My R learning Journal

Table of Contents

[Data Structure Overview 1](#_Toc463564025)

# Data Structure Overview

* Variable/symbol is seen as object , they can be vector(like array) as well
* Everything in R is an object, with **vector** being the most basic , example of basic objects are:
  + Character
  + Numeric
  + Integer
  + Complex
  + Logical
* Vector cannot contain mix class of object except the type of vector call “List”, when printing list, they have double bracket
* Empty vector : vector()
* ## is comment
* Matrices are vectors with dimension attribute.
  + X<- 1:3
  + Y<-10:12
  + cbind(x,y)
    - X Y
    - 1 10
    - 2 11
    - 3 12
  + rbind(x,y)
    - X 1 2 3
    - Y 10 11 13
* Factor is like integer vector, it can be unordered or ordered
  + Treated specially by lm() and glm()
    - Gl(3,10) generate 3 level with 10 element
  + Self-describing
  + Print out slightly differently
    - Levels (sorted alphabetically)
    - Levels can be customized explicitly using “x<-factor(c(factorssss),levels = c(string1,string2))”
  + Table(factor) to give frequency count
  + Unclass(factor) to bring it down to integer vector
  + Summary function for factor will displays the number of times each value (or 'level') occurs in the data.
* Missing Value is object too
  + NAN is used for undefined mathematical operation
  + NA is used for everything else
  + To test for na and nan use is.na() and is.nan()
  + There are integer NA,character NA
  + Nan value is NA but converse is not true
* Data frames are used to store tabular data
  + Special list where all element have same length but can be different type
    - Same length so it’s a “square” table
  + Element is like a column, length of element is like a row
  + Each row has a name which can be helpful for query
  + Created using read.table() or read.csv()
  + Converted to matrix using data.matrix()
  + Nrow(X) to get number of row
  + Ncol(x) to get number of column
* R object can have names to write readable code
  + X<-1:3
  + Names(x)<-c(“foo”,”bar”,”norf”)
  + Printing X result in displaying of header foo bar and norf, followed by 1 2 3
  + Dimname(matrix) for naming matrices,
  + Dimname(m)<-list(c(“a”,”b”),c(“c”,”d”))

# Reading Data

## Read.table,read.csv

Read table(separate by space) and read.csv(separate by comma) is for reading tabular data and is more commonly used. These function is for reading files with format which has row and columns and it will **return a data frame** in R.

Important arguments:

* [needed]File, the name of a file, or a connection
* Header, logical indicating if the file has a header line
* Sep, a string indicating how the columns are separated
* colClasses , a character vector indicating the class of each column in the dataset
* nrows, the number of rows in the dataset
* comment.char, a character string indicating the comment character
* skip, the number of lines to skip from the beginning
* [Default is true]stringsAsFactors, should character variables be coded as factors?

Reading larger dataset

* Read help page for read.table
* Rough gauge of how much memory is needed, if size is larger than RAM, stop
* Set comment to be blank if not needed comment.char= “”
* Use colClasses to significantly reduce load on machine
  + Tip for getting class type if unknown:
    - Set nrow to be a small number , like 100 or 1000 ( it can also help R to calculate the required RAM)
    - Use sapply and calling class function to save the class type

Calculating Memory Requirement

Data frame with 1500000 rows and 120 columns. All are numeric data.

Memory required = 1500000 x 120 x 8 bytes / numeric = 14000000000 bytes

14000000000 bytes / 220 bytes/MB = 1373.29 MB

Taking into account the overhead for reading data, we should x2 the above number.

Textual format

* Output: dumping and dputing
* The resulting file preserve the metadata (sacrificing some readability) like the class of each column
* Input: use source or dget to read such file
* Editable and potentially recoverable
* Longer lived but not space efficient
* Easier for version control like git (unlike binary)

## readLines

readLines ,for reading lines of a text file, any type of file will do as it will **return character vector** in R

## Source

For reading R code. **Return function**

## Dget

For reading R object that have been dparsed into text files. Usually usable on a single R object only.

## Load and unserialize

Reading binary objects into R

# Writing Data

## Write.table

## writeLines

## dump

Can be use on multiple R object, to be read back using source later on

## dput

Takes an arbitrary R object , and create some R code for recreation in R



## save

## serialize

# Connections

When using readtable, we have established connection to the file. Connection abstract out the mechanism for connecting to different types of objects that are external to R.

Connections is basically used for more unique data source when the source needs to be highly customized. Otherwise, read.table or read.csv would have suffix.

## File

For connections related to text file etc.

## Gzfile

For connections related to compression file with the gzip algorithm

## Bzfile

For connections related to compression file with the bzip2 algorithm

## url

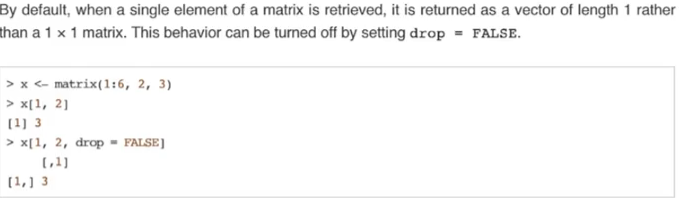
To connect to a url, it should output something like a html file.

## Subsetting

It is used to extract subsets of R objects.

[ always returns an object of same class as the original;  
can also be used to select more than one element.

* X[1]
* x[1:4]
* x[x> “a”] due to lexicographical ordering to letter, we get back “b” to “z”
* we can also create logical vector like u<-x>”a” and print it out.
* This works slightly differently for matrix, be it single column or row or an element in matrix, we get back vector



[[ is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object may not necessarily be a list or data frame  
This is because a list can contain many different class. We can single out the element that we want using the double square bracket.

For example:

x[c(3,5,7)]

x[c(-2, -10)] to get except 2nd and 10th element

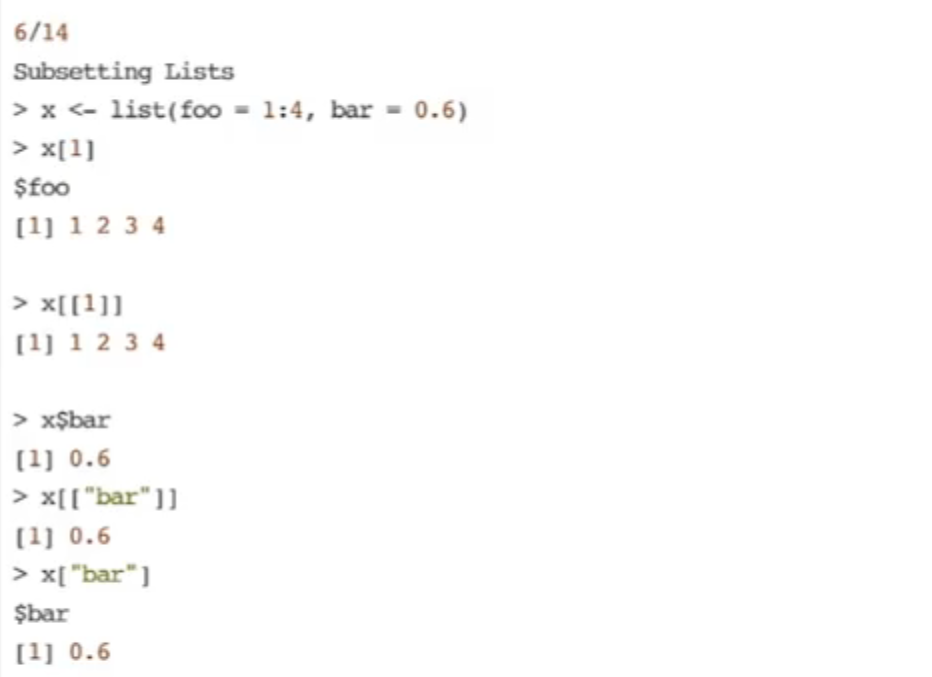
# [**Select first 4 rows of a data.frame in R**](http://stackoverflow.com/questions/2667673/select-first-4-rows-of-a-data-frame-in-r)

head(dnow,4) ## default is 6

get nth row

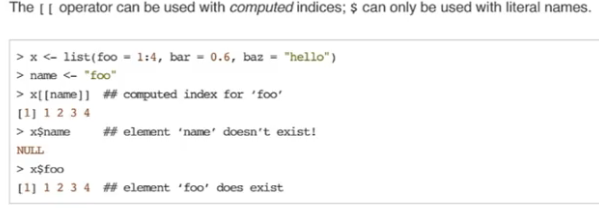
x[r,]

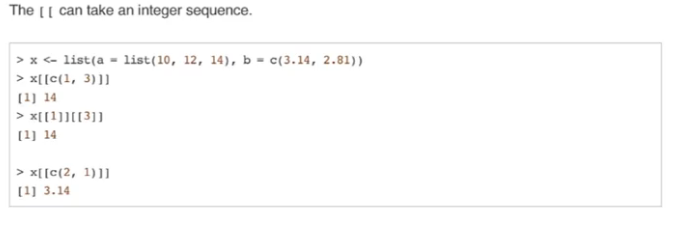
bar <- subset(foo, location == "there")



In the above example, x[1] and x[[1]] is different. X[1] is **a list** that contains 1 to 4. X[[1]] is just **1 to 4.**

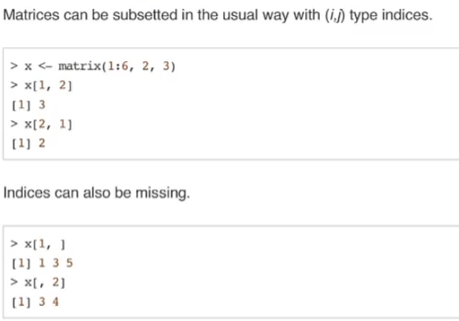
If we want multiple **element,** we have to use single bracket. i.e. x[c(1,2)]





$ is used to extract element of a list or data frame by name, remember that all object can have a name.

Subsetting Matrix



Partial Matching

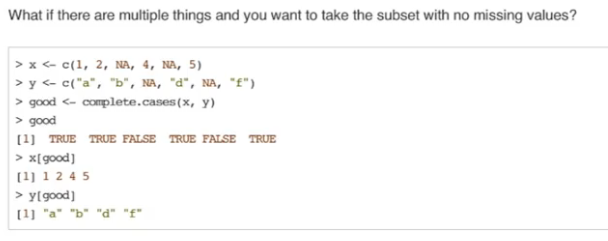


Removing NA values

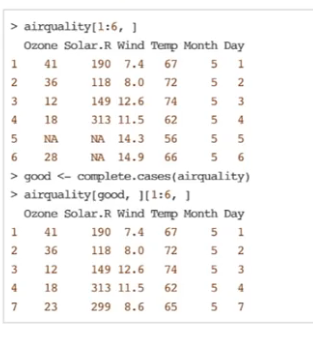
* Bad<-Is.na(x) and then use x[!bad]



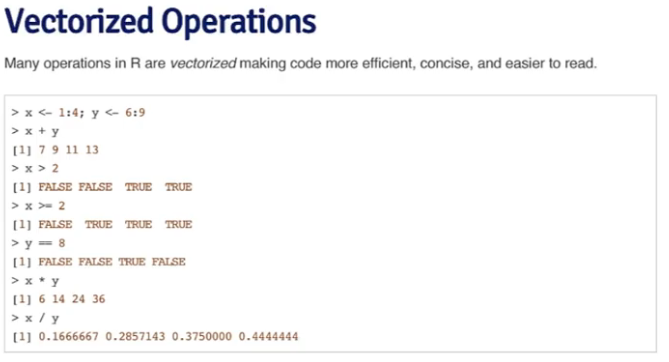
* Complete.cases(x)



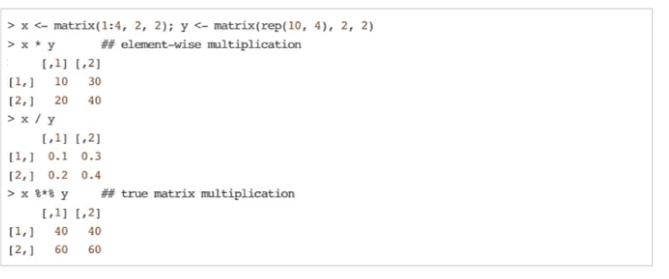
For data frame , use dataframe[complete.cases(dataframe),]



Vectorized operation



Vectorized Matrix Operations



*Note for operations: If vectors are of different lengths, R ‘recycles’ the shorter vector until it is the same length as the longer vector*

Behind the scenes, R is 'recycling' the 2 to make a vector of 2s and the 100 to make a vector of 100s.In other words, when you ask R to compute z \* 2 + 100, what it really computes is this: z \* c(2, 2, 2) +c(100, 100, 100).

Sequence

Seq(0,10,0.5) means 0 to 10, increment of 0.5

Seq(5,10,length=30) 30 numbers between 5 to 10 in equal increment

let's say that rather than repeating the vector (0, 1, 2) over and over again, we want our vector to contain 10 zeros, then 10 ones, then 10 twos. We can do this with the `each` argument. Try rep(c(0, 1, 2), each = 10) instead of rep(c(0,1,2),times=10).

Vectors

paste(my\_name,collapse=" ") join element in my\_name vectors, separated by string stated in collapse.

paste(1:3,c("X", "Y", "Z"),sep="") results in:

[1] "1X" "2Y" "3Z"

Creating normalized random dataset

y<-rnorm(1000)

z<-rep(NA,1000)

my\_data<-sample(c(y,z),100)

Logical

* && evaluate the first member of a vector, & evaluate against all member. This applies for | and || as well.
* And is always evaluated first
* Xor The xor() function stands for exclusive OR. If one argument evaluates to TRUE and one argument evaluates to FALSE, then this function will return TRUE, otherwise it will return FALSE.

Other

Sample(10) give sampling of integers from 1 to 10 without replacement.

The which() function takes a logical vector as an argument and returns the indices of the vector that are TRUE. For example which(c(TRUE, FALSE, TRUE)) would return the vector c(1, 3).

Like the which() function, the functions any() and all() take logical vectors as their argument. The any() function will return TRUE if one or more of the elements in the logical vector is TRUE. The all() function will return TRUE if every element in the logical vector is TRUE.

# Function

The last expression evaluated is the returned item.

We can set the default value by stating them in the argument.

You can also explicitly specify arguments in a function. When you explicitly designate argument values by name, the ordering of the arguments becomes unimportant. For instance: remainder(divisor = 11, num = 5).

we can pass function into function, for instance

evaluate <- function(func, dat){

func(dat)

}

# Time

* Reference to 1/1/1970
* Sys.time() actually has class of POSIXct and POSIXt
* By default, Sys.time() returns POSIXct but we can coerce result to POSIXlt with as.POSIXlt(Sys.time())

# Control

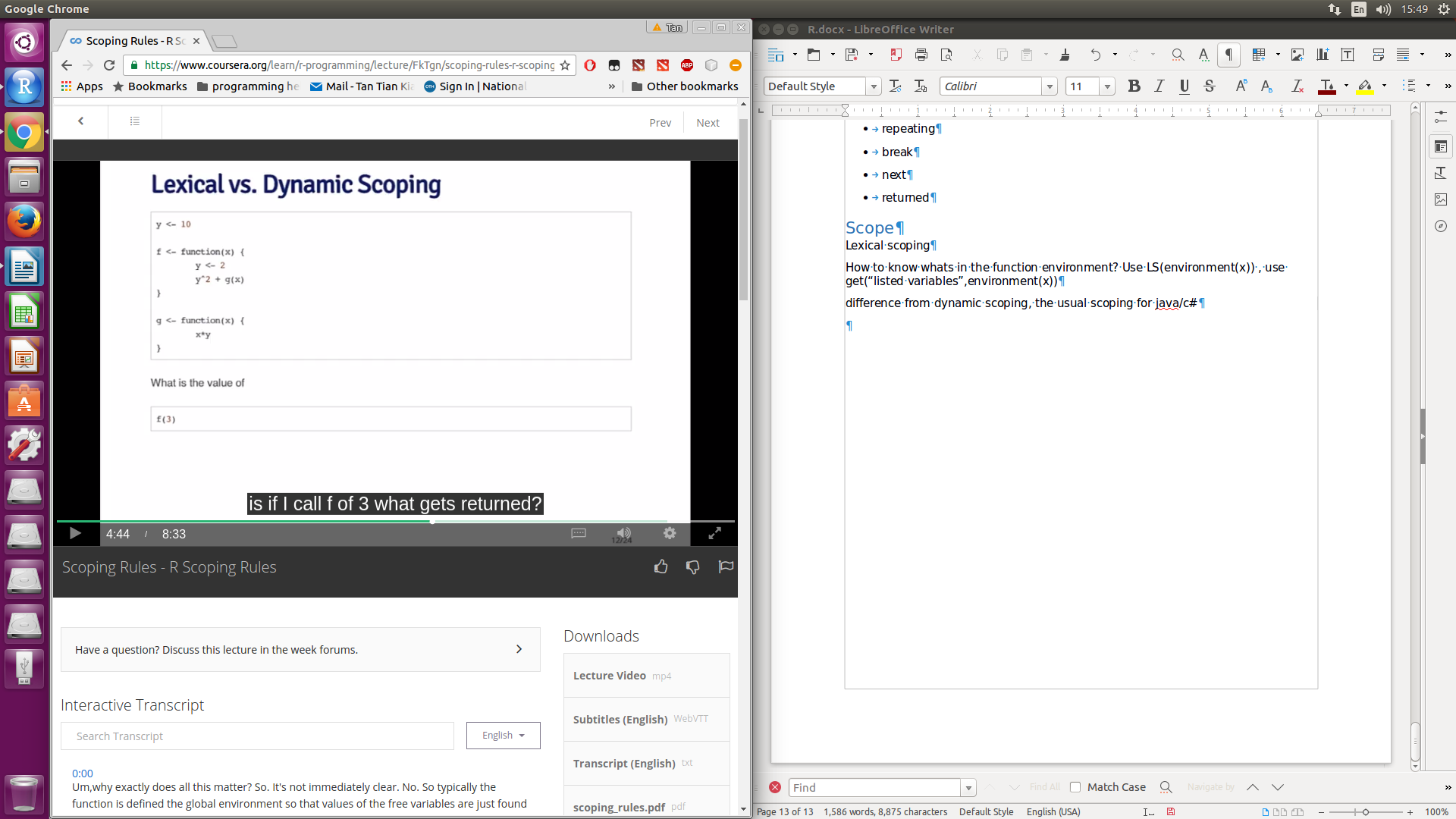
* If,else
* for
* while
* repeating
* break
* next
* returned

# Scope

Lexical scoping

How to know whats in the function environment? Use LS(environment(x)) , use get(“listed variables”,environment(x))

difference from dynamic scoping

Lexical scoping: y in the function g is looked up in the environment in which the function was defined. Therefore value of y is 10

In dynamic scoping, value of y is looked up in the environment from which the function was called, therefore y is 2.

# Date and time

POSIXct is a very large integer , can be stored in data frame

POSIXlt store time as a list

# Looping on Command line

lapply: Loop over a list and evaluate a function on each elements

* I it returns a list, regardless of the input
* use anonymous function in lapply:
* >lapply(x,function(elt) elt[,1]) where x is a list with 2 matrices

sapply: same as lapply but try to simplify the resulting

* If the result is a list where every elements is length 1, then a vector is returned
* if the result is a list where every element is a vector of the same length, matrix is returned.
* If it dont know, a list is returned

Vapply: sapply guess the wanted result, but vapply allow us to specify it.

apply:Apply a function over the margins of an array

* used to apply function to rows or column of matrix
* can be used with general arrays
* function(x,margin,fun,…)
* margin is an integer vector indicating which margin should be retained, 1 to preserve the row, 2 to preserve column
  + rowSums=apply(x,1,sum)
  + rowMeans=apply(x,1,mean)
  + colSums=apply(x,2,sum)
  + colMeans=apply(x,2,mean)
    - colMeans(x, na.rm = FALSE, dims = 1)
* x<-matrix(rnorm(200),20,10)
* apply(x,1,quantile,probs=c(0.25,0.75))
* possible to loop a 3 dimension matrix
  + array(rnorm(2\*2\*10),c(2,2,10))
  + apply(a,c(1,2),mean)

tapply:Apply function over subsets of a vector

* tapply(x,index,fun,…,simplify=true)
* x is vector
* index is factor to identify which group of each element of the vector is in
* fun is the function to apply

mapply:Multivariate version of lapply

* Apply a function in parallel over a set of different arguments
* Str(mapply) function (FUN,…,MoreArgs=NULL,Simplify=true,use.names=true)
* Suppose we have 2 list, mapply will be easier to apply the same function to 2 list.
* For instance:
  + List(rep(1,4),rep(2,3),rep(3,2),rep(4,1)) got 4 list here but to type them out is hard, we can use mapply like this:
    - Mapply(rep,1:4,4:1) from this we know 1 repeat 4 times, 2 repeat 3, 3 repeat 2, 4 repeat 1.
  + It basically means apply a function to multiple set of arguement

Split

inconjuction with lapply, we can use split to split up object.

* Split vector with factor , note that factor can be something like data$Month
* Split(x,f,drop=false,…) drop indicates whether empty factors level should be dropped.
  + Drop is usually use for multi level factor
* One usage is lapply(split(x)) , but we can use tapply to do the same thing
* Split data frame is more commonly seen

# Str function

Its comparable to the args function, in addition,shows a more detailed summary of a function/object

# Generating random numbers

## Dnorm

Density, dnorm(x,mean = 0 , sd =1 , log = false)

## Rnorm

Random number, set.seed to ensure reproducibility, we we set seed, it is like setting the sequence of random variable that’s going to occur

## Pnorm

Cumulative distribution, pnorm(q,mean =0,sd =1, lower.tail=true, log.p=false)

If lower tail is false, it evaluates the upper tail

## Qnorm

Quantile, qnorm(p,mean =0,sd=1,lower.tail=true,log.p=false)

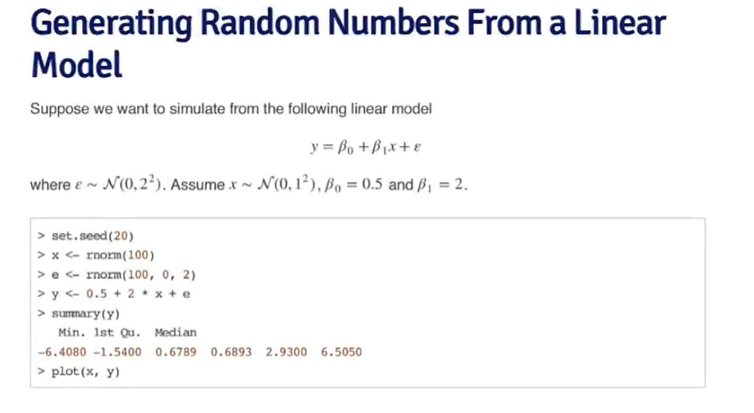
Rpois

Generate poisson data.  
Rpois(10,1) means 10 poisson random variable with the rate of 1 , note that in poisson distribution, mean will be roughly equal to the rate.

Cumulative distribution

Ppois(2,2)

Generating random number from a linear model



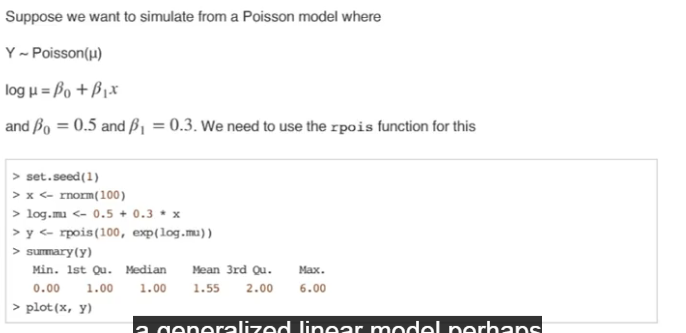
1. First set the seed to ensure reproducibility
2. Generate 100 random variable in x which has standard normal distribution
3. E(error term) will have normal distribution of mean 0 and sd of 2
4. Enter y formula , summary y, plot x against y

What if x is binary instead of rnorm

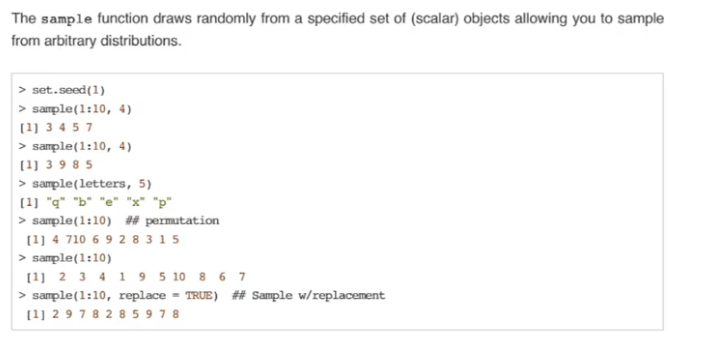


We can use rbinom(100,1,0.5) 100 binomial random variable,n=1, p=0.5 hence probability of getting 1 is going to be 0.5.

Simulate poisson model



Random sampling



sample(c(0,1),100,replace = TRUE,prob=c(0.3,0.7)) , note we can also set the probability in sampling.

R profiler

Examine how much time is spent in different parts of the program.

Using system time comparing difference between user time, elapse time and system time.

Rprof() function, not to be used with system time function, its default sampling interval is 0.02 second to print out the call stack. If code runs too quickly, faster than 0.02 sec, then it’s not necessary to use the profiler anyway.

summaryRprof() function

Replicate

my\_pois<-replicate(100,rpois(5,10)) to replicate Generate 5 random values from a Poisson distribution with mean 10 . 100 times.