Hand on with R - II

Task 9: Functions

Fundamental building block

- small pieces of reusable code
- Characterized by the name followed by parenthesis
- Built in
- User defined

Sys.Date()

Will return your computer's environment date

mean()

take vector as input and return the mean of the vector

try

>mean(c(3,7,9))

Let write our first function

1) Declaration: define your function

```
My_first_function <- function(x){
  return (x)
}
```

2) Call your function

```
My_first_function('function 1')
```

R computation 2 - slogan

1) Everything that exists is an object.
2) Everything that happens is a function call

To see the source code of a function…..

Type the function name

My_first_function

Let replicate mean function called my_mean which take a vector as input

```
my_mean <- function(my_vector) {
  # Write your code here!
  # Remember: the last expression evaluated will be returned!
}
```

After you create the function test it with the following vector

c(4,5,10)

Function with default arguments. You can set default values for a function's arguments

Make a function to calculate reminder, passing number and the divisor as the arguments and the function return the reminder. **Also pass default values to the function.**

```
remainder <- function(num , divisor) {
  your code!
}
```

Let test the reminder function by calling the function

>remainder(10)

What will be the output?

Similarly try

>remainder(11,5)

You can also explicitly specify the argument. For example

remainder(divisor = 5 , num = 67)

Strength of R – R can also partially match arguments

Try

remainder(14, div = 12)

But you should code as easy as possible so that one can understand or debug

how do you see the arguments besides using the help function (? Function name)

>args(remainder)

Did you notice there is something new the above in the above argument?

Like Functional programming - you can pass functions as arguments!

Just like you pass data to a function you can pass a function to a function

Let us give a try

1) Create a function that uses two arguments
    a. Func
    b. Dat

The objective is to pass a function name (sum, mean, median, etc) and a vector/ data to evaluate

Let make evaluate function

evaluate <- function(func, dat){

write your code!

}

Then calculate the standard deviation of the following vector c(1.4, 3.6, 7.9, 8.8)

Let's work on anonymous functions work. For the first argument of the evaluate function we're going to write a tiny function that fits on one line. In the second argument, we'll pass some data to the tiny anonymous function in the first argument.

evaluate(function(x){x+1}, 6)

Notice that the first argument is an anonymous function I.e., with a function name which takes x as an argument and returns x+1. The second argument is a number/ data

- Use the evaluate function to return the second number of the vector c(10,15,18)
- Use the evaluate function to return the last element of the vector c(10,15,18) (hint : length)

Paste function

Type ?paste to see the documentation

… means ellipses / one or more R objects

How can we use the ellipses from a function

```
Func1 <- function(...){
  args <- list(...)
  place <- args[["place"]]
  adjective <- args[["adjective"]]
  noun <- args[["noun"]]
  paste("News from", place, "today where", adjective, "students took to the streets in protest of the new", noun, "being installed on campus.")
}
```

Lets use the function Func1.

Make sure to pass the arguments

Binary functions: take two inputs left and right (+, -, /, *)

You can make your own binary function in R : User defined binary operator

%[whatever]%

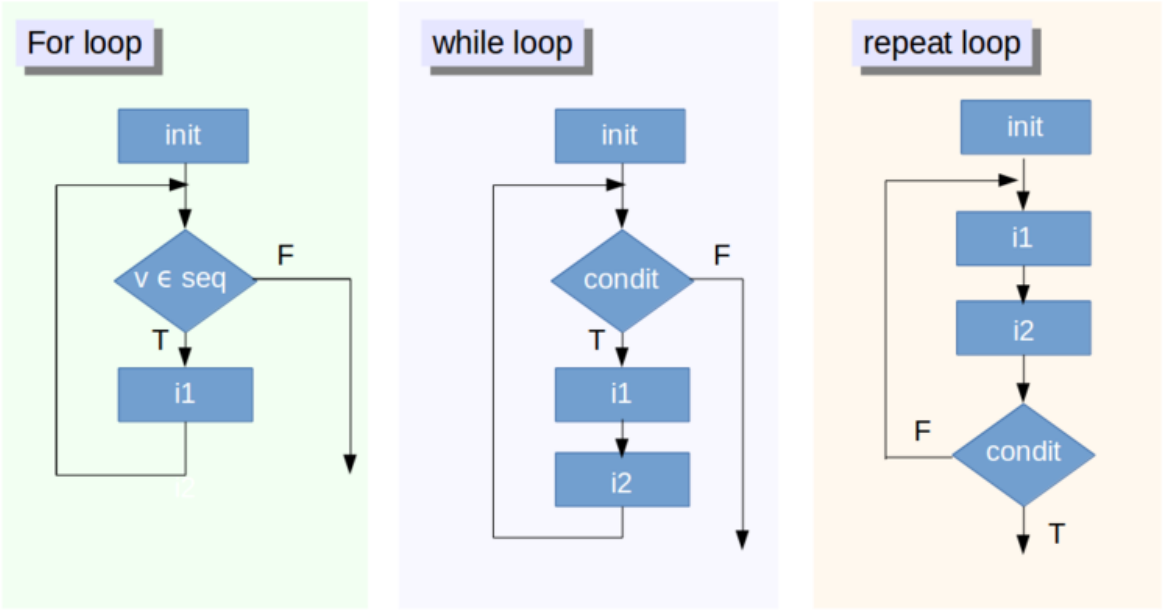
Write your own binary operator below from absolute scratch! Your binary operator must be called %con% so that the expression:
"Good" %con% "job!" will evaluate to: "Good job!"

```
‘%con%’ <- function(){ # Remember to add arguments!
 }
```

Now call your binary operator and print “This”, “is”, “an”, “interesting”, “session”)

Task 12: Loops

"Looping", "cycling", "iterating" or just replicating instructions is an old practice that originated well before the invention of computers. It is nothing more than automating a multi-step process by organizing sequences of actions or 'batch' processes and by grouping the parts that need to be repeated.



Three kind of loop in R (or any other programming language)

The FOR Loop.

Syntax:

```
for (val in sequence)
{
   statement
}
```

rep()

Often we want to start with a vector of 0's and then modify the entries in later code. R makes this easy with the replicate function rep()

rep(0, 10) makes a vector of of 10 zeros.

```
>x <- rep(0,10)
>x
[1] 0 0 0 0 0 0 0 0 0 0
```

```
# rep() will replicate almost anything
>x <- rep(2,6)
>x
[1] 2 2 2 2 2 2
>x <- rep('abc',5)
>x
[1] "abc" "abc" "abc" "abc" "abc"
>x <- rep(1:4,5)
>x
[1] 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
```

The first statement

```
for(i in x){
}
```

'for loops' let us repeat (loop) through the **elements in a vector** and run the same code on each element

Let's see an example

```
for (j in 1:5)
{
  print(j^2)
}
```

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

We can also capture the results of our loop in a list, matrix or a data frame
First we create a vector and then we fill in its values

```
n = 5
x = rep(0,n)
for (j in 1:n)
{
  x[j] = j^2
}
x
[1] 1  4  9 16 25
```

You always wanted to know the sum of the first 100 squares.

```
n = 100
x = rep(0,n)
```

```
for (j in 1:n)
{
  x[j] = j^2
}
sum(x)
```

For loops are truly valuable when the calculation is more complicated and we can't do it exactly or with built in R functions.

Nested For loop

Loop with in a loop
Let see an example:

```
n= 5
for (i in seq(n))
  {
  for( j in seq(i))
    {
       cat(paste("*"))
  }
  print("", quote = FALSE)
}
```

Output

```
*[1]
**[1]
***[1]
****[1]
*****[1]
```

The While loop

```
while (test_expression)
{
   statement
}
```

Let’s see an example

```
n <- 5

while (n>0)
{
```

```
 j = 1
 while(j <= n)
 {
  cat(paste("*"))
  j = j+1
 }
 print("", quote = FALSE)
 n <- n-1
 }
```

Output

```
***** [1]
**** [1]
*** [1]
** [1]
* [1]
```

Break and next

A break statement is used inside a loop (repeat, for, while) to stop the iterations and flow the control outside of the loop.

A next statement is useful when we want to skip the current iteration of a loop without terminating it.

An Example

```
x <- 1:10
for (val in x) {
 if (val == 3){
  next
 }
 else if (val ==7){
  break
 }
 print(val)
}
```

Output

```
[1] 1
[1] 2
[1] 4
[1] 5
[1] 6
```

Task 11: lapply and sapply function

Loop function in R

Close relatives are (vapply() and tapply() next task)

Offers a concise and convenient means to implement the split-apply-combine strategy for data analysis

For this exercise, we will use churn data set.

Type

```
> library(C50)
> data(churn)
```

Load the train churn set in a new variable

head()
Use the head function to get a glimpse of the data

dim()
dim function will give you the dimension if the data.

```
> dim(churnTrain)
[1] 3333   20
```

Show that the churnTrain has 3333 observation and 20 variables

Class()

That just tells us that the entire dataset is stored as a 'data.frame', which doesn't answer our question. What we really need is to call the class() function on each individual column. While we could do this manually (i.e. one column at a time) it's much faster if we can automate the process.

Sounds like a loop!

lapply()

The lapply() function takes a list as input, applies a function to each element of the list, then returns a list of the same length as the original one. Since a data frame is really just a list of vectors, we can use lapply() to apply the class() function to each column of the churnTrain dataset. Let's see!

```
> class_list <- lapply(churnTrain,class)
```

Observe first argument is data set and the second is the function without brackets.

Print the class_list

So ‘l’ stands for list that will return a list. Let’s validate

class(class_list)

This will return a list of size 20. One element for each column/ variable. Remember, list is used to store data of different data type. We can make the output more compact by if we could represent as a vector rather than a list.

In this case, since every element of the list returned by lapply() is a character vector of length one (i.e. "integer" and "vector"), class_list can be simplified to a character vector. To do this manually, type

>as.character(class_list).

sapply() allows you to automate this process by calling lapply() behind the scenes, but then attempting to simplify (hence the 's' in 'sapply') the result for you.

Let's check

> class_vect <- sapply(churnTrain,class)

> class(class_vect)

It should return a "character"

Let's practice lapply and sapply some more….

The churnTrain set has predictor variable

If I want to know the mean of each of the variable?

Use the apply function!

Hint: use a new variable to store the predictors variables. Then use the lapply function to calculate the mean of each of the variables.

Likewise use sapply()!

try other functions in the lapply function like sum, sd, median, unique etc.

range()

Return the minimum and the maximum of its argument. Use lapply or sapply to get the range

As the range function return two dimensions compared to other functions the sapply function returns a matrix instead of a vector.

Use class function to validate.

Can you make a user defined function using lapply function?

Task 12: vapply and tapply

In the last lesson, you learned about the two most fundamental members of R's *apply family of functions: lapply() and sapply(). Both take a list as input, apply a function to each element of the list, then combine and return the result. lapply() always returns a list, whereas sapply() attempts to simplify the result

You may want to reacquaint yourself with the data by using functions like

dim(),
head(),
tail()
str(),
and summary()

unique() function returns the unique values. Therefore, sapply(churnTrain, unique) returns a list containing one vector of unique values for each column of the dataset.

Sometimes you want the correct format in that case sapply() tries to 'guess' the correct format of the result, vapply() allows you to specify it explicitly. If the result doesn't match the format you specify, vapply() will throw an error, causing the operation to stop.

Try vapply(TrainChurn, unique, numeric(1)), which says that you expect each element of the result to be a numeric vector of length 1. Since this is NOT actually the case, YOU WILL GET AN ERROR.

Remember sapply(TrainChurn, class) will return a character vector containing the class of each column in the dataset.

If we wish to be explicit about the format of the result we expect, we can use vapply(ChurnTrain , class, character(1)). The 'character(1)' argument tells R that we expect the class function to return a character vector of length 1 when applied to EACH column of the churn dataset. Try it

Lets look at tapply() function

Use ?tapply to pull up the documentation

Use table() to see how many account length fall into each group.

> table(churnTrain$account_length)

> table(churnTrain$total_day_minutes)

tapply(churnTrain$account_length, churnTrain$total_day_minutes ,mean) will tell us the proportion of account length within each total day minutes plan group.

Now let see how many customer have churned within each account length.

> tapply(churnTrain$account_length, churnTrain$churn ,summary)

What is the maximum account length (in days) for the customers who have churned?

1: 102.7
2: 5.00
3: 127.00
4: 225.0
5: 243.00

Task 13: Looking at Data

Whenever you're working with a new dataset, the first thing you should do is look at it!

- What is the format of the data?
- What are the dimensions?
- What are the variable names?
- How are the variables stored?
- Are there missing data?
- Are there any flaws in the data?

Read the file(don't worry about the command ….

>data <- read.csv(file = "path…/plant.csv")

This task will teach you how to answer these questions and more using R's built-in functions.

List the variables in your work space.

Use the class() function to get a clue as to the overall structure of the dataset ?

Data frame

Since the dataset is stored in a data frame, we know it is rectangular. In other words, it has two dimensions (rows and columns) and fits neatly into a table or spreadsheet. Use dim(plants) to see exactly how many rows and columns we're dealing with.

Use could also use the nrow() command to get the number of rows

> nrow(plants)

And

>ncol(plants)

to get the number of columns

If you are curious as to how much space the dataset is occupying in memory, you can use object.size(plants)

> object.size(plants)

Now that you have an idea of the overall structure of the dataset, let's look inside the dataset.

>names(plants)

Will give you a character vector of all the name of the columns

Usually the dataset is large enough, so it is impractical to view the entire table, so

head() function allows you to preview the top of the dataset

By default it give the first 6 rows as output. Now, suppose you want 10 rows?

tail() function is similar to the head function except it show the last few rows.

After previewing the top and bottom of the data, you probably noticed lots of NAs, which are R's placeholders for missing values. Use summary(plants) to get a better feel for how each variable is distributed and how much of the dataset is missing.

> summary(plants)

You can see that R truncated the summary for Active_Growth_Period by including a catch-all category called 'Other'. Since it is a categorical/factor variable, we can see how many times each value occurs in the data with table(plants$Active_Growth_Period).

Now the best function:

str()

for understanding the structure of the data

try it ?

The beauty of str() is that it combines many of the features of the other functions you've already seen, all in a concise and readable format. At the very top, it tells us that the class of plants is 'data.frame' and that it has 5166 observations and 10 variables. It then gives us the name and class of each variable, as well as a preview of its contents.

Task 14: Simulation

One of the great advantages of using a statistical programming language like R is its vast collection of tools for simulating random numbers.

Assumption is familiarity with common probability distributions!

We have already seen the function

>sample()

Let's simulate rolling four six-sided dice:

>sample(1:6, 4, replace = TRUE).

This command instructs R to randomly select four numbers between 1 and 6, WITH replacement.

Question: Let sample 10 numbers between 1 and 20 without replacement?

LETTERS/letters is predefined variable in R. Take a look!

>sample(LETTERS)

Take a sample of all the 26 letters equal to the size of the sample

Now, suppose we want to simulate 100 flips of an unfair two-sided coin. This particular coin has a 0.3 probability of landing 'tails' and a 0.7 probability of landing 'heads'.

tail represented by 1 and head by 0

Use sample() to draw a sample of size 100 from the vector c(0,1), with replacement. Since the coin is unfair, we must attach specific probabilities to the values 0 (tails) and 1 (heads) with a fourth argument, prob = c(0.3, 0.7). Assign the result to a new variable called flips.

> flips <- sample(c(0,1), 100 , prob = c(0.3,0.7), replace = TRUE)

View the content

Use the sum function to verify

> sum(flips)

A coin flip is a binary outcome (0 or 1) and we are performing 100 independent trials (coin flips), so we can use rbinom() to simulate a binomial random variable. Pull up the documentation for rbinom() using

?rbinom

Each probability distribution in R has an r*** function (for "random"), a d*** function (for "density"), a p*** (for "probability"), and q*** (for "quantile"). We are most interested in the r*** functions in this lesson, but I encourage you to explore the others on your own.

> rbinom(1,size = 100, prob = 0.7) this is the probability of success

Equivalently, if we want to see all of the 0s and 1s, we can request 100 observations, each of size 1, with success probability of 0.7. Give it a try, assigning the result to a new variable called flips2.

> flips2 <- rbinom(n=100, size =1, prob = 0.7)

Calculate the sum(), it should around 70.

Normal distribution

The standard normal distribution has mean 0 and standard deviation 1. As you can see under the 'Usage' section in the documentation, the default values for the 'mean' and 'sd' arguments to rnorm() are 0 and 1, respectively. Thus, rnorm(10) will generate 10 random numbers from a standard normal distribution. Give it a try.

To get a better understanding pull up the documentation.

Now dot the same for mean = 100 and sd = 25

There are other functions like Poisson distribution and other.

Simulation is practically a field of its own and we've only skimmed the surface of what's possible. I encourage you to explore these and other functions further on your own.

Task 15: Date and Time

Dates are represented by the **'Date' class** and times are represented by the **'POSIXct' and 'POSIXlt' classes.** Internally, dates are stored as the number of days since 1970-01-01 and times are stored as either the number of seconds since 1970-01-01 (for 'POSIXct') or a list of seconds, minutes, hours, etc. (for 'POSIXlt')

Let start!

Get he system date and store in a variable d1

> d1 <- Sys.Date()

Use the class function to confirm d1's class

Now use the unclass() function to look at what date look like internally

This will give you the number of days since 1970-01-01!

However, just print the date on the console

Now suppose you want to make a reference to a date before 1970?

Store the date in a new variable d2

> d2 <- as.Date("1969-01-01")

Use unclass() function to see what d2 looks like

Now lets us work with time (how R stores time). Use the Sys.time() and store in t1 variable.

View the content in t1

Use the class() function to see the class

As mentioned earlier, POSIXct is just one of two ways that R represents time information. (You can ignore the second value above, POSIXt, which just functions as a common language between POSIXct and POSIXlt.) Use unclass() to see what t1 looks like internally -- the (large) number of seconds since the beginning of 1970. -- the (large) number of seconds since the beginning of 1970.

unclass(t1)

Now suppose we just want the minutes from the time stored in t1, we can access them t1$min

> t1$min

Now let us extract some more useful information from any of the objects

Try the following

weekdays(),

months(),
and quarters()

Often, the dates and times in a dataset will be in a **format that R does not recognize**. The strptime() function can be helpful in this situation.
To see how it works, store the following character string in a variable called t3: "July 19, 2017 08:34" (with the quotes).
> t3 <- "July 19, 2017 08:34"

Use use strptime(t3, "%B %d, %Y %H:%M") Format and store in t4 and print it.

> t4
[1] "2017-07-19 08:34:00 CDT"

Finally there are other operations that you can perform on date and time function, including arithmetic (+,-) and comparison (<,>,==,etc)

Try Sys.time() > t1

So time has passed, but how much?

There are other operation like difftime() function that gives you more control, that allows you to specify the units
> difftime(Sys.time(), t1, units = 'days')

Task 16: Basic Graphs

For this task we will use the cars dataset. To load the car dataset type

>data(cars)

Our main goal is to introduce various plotting functions and their arguments. All the output would look more interesting with larger, more complex data sets.

Pull help page for cars

Explore the data: head(), tail() , summary(), str(), dim(), name(), nrow()

Plotting
Use plot() function – scatter plot

> plot(cars)

R tries very hard to give you something sensible given the information. As this dataset has just two columns. So R assumes that you want to plot one column versus the other.
Second, we have not provided labels for the axis, so R take the columns name as the labels
Third, it creates axis on its own.
Fourth, it takes all the default values

Open help page for plot
>?plot

Now use the plot() function to plot speed on the X axis and distance on the y axis

There are other ways to call the plot function – using a formula
>plot(dist ~ speed, cars)

‘ ~ ’ is a tilde function left ~ right (dependent ~ predictors)

Remember ‘…’ ellipses

Recreate the plot with the label of the x-axis set to "Speed"

xlab = “speed”
ylab = “stopping”
main title = “My Plot”
sub = “Subtitle”

let change the color

Plot cars while limiting the x-axis to 10 through 15. (Use xlim = c(10, 15) to achieve this effect.)

We can also plot triangles

> plot(cars, pch =2)

Now let us level up

Load dataset mtcars and explore

Use boxplot() with formula = mpg ~ cyl and data = mtcars to create a box plot.

What does the plot show?
The plot shows that mpg is much lower for cars with more cylinders. Note that we can use the same set of arguments that we explored with plot() above to add axis labels, titles and so on.

We can also use hist() when looking at a single variable.

Plot histogram mtcars$mpg

Task 17: Advance Graphs – ggplot2
Grammar of Graphs

How to make plots?
Geoms, stats, Scale, Facets and coordinate system

Install.package(“ggplot2”)

qplot()

Wraps up all the details of ggplot with a familiar syntax borrowed from plot
Additional features:
- Automatically scales data
- Can produce any type of plot
- Facetting and margins
- Creates objects that can be saved and modified

Let use the diamond dataset for the qplot

```
data("diamonds")
str(diamonds)
```