NATIONAL ENGINEERING CENTER
University of the Philippines
Diliman, Quezon City

4.0 Basic R Programming

Eugene Rex L. Jalao, Ph.D.

Associate Professor

Department Industrial Engineering and Operations Research

University of the Philippines Diliman

@thephdataminer

Module 6 of the Business Intelligence and Analytics Certification of UP NEC and the UP Center for Business Intelligence

Outline for this Training

- Introduction to R and R Studio
- Data Types and Operators
  - Case Study on R Scripting
- Reading, Manipulating and Writing Data
  - Case Study on Dataset Analysis with ETL
- **Basic R Programming**
  - **Case Study: Writing Functions**
- Graphics and Plotting
- Deploying R and Dashboard Generation
  - Case Study: Deploying a Simple Dashboard
- Deploying R with C#
  - Case Study: A Simple Standalone GUI For R Apps

Outline for this Session

- Control Structures
- Functions
- Sourcing Scripts

Control Structures

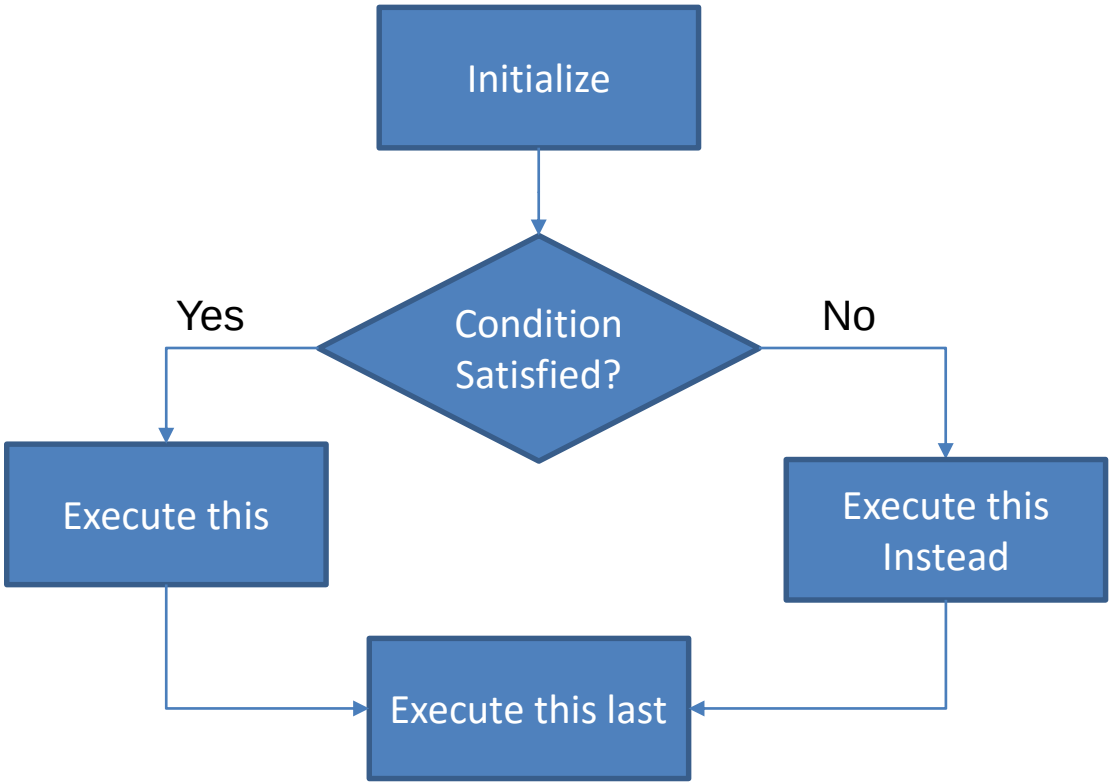
Definition 4.1: Control Structures

- Control structures in R allow you **to control the flow of execution** of the program. Examples are:
  - if, else: testing a condition
  - for: execute a loop x number of times
  - while: execute a loop while a condition is true
  - repeat: execute an infinite loop
  - break: break the execution of a loop
  - next: skip an interaction of a loop
  - return: exit a function
- Most control structures are not used in interactive sessions, but rather when writing functions or longer expressions.

Control Structures

Definition 4.2: Conditional Statements

- Conditional Statements control the flow of the execution



Control Structures

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

Control Structures

Example 4.1: Conditional Statements

```
#If then Else
x=4
if(x > 3) {
   y <- 10
} else {
   y <- 0
}
y
```

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


E.R. L. Jalao, Copyright UP NEC,
eljalao@up.edu.ph

Control Structures

- Vectorized If then Else
  - Previous If Else statements are done on an **element wise basis**
  - Some algorithms may require **vector based** if-then-else results
  - `ifelse(logicalvector, truevalue, falsevalue)`
  - This function returns the same size of the logicalvector

Control Structures

Example 4.2: Vectorized Conditional Statements

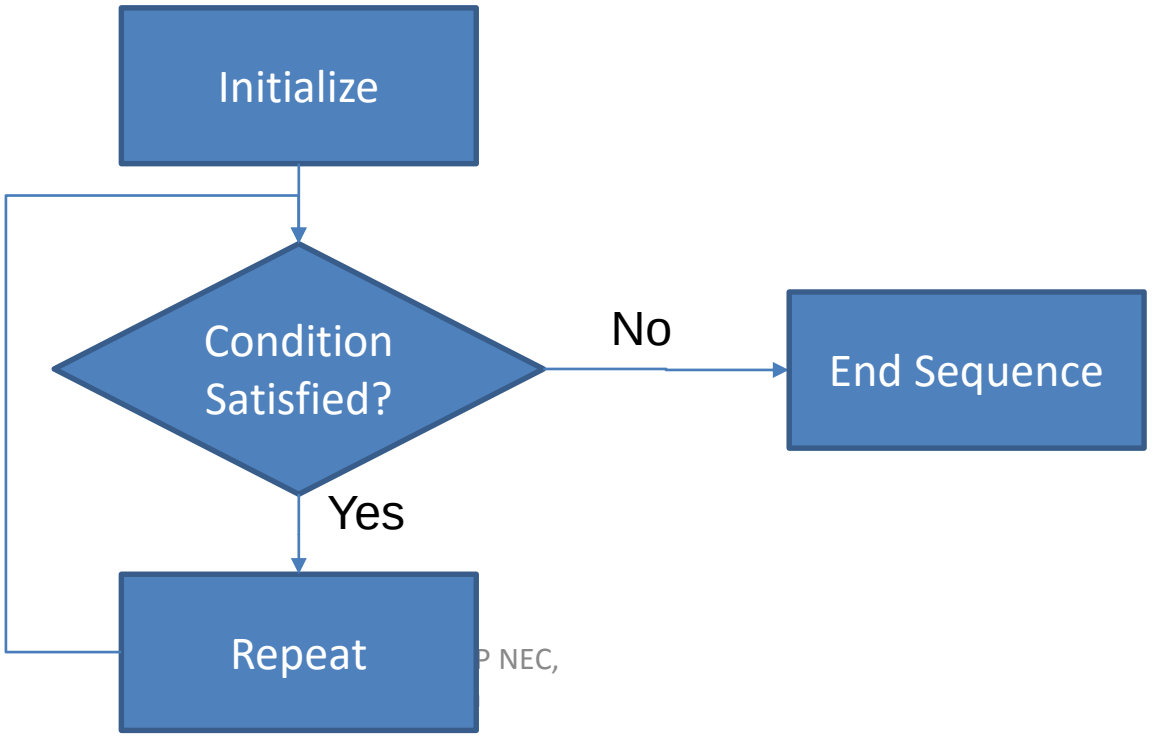
- `#If Else`
- `x <- c(6:-4)`
- `x`
- `sqrt(x)`
- `sqrt(ifelse(x >= 0, x, NA))`

```
> #If Else
> x <- c(6:-4)
> x
 [1]  6  5  4  3  2  1  0 -1 -2 -3 -4
> sqrt(x)
 [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000 0.000000      NaN      NaN
[10]      NaN      NaN
Warning message:
In sqrt(x) : NaNs produced
> sqrt(ifelse(x >= 0, x, NA))
 [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000 0.000000       NA       NA
[10]       NA       NA
>
```

Control Structures

Definition 4.3: Loops

- A loop is a programming control sequence where certain statements are repeated *n* times or until a condition is not met



Control Structures

Definition 4.4: While Loops

- While loops begin by **testing a condition**. If it is true, then they **execute the loop body**. Once the loop body is executed, the condition is tested again, and so forth.

```
count <- 0
while(count < 10) {
  print(count)
  count <- count + 1
}
```

- While loops can potentially result in infinite loops if not written properly. Use with care!

Control Structures

Example 4.3: While Loops

```
x <- 1
while(x < 5){
    x <- x+1
    print(x)
}
```

```
> x <- 1
> while(x < 5){
+ x <- x+1
+ print(x)
+ }
[1] 2
[1] 3
[1] 4
[1] 5
>
```


E.R. L. Jalao, Copyright UP NEC,
eljalao@up.edu.ph

Control Structures

Definition 4.5: For Loops

- for loops take an **iterator 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)
}
```

- This loop takes the i variable and in each iteration of the loop gives it values 1, 2, 3, ..., 10, and then exits.

Control Structures

Example 4.4: For Loops

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

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


E.R. L. Jalao, Copyright UP NEC,
eljalao@up.edu.ph

Control Structures

Example 4.5: Nested Loops

- for loops can be nested.

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

- Be careful with nesting though. Nesting beyond 2-3 levels is often very difficult to read/understand.

Control Structures

- Vector Based and Cursor Based Algorithms
  - Set Based Algorithms are more efficient as compared to cursor based algorithms
  - As much as possible use Set Based Algorithms

Control Structures

Example 4.6: Vector versus Cursor Based Algorithms

```
zVec <- sample(0:999, 10000000, replace=T)
#Cursor Based Example
avg = mean(zVec)
dev = 0
for(i in 1:length(zVec)){
  dev = dev + (zVec[i]-avg)^2
}
sdev = (dev/(length(zVec)-1))^0.5
#Vector Based Example
vdev = (sum((zVec-mean(zVec))^2/(length(zVec)-1)))^0.5
```

Control Structures

Example 4.6 (Cont.) : Vector versus Cursor Based Algorithms

- Which took longer?

```
> #Cursor Based Example
> avg = mean(zVec)
> dev = 0
> for(i in 1:length(zVec)){
+ dev = dev + (zVec[i]-avg)^2
+ }
> sdev = (dev/(length(zVec)-1))^0.5
> sdev
[1] 288.6938
> #Vector Based Example
> vdev = (sum((zVec-mean(zVec))^2/(length(zVec)-1)))^0.5
> vdev
[1] 288.6938
```

Outline for this Session

- Control Structures
- **Functions**
- Sourcing Scripts

Functions

Definition 4.6: Functions

- As we have seen informally along the way, the R language allows the user to create objects called **"functions"**
- These are true R functions that are stored in a special internal form and may be used in **further expressions** and so on.
- In the process, the language gains enormously in power, convenience and elegance, and learning to write useful functions is one of the main ways to make your use of R **comfortable and productive.**

Functions

- Functions are created using **the function() directive** and are stored as R objects just like anything else. In particular, they are R objects of class "function".

```
f <- function(<arguments>) {
## Do something interesting
}
```

- Functions in **R are "first class objects",** which means that they can be treated much like any other R object. Importantly,
  - Functions can be passed as arguments to other functions
  - Functions can be nested, so that you can define a function inside of another function

Functions

- Functions are called **by their name and arguments** are passed
  - `f(<arguments>)`

Functions

Example 4.7: Functions

```
#Declare a Function
squarethis <- function(number){
  return(number^2)
}

#Call the Function
squarethis(1)
squarethis(4)
squarethis(12)
```

```
> #Declare a Function
> squarethis <- function(number){
+ return(number^2)
+ }
> squarethis(1)
[1] 1
> squarethis(4)
[1] 16
> squarethis(12)
[1] 144
>
```

Functions

- The return value of a function is **the last expression** in the function body to be evaluated.
- Functions have named **input arguments** which potentially have default values.
  - The formal arguments are the arguments included in the function definition
  - Not every function call in R makes use of all the formal arguments
  - Function arguments can be missing or might have default values

Functions

- Argument Matching
  - R functions arguments can be matched by position or by name. So the following calls to function sd are all equivalent
    - `mydata <- rnorm(100)`
    - `sd(mydata)`
    - `sd(x = mydata)`
    - `sd(x = mydata, na.rm = FALSE)`
    - `sd(na.rm = FALSE, x = mydata)`
    - `sd(na.rm = FALSE, mydata)`
  - It's not recommend to mess around with the **order** of the arguments too much, since it can lead to some confusion.

Functions

- Argument Matching
  - You can mix positional matching with **matching by name**.
  - When an argument is matched by name, it is “taken out” of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

Functions

Example 4.8: Function Argument Matching

- `args(lm)`
- `#The following two calls are equivalent.`
- `mydata= read.csv("deliverytime.csv")`
- `lm(data = mydata, deltime ~ ncases, model = FALSE, 1:100)`
- `lm(deltime ~ ncases, mydata, 1:100, model = FALSE)`

Functions

```
> #The following two calls are equivalent.
> mydata= read.csv("deliverytime.csv")
> lm(data = mydata, deltime ~ ncases, model = FALSE, 1:100)

Call:
lm(formula = deltime ~ ncases, data = mydata, subset = 1:100,
    model = FALSE)

Coefficients:
(Intercept)       ncases
      3.321        2.176

> lm(deltime ~ ncases, mydata, 1:100, model = FALSE)

Call:
lm(formula = deltime ~ ncases, data = mydata, subset = 1:100,
    model = FALSE)

Coefficients:
(Intercept)       ncases
      3.321        2.176

>
```

Functions

- Argument Matching
  - Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list
  - Named arguments also help if you can remember the name of the argument and not its position on the argument list (plotting is a good example).
  - Function arguments can also be partially matched, which is useful for interactive work.
  - The order of operations when given an argument is
    - 1 Check for exact match for a named argument
    - 2 Check for a partial match
    - 3 Check for a positional match

Functions

- Defining a Function
  - `f <- function(a, b = 1, c = 2, d = NULL) {`
  - `}`
- In addition to not specifying a default value, you can also set an argument value to NULL.

Functions

- Lazy Evaluation
  - Arguments to functions are evaluated lazily, so they are evaluated only as needed.
    - `f <- function(a, b) {`
    - `a^2`
    - `}`
    - `f(2)`
  - This function never actually uses the argument b, so calling f(2) will not produce an error because the 2 gets positionally matched to a.

Outline for this Session

- Control Structures
- Functions
- **Sourcing Scripts**

Sourcing Scripts

- R can read a R file and execute its contents without typing
- Can be used for functions that are used all the time
- Methodology:
  - Write an entire script then save as an R file
  - Use the source() function

Sourcing Scripts

- Saving a Script
  - Type the following script as a new R script by clicking on File->New File->R Script

```
#Declare this function
arithmeticsum <- function(number){
  return(number/2*(1+number))
}
```

  - Save the file in the current working directory as "arithmeticsum.R"

Sourcing Scripts

Example 4.9: Sourcing Scripts

- Type the following script as a new R script by clicking on File->New File->R Script

```
#Source the Function
source('arithmeticsum.R')

#Call the Function
arithmeticsum(3)
arithmeticsum(10)
arithmeticsum(100)
```

```
> #Source the Function
> source('arithmeticsum.R')
> #Call the Function
> arithmeticsum(3)
[1] 6
> arithmeticsum(10)
[1] 55
> arithmeticsum(100)
[1] 5050
>
```

Case 3

- Writing Functions

Outline for This Session

- If Then Else
- Loops
- Functions
- Sourcing Functions