R basics

Dephan

2022-11-19

Variables

- variables are like boxes you put stuff
- you can put things inside by using <- or =
- you can use box without knowing what is in it

```
a = 4
a <- 4
can = c(TRUE, FALSE)
# = is for assigning

# == is equal to
```

Some special numerical constants

1. Inf = infinite
2. NaN = not a number
3. NA = missing

BASICS

getwd() # current directory

setwd(" ") # to change directory and use /

dir() # shows the files in your directory

ls() # list of files in environment

install.packages("package name") # install package

library("package name") # load package and activate it

save(object,file="filename.RData") # save your object as binary

load() # reload data

save.image("Filename.RData") # save your environment

?function name # for help file for function

```
?seq
```

??term #search for a term

```
??"deviation"  or  ??"+"
```

Type of DATA

- Logical TRUE, FALSE
- Numeric 5, 7.9, 100.6
- Character “one”, “two”, “three” Characters are always between “ ”
- Vector - should contain a list of things of only one a single data type either logical, numeric or character.

the function c() is used for making vectors

```
v1 <- c(1,2,4,5,6,7)  #numeric vector

v1
```

```
[1] 1 2 4 5 6 7
```

```
 v2 <- c("one","two","three")  #character vector

 v2
```

```
[1] "one"   "two"   "three"
```

```
 v3 <- c(TRUE,TRUE,TRUE,FALSE)  #logical vector

 v3
```

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

Logical < numeric < character

if you have different data types combined together it will upgrade the lower data type to the higher data type. Logicals are either TRUE or FALSE so if combine it with a numeric, the logical will be upgraded to numeric. If a numeric is combined with a character the numeric is upgraded to character.

```
c(v1,v3)
```

```
 [1] 1 2 4 5 6 7 1 1 1 0
```

```
class(c(v1,v3)) # here v1 is a numeric vector while v3 is a logical when combined with the data type wi
```

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

```
c(v1,v2,v3)
```

```
 [1] "1"     "2"     "4"     "5"     "6"     "7"     "one"   "two"   "three"
[10] "TRUE"  "TRUE"  "TRUE"  "FALSE"
```

```
class(c(v1,v2,v3)) # v3 is a character so the entire data type is upgraded to character
```

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

Matrix - is two dimensional array like an excel file

```
 y<- matrix(1:20,nrow = 5, ncol = 4)  #5 x 4 numeric  matrix.
 y
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
```

for making a matrix we use function matrix()

we can only have one data type either a numeric matrix, character matrix or logical matrix never a mix

```
b = matrix(1:20, 5, 4)
b
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
```

r fills on column basics

so if wanna fill by rows use

```
j = matrix(1:20, 5, 4, byrow = TRUE)
j
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   13   14   15   16
[5,]   17   18   19   20
```

Working with Data types

length(object) # number of elements in variable

```
 measurements = c(1,2,3,4,5,6,7,7,8)

length(measurements)
```

```
[1] 9
```

```
str(measurements)
```

```
 num [1:9] 1 2 3 4 5 6 7 7 8
```

str(object) # structure of object

```
str(measurements)
```

```
 num [1:9] 1 2 3 4 5 6 7 7 8
```

`class(object)` # class or type of an object

as. # force to a certain type

is. # is of certain type ?

`as.numeric()` # Force to a numeric

`as.character()` # Force to a character

`is.numeric()` # to check numeric in a variable

`is.character()` # to check if its a character

`is.matrix()` # to check if its a matrix

Creating Vectors & Matrices

Vectors

`c(object, object,.......)` # combine object to vector

`seq(from, to, by)` # A numerical sequence

`rep(object, times)` # repeat an object/ number

```
 seq(1, 100, 2)
```

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

```
rep("A",3)
```

```
[1] "A" "A" "A"
```

Matrix

matrix(object, nrow=, ncol=) # create matrix

cbind(object,object...) # combine objects as columns

rbind(object,object...) #combine objects as rows

m = matrix(" ", 10, 10) # empty character matrix

Vector examples

```
v1 <- 1:4

v2 = seq(1, 100, 7)

v3 = rep(1,4)

v4 = rep("A",5)
```

Matrix

```
m1 = matrix("", 10, 10)
m2 = matrix (NA, 10, 10)

m3 = cbind(v2,v3)  # cbind combines matrix as columns
```

```
Warning in cbind(v2, v3): number of rows of result is not a multiple of
vector length (arg 2)
```

```
m4 = rbind(v2,v3) # rbind combines matrix by rows
```

```
Warning in rbind(v2, v3): number of columns of result is not a multiple of
vector length (arg 2)
```

#Indexing vectors

- we use [] for selecting from vectors or matrices

```
z= c("a","b","c","d","e","f","g","h")
   # to select the fifth vector from z

z[5]
```

```
[1] "e"
```

```r
z[1:4]
```

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

```r
z[c(1:4,7)]
```

```
[1] "a" "b" "c" "d" "g"
```

```r
# to select the fifth vector from z
```

```r
g = matrix(1:50, 5, 10, byrow = TRUE)
g
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    3    4    5    6    7    8    9    10
[2,]   11   12   13   14   15   16   17   18   19    20
[3,]   21   22   23   24   25   26   27   28   29    30
[4,]   31   32   33   34   35   36   37   38   39    40
[5,]   41   42   43   44   45   46   47   48   49    50
```

```r
g[1:3,5]
```

```
[1]  5 15 25
```

```r
g[5,4:6]
```

```
[1] 44 45 46
```

```r
g[,5]
```

```
[1]  5 15 25 35 45
```

```r
#matrix(row,column)

g[4,]
```

```
 [1] 31 32 33 34 35 36 37 38 39 40
```

Advanced Data types

Data Frame is not a matrix, can contain multiple basic data types and we put into it and we can make a single two dimensional matrix with the function `data.frame`. Every column can have a different data type

```
v1 = c(1,2,3,4)

v2 = c("red","white","red",NA)

v3 = c(TRUE,TRUE,FALSE,TRUE)

my_df = data.frame(v1,v2,v3)

my_df
```

```
  v1    v2    v3
1  1   red  TRUE
2  2 white  TRUE
3  3   red FALSE
4  4  <NA>  TRUE
```

```
# df is variable which is a data frame
```

List is not a vector and can contain anything. We use function`list()`

```
my_list = list(name="Dephan",
               numbers= v1,
               age = 25)

#"Dephan" is a character vector, v1 is a numeric vector, age is a numeric

my_list
```

```
$name
[1] "Dephan"

$numbers
[1] 1 2 3 4

$age
[1] 25
```

```
str(my_list) # here we can use the str function for complex data types
```

```
List of 3
 $ name   : chr "Dephan"
 $ numbers: num [1:4] 1 2 3 4
 $ age    : num 25
```

Factor is a categorical variable like males or females

```
sex = as.factor(c(rep("males",20),
                  rep("female",30)))
sex
```

```
 [1] males  males  males  males  males  males  males  males  males  males
[11] males  males  males  males  males  males  males  males  males  males
[21] female female female female female female female female female female
[31] female female female female female female female female female female
[41] female female female female female female female female female female
Levels: female males
```

Comments # whatever comes after this is ignored by R

#indexing a list [[ ]] to select things use [[ ]] from lists. $ is select always name elements of list so it is easier to select from list.

```
my_list = list(name="Dephan",
               numbers= v1,
               age = 25,
               matrix = matrix(c(1,0,0,1),nrow = 2,ncol =2))
my_list$number[c(2,3)] # to select 2nd and 3rd elements from number.
```

```
[1] 2 3
```

```
my_list$name
```

```
[1] "Dephan"
```

```
my_list[[1]][1]
```

```
[1] "Dephan"
```

```
my_list[[3]][1]
```

```
[1] 25
```

```
my_list$matrix[2]
```

```
[1] 0
```

#matrix and data.frame functions

`nrow(matrix)` # number of rows

`ncol(matrix)` # number of columns

`rownames(matrix)` #names of rows

`colnames(matrix)` #names of columns

```
nrow(g)
```

```
[1] 5
```

```r
ncol(g)
```

```
[1] 10
```

```r
rownames(g) = c("r1", "r2", "r3","r4","r5")

colnames(g) = c("c1","c2","c3","c4","c5","c6","c7","c8","c9","c10")

g
```

```
   c1 c2 c3 c4 c5 c6 c7 c8 c9 c10
r1  1  2  3  4  5  6  7  8  9  10
r2 11 12 13 14 15 16 17 18 19  20
r3 21 22 23 24 25 26 27 28 29  30
r4 31 32 33 34 35 36 37 38 39  40
r5 41 42 43 44 45 46 47 48 49  50
```

```r
g[,"c2"]
```

```
r1 r2 r3 r4 r5
 2 12 22 32 42
```

```r
colnames(g) = paste0('measurements',seq(1,10)) # to give column names

rownames(g) = paste0('Tree',seq(1,5)) # to give rows names

g
```

```
      measurements1 measurements2 measurements3 measurements4 measurements5
Tree1             1             2             3             4             5
Tree2            11            12            13            14            15
Tree3            21            22            23            24            25
Tree4            31            32            33            34            35
Tree5            41            42            43            44            45
      measurements6 measurements7 measurements8 measurements9 measurements10
Tree1             6             7             8             9            10
Tree2            16            17            18            19            20
Tree3            26            27            28            29            30
Tree4            36            37            38            39            40
Tree5            46            47            48            49            50
```

```r
rownames(g) # to know row names
```

```
[1] "Tree1" "Tree2" "Tree3" "Tree4" "Tree5"
```

```r
colnames(g) # to know column names
```

```
 [1] "measurements1"  "measurements2"  "measurements3"  "measurements4"
 [5] "measurements5"  "measurements6"  "measurements7"  "measurements8"
 [9] "measurements9"  "measurements10"
```

Transpose matrix `t(matrix)` # fills rows and columns around

```
g
```

```
      measurements1 measurements2 measurements3 measurements4 measurements5
Tree1             1             2             3             4             5
Tree2            11            12            13            14            15
Tree3            21            22            23            24            25
Tree4            31            32            33            34            35
Tree5            41            42            43            44            45
      measurements6 measurements7 measurements8 measurements9 measurements10
Tree1             6             7             8             9            10
Tree2            16            17            18            19            20
Tree3            26            27            28            29            30
Tree4            36            37            38            39            40
Tree5            46            47            48            49            50
```

```
t(g)
```

```
              Tree1 Tree2 Tree3 Tree4 Tree5
measurements1     1    11    21    31    41
measurements2     2    12    22    32    42
measurements3     3    13    23    33    43
measurements4     4    14    24    34    44
measurements5     5    15    25    35    45
measurements6     6    16    26    36    46
measurements7     7    17    27    37    47
measurements8     8    18    28    38    48
measurements9     9    19    29    39    49
measurements10   10    20    30    40    50
```

Control strucures

They are conveyor belts that guides our variables to their correct destination based on a fixed algorithm

Branching we use: `if else if switch`

looping we use : `while for repeat`

if, else and Switch

something unknown in the box we can use `if` statement.

```
box = "red"

if(class(box) == "character") {
  print("yes")
} else{
    print("no")
  }
```

```
[1] "yes"
```

`if` is followed by a statement between ( ) which evaluates to frue or false. Here in the above code our statement is (class(box) == "character"). our conditions are set between `{ }` If its a character it will print "yes" else "no"

```
# another example
random_number = runif(1) #runif() generates a random number

if(random_number < 0.5) {
  cat("random_number is smaller than 0.5")
} else {
  cat("random_number is greater than 0.5")
}
```

```
random_number is smaller than 0.5
```

```
# cat() is a function like print()
```

switch uses `()`

```
switch(class(random_number),
       logical = cat("logical"),
       numeric = cat("numeric"),
       character = cat("character"))
```

```
numeric
```

else if statement

else if allows us to make multiple routes. here first route is if character, second route if its numeric, the third route others

```
if(class(random_number) == "character") {
  cat("character")
} else if(class(random_number) == "numeric"){
  cat("numeric")
} else {
    cat("others")
  }
```

```
numeric
```

to check if all number are smaller than 5 we use `all` statement

```
1:10 < 5
```

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

```
all(1:10 > 5)
```

```
[1] FALSE
```

```r
x = c(1,2,3,4)

if(all(x < 5)) {
  print("all smaller than 5")
}
```

```
[1] "all smaller than 5"
```

to check iF any number in the vector smaller than 5 uses `any` statement

```r
1:10 < 5
```

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

```r
any(1:10 < 5)
```

```
[1] TRUE
```

```r
x = c(6,2,7,8,9)

if(any(x < 5)) {
  print("at least 1 number is smaller than 5")
}
```

```
[1] "at least 1 number is smaller than 5"
```

Practice else if statements by doing them otherwise.

for, while, repeat

```r
box = 1000                # put 1000 in the box

#FOR

  for (x in 1:10) {         # X will take the values 1,2,3....10
  box = box - x             # Take X element out of the box
  }

box   # first 1000-1, 999 -2, 997 - 3,.........955-10
```

```
[1] 945
```

you can use a `for` statement when you know how many times you wanna do something or how often you wanna do something.

`while` statement when you don't know how many times you wanna do something. it will check if something is true and continue until it is not true.

```
#WHILE
 box = 1000                 # put 1000 is in the box
takeout = 1                 # number of elements we take out

while(takeout <= 10 ){
  box = box - takeout                #take them out
  takeout = takeout +1              #increase the number to take out
}


y = runif(1)
count = 0

while(y < 0.9) {
  count = count + 1
  cat("count = ", count, ",value =",y, "\n")
  y = runif(1)
}
```

```
count =  1 ,value = 0.4590657
count =  2 ,value = 0.3323947
count =  3 ,value = 0.6508705
count =  4 ,value = 0.2580168
count =  5 ,value = 0.4785452
count =  6 ,value = 0.7663107
count =  7 ,value = 0.08424691
count =  8 ,value = 0.8753213
count =  9 ,value = 0.3390729
count =  10 ,value = 0.8394404
count =  11 ,value = 0.3466835
count =  12 ,value = 0.3337749
count =  13 ,value = 0.4763512
count =  14 ,value = 0.8921983
count =  15 ,value = 0.8643395
count =  16 ,value = 0.3899895
count =  17 ,value = 0.7773207
```

An example

```
even <- seq(2,100, by =2)  # a vector from 2 to 100
total <- 0
for(number in even){           # number will be 2
  total = total + number       # add number to total
}

total
```

```
[1] 2550
```

```
# what happens  ?
#2,4,6,8.......100
# total = 6 + 6 = 12
```

* & and && *

This also applicable to | and || (or) & is vectorised can be used for logical vectors

```
v1 = c(1,2,3,4,5,6)

v1 = 4 & v1 > 4

v1
```

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

```
#`&` for selecting from vectors
```

&& is not vectorised and it takes the first element from a vector make sure to use it only for single values.

```
TRUE && FALSE
```

```
[1] FALSE
```

```
c(TRUE,FALSE) && TRUE
```

```
Warning in c(TRUE, FALSE) && TRUE: 'length(x) = 2 > 1' in coercion to
'logical(1)'
```

```
[1] TRUE
```

```
#Warning: 'length(x) = 2 > 1' in coercion to 'logical(1)'[1] TRUE

# use `&&` for `if` statements
```

rule of thumb use `&&` for `if` statements, `&` for selecting from vectors

if statements

in if statements you need a single logical value so the output from `&` and `|` cannot be used directly *the if statement will just take the first comparison

In Vector Comparisons

- logical vectors can be used as indexes

```
x = 10:1
x < 5
```

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

```
x
```

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

```
x[c(TRUE,FALSE)] # this is to give all the even numbers as 10 is TRUE and 9 is FALSE 8 is TRUE 7 is FAL
```

```
[1] 10  8  6  4  2
```

```
# use logical vector as an index

g = x[x<5]      # here we subset x  and numbers smaller than 5

g
```

```
[1] 4 3 2 1
```

```
x = 1:100

x[x < 30 & x >10 & x%%2==0]
```

```
[1] 12 14 16 18 20 22 24 26 28
```

```
# use of &
```

Vector Statements

* Combing logical vectors, advanced selection in vectors

* A & B - pairwise AND

* A|B - pairwise OR

```
x = c(1,2,3,4,5,6,7,8,9,10)

x > 3 & x < 7
```

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

```
subset = (x > 3 & x < 7)

x[subset]
```

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

* this work for matrices also

```
m1 = matrix(1:9, 3, 3)
m1
```

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

```
col1lower3 = m1[, 1 ] < 3          # selects column 1 smaller than 3
col1lower3
```

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

```
m1[col1lower3,]                    # select row for which col 1 > 3
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
```

```
colequalto3 <- m1[,1] ==3
m1[colequalto3,]                   # select rows for which col 1 == 3
```

```
[1] 3 6 9
```

SPECIAL control structures

- *WARNINGS*

if (x <= 0) warning ("this might go horribly wrong")

- *ERRORS*

if(x = 0) stop('this will go wrong")

- *Try catch* to the code and continue even if we get error

```
tryCatch(expression,                        # this might stop
         error = function(e){                # catch the error
           print("trying to recover")         # handle it
         },
         finally = print("hello"))             # always do this after
```

```
[1] "hello"
```

```
function (...)  .Primitive("expression")
```

Advanced looping

- `lappy(X,FUNction,.....)`. lapply is linearly apply

*Repeat a function to each element in a vector or list

```
x <- 1:10
lapply(x,"/", 8)   # go through all the vectors x and divide by 8
```

```
[[1]]
[1] 0.125

[[2]]
[1] 0.25

[[3]]
[1] 0.375

[[4]]
[1] 0.5

[[5]]
[1] 0.625

[[6]]
[1] 0.75

[[7]]
[1] 0.875

[[8]]
[1] 1

[[9]]
[1] 1.125

[[10]]
[1] 1.25
```

```
m <- matrix(1:100,20,5)

apply(m, 2, mean)    # calculate mean of column
```

```
[1] 10.5 30.5 50.5 70.5 90.5
```

```
# lapply example

my_list = list(1:5, c(1,2,3,NA)) # first element 1:5 second element c(1,2,3,4)

#calculate the mean for each element in mylist
lapply(my_list, mean)
```

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

[[2]]
[1] NA
```

```
# na.rm is a parameter to the mean function to remove na, use add it to lapply

lapply(my_list,mean,na.rm = TRUE)
```

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

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

apply n

```
# apply example

mymatrix = matrix(1:50,10,5)

apply(mymatrix,1,mean)  # mean value per  row
```

```
 [1] 21 22 23 24 25 26 27 28 29 30
```

```
apply(mymatrix,2,mean)   # mean value per column
```

```
[1]  5.5 15.5 25.5 35.5 45.5
```

1. lapply and apply are sometimes more efficent

2.speed wise depending on your cpu

3. memory wise
4. Arithmetic function should be quoted "+"
5. `lapply` and `apply` gives back a list so we should unlist it

```
unlist(lapply(1:2,"+",5))
```

```
[1] 6 7
```

FUNCTIONS

- Factories(functions) contain boxes (variable) and conveyor belts (control statements)
- multiple bpxes can go in (function parameters)

- boxes for intermediate stuff(local variables)
- but only one bpx can come out (returned value)
- define a function by using the keyword `function`
- to return a box use the special control statement `return`

```
# basic function example

box1 = 4
box3 = 5
box2 = 3

boxfactory <- function(box1, box2, box3){
  fbox <- (box1 - box2) * box3
  if (box1 < box2) {
    return (box1)
  } else if (box1 > box3){
    return (box2)
  }
  return(fbox)
  }

# function parameters box1 box2 box3
#local variable fbox
```

Function parameters some theory

*pass-by-value used in python

Function parameters get copied into the function, changing them does not alter the state of the variable that was passed

- pass-by-reference used in C#
  - Function parameters are references to the variable passed into the function, changing them alters the state of the variable that was passed
- pass-by-promise used in R
  - Function parameters are references to the variables passed into the function, when changing them, a copy is made. This way the state of the variable that was passed is not changed.

In R function parameters get copied when changed

*inside a function

*pass-by-promise

*So updating them is not visible form the outside

```
exampleFun <- function(p1){
p2 <- p1 * 8             # p1 is still a reference to myvar
p1 <- p2 + 5             # p1 is now a COPY of myvar
return( 1)
}

myvar <- 5

exampleFun(myvar)
```

```
[1] 1
```

Functions: Default parameters

We can set a reasonable default for some function parameters.

- alpha = 0.05, FDR = 10, n.perm = 1000

FDR false discvovery rate n.perm number of permutations

```
exp <- 5 # Exponent in parent scope

someFunction <- function(inParam, exponent = 2){
intern <- (inParam)^exponent # Calculation
return(intern)
}

someFunction(5)
```

```
[1] 25
```

```
someFunction(5,exp)
```

```
[1] 3125
```

Functions: dot dot dot

Functions can be variadic.

*A variadic function is a function which accepts a variable number of arguments

- We can use ... to specify variadic functions in R
- Example: sum
- We don't know beforehand how many elements the user would like to add up

```
function(...,na.rm = FALSE) .Primitive("sum")
```

```
function(...,na.rm = FALSE) .Primitive("sum")
```

```
sum(1,2,3,4,5,6,7)
```

```
[1] 28
```

```
mysum <- function(...){
       count <- 0                 # Initially the count is 0
      for(x in list(...)){        # Go through the parameters
         count <- count + x          # Add them to count
}
        return(count) # Return our count to the user
}

mysum()
```

```
[1] 0
```

```
mysum(1,2,3)
```

```
[1] 6
```

```
variadicTest <- function(...){
return(list(...))                   # Return the parameters
}

variadicTest(param1 = 15)           # Using named parameters
```

```
$param1
[1] 15
```

```
variadicTest(param1 = 15, test =1:10)
```

```
$param1
[1] 15

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

```
someFunction <- function(inParam){         #inparam input parameters
  intern <- (inParam)^2
  return(intern)
}

someFunction(8)
```

```
[1] 64
```

```
#intern
#Error: object 'intern' not found
#intern is not visible from outside
#This is called the scope of the variable
# We can however access variables in our parent scope
#This is considered bad practice, but sometimes we need to...
```

intern is not visible from outside

This is called the scope of the variable

We can however access variables in our parent scope

This is considered bad practice, but sometimes we need to...

Possible reasons :

1. WE ARE BEING LAZY
2. SAVE RAM
3. Plot functopn use it to read environment settings

```
# example of being lazy

exponent <- 5

someFunction <- function(inParam){
  intern <- (inParam)^exponent
  return(intern)
}

someFunction(5)
```

```
[1] 3125
```

```
#never write function like the above.
# when writing function make sure all the variables are defined as input parameters or local variables
```

How should the function look like ?

```
exponent <- 5 # a global variable defined in global scope
#define a function
somefunction <- function(inParam,exponent){
  intern <- (inParam)^exponent
  return(intern)
}

somefunction(5, exponent)
```

```
[1] 3125
```

```
# in the above code if the value of exponent is changed it doesnt effect the end result of the code
```

```r
#Another example of making a function

dobox <- function(n.row = 5,b.length =5){
  for(x in 1:n.row){                #for(var in seq) expr
    xrow <- rep("X",b.length)
    cat(xrow,
        "\n")
  }
}
dobox()
```

```
X X X X X
X X X X X
X X X X X
X X X X X
X X X X X
```

```r
dobox(7,9)
```

```
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
X X X X X X X X X
```

```r
# if you write a function all the variable should be input parameters or temporary variables.
```

Brackets Overview

(and)

- Used when you call a function
- Used in control structure statements

[and]

- Specify an index in a vector, matrix or data.frame

[[and]]

- Specify an index in a list

{ and }

- Defines blocks of code, is used to surround expressions
- What expressions belong to an if statement
- What expressions belong to this function

Escaping the inevitable

- About strings, they are enclosed using:" or '

*Combine strings using:`paste`

*Print them to screen using:`print`

*Print them to anywhere (screen and file) using:`cat`

Forgetting to close string

*Forgetting to close a string happens (a lot)

*In R, no command after will produce output

*Then look at the symbol in front of your cursor

```
# example

print(paste("Hello", "world")) # To screen
```

```
[1] "Hello world"
```

```
cat(paste("Hello", "world"), file = "out.txt")  # To a file
```

So what if we want to print " to a file ?

We need to 'escape' the character,

characters that need escaping:

```
#1. Quotes: \" and \'

# 2.Newline: \n

# 3.Tab: \t

# 4.Backslash: \\

# 5.Backspace: \b


cat("hello\n")
```

```
hello
```

```
cat("hello: \"\n")
```

```
hello: "
```

When using cat, we print verbatim, meaning we need to make sure to add the end of line element, otherwise R continues on the same line

```r
cat("Hello", "World\n", sep=",")
```

```
Hello,World
```

```r
cat("Hello", "World\n", sep=" ")
```

```
Hello World
```

```r
cat("Hello", "World\n", sep="-")
```

```
Hello-World
```

Uniform distribution

Every value has the same chance of being drawn

runif()

*Gaussian distribution

Value near the mean have a higher chance of being drawn

rnorm()

- Poisson distribution

Numbers at the low end of the distribution have a higher chance of occurring

rpois()

If you need to have repeatable randomness

Use `set.seed`