
R fundamentals 03: Elements of programming, R Markdown (Sept. 19, 2017)

In this section we will learn Conditional Statements and loops, essential coding skill for any programing language. As the last part of R fundamentals series, we will study elements of R programming and R markdown.

Recorded Stream

Unfortunately the stream was not captured, due to the display directing the output to the projector. Sorry for this :-(However, this will be fixed during the next iteration of this lecture.

Conditional statements and operators in R

R provides syntax to evaluate conditions.

```
if (condition) {
  expression
}
```

As an example,

```
# Assign x to -8
x <- -8
# Print if x is negative
if (x < 0) {
  print("x is a negative number")
}
```

```
## [1] "x is a negative number"
```

Classwork/Homework: Change x to 5 and re-run the above code. What does that print?

`if...else` statements are evaluated thus:

```
if (condition) {
  expression 1
}
else {
  expression 2
}
```

Classwork/Homework: Do the previous classwork/homework with the else statement printing "x is positive or zero", when it is.

We can also make `else if` statements as follows:

```
if (condition 1) {
  expression
} else if (condition 2){
  expression
} else {
  expression
}
```

R provides logical operators:

- AND with symbol &
- OR with symbol |
- NOT with symbol !

The **AND** truth table

```
# AND truth table
TRUE  & TRUE   : TRUE
TRUE  & FALSE  : FALSE
FALSE & TRUE   : FALSE
FALSE & FALSE  : FALSE
```

Example: `x <- 5` and `(x >3) & (x <8)` will evaluate to `TRUE`.

The **OR** truth table

```
# OR truth table
TRUE  | TRUE   : TRUE
TRUE  | FALSE  : TRUE
FALSE | TRUE   : TRUE
FALSE | FALSE  : FALSE
```

Example: `x <- 5` and `(x >5) | (x <8)` will evaluate to `TRUE`.

The **NOT** truth table

```
# NOT truth table
!TRUE  : FALSE
!FALSE : TRUE
```

Example: `x <- 5` and `!((x >5) | (x <8))` will evaluate to `FALSE`.

Classwork/Homework: Create an R script that returns the max value of a vector x with length 3. Don't use the aid of an auxiliary variable.

Note: R comes out of the loop as soon as the condition is met. It will **not** evaluate further conditions down the line even if they satisfy.

```r
# Assign x to 6
x <- 6
# Check if x is divisible by 2 or 3, if so print it is
# Otherwise print it is not
if (x %% 2 == 0) {
  print("x is divisioble by 2")
} else if(x %% 3 == 0) {
  print("x is divisible by 3")
} else {
  print("x is neither divisible by 2 nor 3")
}
```

```
## [1] "x is divisioble by 2"
```

We can apply logical operators on vectors and matrices - works element wise:

`c(TRUE,TRUE,FALSE) & c(FALSE,TRUE,TRUE)` will evaluate to `FALSE TRUE FALSE`.

`c(TRUE,TRUE,FALSE) | c(FALSE,TRUE,TRUE)` will evaluate to `TRUE TRUE TRUE`.

Also, we have `&&` and `||` operators in R. The difference between single `&`and `&&`is that, `&&` will only look at the first elements of the vectors. Similarly for `||`. Thus,

`c(TRUE,TRUE,FALSE) && c(FALSE,TRUE,TRUE)` will evaluate to `FALSE` and

`c(TRUE,TRUE,FALSE) | c(FALSE,TRUE,TRUE)` will evaluate to `TRUE`.

We can check two objects are equal by the `==` sign or inequality using `!=` operator.

```r
# Check the quality of logical objects
TRUE == TRUE
```

```
## [1] TRUE
```

```r
# Check the quality of logical objects
TRUE == FALSE
```

```
## [1] FALSE
```

```r
# Compare strings
"hi" == "hello"
```

```
## [1] FALSE
```

```r
# Or numbers
2 == 2
```

```
## [1] TRUE
```

```r
# Check inequlaity
2 != 3
```

```
## [1] TRUE
```

In fact, we can use comparison operators for objects!

```r
# Use comparison on objects
"hello" >= "goodbye"
```

```
## [1] TRUE
```

```r
# How about on logicals?
TRUE < FALSE
```

```
## [1] FALSE
```

Why is the comparison `TRUE < FALSE` evaluates to `FALSE` despite the fact that `T > F` in R? Becuase logicals are *coerced*. `TRUE` coresponds to 1 and `FALSE` to 0. We can also use comparison operators on vectors.

Classwork/Homework:

1. Compare two vectors of equal length and output the result.
2. How about vectors of unequal length? Why does the result makes sense?
3. Compare two matrices and lists and explain how R handles such comparisons.

Functions and data structures in R

The general syntax of a function:

```r
my_function <- function(arg 1, agr 2) {
  body of the function
}
```

Unlike some other languages, in R, there is no special syntax for naming a function. One just defines like any other vector.

Consider a simple function,

```r
# The add function
add <- function(x, y=1) {
  x+y
}
```

If y is not given any value, it will take the default argument. There is no difference between the functions we define and R supplied functions. There are three components for every function, the formal, the body and the environment.

```r
# The add function
add <- function(x, y=1) {
  x+y
}
# Print the formals arguments
formals(add)
```

```
## $x
##
##
## $y
## [1] 1
```

```
  # Print the body
  body(add)
```

```
## {
##     x + y
## }
```

```
  # Print the environment
  environment(add)
```

```
## <environment: R_GlobalEnv>
```

Environment is typically invisible, but it is important on how the function behaves. We will soon see scope for functions. In this case the environment is the global environment.

Classwork/Homework: What does this function do?

```
  f <- function(x) {
          if (x<0) {
              -x
          } else {
               x
            }
        }
```

Functions are just objects in R and act like any other object. You can assign a function to any other object and the object will behave exactly like the assigned function. Function need not have any name - they are called anonymous functions. Anonymous functions has to be called in one line. For example, `(function (x) {x+1})(2)` is an increment function that will produce the output 3.

Scoping

R looks into the function scope for values.

```
  # Assign the value 10 for x
  x <- 10
  # Define a function f that returns a vector
  f <- function() {
    x <- 1
    y <- 3
    c(x,y)
  }
  # Print the function
  f()
```

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

If the value is not in the scope, it looks one level above.

```
# Assign the value 10 for x
x <- 10
# Define a function f that returns a vector
f <- function() {
  y <- 3
  c(x,y)
}
# Print the function
# Note x is not in the scope
f()
```

```
## [1] 10  3
```

Classwork/Homework: Use `rm(x)` in the above code to remove x from the global environment and report what happens when you call the function.

Scoping defines *where* and not *when* to look for a function.

```
# Define a function
f <- function() x
x <- 15
f()
```

```
## [1] 15
```

```
x <- 20
f()
```

```
## [1] 20
```

This is a highly undersirable behaviour. So always as a practice design functions using the variables you'll use and not leave it to the environment.

Lookup works the same for functions when it comes to scoping.

```
# Define a function l
l <- function() x + 20
# Define another function and define l inside f
f <- function() {
      l <- function(x) x +15
      l(20)
}
f()
```

```
## [1] 35
```

If you use name like a function, R ignores any non-function objects when it looks it up. Here is an example:

```r
  # Define c
  c <- 20
  # Use c as a combine
  # R ignores the function aspect of c
  c(c=c)
```

```
##  c
## 20
```

Each call to a function has its own clean environment.

```r
  # Define a function
  f <- function() {
    if (!exists("a")) {
      a <- 10
    } else {
      a <- a + 1
    }
    print(a)
  }
 f()
```

```
## [1] 10
```

Everytime the working environment for this function is empty and so it is going to print only 10.

A note on data structures:

Atomic vectors are of six types: logical, integer, double, character, complex and raw.

Linked lists aka lists are like vectors but can hold multiple data types. Lists can also contain lists.

Every data structure has two key properties. The `typeof()` function that specifies the type of the data structure and `length()` function that gives the length of the data structure.

Another important object in R is the `NULL` that specifies the absence of elements in a data structure. Compare this with `NA` that specifies the absense of values in a data structure. This can be seen as follows:

```r
  # Find the type and length of NULL
  typeof(NULL)
```

```
## [1] "NULL"
```

```r
  length(NULL)
```

```
## [1] 0
```

```r
  # Find the type and length of NA
  typeof(NA)
```

```
## [1] "logical"
```

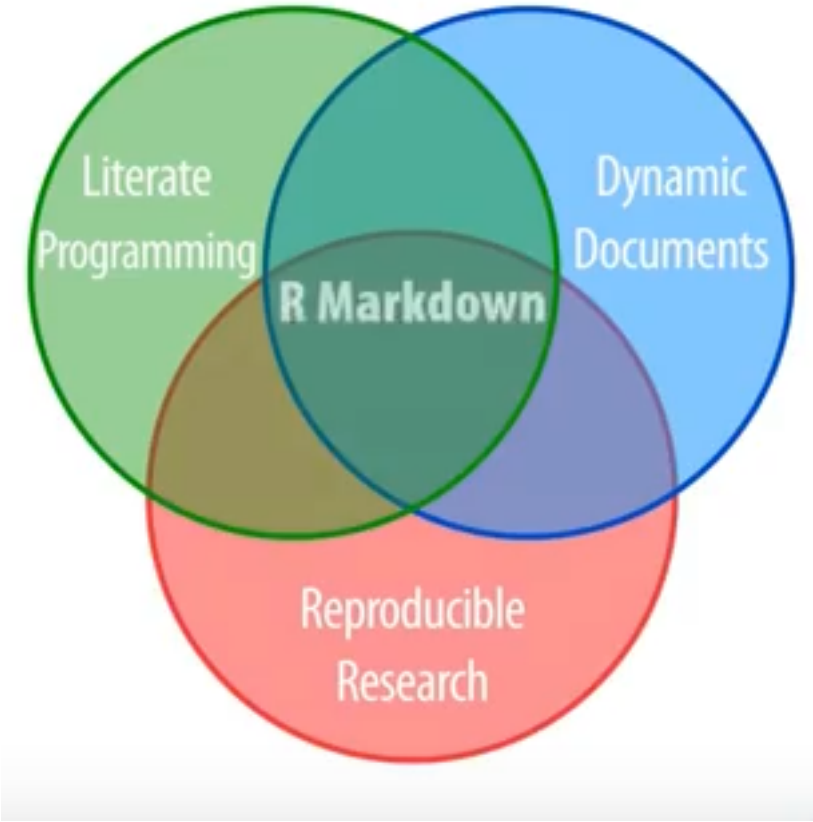

Figure 1:

```
length(NA)
```

```
## [1] 1
```

Quiz: How do you test for `NA` in a vector?

In fact there are many types of `NA` depending on the data type, one for each atomic type (like `NA` for integer etc.). Missing values are contagious - any mathematical operation involving `NA` will result in NA. Any logical comparison involving `NA` will result in `NA` too (like `NA == NA`).

R Markdown

R Markdown is a writing tool that allws us to write pain texts while the formating is kind of taken care by the system. Helps focussing on the content than on formatting - which is extremely important. Several cool features including adding math equations (latex).

R Markdown combines best features of programming elements:

Why write in R Markdown?

- Formatting stays where it should
- Lightning fast to type - no mouse required
- Easy to read when marked up
- Easy to write
- More than anything **REPRODUCIBLE!!!**

R markdown demo (in class)

Note: Never ever change directories in R Markdown documents - just don't do it. You need to put all images inside the folder you are working on to include in the R Markdown document.

It also renders the final report in different formats:

One can also create interactive slide shows with shiny.

Classwork/Homework:

1. Complete Homwork 02 as a R Markdown file, Knit to html and submit both the markdown and the html file. Make good comments for the codes you write. Make the codes visible in the document.
2. How do you include images in R Markdown? Include an image of your choice in the homework.

Selected materials and references

An Introduction to R

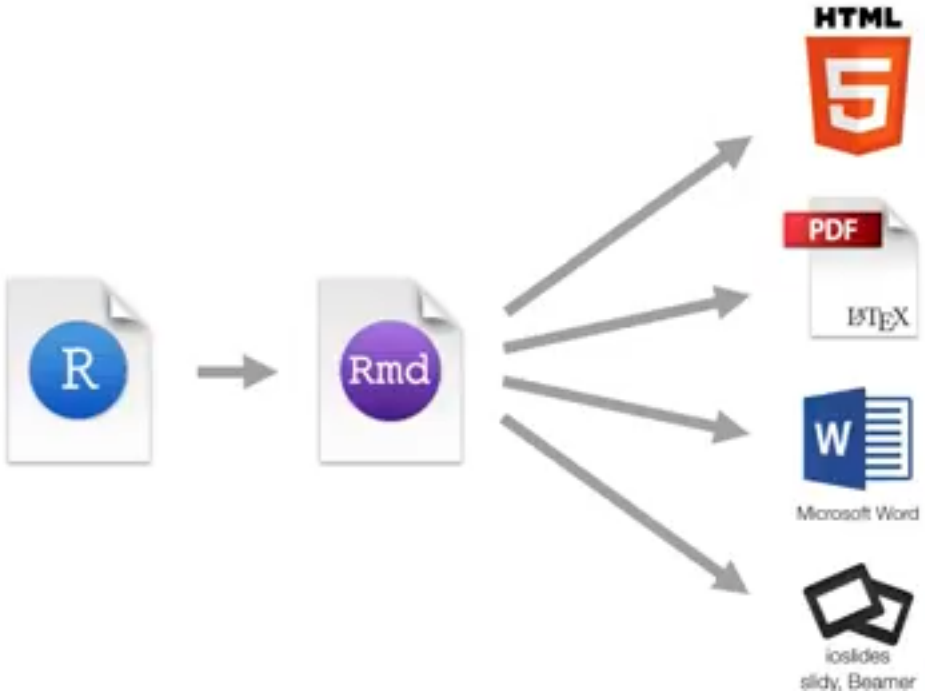

Figure 2: