It
# The linkedin vector has already been defined for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)

# Adapt/extend the for loop
for (li in linkedin) {
  if (li > 10) {
    print("You're popular!")
  } else {
    print("Be more visible!")
  }

  # Add code to conditionally break iteration
  if (li > 16) {
    print("This is ridiculous, I'm outta here!")
    break
  }

  # Add code to conditionally skip iteration
  if (li < 5) {
    print("This is too embarrassing!")
    next
  }

  print(li)
}
```

```
## [1] "You're popular!"
## [1] 16
## [1] "Be more visible!"
## [1] 9
## [1] "You're popular!"
## [1] 13
## [1] "Be more visible!"
## [1] 5
## [1] "Be more visible!"
```

```
## [1] "This is too embarrassing!"
## [1] "You're popular!"
## [1] "This is ridiculous, I'm outta here!"
```

As stated earlier, if a solution can be done without loops, via vectorization or matrix algebra, then avoid the loop. It is particularly important to avoid nested loops. Loops inside other loops are extremely *slow* in R.

Introduction to Functions

Most of you might be knowing the concept of functions. We have already used functions in our program before. Remember the time you created a list using *list*() fucntion. You can think function as some kind of black box. You can give input to this black box and the black box processes the input and it returns the output. For example: *sd* function for calculating standard deviation. We can start with this example:

If you want to learn about the arguments to be passed to a function you may use args(sd) If you want to know more about the function: ?sd or u may use help(sd) You may try the following lines in the console:

```
values<- c(1,5,6,7)
sd(values)
```

```
## [1] 2.629956
```

```
args(sd)
```

```
## function (x, na.rm = FALSE)
## NULL
```

Before even thinking of using an R function, you should clarify which arguments it expects. All the relevant details such as a description, usage, and arguments can be found in the documentation. To consult the documentation on the sample() function, for example, you can use one of following R commands: help(sample) ?sample

- Consult the documentation on the mean() function: ?mean or help(mean).
- Inspect the arguments of the mean() function using the args() function.

```
# Consult the documentation on the mean() function
help(mean)

# Inspect the arguments of the mean() function
args(mean)
```

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

The documentation on the mean() function gives us quite some information:

The *mean*() function computes the arithmetic mean. The most general method takes multiple arguments: x and .... The x argument should be a vector containing numeric, logical or time-related information. Remember that R can match arguments both by position and by name. Can you still remember the difference? You'll find out in this exercise!

Once more, We'll be working with the view counts of your social network profiles for the past 7 days. These are stored in the linkedin and facebook vectors and have already been defined earlier.

```
###Use A Function
# The linkedin and facebook vectors have already been created for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)
facebook <- c(17, 7, 5, 16, 8, 13, 14)
```

```
# Calculate average number of views
avg_li = mean(linkedin)
avg_fb = mean(facebook)

# Inspect avg_li and avg_fb
avg_li
```

```
## [1] 10.85714
```

```
avg_fb
```

```
## [1] 11.42857
```

If you've had a good look at the documentation, you'll know by now that the mean() function also has this argument, na.rm, and it does the exact same thing. By default, it is set to FALSE, as the Usage of the default S3 method shows:

mean(x, trim = 0, na.rm = FALSE, ...) Let's see what happens if your vectors linkedin and facebook contain missing values (NA).

```
# The linkedin and facebook vectors have already been created for you
linkedin <- c(16, 9, 13, 5, NA, 17, 14)
facebook <- c(17, NA, 5, 16, 8, 13, 14)

# Basic average of linkedin
mean(linkedin)
```

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

```
# Advanced average of linkedin
mean(linkedin, na.rm = TRUE)
```

```
## [1] 12.33333
```

Writing R Functions

Let us write our own function named *triple()*

```
my_fun <- function(arg1, arg2){

}
```

The last expression evaluated in anR function becomes the return value. You can also explicity specify the return value by using the *return* statement.

```
triple <- function(x){
  3*x

}
triple(6)
```

```
## [1] 18
```

Using optional arguments in function

```
math_magic<-function(a,b=1){
  a*b +a/b
}
math_magic(4)
```

```
## [1] 8
```

Scope of Functions:

To understand the scope, lets examine a simple program

```
y<-2
function_scoping<-function(x){
  sum=x+y
  print(sum)
}
x<-1  # Global or free variable
function_scoping(x)
```

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

From the above code we can see that function_scoping has one parameter x. Before calling the function, we have assigned a value to x and then passed as a parameter. There is available y inside the function with a value no value specified. Now, whats the difference between these two variables? variable y, whose value is not specified in function and not passed as an argument of the function, such variables are called free variables. So, R first looks at the global or workspace, if it finds variables and values here, it will directly assign and use it. **Practice Functions**

- Create a function pow_two(): it takes one argument and returns that number squared (that number times itself).
- Call this newly defined function with 12 as input.
- Next, create a function sum_abs(), that takes two arguments and returns the sum of the absolute values of both arguments.
- Finally, call the function sum_abs() with arguments -2 and 3 afterwards.

```
##Write Your Own Function
# Create a function pow_two()
pow_two <- function(x) {
  x ^ 2
}

# Use the function
pow_two(12)

# Create a function sum_abs()
sum_abs <- function(x, y) {
  abs(x) + abs(y)
}

# Use the function
sum_abs(-2, 3)
```

R packages

We have seen many R functions till now.How come these functions are built in? Well, all of these built-in functions are part of R packages that are loaded in your R session. R packages are bundles of code,data, documentation, and tests that are easy to share with others. For example, *mean*, *list* etc.. are all part of the base package which contains the basic functionality to use R. Some packages wont come bundled with

R base version. But don't worry, you can install it easily from inside R, using *install.* packages function which is actually a function inside *utils* package. This function goes to CRAN(Comprehensive R Archive Network)- a repository where thousands of packages are available. The function downloads the package file and installs the package on your system. All this can be done using a single command in R.. For example: *install.packages("ggplot2")* will install ggplot package in R

To load a package in R: To load package in R environment and to start using it we can use the *library* function.

example: library(ggplot2)

An alternative to *library()* is *require()*. You may use Google for finding out cool packages in R.

The apply() family

When you are using R, you will often encounter vectors and lists containing all sorts of information. We have seen earlier that the for loop helps in iterating over all kinds of data structures. But there is even an easier way.

lapply()

Let's see one example:

```
del <- list(pop=19000000, place =c("Janpath","Saket","Ansari nagar","Gautham Nagar"),capital= TRUE)


#i can print the info in two ways
for(info in del){
  print(class(info))
}
```

```
## [1] "numeric"
## [1] "character"
## [1] "logical"
```

```
#Or I can use lapply for the same
lapply(del,class)
```

```
## $pop
## [1] "numeric"
##
## $place
## [1] "character"
##
## $capital
## [1] "logical"
```

```
#another example to find the number of characters in each value

cities<-c("New Delhi","Chennai","Bangalore","Hyderabad","Indore")
num_chars<-c()
for(i in 1:length(cities)){
  num_chars[i]<-nchar(cities[i])
}
```

```
#Same can be done using lapply
lapply(cities,nchar)
```

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

```
# To convert the list to a vector
unlist(lapply(cities,nchar))
```

```
## [1] 9 7 9 9 6
```

Notice here that output is alist, although the input is a vector. This is something important about lapply, irrespective of the data structure it always return a list. If you want to change that to a a vector you may use unlist. By using *lapply* the code looks more intuituve and readable. You may write your own function and use it along with lapply.

sapply()

We have seen how to apply a function over each and every element of a list or vector. The key thing about lapply is that its output is always a list. *sapply* will return the output as a vector. It is short for 'simplify apply'.

Lets see on example:

```
del <- list(pop=19000000, place =c("Janpath","Saket","Ansari nagar","Gautham Nagar"),capital= TRUE)
sapply(cities,nchar)
```

```
## New Delhi   Chennai Bangalore Hyderabad    Indore
##         9         7         9         9         6
```

vapply()

Till now we have seen *lapply*() and *sapply*()

- lapply()
  - Apply functions over list or vector
  - output = list
- sapply()
  - Apply functions over list or vector
  - try to simplify list to array/vector

- vapply()
  - Apply functions over list or vector
  - **Explicitly specify the output format**

We will use an earlier example to demonstrate vapply()

```
vapply(cities, nchar, numeric(1))
```

```
## New Delhi   Chennai Bangalore Hyderabad    Indore
##         9         7         9         9         6
```