R Fundamentals

You might have seen by now that there are two places in Rstudio [The editor which we'll be using throughout this course] has two places where you can put your code snippets for execution. Console and R script . I'll advice to always write your codes in a script not on console even when you are just practicing, especially in the beginning. Because this is the best way to conserve your code and comments and will come very handy for reference later on.

Object Assignment

Lets begin. We'll start with introduction to R object assignment. objects in R are value containers. Lets look at few examples:

```
x=5
```

By typing in the above line and executing by selecting the code and hitting run [shortcut ctrl+ Enter also works], you just created an R object named "x" which contains value 5. you can double check by typing in just x and execute that .

```
x
```

```
## [1] 5
```

You can see the output to be 5 , the value which is stored in the object named "x". Now lets try typing in "X", [capital x].

```
X
```

```
## Error in eval(expr, envir, enclos): object 'X' not found
```

you get the above mentioned error, why? because R is case sensitive. "x" is different from "X".You haven't yet created object named "X".So you get the error `object 'X' not found` Keep that in mind here on-wards, especially if you have already been working with a language like SAS which is not case sensitive.

Object Name Restrictions

Lets create another R object

```
1.more=2.34
```

But above written code throws an error:

```
Error: unexpected symbol in "1.more"
```

Because that is a syntactically incorrect object name. Avoid special characters also, because they might hold some other meaning in the context of R syntax, you'll end up getting such error. Essentially, Keep your names simple.

```
one.more=2.34
onemorerobject=2.34
```

A syntactically correct name can contain `letters` *[both upper & lower case]* , `.` , `number` and `_` [underscore]. But it should not start with a `number` or a `.`

There are few reserved words as well which should not be used as object names:

```
if else repeat while function for in next break

TRUE FALSE NULL Inf NaN NA NA_integer_ NA_real_ NA_complex_ NA_character_
```

Data object

There can be many type of R object depending on how are we creating them. We'll learn many ways to do that as we progress in this course. However there are limited types R `data` objects. Three major basic type of R `data` objects are

- numeric
- character
- logical

Numeric data objects are simply numbers of any type; integers , decimals etc. We have already scene many examples of such numeric object. [there are specific types for integers, doubles etc , which you will encounter, I'm leaving those for you to explore on your own]

Character type data objects are the ones which contain character/string values.

```
x="Hadley Wickham"
```

Now if you simply type in `x` and run, the value `Hadley Wickham` gets displayed on your console in quotes `""` because its a character value.

```
x
```

```
## [1] "Hadley Wickham"
```

Logical data object can take three values : `TRUE` , `FALSE` , `NA`

you can create object containing these values as we have been doing so far. However keep in mind that these need to be without quotes and all in capital letters. If its within quotes then it is considered as just another character value. If its not in caps , then it is considered as another R object with the same name. This will be more clear after you execute following codes:

```
x=TRUE
x
```

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

This works alright without any error messages. Using `True` throws an error as discussed before.

```
x=True
```

```
## Error in eval(expr, envir, enclos): object 'True' not found
```

If you find typing in complete words `TRUE`, `FALSE` a hassle you can simply use `T` and `F` instead. However displayed/stored value will still be `TRUE` and `FALSE`.

```
x=F
x
```

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

A Sneak Peak on R Functions

If you were a little intimidated by learning a full fledged programming language, don't worry, R isn't one. In fact , at basic levels , R can be taken as being close to excel. You'll find functions for almost everything. Although as we progress further we'll discover more sophisticated aspects of R.

Lets use our first function. `class` . It tells us what class the R object falls in. Argument that it takes is the object name and it returns class of the object . Lets quickly see some examples:

```
x="bunny"
y=F
z=4.5
class(x)
```

```
## [1] "character"
```

```
class(y)
```

```
## [1] "logical"
```

```
class(z)
```

```
## [1] "numeric"
```

One more example of a function that i wanted to take up here was the one which can transform an object to another class. For example here , the object v1 is of class character. Although it a number for us, but for R , anything within quotes is a character.

```
v1="23.45"
class(v1)
```

```
## [1] "character"
```

Lets convert it to a number now.

```
v2=as.numeric(v1)
class(v2)
```

```
## [1] "numeric"
```

you can see v2 is of class numeric. This new function that I used `as.numeric`, coverts any given input to class numeric. Sometimes its not possible to convert them to class as intended. Then the result value is `NA`.

```
v1="King"
class(v1)
```

```
## [1] "character"
```

```
v2=as.numeric(v1)
```

```
## Warning: NAs introduced by coercion
```

```
v2
```

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

There are lot of these `as.*` functions such as `as.character`, `as.data.frame` etc. We'll get to use and know about these as we progress in the course. Your quick "intro" to functions in R is done. Of course we'll keep encountering more and more functions later on.

One last thing , say i want to know what a particular function does, i can find that out by looking at the function documentation. But how to find that documentation? It's really simple. you can simply time `help(functionname)` or `?functionname` and documentation will open on the right bottom page in Rstudio [after executing that bit of code which you wrote , here on-wards I will assume execution of code is self implied to see the results]. If any function with the name you have given does not exist then you'll get appropriate `Not Found` response from Rstudio interface. Lets find out what function `sum` does:

```
?sum
```

Once you execute the code written above you can see that there is documentation of the function has opened. If you scroll to bottom of that page, in most of the cases , you'll find some examples such as one here taken from documentation of the function `sum`

```
## Pass a vector to sum, and it will add the elements together.
sum(1:5)
```

```
## [1] 15
```

If you pass on a function name which does exist :

```
?smu
```

```
## No documentation for 'smu' in specified packages and libraries:
## you could try '??smu'
```

What does `??smu` does , it returns list of all R object names in various packages which are loaded in the current session of R containing the word "smu".

```
??smu
```

You'll find links of related functions on the same page where the documentation of existing functions appears. From this list you can select the function which you were looking for did not know the complete name. If you know nothing about the function name, Google will be a better place to start with whatever information you have at hand.

Basic Data Operation with Functions and Operators in R

In this section we'll learn how to do basic operations with R, for all basic data types.

Numeric Operations with R functions and operators

Basic numeric operations such as addition, subtraction, multiplication etc can be simply done with naturally associated operators. Here are couple of examples:

```
x=2
y=8
x+y
```

```
## [1] 10
```

```
x-y
```

```
## [1] -6
```

```
x*y
```

```
## [1] 16
```

```
x/y
```

```
## [1] 0.25
```

For evaluating expressions like x^y you can use either symbol ^ or ** like this:

```
x^y
```

```
## [1] 256
```

```
x**y
```

```
## [1] 256
```

You can combine multiple such calculations into one using parenthesis etc.

```
(x+y-(x/y))**(x/10)
```

```
## [1] 1.576888
```

You notice that value of such expression is showed as output in your console , you can assign the same to an R object if you intend to store and use it in future

```
z=(x+y-(x/y))**(x/10)
z
```

```
## [1] 1.576888
```

As you can see that , now z contains value `1.576888` which was result of that complex expression we wrote. If we intend to use the result of the complex expression we can use z instead.

There are many function to evaluate many mathematical expressions such as logarithmic or trigonometric expressions. Here are few examples given below.

```
tan(10)
```

```
## [1] 0.6483608
```

```
log(2^14,10)
```

```
## [1] 4.21442
```

```
log(2^14)
```

```
## [1] 9.704061
```

```
log(2^14,2)
```

```
## [1] 14
```

You can see that a function can have sometimes multiple arguments [example that we have taken here is of function `log`]. Second argument is used to specify `base` of logarithmic expression. Default value for which is `e`, natural log. So when you don't specify any second argument , default value is used instead. You'll find that most of the function in R have a lot of arguments but many of them have set defaults and in practice you'll be mostly using only few arguments. So don't get intimidated by many arguments present in any function documentation. In practice you'll almost never have to use all of them.

`Functions for you to explore : round , exp, factorial`

Character Operations with functions in R

Character operations in R are done with many functions available pertaining to specific tasks , we'll discuss quite a few here relating to basic character operation, but do keep in mind that this by no means an exhaustive list.

Concatenation

Concatenation is combining two character strings to form a new one. Basically you "paste" two or more character string together to form a new one. And you have guessed it right, the function which we use to this is named `paste`. Lets see some examples:

```
x="Sachin"
y="Tendulkar"
z="Cricket"
paste(x,y,z)
```

```
## [1] "Sachin Tendulkar Cricket"
```

Function `paste` combines input strings. You might have noticed that in the end result , input strings are separated by a space. This space is basically default value for the argument `sep` which is short for separator. Lets see if we can use some other value as separator.

```
name=paste(x,y)
profile=paste(name,z,sep=":")
profile
```

```
## [1] "Sachin Tendulkar:Cricket"
```

You can see that in the second instance of usage of function `paste` we have used separator `:` which shows up in the end result , input string object `name` and `z` are being separated by `:`.

You can use anything as a separator. In next few examples we use `$` and `%2` as separator.

```
paste(x,y,z,sep="$")
```

```
## [1] "Sachin$Tendulkar$Cricket"
```

```
paste(x,y,z,sep="%2")
```

```
## [1] "Sachin%2Tendulkar%2Cricket"
```

Substitution

Substitution is where you substitute a part of string with another "word" or "character". Say we want to convert "1612-Lone Tower-Mumbai" to "1612/Lone Tower/Mumbai". Here we are substituting `-` with `/` in the first string. Lets see which function to use, to do that in R

```
address="1612-Lone Tower-Mumbai"
newAd=sub("-","/",address)
```

Here first argument is what you are looking for in the string, second argument is what are you going to replace it with, third argument is the string where this replacement will happen. Lets look at `newAd`

```
newAd
```

```
## [1] "1612/Lone Tower-Mumbai"
```

This isn't what we were expecting, it turns out that function `sub` ; only substitutes first occurrence of the pattern with replacement. What to do if we wanted to replace both occurrences of `-` with `/` at once. Well, we could try to feedback the outcome to another call to function `sub`. Lets try that

```
newAd=sub("-","/",newAd)
newAd
```

```
## [1] "1612/Lone Tower/Mumbai"
```

This does the job, but its not a very good solution. Imagine if we had more than two occurrences of `-` in our input string. It doesn't makes sense to repeatedly feed in the output of function `sub` back to itself.

We have another function in R which does global substitution at once. Function is named short for global substitution : `gsub`

```
newAd1=gsub("-","/",address)
newAd1
```

```
## [1] "1612/Lone Tower/Mumbai"
```

`gsub` substitutes all occurrences of `-` in the input string at once in the given example above.

Part Extraction

You can extract part of character string using the function `substr`. For example if I wanted extract port number from this ip listing "192.168.23.134:219", I can do:

```
ip=" 192.168.23.134:219"
# Anything between quotes is a string. Doesn't matter if you see numbers here
port=substr(ip,17,20)
port
```

```
## [1] "219"
```

Here first argument is the input string, second argument is the character position in the input string where the sub-setting starts, third argument is where the sub-setting ends. Few things to note

- Character numbering within the string starts with 1, not 0 as opposed to some other standard programming languages
- All characters are counted including special characters. [e.g. `.` or spaces in previous example]

** Counting Number of Characters in A string **

We use function `nchar` to find out number of characters in a string. Here is an example:

```
x="Sa  chin-$.  ?/"
nchar(x)
```

```
## [1] 15
```

Logical/Conditional Operations in R with functions and Operators

Single Conditions

You can use simple operators `>` , `<` , `==`,`!=`,`<=`,`>=` for doing logical operations. Result of these logical operations are logical values. Here are some examples:

```
x=7
y=9
x>y
```

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

```
x<y
```

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

```
x==y
```

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

Note that for equality check, you need two `=` signs as `==`, because one `=` is reserved for value assignment.

As always you can store these results for further usage in another R object as well.

```
z=x>y
z
```

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

Avoid using `> or <` like operators for character values. If you do , you will not get an error; however you'll get a result based on dictionary/lexicographic order which might not be the output you are looking for.

Combining Multiple Conditions

You can combine multiple condition with and `&` and or `|` operators. Lets say we want to find whether x falls in the range `[1,19]` or not in below example.

```
x=10
x>=1 & x<=19
```

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

```
y="SAchin"
y=="Sachin" | y=="SACHIN"
```

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

you can combine as many conditions you want using `&` and `|` symbols.

Note that the result above is false because strings inside quotes are case sensitive. "SAchin" is not same as "Sachin" or "SACHIN".

Higher Data Types: Vector, Lists and Data Frames

Basic data types are only the building blocks . Eventually what you'll use in practice when solving real life data analysis problems is collections of these data types. Vectors , Lists and Data frames come to your rescue in R. Lets learn about these .

Vectors and Vector Access

Vectors are most used collections of basic data types in R. As you learn more and more you'd find how central vectors for data processing in R. In the immediate example we'd see a quick way to combine basic data types to create vectors

```
x=c(2,4,89,-10,5,6)
is.vector(x)
```

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

An important property of the vectors is:

- They can contain only one type of basic data type, either numeric or character. [Logical values are stored as numbers only at the back end]

if you do `class(vector)`, result will be type of basic data type which the vector contains.

```
class(x)
```

```
## [1] "numeric"
```

Accessing Vector Elements with Numerical Indices

Once you have collection of basic data types, next step is to know how you can access individual elements or subsets of the collection. You can do so by passing index numbers. Here is how you can access individual elements of a vector.

```
x
```

```
## [1]   2   4  89 -10   5   6
```

```
x[3]
```

```
## [1] 89
```

You can see that `x[3]` gives third element of the vector x. if you want to access multiple elements, you can pass vector of indices.

```
z=x[c(1,2,5)]
is.vector(z)
```

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

This gives you first, second and fifth element as output. This output is also a vector.

Vector indices that you pass need not be in order or unique.

```
x[c(3,4,2,2,6,3)]
```

```
## [1]  89 -10   4   4   6  89
```

if index exceeds the number of elements in the vector, NA is returned as output.

```
x[c(2,3,8,5,4,9)]
```

```
## [1]   4  89  NA   5 -10  NA
```

One very important aspect of accessing vector elements with numerical indices is to pass -ve indices. Lets see what happens when we do that:

```
x
```

```
## [1]   2   4  89 -10   5   6
```

```
x[-2]
```

```
## [1]   2  89 -10   5   6
```

You can see that `x[-2]` gives you all elements of x except the second one. This is how you can exclude elements from a vector. Same thing happens if you pass a vector of indices and put a -ve sign in front of it.

```
x[-c(2,3,6)]
```

```
## [1]   2 -10   5
```

You can see that in the output you have all the elements except second, third and sixth.

One important thing about passing vector indices for sub-setting vector is to understand that:

They need not be in order They need not be unique *They need not be less than the number of elements in the vector that you are sub-setting

Here is an example:

```
x[c(4,5,4,40,1,1,2,3,10,2,1,4,6)]
```

```
##  [1] -10   5 -10  NA   2   2   4  89  NA   4   2 -10   6
```

Wherever the index exceeds the number of elements in the vector, you get an `NA` as result.

Another important thing to know is that you can not pass mixed [positive and negative index] for vector sub-setting. This will throw an error:

```
x[c(2,3,-1)]
```

```
## Error in x[c(2, 3, -1)]: only 0's may be mixed with negative subscripts
```

Accessing Vector Elements with Logical Indices

At times you might not want to access vector elements based on numerical indices. Say for example you simply want to extract elements which are greater 4 from a vector.

```
x
```

```
## [1]   2   4  89 -10   5   6
```

```
L=x>4
L
```

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

You can see that L here is a logical vector, which takes values `TRUE` and `FALSE` for corresponding elements of x. For elements where condition `>4` is true `L` takes value `TRUE` otherwise `FALSE`.

Now if you use this logical vector for sub-setting, output will get only those elements for which L takes value `TRUE`. Lets check this out.

```
x[L]
```

```
## [1] 89  5  6
```

you don't really have to first create a logical vector and then pass it for sub-setting. You can directly put in the condition as well.

```
x[x>4]
```

```
## [1] 89  5  6
```

To negate a logical condition or vector, putting a -ve sign doesn't make sense. Because when you are negating a condition you are flipping the comparison, putting a -ve sign doesn't fit, in that context.

You can negate or; in correct words , flip a condition by putting a `!` sign front of it.

```
x
```

```
## [1]   2   4  89 -10   5   6
```

```
L
```

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

```
!L
```

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

```
x[!L]
```

```
## [1]   2   4 -10
```

Some Handy ways of creating a vector

If we want to create a vector containing numbers from a to b with increment 1, we can simply write `a:b`.

```
x=2:10
x
```

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

we can use function `seq` to generate a vector using a arithmetic sequence as well

```
x=seq(1,5,by=0.8)
```

this will generate a vector with AP which starts at `1` and subsequent elements increasing by `0.8` up till 5. Largest element here will not be greater than 5.

```
x
```

```
## [1] 1.0 1.8 2.6 3.4 4.2 5.0
```

In the `seq` function if we use option `length` instead of `by` then a sequence of the desired length is generated including both first and last elements as specified with first and second argument of the function `seq`. Function chooses increments on its on.

```
x=seq(1,10,length=21)
x
```

```
##  [1]  1.00  1.45  1.90  2.35  2.80  3.25  3.70  4.15  4.60  5.05  5.50
## [12]  5.95  6.40  6.85  7.30  7.75  8.20  8.65  9.10  9.55 10.00
```

Simply writing `seq(a)` is same as writing `1:a`. Why don't you try to see that for yourself.

you can combine multiple vector with same function `c` which we saw earlier.

```
x=1:5
y=seq(2,3,length=5)
z=c(x,y)
z
```

```
##  [1] 1.00 2.00 3.00 4.00 5.00 2.00 2.25 2.50 2.75 3.00
```

you can see that both the vectors have been simply joined keeping duplicate values as well.

One more way to create a new vector is to simply replicate another vector or a single element few times. This can be achieved by using function `rep`

```
rep(2,10)
```

```
##  [1] 2 2 2 2 2 2 2 2 2 2
```

```
rep("a",5)
```

```
## [1] "a" "a" "a" "a" "a"
```

you can pass a vector also to replicate.

```
rep(c(1,5,6),4)
```

```
##  [1] 1 5 6 1 5 6 1 5 6 1 5 6
```

the above code returns a vector which is nothing but repetition of the values 1,5,6 ; four times. If you use the option `each`, output will be slightly different.

```
rep(c("a","b"),4)
```

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

```
rep(c("a","b"),each=4)
```

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

Vector Operations

Vector operations form a very interesting part of R. We have seen several operations between individual data elements using operators and functions. You can use all those operators and functions with vectors as is. However output for the same will require some understanding.

```
x=1:10
y=1:10
x+y
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```
x*y
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

```
x-y
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

```
x/y
```

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

You can see that we used operator `+` for `x` and `y` as if they were basic data type. Result is simply summation *`[or other appropriate operator result]`* of corresponding elements of x and y. This is how simple operations that we have seen between basic data types translate to vectors.

Same goes for functions and other complex operators too. Lets see some more examples.

```
log(x)
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
##  [8] 2.0794415 2.1972246 2.3025851
```

```
2**x
```

```
##  [1]    2    4    8   16   32   64  128  256  512 1024
```

you can see that these operations are carried out for individual elements in the vector and result is again a vector. Same goes for character functions too

```
x=1:10
y=rep("a",10)
paste(x,y,sep="")
```

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

vector specific option in paste

you can use option `collapse` in function paste when using vectors as input. Here is an example

```
z=paste(y,x,sep="")
paste(z,collapse="+")
```

```
## [1] "a1+a2+a3+a4+a5+a6+a7+a8+a9+a10"
```

Lets say we have coefficients in one vector and variable names in another, we can use function paste to quickly make a formula out of it.

```
z
```

```
##  [1] "a1"  "a2"  "a3"  "a4"  "a5"  "a6"  "a7"  "a8"  "a9"  "a10"
```

```
f=round(runif(10),2)
f
```

```
##  [1] 0.57 0.92 0.94 0.69 0.60 0.50 0.36 0.84 0.74 0.45
```

```
paste(f,z,sep="*",collapse="+")
```

```
## [1] "0.57*a1+0.92*a2+0.94*a3+0.69*a4+0.6*a5+0.5*a6+0.36*a7+0.84*a8+0.74*a9+0.45*a10"
```

You must be curious about , what happens if length of the vectors involved in these operations do not match. Savior is cyclic operations.

lets say we want to add following two vectors

```
x=1:10
y=c(1,2,3)
```

What happens is that , smaller length vector values are recycled until they compensate for the difference in length. so `x+y` in the case above will be same as adding these two vectors

1 2 3 4 5 6 7 8 9 10

1 2 3 1 2 3 1 2 3 1

Result will be summation of corresponding elements :

2 4 6 5 7 9 8 10 12 11

Lets see:

```
#you'll get a result but with a warning for length mismatch
# However you will not get this warning if smaller vector's length is
# a multiple of larger vector
x+y
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length
```

```
##  [1]  2  4  6  5  7  9  8 10 12 11
```

These recycling of values to match lengths happens in any kind of vector operations. For example consider this:

```
paste0(c("a","b"),1:10)
```

```
##  [1] "a1"  "b2"  "a3"  "b4"  "a5"  "b6"  "a7"  "b8"  "a9"  "b10"
```

More Utility functions and operators

match

This gives for the first vector elements, index numbers at which they are present in the second vector. If an element of first vector is not present in second, you will get `NA` for an index reference.

```
x=c("k","j","$","1","f")
y=letters
match(x,y)
```

```
## [1] 11 10 NA NA  6
```

The result tells us that first, second and fifth elements of x are present at 11,10 and 6th position in y. Third and fourth element of x are not present in y.

%in%

Functionality of this operator is similar to match function. Although the results might be more intuitive to some. Instead of returning straight up indices, it returns a logical vector telling whether an element is present in another vector or not.

```
x %in% y
```

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

%%

This is modulo operator. It simply returns the value of remainder if you divide the first number by second.

```
100%%32
```

```
## [1] 4
```

Extract odd numbers from the vector 2:99 using modulo operator

which

This function again belongs to family of functions which give you information regarding presence of elements of one vector in another. It returns indices of elements in a vector which fulfill the specified condition

```
x=2:99
which(x%%2!=0)
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46
## [24] 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92
## [47] 94 96 98
```

Note that the returned vector contains indices not the values themselves, you can get the values by passing these indices to vector for sub-setting.

```
x[which(x%%2!=0)]
```

```
##  [1]  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47
## [24] 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93
## [47] 95 97 99
```

sort

As the name suggest , this sorts the input. Default sorting order for numeric is ascending. In alphabetical sorting ,lexicographic/dictionary order is followed.

```
x=c(3,4,5,-10,1,2,0)
sort(x)
```

```
## [1] -10   0   1   2   3   4   5
```

```
y=c("art","Ant","Bat")
sort(y)
```

```
## [1] "Ant" "art" "Bat"
```

If you need to sort this in descending order you can use option `decreasing`.

```
sort(x,decreasing = T)
```

```
## [1]   5   4   3   2   1   0 -10
```

```
sort(y,decreasing = T)
```

```
## [1] "Bat" "art" "Ant"
```

rev

This can be used to simply flip/reverse your vector.No sorting happens by default.

```
rev(y)
```

```
## [1] "Bat" "Ant" "art"
```

```
rev(x)
```

```
## [1]   0   2   1 -10   5   4   3
```

sample

This is a very useful function for taking random sample from your vector. Lets look at a quick example, we'll go into details with subsequently more examples.

```
x=1:100
sample(x,3)
```

```
## [1] 49 37 90
```

```
sample(x,3)
```

```
## [1] 99 18 92
```

You can see that this function `sample` is returning 3 randomly chosen numbers from the vector `x`. You must have also noticed that those 3 numbers in these two iterations are different. Because each randomly chosen numbers differ. This can be avoided by using a fixed seed for the randomizing algorithm.

```
set.seed(1)
sample(x,3)
```

```
## [1] 27 37 57
```

when you run this call to function `sample` with seed `1` , your output is going to be `27 37 57`. Irrespective of what OS/System/Version you are running your codes. This reproducibility of random sampling plays a powerful role later on.

Default call to function `sample` assumes sampling without replacement. You can do sampling with replacement as well by using `replace=T`.

```
x=1:10
sample(x,7,replace=T)
```

```
## [1] 10  3  9 10  7  7  1
```

You can see that few of the numbers sampled are repeated because of sampling with replacement. This way you can take sample which is larger than the input vector itself. This is very useful in bootstrapping your data. We'll learn about that as we progress in the course.

Note: Its not neccessary that repetion of values will always happen when you are doing sampling with replacement .

You can use `sample` function for all kind of vectors. It is not limited to numeric vectors.

```
sample(c("a","b","c"),10,replace=T)
```

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

you can specify percentage distribution for the items being samples as well. Result will be approximately close to that. lets look at an example.

```
x=sample(c("a","b"),100,replace=T,prob=c(0.3,0.7))
table(x)
```

```
## x
##  a  b
## 30 70
```

you can see that every time you run above code, `a` is present approximately 30% of the time and b 70% of the time. Larger the sample size, closer will be the approximation to probability vector supplied. Probability vector need not be adding to one. R internally normalizes the values if summation isn't 1.

unique

This function returns distinct values in a vector. For example if we apply this on the vector `x` which we generated above:

```
unique(x)
```

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

Lists

Vectors are called atomic data type because they can not contain higher data types themselves. List are recursive data types . They can contain all sorts of object and they don't need to be of same type either.

Here is an example

```
x=1:10
y=rep(c("a","b"),each=6)
z=4.56

list1=list(x,y,z)
list1
```

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

```
##  [1] "a" "a" "a" "a" "a" "a" "b" "b" "b" "b" "b" "b"
##
## [[3]]
## [1] 4.56
```

List looks much more complex in comparison to vectors. Here is how to access individual list elements

```
list1[[2]]
```

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

Note the double bracket `[[]]` instead of single bracket `[]` which we used for vectors. Once you have fished out element of the list, they can be accessed the way they are usually accessed. For example if its a vector it can be accessed like a vector. To get third element of the second element of the above list we'd simply write:

```
list1[[2]][3]
```

```
## [1] "a"
```

If you find `[[]]` to be really cumbersome like other people, you can name your list elements while creating the list.

```
list2=list(num=x,char=y,single=z)
list2
```

```
## $num
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $char
##  [1] "a" "a" "a" "a" "a" "a" "b" "b" "b" "b" "b" "b"
##
## $single
## [1] 4.56
```

Notice that now, instead of those `[[]]` indices you simply see `$` signs with the name of elements. To access 3rd element of the 2nd element *[named char]* of this list :

```
list2$char
```

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

```
list2$char[3]
```

```
## [1] "a"
```

Note that you can also access list elements with single bracket `[]`. But the subset that you get is of type list only. Which you'll have to access further accordingly

Data Frames

Data frames are actually a special kind of list. They are list of vectors of equal length. They are close to data sets kind of which we are used to seeing in general.

Lets make one data frame:

```
set.seed(2)
x=round(runif(30),2)
y=sample(c("a","b","c"),30,replace = T)
d=data.frame(x,y)
d
```

```
##       x y
## 1  0.18 a
## 2  0.70 a
## 3  0.57 c
## 4  0.17 c
## 5  0.94 b
## 6  0.94 b
## 7  0.13 c
## 8  0.83 a
## 9  0.47 c
## 10 0.55 a
## 11 0.55 c
## 12 0.24 a
## 13 0.76 a
## 14 0.18 a
## 15 0.41 c
## 16 0.85 c
## 17 0.98 c
## 18 0.23 b
## 19 0.44 b
## 20 0.07 c
## 21 0.66 a
## 22 0.39 a
## 23 0.84 c
## 24 0.15 c
## 25 0.35 a
## 26 0.49 c
## 27 0.15 c
## 28 0.36 c
## 29 0.96 b
## 30 0.13 c
```

You can see the data frame listed in console with 30 values. Number columns that you see on the left are known as `rownames` for data frames. We'll talk about them in detail in some time.

You can view this data by typing in `View(data frame name)` or simply clicking on the data set name in the `Environment` window on top left.

```
View(d)
```

We'll now learn about few useful functions in the context of data frames.

names

This outputs variables names of the data set.

```
names(d)
```

```
## [1] "x" "y"
```

you can also use this to assign new variable names

```
names(d)=c("num","char")
d
```

```
##     num char
## 1  0.18    a
## 2  0.70    a
## 3  0.57    c
## 4  0.17    c
## 5  0.94    b
## 6  0.94    b
## 7  0.13    c
## 8  0.83    a
## 9  0.47    c
## 10 0.55    a
## 11 0.55    c
## 12 0.24    a
## 13 0.76    a
## 14 0.18    a
## 15 0.41    c
## 16 0.85    c
## 17 0.98    c
## 18 0.23    b
## 19 0.44    b
## 20 0.07    c
## 21 0.66    a
## 22 0.39    a
## 23 0.84    c
## 24 0.15    c
## 25 0.35    a
## 26 0.49    c
## 27 0.15    c
## 28 0.36    c
## 29 0.96    b
## 30 0.13    c
```

rownames

this gives row names of the data set. As function `names`, row names can be used to assign row names.

```
rownames(d)
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14"
## [15] "15" "16" "17" "18" "19" "20" "21" "22" "23" "24" "25" "26" "27" "28"
## [29] "29" "30"
```

```r
rownames(d)=paste0("R",1:30)
d
```

```
##      num char
## R1  0.18    a
## R2  0.70    a
## R3  0.57    c
## R4  0.17    c
## R5  0.94    b
## R6  0.94    b
## R7  0.13    c
## R8  0.83    a
## R9  0.47    c
## R10 0.55    a
## R11 0.55    c
## R12 0.24    a
## R13 0.76    a
## R14 0.18    a
## R15 0.41    c
## R16 0.85    c
## R17 0.98    c
## R18 0.23    b
## R19 0.44    b
## R20 0.07    c
## R21 0.66    a
## R22 0.39    a
## R23 0.84    c
## R24 0.15    c
## R25 0.35    a
## R26 0.49    c
## R27 0.15    c
## R28 0.36    c
## R29 0.96    b
## R30 0.13    c
```

dim , nrow, ncol

`dim` gives dimension of the data set in terms of number rows and vector. Output is a vector of length 2. You can get individual row numbers and column numbers using `nrow` and `ncol`.

```r
dim(d)
```

```
## [1] 30  2
```

```r
nrow(d)
```

```
## [1] 30
```

```r
ncol(d)
```

```
## [1] 2
```

str

`str` gives you a quick glimpse of the data in terms of variable names, their type and first few values.

```
str(d)
```

```
## 'data.frame':    30 obs. of  2 variables:
##  $ num : num  0.18 0.7 0.57 0.17 0.94 0.94 0.13 0.83 0.47 0.55 ...
##  $ char: Factor w/ 3 levels "a","b","c": 1 1 3 3 2 2 3 1 3 1 ...
```

you can see here that character variable type is given as `Factor` rather than character. Character variables are by default stored as factors which are nothing but integers assigned to different levels/unique values for character variable. This allows R to save on memory because integers take lesser space that character values. If you want to force the type , you can always do that for that particular column.

```
d$char=as.character(d$char)
str(d)
```

```
## 'data.frame':    30 obs. of  2 variables:
##  $ num : num  0.18 0.7 0.57 0.17 0.94 0.94 0.13 0.83 0.47 0.55 ...
##  $ char: chr  "a" "a" "c" "c" ...
```

Accessing and Sorting Data Frames

To understand data frame access , just imagine it to be a vectors in both the dimensions, row and columns [although this is not correct technically].

before we start exploring how to pass row,column indices to a data set for access, let me show how we can use pre loaded data set in R. We'll be using one such data set right now : mtcars.

```
data(mtcars)
d=mtcars
dim(d)
```

```
## [1] 32 11
```

to know more about this data , do `?mtcars`.

Lets say we want to get 3rd value in the 6th column. We'll do as following.

```
d[3,6]
```

```
## [1] 2.32
```

As you can see this returns `2.32`. Which is what we wanted to extract. Now lets talk about how did we pass these indices. `[,]` inside the square bracket; numbers/vectors on the left hand side of the `,` represent row indices and on right hand side , represent column indices. If you leave any of these places blank then all members *[row or column]* of that dimension are included in the output. Lets go through some example to understand this better

```r
#3rd row , all columns
d[3,]
```

```
##             mpg cyl disp hp drat   wt  qsec vs am gear carb
## Datsun 710 22.8   4  108 93 3.85 2.32 18.61  1  1    4    1
```

```r
# all rows, 6 th colunmn
d[,6]
```

```
##  [1] 2.620 2.875 2.320 3.215 3.440 3.460 3.570 3.190 3.150 3.440 3.440
## [12] 4.070 3.730 3.780 5.250 5.424 5.345 2.200 1.615 1.835 2.465 3.520
## [23] 3.435 3.840 3.845 1.935 2.140 1.513 3.170 2.770 3.570 2.780
```

```r
# multiple rows, multiple column
d[c(3,4,20),c(1,4,6)]
```

```
##                 mpg  hp    wt
## Datsun 710     22.8  93 2.320
## Hornet 4 Drive 21.4 110 3.215
## Toyota Corolla 33.9  65 1.835
```

```r
# Excluding rows or columns by passing indices with negative sign
d[-(3:17),-c(1,4,6,7)]
```

```
##                 cyl  disp drat vs am gear carb
## Mazda RX4         6 160.0 3.90  0  1    4    4
## Mazda RX4 Wag     6 160.0 3.90  0  1    4    4
## Fiat 128          4  78.7 4.08  1  1    4    1
## Honda Civic       4  75.7 4.93  1  1    4    2
## Toyota Corolla    4  71.1 4.22  1  1    4    1
## Toyota Corona     4 120.1 3.70  1  0    3    1
## Dodge Challenger  8 318.0 2.76  0  0    3    2
## AMC Javelin       8 304.0 3.15  0  0    3    2
## Camaro Z28        8 350.0 3.73  0  0    3    4
## Pontiac Firebird  8 400.0 3.08  0  0    3    2
## Fiat X1-9         4  79.0 4.08  1  1    4    1
## Porsche 914-2     4 120.3 4.43  0  1    5    2
## Lotus Europa      4  95.1 3.77  1  1    5    2
## Ford Pantera L    8 351.0 4.22  0  1    5    4
## Ferrari Dino      6 145.0 3.62  0  1    5    6
## Maserati Bora     8 301.0 3.54  0  1    5    8
## Volvo 142E        4 121.0 4.11  1  1    4    2
```

```r
# Selecting column by their names
d[,c("wt","mpg")]
```

```
##                     wt  mpg
## Mazda RX4          2.620 21.0
## Mazda RX4 Wag      2.875 21.0
## Datsun 710         2.320 22.8
```

```
## Hornet 4 Drive      3.215 21.4
## Hornet Sportabout   3.440 18.7
## Valiant             3.460 18.1
## Duster 360          3.570 14.3
## Merc 240D           3.190 24.4
## Merc 230            3.150 22.8
## Merc 280            3.440 19.2
## Merc 280C           3.440 17.8
## Merc 450SE          4.070 16.4
## Merc 450SL          3.730 17.3
## Merc 450SLC         3.780 15.2
## Cadillac Fleetwood  5.250 10.4
## Lincoln Continental 5.424 10.4
## Chrysler Imperial   5.345 14.7
## Fiat 128            2.200 32.4
## Honda Civic         1.615 30.4
## Toyota Corolla      1.835 33.9
## Toyota Corona       2.465 21.5
## Dodge Challenger    3.520 15.5
## AMC Javelin         3.435 15.2
## Camaro Z28          3.840 13.3
## Pontiac Firebird    3.845 19.2
## Fiat X1-9           1.935 27.3
## Porsche 914-2       2.140 26.0
## Lotus Europa        1.513 30.4
## Ford Pantera L      3.170 15.8
## Ferrari Dino        2.770 19.7
## Maserati Bora       3.570 15.0
## Volvo 142E          2.780 21.4
```

```
# Excluding columns by their names [ this is a little tricky because simple negative sign doesn't work]
d[,!(names(d) %in% c("wt","mpg"))]
```

```
##                     cyl  disp  hp drat  qsec vs am gear carb
## Mazda RX4             6 160.0 110 3.90 16.46  0  1    4    4
## Mazda RX4 Wag         6 160.0 110 3.90 17.02  0  1    4    4
## Datsun 710            4 108.0  93 3.85 18.61  1  1    4    1
## Hornet 4 Drive        6 258.0 110 3.08 19.44  1  0    3    1
## Hornet Sportabout     8 360.0 175 3.15 17.02  0  0    3    2
## Valiant               6 225.0 105 2.76 20.22  1  0    3    1
## Duster 360            8 360.0 245 3.21 15.84  0  0    3    4
## Merc 240D             4 146.7  62 3.69 20.00  1  0    4    2
## Merc 230              4 140.8  95 3.92 22.90  1  0    4    2
## Merc 280              6 167.6 123 3.92 18.30  1  0    4    4
## Merc 280C             6 167.6 123 3.92 18.90  1  0    4    4
## Merc 450SE            8 275.8 180 3.07 17.40  0  0    3    3
## Merc 450SL            8 275.8 180 3.07 17.60  0  0    3    3
## Merc 450SLC           8 275.8 180 3.07 18.00  0  0    3    3
## Cadillac Fleetwood    8 472.0 205 2.93 17.98  0  0    3    4
## Lincoln Continental   8 460.0 215 3.00 17.82  0  0    3    4
## Chrysler Imperial     8 440.0 230 3.23 17.42  0  0    3    4
## Fiat 128              4  78.7  66 4.08 19.47  1  1    4    1
## Honda Civic           4  75.7  52 4.93 18.52  1  1    4    2
## Toyota Corolla        4  71.1  65 4.22 19.90  1  1    4    1
```

```
## Toyota Corona         4 120.1  97 3.70 20.01  1  0    3    1
## Dodge Challenger      8 318.0 150 2.76 16.87  0  0    3    2
## AMC Javelin           8 304.0 150 3.15 17.30  0  0    3    2
## Camaro Z28            8 350.0 245 3.73 15.41  0  0    3    4
## Pontiac Firebird      8 400.0 175 3.08 17.05  0  0    3    2
## Fiat X1-9             4  79.0  66 4.08 18.90  1  1    4    1
## Porsche 914-2         4 120.3  91 4.43 16.70  0  1    5    2
## Lotus Europa          4  95.1 113 3.77 16.90  1  1    5    2
## Ford Pantera L        8 351.0 264 4.22 14.50  0  1    5    4
## Ferrari Dino          6 145.0 175 3.62 15.50  0  1    5    6
## Maserati Bora         8 301.0 335 3.54 14.60  0  1    5    8
## Volvo 142E            4 121.0 109 4.11 18.60  1  1    4    2
```

Now we understand how to access various elements of a data set individually as well as in a group. Next is to understand sorting of the data. Its understandable that sorting of a data means rearranging rows by considering values of one or more columns in ascending or descending order.

This can be achieved by using function order in row indices.

```
d[order(d$vs,d$wt),c("vs","wt","mpg","gear")]
```

```
##                    vs    wt  mpg gear
## Porsche 914-2       0 2.140 26.0    5
## Mazda RX4           0 2.620 21.0    4
## Ferrari Dino        0 2.770 19.7    5
## Mazda RX4 Wag       0 2.875 21.0    4
## Ford Pantera L      0 3.170 15.8    5
## AMC Javelin         0 3.435 15.2    3
## Hornet Sportabout   0 3.440 18.7    3
## Dodge Challenger    0 3.520 15.5    3
## Duster 360          0 3.570 14.3    3
## Maserati Bora       0 3.570 15.0    5
## Merc 450SL          0 3.730 17.3    3
## Merc 450SLC         0 3.780 15.2    3
## Camaro Z28          0 3.840 13.3    3
## Pontiac Firebird    0 3.845 19.2    3
## Merc 450SE          0 4.070 16.4    3
## Cadillac Fleetwood  0 5.250 10.4    3
## Chrysler Imperial   0 5.345 14.7    3
## Lincoln Continental 0 5.424 10.4    3
## Lotus Europa        1 1.513 30.4    5
## Honda Civic         1 1.615 30.4    4
## Toyota Corolla      1 1.835 33.9    4
## Fiat X1-9           1 1.935 27.3    4
## Fiat 128            1 2.200 32.4    4
## Datsun 710          1 2.320 22.8    4
## Toyota Corona       1 2.465 21.5    3
## Volvo 142E          1 2.780 21.4    4
## Merc 230            1 3.150 22.8    4
## Merc 240D           1 3.190 24.4    4
## Hornet 4 Drive      1 3.215 21.4    3
## Merc 280            1 3.440 19.2    4
## Merc 280C           1 3.440 17.8    4
## Valiant             1 3.460 18.1    3
```

As you can see this outputs a sorted data set which first sorted by variable `vs` and then with in the groups created by `vs` values , data is sorted by variable `wt`. Default order of this sorting is ascending. By putting a negative sign in front of the numeric variable name inside the order function you can achieve descending order sorting for that particular variable.

```
d[order(d$vs,-d$wt),c("vs","wt","mpg","gear")]
```

```
##                     vs    wt  mpg gear
## Lincoln Continental  0 5.424 10.4    3
## Chrysler Imperial    0 5.345 14.7    3
## Cadillac Fleetwood   0 5.250 10.4    3
## Merc 450SE           0 4.070 16.4    3
## Pontiac Firebird     0 3.845 19.2    3
## Camaro Z28           0 3.840 13.3    3
## Merc 450SLC          0 3.780 15.2    3
## Merc 450SL           0 3.730 17.3    3
## Duster 360           0 3.570 14.3    3
## Maserati Bora        0 3.570 15.0    5
## Dodge Challenger     0 3.520 15.5    3
## Hornet Sportabout    0 3.440 18.7    3
## AMC Javelin          0 3.435 15.2    3
## Ford Pantera L       0 3.170 15.8    5
## Mazda RX4 Wag        0 2.875 21.0    4
## Ferrari Dino         0 2.770 19.7    5
## Mazda RX4            0 2.620 21.0    4
## Porsche 914-2        0 2.140 26.0    5
## Valiant              1 3.460 18.1    3
## Merc 280             1 3.440 19.2    4
## Merc 280C            1 3.440 17.8    4
## Hornet 4 Drive       1 3.215 21.4    3
## Merc 240D            1 3.190 24.4    4
## Merc 230             1 3.150 22.8    4
## Volvo 142E           1 2.780 21.4    4
## Toyota Corona        1 2.465 21.5    3
## Datsun 710           1 2.320 22.8    4
## Fiat 128             1 2.200 32.4    4
## Fiat X1-9            1 1.935 27.3    4
## Toyota Corolla       1 1.835 33.9    4
## Honda Civic          1 1.615 30.4    4
## Lotus Europa         1 1.513 30.4    5
```

Combining Data Sets

There are three ways in which one or more data sets can be combined:

- Horizontal Stacking
- Vertical Stacking
- Merging based on Keys

Horizontal stacking can be achieved by `cbind`. Requirement is that number of rows in all the input data sets should be same. Similarly vertical stacking can be done with `rbind`, requirement being same number of columns.

I'm leaving the vertical and horizontal stacking part for you to explore on your own. To explore key based merging we'll be using functions `join_*` which is found in package `dplyr`. You can install by entering name `dplyr` in the install window which appears after you click on `Install` button in `Packages` tab in `Rstudio`.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
df1 = data.frame(CustomerId=c(1:6),Product=c(rep("Toaster",3),rep("Radio",3)))
df2 = data.frame(CustomerId=c(3,4,7,8),State=c(rep("Alabama",2),rep("Ohio",2)))

inner=inner_join(df1,df2,by="CustomerId")
left=left_join(df1,df2,by="CustomerId")
right=right_join(df1,df2,by="CustomerId")
full=full_join(df1,df2,by="CustomerId")
semi=semi_join(df1,df2,by="CustomerId")
anti=anti_join(df1,df2,by="CustomerId")
```

inner join : contains rows have key values which are common to both the data sets.

```
inner
```

```
##   CustomerId Product   State
## 1          3 Toaster Alabama
## 2          4   Radio Alabama
```

left join: contains all rows having key values which are present in left data sets.

```
left
```

```
##   CustomerId Product   State
## 1          1 Toaster    <NA>
## 2          2 Toaster    <NA>
## 3          3 Toaster Alabama
## 4          4   Radio Alabama
## 5          5   Radio    <NA>
## 6          6   Radio    <NA>
```

right join: contains all rows having key values which are present in right data sets.

```
right
```

```
##   CustomerId Product   State
## 1          3 Toaster Alabama
## 2          4   Radio Alabama
## 3          7    <NA>    Ohio
## 4          8    <NA>    Ohio
```

full join: contains rows with all the key values.

```
full
```

```
##   CustomerId Product   State
## 1          1 Toaster    <NA>
## 2          2 Toaster    <NA>
## 3          3 Toaster Alabama
## 4          4   Radio Alabama
## 5          5   Radio    <NA>
## 6          6   Radio    <NA>
## 7          7    <NA>    Ohio
## 8          8    <NA>    Ohio
```

semi join : contains columns only from the first data set but only those ids which are common to both the data sets

```
semi
```

```
##   CustomerId Product
## 1          3 Toaster
## 2          4   Radio
```

anti join: contains columns only from the first data set but only those ids which are exclusive to the first/left data set

```
anti
```

```
##   CustomerId Product
## 1          6   Radio
## 2          5   Radio
## 3          2 Toaster
## 4          1 Toaster
```

Remember that if a key value is present only in one data set, corresponding variable values coming from another data set are going to be missing [NA].

For loops

For loops enable you to do iterative operations. Here is an example :

```
x=sample(1:100,10)
for (i in 1:10){
  print(x[i]**2)
}
```

```
## [1] 7921
## [1] 3844
## [1] 676
## [1] 7056
## [1] 1764
## [1] 1369
## [1] 1936
## [1] 441
## [1] 49
## [1] 9604
```

Syntactically for loop is very straight forward. There are few parts. `i` is the iteration counter. It takes values from the vector which comes after the keyword `in`. In this case vector is `1:10`. There is no necessity of this vector to contain contiguous values.

with in the curly braces `{}` , you can put in whatever you want to do iteratively. There can be multiple lines of codes inside. Lets take an example where we want to print squares of even numbers only. We can customize the vector from which our counter is going to take values.

```
for(j in which(x%%2==0)){
  print(x[j]**2)
}
```

```
## [1] 3844
## [1] 676
## [1] 7056
## [1] 1764
## [1] 1936
## [1] 9604
```

Writing your own functions

We'll start with an example for a function gives range of a numeric vector. If you pass a character vector it prints a error message.

```
myfunc=function(x){
  range=max(x)-min(x)
  return(range)
}

myfunc(1:10)
```

```
## [1] 9
```

Elements of function code are `name` of the function. Make sure that it is not same as existing R functions. You can quickly check for that by looking up documentation for that function. If function with the same name exists , documentation will show up.

Next part is arguments which the function takes, they are listed in brackets beside the keyword `function`. Although in the example , we have given only one argument, a function can take multiple arguments as well.

inside the curly braces whatever processing you want to do is contained. In the end line there you return the result of your computation by using function `return`. Returning a result is not necessary. There can be functions which do something and just print the results , instead of returning anything.

Functions try to process whatever input you pass, if input does not make sense in that context you'll get an error.

```
myfunc(rep("a",10))
```

```
## Error in max(x) - min(x): non-numeric argument to binary operator
```

You can take care of this by processing different kind of inputs differently, *[by using if else blocks]*

```
myfunc=function(x){
if(class(x) %in% c("numeric","integer")){
    return(max(x)-min(x))
  }else{
    print("Vector is not of supported type")
  }
}
myfunc(1:10)
```

```
## [1] 9
```

```
myfunc(rep("a",10))
```

```
## [1] "Vector is not of supported type"
```

Limitation to a function is that it can not return more than one object. If you need to return multiple values , you can do so by binding them into one object. Lets create a function which returns a five point summary for any vector. *[min,max,mean,sd,median]*

```
mysummary=function(x){
  s1=min(x)
  s2=max(x)
  s3=mean(x)
  s4=sd(x)
  s5=median(x)
  summary=list(min=s1,max=s2,mean=s3,sd=s4,median=s5)
  return(summary)
}
mysummary(2:90)
```

```
## $min
## [1] 2
##
## $max
## [1] 90
##
```

```
## $mean
## [1] 46
##
## $sd
## [1] 25.83602
##
## $median
## [1] 46
```

you can access these results as follows

```
t=mysummary(2:90)
t$median
```

```
## [1] 46
```

```
t$max
```

```
## [1] 90
```

Next we'll learn about how to give default arguments to your function. There is no complex process involved here. While declaring your arguments , you can pass default values. Lets understand that with an example. Modifying the function that we wrote above, we'll now write a function, which takes input a vector x and a value y. Function first multiplies the vector x with value y and then gives the summary.

```
mysummary2=function(x,y){
  x=x*y
  s1=min(x)
  s2=max(x)
  s3=mean(x)
  s4=sd(x)
  s5=median(x)
  summary=list(min=s1,max=s2,mean=s3,sd=s4,median=s5)
  return(summary)
}
mysummary2(1:10,2)
```

```
## $min
## [1] 2
##
## $max
## [1] 20
##
## $mean
## [1] 11
##
## $sd
## [1] 6.055301
##
## $median
## [1] 11
```

Call to function with both arguments being passed properly gives you an expected result. Now lets try to miss out on one of the arguments.

```
mysummary2(1:10)
```

```
## Error in mysummary2(1:10): argument "y" is missing, with no default
```

This throws an error for missing arguments. Lets give some default values to our arguments

```
mysummary2=function(x=2:30,y=1){
  x=x*y
  s1=min(x)
  s2=max(x)
  s3=mean(x)
  s4=sd(x)
  s5=median(x)
  summary=list(min=s1,max=s2,mean=s3,sd=s4,median=s5)
  return(summary)
}
mysummary2(1:10,4)
```

```
## $min
## [1] 4
##
## $max
## [1] 40
##
## $mean
## [1] 22
##
## $sd
## [1] 12.1106
##
## $median
## [1] 22
```

```
mysummary2(1:10)
```

```
## $min
## [1] 1
##
## $max
## [1] 10
##
## $mean
## [1] 5.5
##
## $sd
## [1] 3.02765
##
## $median
## [1] 5.5
```

```
mysummary2()
```

```
## $min
## [1] 2
##
## $max
## [1] 30
##
## $mean
## [1] 16
##
## $sd
## [1] 8.514693
##
## $median
## [1] 16
```

You can see that all kind of calls to function work. Whatever arguments is missing, takes default value inside the function.

Last thing we'll talk about is using name of the arguments while calling your function.If you do not use name of the argument [as we have done in multiple last examples], R figures out which value belongs to which argument by their position in the input. But if you use explicit names of the arguments , position doesn't matter.

See here :

```
mysummary2(x=1:10,y=4)
```

```
## $min
## [1] 4
##
## $max
## [1] 40
##
## $mean
## [1] 22
##
## $sd
## [1] 12.1106
##
## $median
## [1] 22
```

```
mysummary2(y=4,x=1:10)
```

```
## $min
## [1] 4
##
## $max
## [1] 40
##
```

```
## $mean
## [1] 22
##
## $sd
## [1] 12.1106
##
## $median
## [1] 22
```

```
mysummary2(y=3)
```

```
## $min
## [1] 6
##
## $max
## [1] 90
##
## $mean
## [1] 48
##
## $sd
## [1] 25.54408
##
## $median
## [1] 48
```

```
mysummary2(x=3:10)
```

```
## $min
## [1] 3
##
## $max
## [1] 10
##
## $mean
## [1] 6.5
##
## $sd
## [1] 2.44949
##
## $median
## [1] 6.5
```

Optimizing for loops

Before we start with it , you need to understand a little about, how to convert a vector to matrix, so that the example makes sense from the very beginning.

```
x=1:9
```

Here, x is a simple vector containing 9 values. We can put these into mXn matrix by telling R in how many rows and columns we want in the resultant matrix.

```
matrix(x,3,3)
```

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

This gives you a simple matrix with 3 rows and 3 columns. By default values are filled in columns of matrix in sequence. You can alter that behavior by explicitly setting `byrow` to `FALSE`.

```
n=100000
myvec=sample(1:10, n*9, TRUE)
X = as.data.frame(matrix(myvec, n, 9))
```

Here we are creating myvec which contain 10^{5}*9 random values between 1 to 10. using `matrix` we rearrange this into a matrix of 10^{5} rows and 9 columns. Function `as.data.frame` simply coerces this matrix into a data frame. this is what X looks like. [printing top 6 values]

```
head(X)
```

```
##   V1 V2 V3 V4 V5 V6 V7 V8 V9
## 1  4  7  6  7  4  8  8  1  8
## 2  1  2  6  3  2  5  2  6  7
## 3  2  2  7  9  2 10  2  4  7
## 4  2 10  3  4  5  2  5  1  5
## 5  8  3  9  1  5  3 10  8  1
## 6  3  3  9  4  6  9  3  7  6
```

Now lets talk about the real problem. The data frame that you see here , your task is to create a 10th variable here according to following rules

1) If current and previous values of 6th variable and current and previous value of 3rd variable are equal then variable is equal to summation of cu current value of 9th variable and previous value of 10th variable
2) Otherwise it simply takes value equal to current value of 9th variable

A simple for loop will be as follows:

```
for (i in 1:nrow(X)){

    if (i > 1) {
      if ((X[i,6] == X[i-1,6]) & (X[i,3] == X[i-1,3])) {
        X[i,10] = X[i,9] + X[i-1,10]
      }
      else {
        X[i,10] = X[i,9]
      }
    }
    else {
      X[i,10] = X[i,9]
    }
  }
```

To asses its performance (run time), we are going to wrap it up in a function, then we can use function `system.time` to find how much exact time it takes to execute this call.

```
dayloop2 = function(temp){

  for (i in 1:nrow(temp)){

    if (i > 1) {
      if ((temp[i,6] == temp[i-1,6]) & (temp[i,3] == temp[i-1,3])) {
        temp[i,10] = temp[i,9] + temp[i-1,10]
      }
      else {
        temp[i,10] = temp[i,9]
      }
    }
    else {
      temp[i,10] = temp[i,9]
    }
  }
  return(temp)
}

(t1=system.time(dayloop2(X)))
```

```
##    user  system elapsed
##  66.968  23.293  91.270
```

Performance that you see here is not very impressive . Imagine X which is 100 times this . How can we make this better. Although writing a time efficient program depends on the context of the problem, however there are couple of simple pointers which you can follow and we'll demonstrate their effectiveness by improving performance of the function written above [In effect we will learn how to work around `for loop`'s inefficiency because that is the bottleneck in the function we are optimizing]

First pointer : Accessing data.frame is a time consuming process. Reduce it as much as you can.

in the context of the current problem; in the previous implementation we are assigning value to 10th variable in each iteration. What we could do is to store this values in separate vector instead and then when the for loop is done, attach it to data set.

Lets do that , look at the code below . instead of assigning values to `X[i,10]`, we are storing the values in the vector `res` and at the end we are adding it to the data frame. lets see how much performance improvement we see.

```
dayloop2_A = function(temp){

  res = numeric(nrow(temp))

  for (i in 1:nrow(temp)){

    if (i > 1) {
      if ((temp[i,6] == temp[i-1,6]) & (temp[i,3] == temp[i-1,3])) {
        res[i] = temp[i,9] + res[i-1]
      } else {
        res[i] = temp[i,9]
      }
```

```
    } else {
      res[i] = temp[i,9]
    }
  }
  temp$V10 = res
  return(temp)
}

(t2=system.time(dayloop2_A(X)))
```

```
##    user  system elapsed
##   9.496   0.063   9.617
```

We see a significant improvement in the efficiency by a factor of 9.49 . lets see if we can do better. We'll keep on modifying the previous function in every subsequent step and see the improvement.

Second Pointer : Find out iterative operations which can be replaced with vector operations [which will happen in one go, and are much better optimized internally]

One very prominent iterative operation that we do is to check if current values of 3rd and 6th variables are same as their previous values. How can we replace that with a vector operation. Lets go through simpler example first to understand the idea and then we'll implement the same idea into our code of function which we are trying to optimize.

```
set.seed(2)
x=sample(1:5,20,replace = T)
x
```

```
##  [1] 1 4 3 1 5 5 1 5 3 3 3 2 4 1 3 5 5 2 3 1
```

Lets say we want to get the result of comparing values in x with their immediate previous values. We can always write a for loop to do so. We will construct a logical vector, where we will set the first value as FALSE because first value of x has nothing to compare against.

```
y=FALSE
for(i in 2:20){
  if(x[i]==x[i-1]){
    y[i]=TRUE
  }else{
    y[i]=FALSE
  }
}
y
```

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

Now lets see if we can get this without writing a for loop. If you think about it, you will realize that comparing values of x with their previous counterparts is same as comparing x with its shifted counterpart.

here is x except its first value

```
x[-1]
```

```
##  [1] 4 3 1 5 5 1 5 3 3 3 2 4 1 3 5 5 2 3 1
```

here is x except its last value

```
x[-length(x)]
```

```
##  [1] 1 4 3 1 5 5 1 5 3 3 3 2 4 1 3 5 5 2 3
```

we can simply compare these two vectors , which will result into a logical vector. If we put a FALSE value at the beginning too, we'd have achieved the comparison without writing a for loop.

```
z=c(FALSE,x[-1]==x[-length(x)])
z
```

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

In the example given above , we achieved the comparison and stored in a object without writing a for loop. Lets implement the same idea in out function that we are trying to optimize.

```
dayloop2_B = function(temp){
  # Using the idea explained above to do comparsion without using for loop. the result
  # is stored in vector cond.
  cond = c(FALSE, (temp[-nrow(temp),6] == temp[-1,6]) &
             (temp[-nrow(temp),3] == temp[-1,3]))
  # along with that we are also getting rid of assigning values of res to temp[,9]
  # when condition is not true. Instead of that , from the very beginning we are
  # assigning it equal to temp[,9]. We'll change it whenever the condition is true
  res = temp[,9]

  for (i in 1:nrow(temp)) {
    if (cond[i]) {res[i] = temp[i,9] + res[i-1]}
  }

  temp$new = res
  return(temp)

}

(t3=system.time(dayloop2_B(X)))
```

```
##    user  system elapsed
##   0.062   0.002   0.063
```

We see a fur further significant improvement in the efficiency by a factor of 1448.73 .

Third Pointer: For loops do not need to run on a contiguous vector of indices . Subset the index vector for values, over which you really need to do processing.

Lets use this tip to further optimize our function:

```
dayloop2_c = function(temp){
  cond = c(FALSE, (temp[-nrow(temp),6] == temp[-1,6]) &
             (temp[-nrow(temp),3] == temp[-1,3]))

  res = temp[,9]

  # previously we were running for loop for all rows in data frame
  k=1:nrow(temp)
  # we'll subset it to only those value where the condition is true
  # because we need to change the value of "res" in only those cases

  k=k[cond]

  for (i in k) {
    res[i] = res[i] + res[i-1]
  }

  temp$new = res
  return(temp)
}

(t4=system.time(dayloop2_c(X)))
```

```
##    user  system elapsed
##   0.008   0.001   0.010
```

We have improved performance , following these three tips by a factor of 9127 . This is a very significant improvement. Keep the three pointers in mind and you are good.

We'll conclude our discussion on R programming here. We'll keep on learning more of R programming as we progress in the course , but that'll be more off contextual kind. As far as the basics of R are concerned we are covered here.

In case of any doubts , please take to Q&A forum on LMS.

Prepared By: lalit Sachan

Contact: lalit.sachan@edvancer.in