DATA SCIENCE SPECIALIZATION (JOHN HOPKINS UNIVERSITY)

1. R Programming :

a. Syllabus:
   i. Week 1: Overview of R, R data types and objects, reading and writing data
   ii. Week 2: Control structures, functions, scoping rules, dates and times
   iii. Week 3: Loop functions, debugging tools
   iv. Week 4: Simulation, code profiling

b. Grading Policy:
   i. Week 1 Quiz - 20%
   ii. Week 2 Quiz - 10%
   iii. Week 3 Quiz - 5%
   iv. Week 4 Quiz - 10%
   v. Programming Assignment 1 (Air Pollution) - 20%
   vi. Programming Assignment 2 (Lexical Scoping) - 10%
   vii. Programming Assignment 3 (Hospital Quality) - 25%
   viii. swirl Programming Assignment (practice) - 0%

c. Course Textbook:
   i. R Programming for Data Science
   ii. Art of Data Science
   iii. R Studio Cheat Sheet

d. Initial Commands (mac):

Getting your working directory →

```
> getwd()
[1] "C:/Users/vjoshi/Desktop/RProgramming"
> setwd("C:/Users/vjoshi/Desktop/CourseraDataScientist/RProgramming")
> dir()
[1] "DATA SCIENCE SPECIALIZATION-RProgramming.docx"
> ls()                    → lists objects in the dir
character(0)
> source()  → source function loads the .r file to run in command window
```

Week 1: Overview of R, R data types and objects, reading and writing data

By the end of week 1 you should be able to:
1. Install the R and RStudio software packages
2. Download and install the swirl package for R
3. Describe the history of the S and R programming lectures
4. Describe the differences between atomic data types
5. Execute basic arithmetic operations
6. Subset R objects using the "[", "[[", and "$" operators and logical vectors
7. Describe the explicit coercion feature of R
8. Remove missing (NA) values from a vector

1) Overview & History of R:

R is a dialect of S Language → S developed at Bell Labs → R created in 1991 in NZ by Ross Ihaka & Robert Gentlemen → quite lean → functionality divided into modular packages → Free software → which means :
Freedom to run program for any purpose **(freedom 0)**.
Freedom to study how program works, adapt it your needs **(freedom 1)**. Access to source code is a precondition to this.
Freedom to redistribute copies, help ur neighbor **(freedom2)**
Freedom to improve the program & release improvements to public, so that whole community benefits **(freedom 3)**

Design of R system: R system divided into 2 conceptual parts : 1) The "base" R system that you download from CRAN, 2) Everything else. R functionality divided into number of packages: a) the "base" R system contains, among other things, the base package which is required to run R and contains most fundamental functions, b) other packages contained in the base system include utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, tcltk, stats4, c) recommended packages: boot, class, cluster, codetools, foreign, KernSmooth,lattice, mgcv, nlme, rpart, survival, MASS, spatial, nnet, Matrix.

2) Getting Help:

```
> library(datasets)→ loading datasets pkg
>data() → list all the datasets in datasets package
> data("airquality") → airquality dataset from datasets package
>summary(airquality)
    Ozone         Solar.R        Wind           Temp          Month
 Min.   : 1.00  Min.   : 7.0  Min.   : 1.700  Min.   :56.00  Min.   :5.00
0
 1st Qu.: 18.00  1st Qu.:115.8  1st Qu.: 7.400  1st Qu.:72.00  1st
Qu.:6.000
 Median : 31.50  Median :205.0  Median : 9.700  Median :79.00
Median :7.000
 Mean   : 42.13  Mean   :185.9  Mean   : 9.958  Mean   :77.88  Mea
n   :6.993
 3rd Qu.: 63.25  3rd Qu.:258.8  3rd Qu.:11.500  3rd Qu.:85.00  3r
d Qu.:8.000
 Max.   :168.00  Max.   :334.0  Max.   :20.700  Max.   :97.00  Max.   :
9.000
 NA's   :37      NA's   :7
      Day
 Min.   : 1.0
 1st Qu.: 8.0
 Median :16.0
 Mean   :15.8
 3rd Qu.:23.0
 Max.   :31.0
> view(airquality)  → view the airquality dataset.
```

3) R Console Input & Evaluation:

# character indicates a comment.

```
> x <- 5                  ## nothing printed
> x                       ## auto printing occurs
[1] 5
> print(x)                ## explicit printing
[1] 5
```

The [1] indicates that x is a vector and 5 is the first element

4) R Data Type: Objects & Attributes: Everything in R → object

R has five basic or atomic classes of objects: character, numeric (real numbers), integer, complex, logical (TRUE/FALSE).
The most basic object is a 'vector' → vector can only contain objects of same class → BUT one exception is 'list', which can be represented as a vector but can contain objects of different classes.
Empty vectors can be created with '**vector()**' function. A vector() function has two basic arguments → i) class of object (type of object you want to have in the vector), ii) length of vector itself.

```
> vector(mode = "integer", length = 3)
[1] 0 0 0
> x <- vector("logical", length = 4)
> x
[1] FALSE FALSE FALSE FALSE
> x <- vector("complex", length = 4)
> x
[1] 0+0i 0+0i 0+0i 0+0i
> x <- vector("character", length = 4)
> x
[1] "" "" "" ""
```

Numbers → in R treated as numeric objects (i.e. double precision real numbers→ integer explicitly suffix **L** → entering 1 gives a numeric object; entering 1L gives a integer object.

```
> class(1)
[1] "numeric"
> class(1L)
[1] "integer"
```

NaN (not a Number) represents an undefined value (can also be thought as a missing value

Attributes → R objects can have attributes → names, dimnames, dimensions (matrices, array), class, length, other user defined attributes/metadata. Attributes of an object can be accessed using the **attributes()** function.

5) R Data Types: Vectors & Lists

Creating Vectors → c() function to create vectors of objects

```
> x <- c (0.5, 0.6)          #  numeric
> x <- c (T,F)               # logical
> x <- c ("a", "b", "c")     # character
> x <-  1:15                 # integer
> x <- c (1+0i,2+0i)         # complex
```

Mixing objects →

```
> x <- c (0.5, "a")          #  character
> x <- c (TRUE, 2)           #  numeric
```

When different objects are mixed in a vector, coercion occurs so that every element in the vector is of same class. Example above is of 'implicit coercion'

Explicit Coercion → objects can be explicitly coerced from one class to ano

```
> x <- 0:5
> class(x)
[1] "integer"
> as.numeric(x)
[1] 0 1 2 3 4 5
> as.logical(x)
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5"
> as.complex(x)
[1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i
```

Non-sensical coercion results in NAS

Lists → special type of vector that can contain elements of different classes.

```
> x <- list(1, "a", TRUE, 1+4i)
> x
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

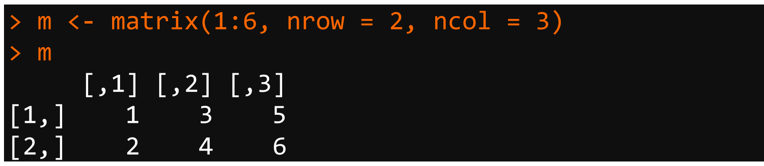
6) R Data Types: Matrices

Matrices are vectors with a dimension attribute→ which is an Integer vector of length 2(nrow, ncol)

```
> m <- matrix(nrow = 2, ncol = 2)
> m
    [,1] [,2]
[1,]   NA   NA
[2,]   NA   NA
> dim(m)
[1] 2 2
> attributes(m)
$dim
[1] 2 2
```

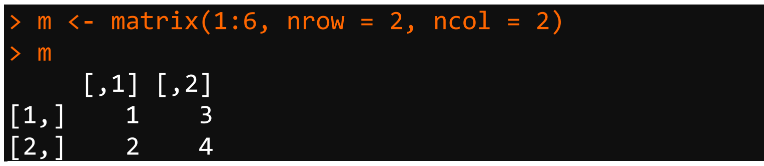
Above attributes(m) gives a list with $dim as a list with vector [1] 2 2

Matrices→ constructed column-wise→



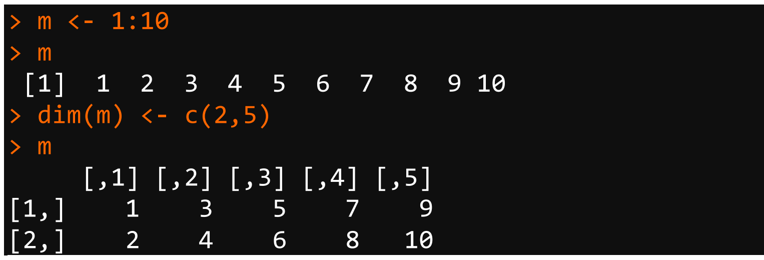
```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

If more elements in matrix than nrow/ncol combination, extra elements not used.



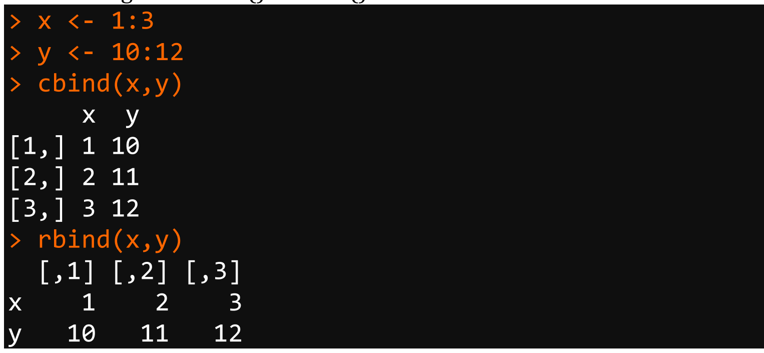
```
> m <- matrix(1:6, nrow = 2, ncol = 2)
> m
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Matrices can also be created directly from vectors by adding a dimension attribute.



```
> m <- 1:10
> m
 [1]  1  2  3  4  5  6  7  8  9 10
> dim(m) <- c(2,5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

cbind & rbind → matrices can also be created by column-binding & row binding with cbind() & rbind()



```
> x <- 1:3
> y <- 10:12
> cbind(x,y)
     x  y
[1,] 1 10
[2,] 2 11
[3,] 3 12
> rbind(x,y)
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

7) R Data Types: Factors

Factors → special types of vector → represent categorical data→ can be ordered or unordered → factor as integer vector where each integer has a label → factors are treated specially by modelling functions like lm() & glm() → using factors with labels is better than using integers because factors are self-describing (having a variable that has values as "male" & "female" is better than a variable that has a value 1 & 2.

```
> x <- factor(c("yes", "no", "yes", "yes", "n
o"))
> x
[1] yes no  yes yes no
Levels: no yes
> table(x)
x
 no yes
  2   3
> unclass(x)
[1] 2 1 2 2 1
attr(,"levels")
[1] "no"  "yes"
```

Order of levels can be set using **levels** argument to factor() → important in linear modeling because the first level is used as the baseline level.

```
> x <- factor(c("yes", "no", "yes", "yes", "n
o"),
+                    levels = c("yes", "no"))
> x
[1] yes no  yes yes no
Levels: yes no
```

8) Data Types: Missing Values

Missing values are denoted by NA or NaN for undefined mathematical operations

- is.na() → test objects if they are NA
- is.nan() → test objects for NaN
- NA values have class also→ int NA, char NA etc
- NaN value is also NA but converse not true.

```
> x <- c(1,2, NA, 4,5)
> is.na(x)
[1] FALSE FALSE  TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE
> x <- c(1,NaN, NA, 4,5)
> is.na(x)
[1] FALSE  TRUE  TRUE FALSE FALSE
> is.nan(x)
[1] FALSE  TRUE FALSE FALSE FALSE
> x <- c(1,NAN, NA, 4,5)
Error: object 'NAN' not found
```

9) R Data Type: Data Frames

Used for storing tabular data

- represented as special list where every element of the list has to have same length
- each element of list can be thought of as a column and length of each element of list is # of rows
- unlike matrices, data frames can store different classes of objects in each column(~ lists)
- data frames have special attribute called **row.names**
- data frames created → read.table() or read.csv()
- can be converted to matrix by calling data.matrix()

```
> x <- data.frame(foo = 1:4, bar = c(T,T,F,F))
> x
  foo   bar
1   1  TRUE
2   2  TRUE
3   3 FALSE
4   4 FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

10) R Data Types: Then Names Attribute

R objects can also have names → very useful for writing readable code and self-describing objects.

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("A", "B", "C")
> x
A B C
1 2 3
> names(x)
[1] "A" "B" "C"
```

list can also have names

```
> x <- list(a=1, b=2, c=3)
> x
$a
[1] 1

$b
[1] 2

$c
[1] 3
```

a, b, c are elements with values 1,2,3 respectively.

Matrices can have names:

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> m
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> dimnames(m) <- list(c("a", "b"), c("c", "d
"))
> m
  c d
a 1 3
b 2 4
```

Summary :

atomic classes→numeric, logical, character, integer, complex vectors→only have elements of the same class ; lists →can have elements of different classes. Factors which are used for coding categorical data, with ordered and unordered data. There are missing values that are represented by NAs, and NANs. Data frames are used to store tabular data or each COM can be of a different class. And finally, all our R objects can have names which can be useful for creating self-describing data.

11) Reading Tabular Data:

read.table & read.csv →reading tabular data
readLines → reading lines of text
source → reading in R code files (inverse of dump)
dget → reading in R code files (inverse of dput)
load → reading in saved workspaces
unserialize → reading R objects in binary form

Writing Data:

write.table
writeLines
dump
dput
save
serialize

read.table → most commonly used for reading data → Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file→arguments in read.table
file, name of file or a connection
header, logical indication if file has header line
sep, a string indicating how the columns are separated
colClasses, a character vector indicating classes of each column
nrows, # of rows in dataset
comment.char, a character string indicating comment character
skip, # of lines to skip from beginning
stringAsFactors, should character variables be coded as factors

default separator for read.table is space whereas for read.csv is comma

12) Reading Large Tables

If dataset > RAM, should stop there
Set comment.char = "" if there are no commented lines in ur file
Use colClasses argument → need to know class of each column in ur data frame → all columns numeric set colClasses = "numeric"
Figuring out classes of each column

```
initial <- read.table("dataset.txt", nrows = 100)
classes <- sapply(initial,class)
tabAll <- read.table("datatable.txt", colClasses = classes)
```

set nrows → helps with memory usage

Calculating Memory Requirements → e.g dataframe with 1,500,000 rows and 120 columns (all are numeric data)
1,500,000 * 120 * 8 bytes/numeric
= 1440000000 bytes
= 1440000000/2^20 bytes/MB
=1,3773.29 MB /2^10 GB
= 1.34 GB

13) Textual Data Formats:

dumping & dputing → preserve metadata

14) Connections: Interfaces to Outside World

file, opens connection to a file
gzfile, opens a connection to a file compressed with gzip
bzfile, opens a connection to a file compressed with bzip2
url, opens a connection to a webpage

```
> str(file)
function (description = "", open = "", blocki
ng = TRUE, encoding = getOption("encoding"),
raw = FALSE,
    method = getOption("url.method", "default
"))
```

15) Subsetting R Objects : Basics

# of operators that can be used to extract subsets of R objects:

- [→ always returns an object of same class as original; can be used to select more than one element(one exception) → if you subset a vector u get a vector, u subset a list u get a list.
- [[→ extract elements of a list or a data frame; it can only be used to extract a single element and the class of returned object will not necessarily be a list or data frame
- $ → used to extract elements of a list or dataframe by name

```
> x <- c("a", "b", "C","d", "e", "f")
> x[1]
[1] "a"
> x[2]
[1] "b"
> x[1:3]
[1] "a" "b" "C"
> x[x > "a"]
[1] "b" "C" "d" "e" "f"
> u <- x > "a"
> u
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> x[u]
[1] "b" "C" "d" "e" "f"
```

16) Subsetting R Objects : Lists

```
> x <- list(foo = 1:4, bar = 0.5)
> x[1]
$foo
[1] 1 2 3 4

> x[2]
$bar
[1] 0.5

> x[[1]]
[1] 1 2 3 4
> x[[2]]
[1] 0.5
> x$bar
[1] 0.5
> x[["bar"]]
[1] 0.5
> x["bar"]
$bar
[1] 0.5
```

because the single bracket always returns a list if I'm going to subset a list

Extracting multiple elements from a list

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hell
o")
> x[c(1,3)]
$foo
[1] 1 2 3 4

$baz
[1] "hello"
```

[[→ operator can be used with computed indices; $ can only be used with literal names

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hell
o")
> x[c(1,3)]
$foo
[1] 1 2 3 4

$baz
[1] "hello"

> name <- "foo"
> x[[name]]
[1] 1 2 3 4
> x$name
NULL
> x$foo
[1] 1 2 3 4
```

[[→ can take a integer sequence

```
> x <- list(a = list(10,11,12), b = c(3.14,2.
81))
> x
$a
$a[[1]]
[1] 10

$a[[2]]
[1] 11

$a[[3]]
[1] 12


$b
[1] 3.14 2.81

> x[[c(1,3)]]
[1] 12
> x[[1]][[3]]
[1] 12
> x[[c(2,1)]]
[1] 3.14
```

17) Subsetting R Object : Matrices

Matrices can be subsetted in the usual way with (i,j) type indices

```
> m <- matrix(1:6, 2,3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> m[1,2]
[1] 3
> m[2,1]
[1] 2
> m[1,]
[1] 1 3 5
> m[,2]
[1] 3 4
```

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1x1 matrix. This behavior can be turned off by setting drop = FALSE

```
> m <- matrix(1:6, 2,3)
> m[2,1]
[1] 2
> m[2,1, drop = FALSE]
     [,1]
[1,]    2
```

Similarly, subsetting a single column or row, will give a vector not a matrix

```
> m <- matrix(1:6, 2,3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> m[1,]
[1] 1 3 5
> m[1,,drop = FALSE]
     [,1] [,2] [,3]
[1,]    1    3    5
```

18) Subsetting R Objects : Subsetting with Names

Partial matching → partial matching of name is allowed with [[& $

```
> x <- list(aardvark = 1:5)
> x
$aardvark
[1] 1 2 3 4 5

> x$a
[1] 1 2 3 4 5
> x[["a"]]
NULL
> x[["a", exact = FALSE]]
[1] 1 2 3 4 5
```

19) Subsetting R Objects: Removing Missing Values

```
20) > x <- c(1,3,NA,5,NA,6)
21) > bad <- is.na(x)
22) > bad
23) [1] FALSE FALSE  TRUE FALSE  TRUE FALSE
24) > x[!bad]
25) [1] 1 3 5 6
```

```
> x <- c(1,2,NA,4,NA,5)
> y <- c("a","b",NA,"d",NA,"f" )
> good <- complete.cases(x,y)
> good
[1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE
> x[good]
[1] 1 2 4 5
> y[good]
[1] "a" "b" "d" "f"
> x <- c(1,NA,NA,4,NA,5)
> y <- c("a","b",NA,"d",NA,"f" )
> good <- complete.cases(x,y)
> good
[1]  TRUE FALSE FALSE  TRUE FALSE  TRUE
> x[good]
[1] 1 4 5
> y[good]
[1] "a" "d" "f"
```

```
> airquality[1:6,]
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
> good <- complete.cases(airquality)
> airquality[good, ][1:6,]
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
7    23     299  8.6   65     5   7
8    19      99 13.8   59     5
```

20) **Vectorized Operations: Matrix**

```
> x <- matrix(1:4, 2,2) ; y <- matrix(rep(10,
4),2,2)
> x
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> y
     [,1] [,2]
[1,]   10   10
[2,]   10   10
> x*y  ## element wise multiplication
     [,1] [,2]
[1,]   10   30
[2,]   20   40
> x/y  ## element wise division
     [,1] [,2]
[1,]  0.1  0.3
[2,]  0.2  0.4
> x %*% y  ## true matrix multiplication
     [,1] [,2]
[1,]   40   40
[2,]   60   60
```

Week 2: Control structures, functions, scoping rules, dates and times

By the end of this week you should be able to:

1. Write an if-else expression
2. Write a for loop, a while loop, and a repeat loop
3. Define a function in R and specify its return value [see Functions Part 1 and Part 2]
4. Describe how R binds a value to a symbol via the search list
5. Define what lexical scoping is with respect to how the value of free variables are resolved in R
6. Describe the difference between lexical scoping and dynamic scoping rules
7. Convert a character string representing a date/time into an R datetime object. [see Dates and Times]

I) Control Structures: Control the flow of execution of the program:

- if,else → testing a condition
- for → execute a loop a fixed # of times
- while → execute a loop while a condition is true
- repeat → execute an infinite loop
- break → break the execution loop
- next → skip an iteration of a loop
- return → exit a function

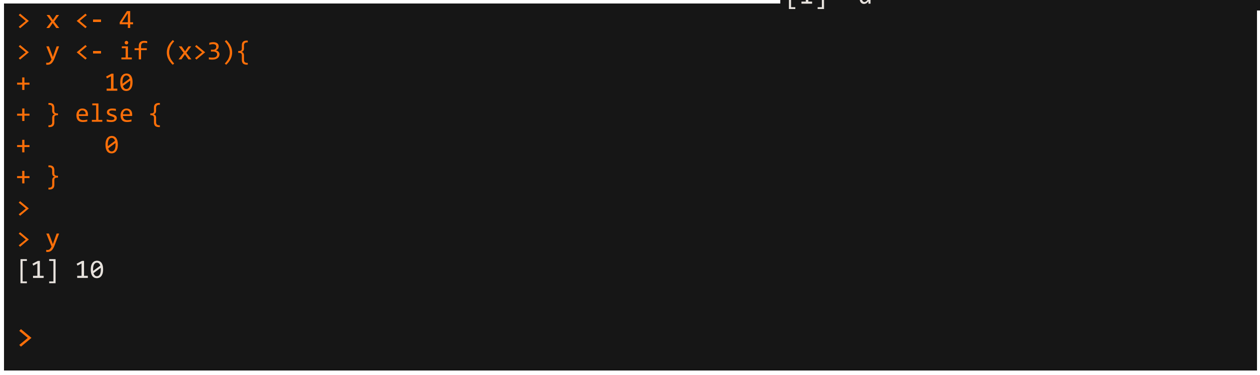
II) Control Structures : If-else

```
if (<condition>){
  ## do something
} else {
  ## do something else
}

if (<condition1>){
  ## do something
} else if (<condition2>){
  ## do something different
} else {
  ## do something different
}

if (x>3){
  y <- 10
} else{
  y <- 0
}

## or
```

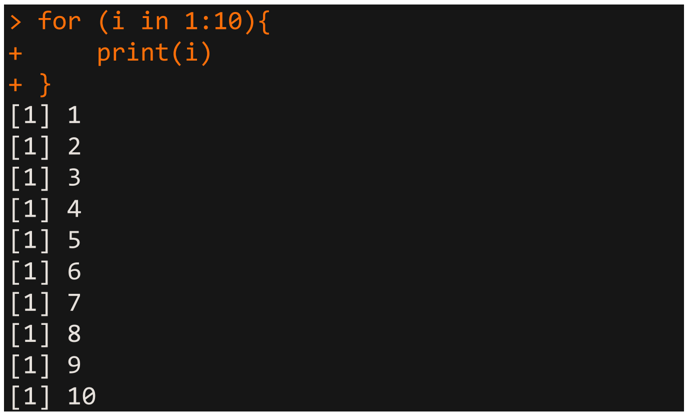


```
> x <- 4
> y <- if (x>3){
+     10
+ } else {
+     0
+ }
>
> y
[1] 10

>
```

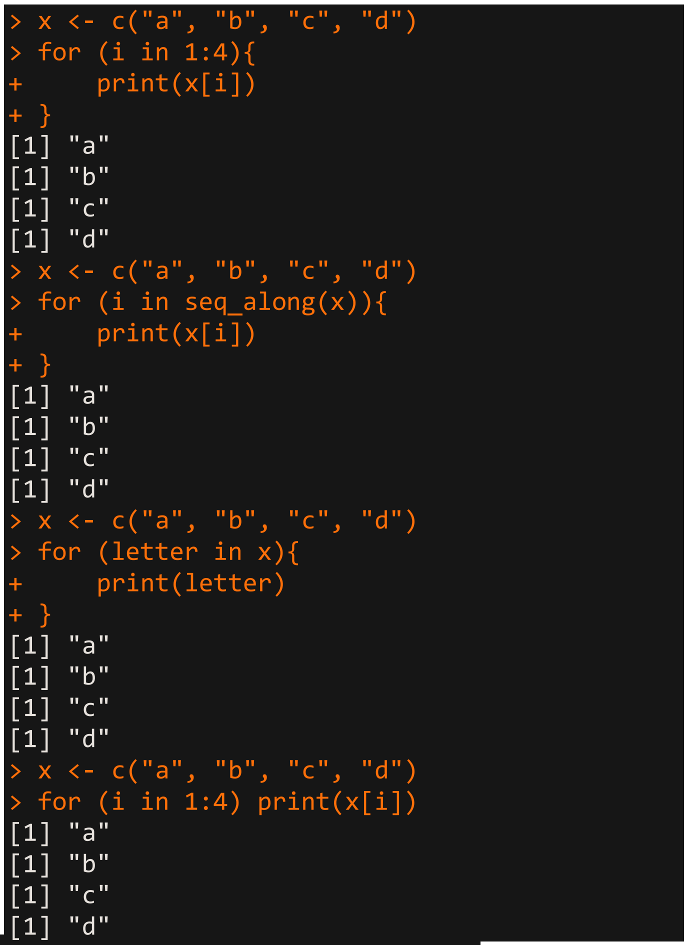
III) Control Structures: For loop

for loops take an interator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc)



```
> for (i in 1:10){
+     print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Below for loops have the same behavior



```
> x <- c("a", "b", "c", "d")
> for (i in 1:4){
+     print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
> x <- c("a", "b", "c", "d")
> for (i in seq_along(x)){
+     print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
> x <- c("a", "b", "c", "d")
> for (letter in x){
+     print(letter)
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
> x <- c("a", "b", "c", "d")
> for (i in 1:4) print(x[i])
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

Nested for loops:

```
> x <- matrix(1:6, 2,3)
> x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> for (i in seq_len(nrow(x))){
+     for (j in seq_len((ncol(x)))){
+         print(x[i,j])
+     }
+ }
[1] 1
[1] 3
[1] 5
[1] 2
[1] 4
[1] 6
```

IV) Control Structures: while loop

While loops begins by testing a condition. If it is true, they execute the loop body, once the loop body is executed the condition is again and so forth → while loops can potentially result in infinite loops.

```
> count <- 0
> while(count <10){
+     print (count)
+     count <- count + 1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
```

Infinite loop example
z <- 5
while (z >=3 && z <=10){
 print(z)
}

```
> z <- 5
> while (z >=3 && z <=10){
+     print(z)
+     coin <- rbinom(1,1,0.5)
+ 
+     if (coin == 1) {
+         z <- z +1
+ 
+     } else{
+         z <- z -1
+     }
+ }
[1] 5
[1] 6
[1] 5
```

```
[1] 6
[1] 7
[1] 8
[1] 7
[1] 6
[1] 7
[1] 6
[1] 5
[1] 6
[1] 5
[1] 6
[1] 7
[1] 6
[1] 7
[1] 6
[1] 5
[1] 6
[1] 7
[1] 6
[1] 7
[1] 6
[1] 5
[1] 6
[1] 7
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

V) Control Structures: Repeat, next , break

Repeat initiates an infinite loop; these are not commonly used in statistical applications. The only way to exit a repeat loop is to call break

```
> x0 <- 1
> tol <- 1e-8
> repeat {
+     x1 <- computeEstimate()
+ 
+     if (abs(x1 - x0) < tol) {
+         break
+     }  else {
+         x0 <- x1
+     }
+ }
```

Ex : computeEstimate() is not a real function, just an example this is a common type of formulation in in many types of optimization algorithms, for example, if you're trying to find the solution to some set of equations, or you're trying to maximize the function, often you'll iterate over and over again. And and you'll stop when the, when the estimates that you're calculating are getting closer and closer together, because that's usually a sign that you're kind of converging to whatever the minimum or maximum of the objective function is next, return → next is used to skip an iteration of a loop

```
> for ( i in 1:100) {
+     if ( i <=20) {
+         ## skip the first 20 interations
+         next
+     }
+     ## do something else
+ }
```

return → function should exit and return a given value

VI) R Function

```
> add_2 <- function(x,y){
+     x + y
+ }
> add_2(3,5)
[1] 8
```

```
> above10 <- function(x){
+     use <- x > 10
+     x[use]
+ }
> above10(15)
[1] 15
> above10(9)
numeric(0)
> above <- function(x,n){
+     use <- x > n
+     x[use]
+ }
> x <- 1:20
> above(x,12)
[1] 13 14 15 16 17 18 19 20
```

Specifing the default value for n

```
> above <- function(x,n =10){
+     use <- x > n
+     x[use]
+ }
> above(x)
 [1] 11 12 13 14 15 16 17 18 19 20
```

```
> columnmean <- function (m) {
+     nc <- ncol(m)
+     means <- numeric(nc)
+     for (i in 1:nc){
+         means[i] <- mean(m[,i])
+     }
+     means
+ }
> columnmean(airquality)
[1]        NA        NA  9.957516 77.882353
6.993464 15.803922
```

Now first two columns have NA .. to remove the NA modify the function

```
> columnmean <- function (m, removeNA = TRUE)
 {
+     nc <- ncol(m)
+     means <- numeric(nc)
+     for (i in 1:nc){
+         means[i] <- mean(m[,i], na.rm = rem
oveNA)
+     }
+     means
+ }
> columnmean(airquality)
[1]  42.129310 185.931507   9.957516  77.8823
53   6.993464  15.803922
```

```
> columnmean(airquality, FALSE)
[1]        NA        NA  9.957516 77.882353
6.993464 15.803922
```

Functions created using function() directive → stored as R objects → objects of class "function"

```
f <- function(<argument){
       ## some code for the function
}
```

Functions can be used to pass arguments to other functions; can be nested. The return value of function is the last expression in the function of the body.

Function Arguments:
Functions have named arguments which potentially have default values

- formal arguments are the arguments included in the function definition
- formals function return a list of all the formal arguments of a function
- not ever y function call in R makes use of the all the formal arguments
- function arguments can be missing or might have default values.

Argument matching → R functions arguments can be matched positionally or by name. All below are equivalent.

```
> mydata <- rnorm(100)
> sd(mydata)
[1] 1.017143
> sd(x = mydata)
[1] 1.017143
> sd(x = mydata, na.rm = FALSE)
[1] 1.017143
> sd(na.rm = FALSE, x = mydata)
[1] 1.017143
> sd(na.rm = FALSE, mydata)
[1] 1.017143
> args(sd)
function (x, na.rm = FALSE)
NULL
> args(lm)
function (formula, data, subset, weights, na.
action, method = "qr",
    model = TRUE, x = FALSE, y = FALSE, qr =
TRUE, singular.ok = TRUE,
    contrasts = NULL, offset, ...)
NULL
```

Lazy Evaluation: Arguments to function are evaluated lazily, they are evaluated only as needed.

```
> f <- function (a,b){
+     a^2
+ }
> f(2)
[1] 4
```

Above function never actually uses the argument b, so calling f(2) will not produce an error because the 2 gets positionally matched to a.

```
> f <- function (a,b){
+     print(a)
+     print(b)
+ }
> f(45)
[1] 45
Error in print(b) : argument "b" is missing, 
with no default
```

"45" got printed first before the error message was triggered. This is because b did not have to be a evaluated until after print(a). Once the function tried to evaluate print(b) it had to throw an error.

VII) Scoping Rules → Lexical Scoping

Consider the following example

```
> f <- function (x,y){
+     x^2 + y/z
+ }
> f(2,3)
Error in f(2, 3) : object 'z' not found
```

Function has 2 formal arguments x & y . body of function has z. In this case z is a free variable.

Lexical scoping in R means that : values of free variables are searched for in the environment in which function was defined.

What is an environment → collection of (symbol, value) pairs, i.e. x is a symbol and 3.14 might be its value. The idea here is that you can define things like global variables, that will be common to a lot of different functions. That you might be defining in your workspace. "z" is free variable which can be defined in global variables.

```
> make.power <- function(n) {
+     pow <- function(x){
+         x ^ n
+     }
+     pow
+ }
```

This function returns another function as its value

```
> cube <- make.power(3)
> square <- make.power(2)
> cube(3)
[1] 27
> square(3)
[1] 9
```

Exploring a Function Closure → what's in a function's env

```
> ls(environment(cube))
[1] "n"   "pow"
> get("n", environment(cube))
[1] 3
> ls(environment(square))
[1] "n"   "pow"
> get("n", environment(square))
[1] 2
```

VIII) Coding Standards

IX) Dates & Times

R → special representation of dates & times → dates are represented by Date class → Times are represented by POSIXct or POSIXlt class → Dates are stored internally as # of days since 1970-01-01 → Same for Times (stored as seconds since 1970-01-01)

Dates are represented by the Date class and be coerced from a character string using the as.Date() function

```
> x <- as.Date("1970-01-01")
> x
[1] "1970-01-01"
> class(x)
[1] "Date"
> unclass(x)
[1] 0
> unclass(as.Date("1970-01-02"))
[1] 1
> unclass(as.Date("1977-11-27"))
[1] 2887
```

Times are represented using POSIXct & POSIXlt class

POSIXct → very large integer → useful class when u want to store times in something like data frame

POSIXlt → list → stores bunch of other useful information like the day of the week, day of the year, month, day of the month

# of generic functions that work on dates and times

weekdays → give the day of the week

months → give the month name

quarters → give the quarter number ("Q1", "Q2", "Q3" or "Q4")

Times can be coerced from a character string using the as.POSIXlt or as.POSIXct function.

```
> x <- Sys.time()
> x
[1] "2016-12-04 15:41:41 EST"
> p <- as.POSIXlt(x)
> p
[1] "2016-12-04 15:41:41 EST"
> names(unclass(p))
 [1] "sec"    "min"    "hour"   "mday"   "mon
"    "year"   "wday"   "yday"   "isdst"  "zon
e"
[11] "gmtoff"
> p$sec
[1] 41.59279
> p$min
[1] 41
> datestring <- c("January 10, 2012 10:40","D
ecember 9, 2011 9:10")
> datestring
[1] "January 10, 2012 10:40" "December 9, 201
1 9:10"
> x <- strptime(datestring,"%B %d, %Y %H:%M")
> x
[1] "2012-01-10 10:40:00 EST" "2011-12-09 09:
10:00 EST"
> class(x)
[1] "POSIXlt" "POSIXt"
```

Week 3: Loop functions, debugging tools

By the end of this week you should be able to:

- Define an anonymous function and describe its use in loop functions [see lapply]
- Describe how to start the R debugger for an arbitrary R function
- Describe what the traceback() function does and what is the function call stack

I. Loop Functions

- lapply → loop over a list & evaluate a function on each element
- sapply → same as lapply but try to simplify the result
- apply → apply a function over the subsets of a vector
- mapply → multivariate version of lapply

An auxiliary function split is also useful, particulary in conjunction with lapply.

lapply → takes three arguments: (1) a list; (2) a function (or name of function) FUN; (3) other arguments via its ...argument. If x is not a list, it will be coerced to a list using as.list

lapply always returns a list, regardless of the class of the input

```
> x <- list(a = 1:5, b = rnorm(10))
> x
$a
[1] 1 2 3 4 5

$b
 [1]  2.2030099 -0.1620209  0.7525460  0.5554
205 -0.1693452 -0.1847290  0.5673729  0.15582
68
 [9] -0.3560610  0.9902906

> lapply(x,mean)
$a
[1] 3

$b
[1] 0.4352311
> x<- 1:4
> x
[1] 1 2 3 4
> lapply(x,runif)
[[1]]
[1] 0.4942846

[[2]]
[1] 0.27266597 0.03314656

[[3]]
[1] 0.8071870 0.5629495 0.4185483

[[4]]
[1] 0.5552167 0.1171390 0.7974070 0.5112462
```

* runif generates random numbers

Anonymous functions→ functions that don't have names

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix
(1:6, 2, 3))
> x
$a
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$b
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

An anonymous function for extracting the first column of each matrix

```
> lapply(x, function(first) first[,1])
$a
[1] 1 2

$b
[1] 1 2
```

sapply → will try to simplify the result of lapply if possible

If the result is a list where every element is length 1, then a vector is returned

If the result is a list where every element is a vector of the same length (>1), matrix is returned

If it can't figure things out, a list is returned

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm
(20,1), d = rnorm(100,5))
> lapply(x,mean)
$a
[1] 2.5

$b
[1] -0.2367064

$c
[1] 1.387059

$d
[1] 5.048353

> sapply(x,mean)
        a          b          c          d 
 2.5000000 -0.2367064  1.3870587  5.0483526 
```

II. Loop functions : apply

apply is used to evaluate a function (often an anonymous one) over the margins of array. It is most often used to apply a function to the rows or columns of matrix. It can be used with general arrays, e.g. taking the average of an array of matrices

apply(X, MARGIN, FUN, ...)

MARGIN a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns. Where X has named dimnames, it can be a character vector selecting dimension names.

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x,2, mean)
 [1] -0.36970556 -0.10721054 -0.11880799  0.1
1256591 -0.36625233 -0.14395375 -0.01988020
0.02133177
```

```
 [9] -0.04641594  0.09505520
> apply(x,1, sum)
 [1]  2.1723020 -4.8581244  1.0043615 -0.1920
396 -0.8918058  2.7182803  4.1140604 -0.55737
33
 [9] -2.9796282 -1.7586960  3.7704862 -1.9243
366 -6.6374823  0.8298561 -2.3624651 -5.25647
88
[17] -8.0046384  0.6201870 -2.6261280  3.9541
942
```

Col/row sums & means of matrix dimension

rowSums = apply(x,1,sum)

rowMeans = apply(x,1 ,mean)

colSums = apply(x,2, sum)

colMeans = apply(x,2, mean)

```
> a <- array(rnorm(2*2*10), c(2,2,10))
> apply(a,c(1,2), mean)
            [,1]       [,2]
[1,] -0.07407854  0.6270023
[2,]  0.47771780 -0.2420869
> rowMeans(a, dims = 2)
            [,1]       [,2]
[1,] -0.07407854  0.6270023
[2,]  0.47771780 -0.2420869
```

III. Loop functions : mapply

mapply is a multivariate apply of sorts which applies a function in parallel over a set of arguments

```
> x <- list(rep(1,4), rep(2,3), rep(3,2), rep
(4,1))
> x
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

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

Instead can be done by mapply

```
> mapply(rep,1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

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

IV. Loop functions : tapply

tapply is used to apply a function over subsets of a vector.

V. Loop functions : split

split takes a vector or other objects and splits it into groups determined by a factor or list of factors

Splitting Data Frame

```
> library(datasets)
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
> s <- split(airquality,airquality$Month)
> lapply(s, function(x) colMeans(x[,c("Ozone",
 "Solar.R", "Wind")]))
$`5`
   Ozone  Solar.R     Wind
      NA       NA 11.62258

$`6`
    Ozone    Solar.R      Wind
       NA 190.16667  10.26667

$`7`
     Ozone     Solar.R       Wind
        NA 216.483871   8.941935

$`8`
   Ozone  Solar.R     Wind
      NA       NA 8.793548

$`9`
   Ozone  Solar.R     Wind
      NA 167.4333  10.1800
> sapply(s, function(x) colMeans(x[,c("Ozone",
"Solar.R", "Wind")]))
               5         6          7
8        9
Ozone         NA        NA         NA        N
A        NA
Solar.R       NA 190.16667 216.483871        N
A 167.4333
Wind    11.62258  10.26667   8.941935 8.79354
8  10.1800
> sapply(s, function(x) colMeans(x[,c("Ozone",
"Solar.R", "Wind")],na.rm = TRUE))
                5         6          7
   8         9
Ozone    23.61538  29.44444  59.115385  59.96
1538  31.44828
Solar.R 181.29630 190.16667 216.483871 171.85
7143 167.43333
Wind     11.62258  10.26667   8.941935   8.79
3548  10.18000
```

Week 4: Simulation, code profiling

By the end of this week you should be able to:

- Call the str function on an arbitrary R object
- Describe the difference between the "by.self" and "by.total" output produced by the R profiler
- Simulate a random normal variable with an arbitrary mean and standard deviation
- Simulate data from a normal linear model

A. The str function

str → compactly display the internal structure of an R object.

Alternative to summary

```
> library(datasets)
> data()
> dim(airquality)
[1] 153   6
> str(airquality)
'data.frame':    153 obs. of  6 variables:
 $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA
 ...
 $ Solar.R: int  190 118 149 313 NA NA 299 99
 19 194 ...
 $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.
6 13.8 20.1 8.6 ...
 $ Temp   : int  67 72 74 62 56 66 65 59 61 6
9 ...
 $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
> x <- rnorm(100,2,4)
> summary(x)
    Min.  1st Qu.   Median     Mean  3rd Qu.
    Max.
-12.4500  -0.7354   1.9960   1.6790   4.5240
 13.2500
> str(x)
 num [1:100] -0.58 2.812 0.398 4.154 1.355 ..
```

B. Generating Random Numbers

Functions for probability distributions in R

- rnorm → generate random Normal variates with a given mean and standard deviation
- dnorm → evaluate the normal probability density(with given mean/SD) at a point (or a vector of points)
- pnorm → evaluate the cumulative distribution function for a Normal distribution
- rpois → generate random Poisson variates with a given rate

Probability distribution functions usually have four functions associated with them. The functions are prefixed with

- d for density
- r for random number generation
- p for cumulative distribution
- q for quantile distribution

Working with Normal distributions requires using these four functions:

```
> str(dnorm)
function (x, mean = 0, sd = 1, log = FALSE)
> str(rnorm)
function (n, mean = 0, sd = 1)
> str(qnorm)
function (p, mean = 0, sd = 1, lower.tail = T
RUE, log.p = FALSE)
> str(pnorm)
function (q, mean = 0, sd = 1, lower.tail = T
RUE, log.p = FALSE)
```

If ϕ is the cumulative distribution function for a standard Normal distribution, then pnorm(q) = ϕ(q) and qnorm(p) = ϕ^{-1}(p)

```
> x <- rnorm(4)
> x
[1] -0.25515964 -0.62510791 -1.22555901 -0.03
352347
> summary(x)
    Min.  1st Qu.   Median     Mean  3rd Qu.
    Max.
-1.22600 -0.77520 -0.44010 -0.53480 -0.19980
-0.03352
> y <- rnorm(4, 20, 2)
> y
[1] 19.53973 18.11787 20.85269 18.93159
> summary(y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Ma
x.
  18.12   18.73   19.24   19.36   19.87   20.
85
```

set.seed ensures reproducibility

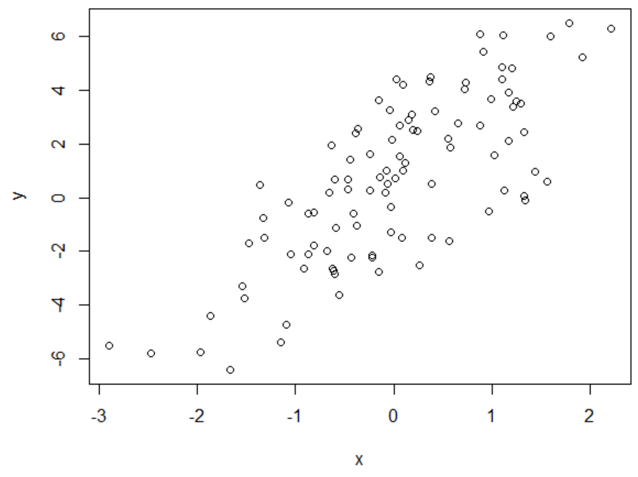
```
> set.seed(1)
> rnorm(4)
[1] -0.6264538  0.1836433 -0.8356286  1.59528
08
> rnorm (4)
[1]  0.3295078 -0.8204684  0.4874291  0.73832
47
> set.seed(1)
> rnorm(4)
[1] -0.6264538  0.1836433 -0.8356286  1.59528
08
```

Generating Poisson data → data is integer

```
> rpois(10,1)
 [1] 1 0 0 0 1 1 2 1 1 4
> rpois(10,3)
 [1] 2 4 6 2 4 1 2 2 0 2
```

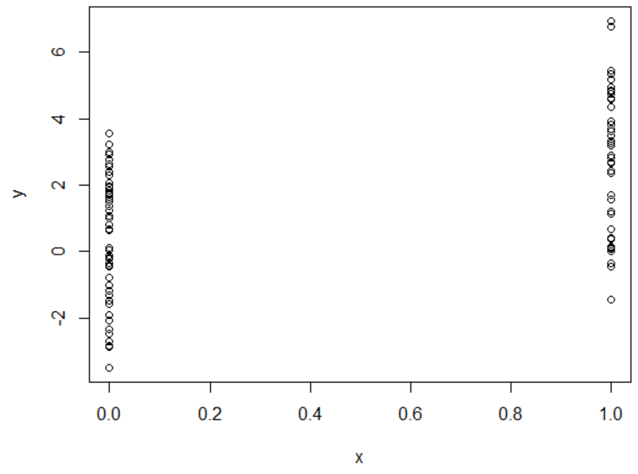
C. Simulating Linear Model

```
> set.seed(20)
> x <- rnorm(100)
> e <- rnorm(100,0,2)
> y <- 0.5 + 2*x + e
> summary(y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Ma
x.
-6.4080 -1.5400  0.6789  0.6893  2.9300  6.50
50
> plot(x,y)
```



What if x is binary

```
> set.seed(10)
> x <- rbinom(100,1,0.5)
> e <- rnorm(100,0,2)
> y <- 0.5 + 2*x + e
> summary(y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Ma
x.
-3.4940 -0.1409  1.5770  1.4320  2.8400  6.94
10
> plot(x,y)
```

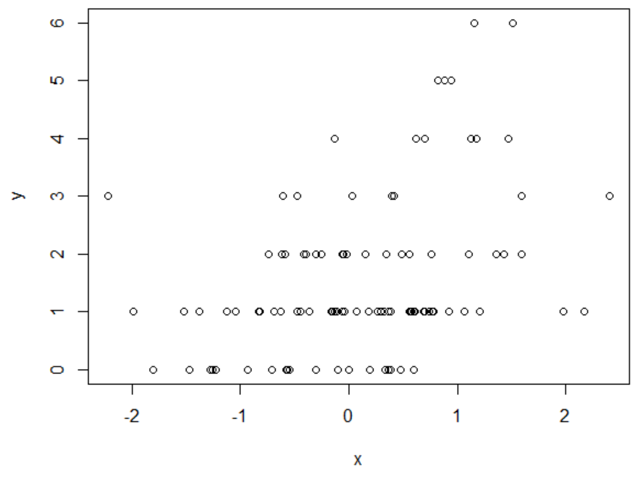


Generate Poisson model

Y ~ Poisson (μ)

$\log \mu = \beta_0 + \beta_1 x$

```
> set.seed(1)
> x <- rnorm(100)
> log.mu <- 0.5+0.3*x
> y <- rpois(100, exp(log.mu))
> summary(y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Ma
x.
   0.00    1.00    1.00    1.55    2.00    6.
00
> plot(x,y)
```



D. Simulation in R : Random Sampling

sample function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions

```
> set.seed(1)
> sample(1:10, 4)
[1] 3 4 5 7
> ?sample
> sample(1:10, 4)
[1] 3 9 8 5
> sample(letters,5)
[1] "q" "b" "e" "x" "p"
> sample(1:10)
 [1]  4  7 10  6  9  2  8  3  1  5
> sample(1:10)
 [1]  2  3  4  1  9  5 10  8  6  7
> sample(1:10, replace = TRUE)
 [1] 2 9 7 8 2 8 5 9 7 8
```