University of Chicago Professional Education

MSCA 37010 Programming for Analytics
Week 4 Lecture Notes

Autumn 2020

❑ Introduction

Instructor: Mei Najim
Email: mnajim@uchicago.edu
Tel: 847-800-9979 (C)
Class Meeting Time: 6:00 - 9:00pm, Mondays (01 Section)
1:30 - 4:30pm, Saturdays (02 Section)
Office Hour: Mondays (9:00pm until last students)
Saturdays (4:30pm until last students)
Or, email with questions first and following by an appointment if needed

Notes: 1) First ten-minute quiz; Breakout groups in zoom; Three 5-minute breaks
2) Set up a weekly discussion group on canvas, allow 24 hours to respond
3) Email questions with the **Section Number (01/02)** in the subject line
4) If it is urgent, feel free to text me directly (847-800-9979)

Week 4 Class Agenda

- Rmakrdown
- Data Preparation
  - dplyr library continued; Pipe Operator (dplyr library or magrittr library)
- Conditional Execution (If, else, and else if statements)
- Looping (For, While, Repeat)

Week 4 Class Agenda

- Data Preparation
  - dplyr library continued: Pipe Operator (dplyr library or magrittr library)
- Conditional Execution (If, else, and else if statements)
- Looping (For, While, Repeat)

❑ Pipe with *magrittr or dplyr*

- Pipes are a powerful tool for clearly expressing a sequence of multiple operations
- The pipe, %>%, comes from the **magrittr** package in the **tidyverse** load automatically or **dplyr**
- The point of the pipe is to simplify your code with %>% so the magrittr/dplyr package is a powerful tool to have in your data wrangling toolkit
- The magrittr package was created by Stefan Milton Bache two primary aims: "to decrease development time and to improve readability and maintainability of code." Hence, it aims to increase **efficiency and improve readability**; and in the process it greatly simplifies your code
- To see why the pipe is so useful, we're going to explore a number of ways of writing the same code; learn the alternatives to the pipe, when you shouldn't use the pipe, and some useful related tools.

❑ Pipe (*%>%*) Operator

- The principal function provided by the ***magrittr*** package is ***%>%***, or what's called the "pipe" operator.
- This operator will forward a value, or the result of an expression, into the next function call/expression.

Example: A function to filter data can be written as:

filter*(data, variable == numeric_value)*

Or, data ***%>%*** ***filter****(variable == numeric_value)*

Both functions complete the same task and the benefit of using %>% may not be immediately evident in this example

❑ Pipe (*%>%*) Operator

- However, when you desire to perform multiple functions its advantage becomes obvious.
  For instance, if we want to filter some data, group it by categories, summarize it, and then order the summarized results we could write it out in three different ways.

  Option 1 - Nested Option

```
arrange(
    summarize(
        group_by(
            filter(mtcars, carb > 1),
            cyl
        ),
        Avg_mpg = mean(mpg)
    ),
    desc(Avg_mpg)
)
```

Or, arrange(summarize(group_by(filter(mtcars, carb > 1), cyl), Avg_mpg = mean(mpg)), desc(Avg_mpg))

❑ Pipe (*%>%*) Operator

- Option 2 - Multiple Object Option:

```
data(mtcars)
attach(mtcars)
a <- filter(mtcars, carb > 1)
b <- group_by(a, cyl)
c <- summarise(b, Avg_mpg = mean(mpg))
d <- arrange(c, desc(Avg_mpg))
print(d)
```

- Option 3 - %>% Option:

```
library(magrittr)
library(dplyr)

mtcars %>%
    filter(carb > 1) %>%
    group_by(cyl) %>%
    summarise(Avg_mpg = mean(mpg)) %>%
    arrange(desc(Avg_mpg))
```

❑ Pipe (*%>%*) Operator

- R is a functional programming language, meaning that everything you do is basically built on functions, you can use the pipe operator to feed into just about any argument call.

  For example, we can pipe into a linear regression function and then get the summary of the regression parameters. Note in this case insert "data = ." into the lm() function.

  When using the %>% operator the default is the argument that you are forwarding will go in as the first argument of the function that follows the %>%.

```
mtcars %>%
    filter(carb > 1) %>%
    lm(mpg ~ cyl + hp, data = .) %>%
    summary()
```

❑ Pipe (*%>%*) Operator

- However, in some functions the argument you are forwarding does not go into the default first position. In these cases, you place "." to signal which argument you want the forwarded expression to go to.

You can also use %>% to feed into plots:

```
library(ggplot2)
   mtcars %>%
   filter(carb > 1) %>%
   qplot(x = wt, y = mpg, data = .)
```

❑ Pipe (*%>%*) Operator

- Additional Functions. In addition to the %>% operator, ***magrittr*** provides several additional functions which make operations such as addition, multiplication, logical operators, re-naming, etc. more pleasant when composing chains using the %>% operator. Some examples follow:

```
# subset with extract:   mtcars %>%
                           subset(, 1:4) %>%
                           head

# add, subtract, multiply, divide:  mtcars %>%
                                      subset(, "mpg") %>%
                                      multiply_by(5) %>%
                                      head

# logical assessments and filters:  mtcars %>%
                                      subset(, "cyl") %>%
                                      equals(4)

# renaming columns and rows:  mtcars %>%
                                head %>%
                                set_colnames(paste("Col", 1:11, sep = ""))
```

❑ Additional Pipe *(%T>%)* Operator

- Additional Pipe Operators: magrittr also offers some alternative pipe operators. Some functions, such as plotting functions, cause the string of piped arguments to terminate. The tee (%T>%) operator allows you to continue piping functions that normally cause termination.

Normal piping terminates with the plot() function resulting in NULL results for the summary() function

```
mtcars %>%
filter(carb > 1) %>%
subset.data.frame(, 1:4) %>%
plot() %>%
summary()
```

Inserting %T>% allows you to plot and perform the functions that follow the plotting function

```
mtcars %>%
filter(carb > 1) %>%
subset.data.frame(, 1:4) %T>%
plot() %>%
summary()
```

❑ Additional Pipe (*%<>%*) Operator

- Additional Pipe Operators: The compound assignment %<>% operator is used to update a value by first piping it into one or more expressions, and then assigning the result.

  Example: Let's say you want to transform the mpg variable in the mtcars data frame to a square root measurement. Using %<>% will perform the functions to the right of %<>% and save the changes these functions perform to the variable or data frame called to the left of %<>%.

  Note that mpg is in its typical measurement

```
head(mtcars)
mtcars$mpg %<>% sqrt
head(mtcars)
```

❑ Additional Pipe (*%$%*) Operator

- The Exposition (*%$%*) Operator is useful when you want to pipe a dataframe, which may contain many columns, into a function that is only applied to some of the columns.

  Example: The correlation (cor) function only requires an x and y argument so if you pipe the mtcars data into the cor function using %>% you will get an error because cor doesn't know how to handle mtcars. However, using %$% allows you to say "take this dataframe and then perform cor() on these specified columns within mtcars."

```
# regular piping results in an error
  mtcars %>%
  subset(vs == 0) %>%
  cor(mpg, wt)

# using %$% allows you to specify variables of interest
  mtcars %>%
  subset(vs == 0) %$%
  cor(mpg, wt)
```

Week 4 Class Agenda

- Data Preparation
  - dplyr library continued: Pipe Operator (dplyr library or magrittr library)
- Conditional Execution (If, else, and else if statements)
- Looping (For, While, Repeat)

❑ Conditional Execution – if…else Statement

- Conditional Execution of statements are often used in an R function
- The if statement uses the following form:

if **(Test Expression)** {

Statement1

} ***else*** {

Statement2

}

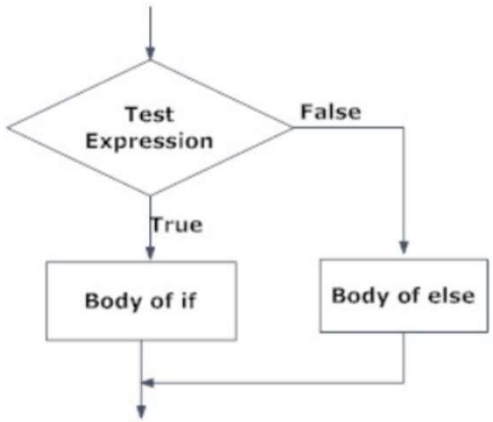


Fig: Operation of if...else statement

❑ Conditional Execution – if…else Example

- Example:

```
x <- -5
  if(x > 0) {
      print ("Non-negative number")
     } else {
      print ("Negative number")
   }
```

Output

```
[1] "Negative number"
```

Also, the above condition can be written in a single line as follows:

```
if(x > 0) print("Non-negative number") else print("Negative number")
```

❑ Conditional Execution – Nested if…else

- if…else Ladder: The if…else ladder (if…else…if) statement allows you execute a block of code among more than 2 alternatives
- The if statement uses the following form:

```
if (Test Expression1) {
    Statement1
 } else if (Test Expression2) {
    Statement2
} else if (Test Expression3) {
    Statement3
} else {
    Statement4
```

❑ Conditional Execution – Nested if…else Example

- Example of nested if…else:

```
x <- 0
if(x > 0){
  print ("Non-negative number")
} else if (x<0){
  print ("Negative number")
} else{
  print("Zero")
}
```

Output
[1] “Zero"

❑ Conditional Execution – ifelse() Function

- Ifelse Function: ***ifelse***(test_expression, x, y)

- test_expression must be a logical vector (or an object that can be coerced to logical). The return value is a vector with the same length as test_expression
- This returned vector has element from x if the corresponding value of test_expression is TRUE or from y if the corresponding value of test_expression is FALSE
- This is to say, the i-th element of result will be x[i] if test_expression[i] is TRUE else it will take the value of y[i]
- The vectors x and y are recycled whenever necessary

Example:

```
> a = c(5,7,2,9)
> ifelse(a %% 2 == 0,"even","odd")
 [1] "odd" "odd" "even" "odd"
```

Week 4 Class Agenda

- Data Preparation
  - dplyr library continued: Pipe Operator (dplyr library or magrittr library)
- Conditional Execution (If, else, and else if statements)
- Looping (For, While, Repeat)

❑ Looping (*for*, *while*, and *repeat*)

- Looping
  - R has three statements that perform explicit looping: ***for***, ***while***, and ***repeat***
  - There are two statements that can be used to explicitly control looping: ***break*** and ***next***

    - The **break** statement causes an exit from the innermost loop that is currently being executed

    - The **next** statement immediately causes control to return to the start of the loop. The next iteration of the loop is then executed if there is one

❑ Looping – *for* Statement

- ***for*** Statement

A ***for*** loop allows a statement to be iterated in a specified sequence and is very valuable when we need to iterate over a list of elements or a range of numbers. Loop can be used to iterate over a ***list, data frame, vector, matrix or any other object***. The braces and square bracket are compulsory. It has the following form:

for *(variable* ***in*** *sequence) statement*

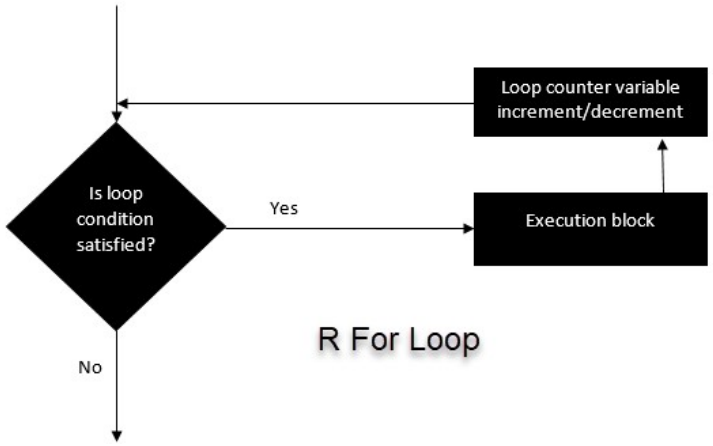
- ***for*** and ***in*** are the **keywords**
- variable is the loop variable
- sequence is the vector of values
- the statement part will often be a grouped statement and hence enclosed with braces { }

❑ Looping – *for* Loop Syntax

- ***for*** Loop Syntax:

  R will loop over all the variables in vector and do the computation written inside the expression

```
for (i in vector) {
   Expression
}
```



❑ Looping – *for* Loop over a Vector

- ***for*** Loop over a Vector:

Example 1: We iterate over all the elements of a vector and print the current value.

```
# Create fruit vector
  > fruit <- c('Apple', 'Orange', 'Banana')
# Create the for statement
  > for (i in fruit){
     print(i)
    }
  [1] "Apple"
  [1] "Orange"
  [1] "Banana"
```

❑ Looping – *for* Loop over a Vector

- *for* Loop over a List:

  Example 2: creates a non-linear function by using the polynomial of x between 1 and 4 and we store it in a list

```
# Create an empty vector
   > list <- c()

# Create a for statement to populate the list
  > for (i in seq(1, 4, by=1)) {
        list[[i]] <- i*i
    }
   print(list)
   [1] 1 4 9 16
```

❑ Looping – *for* Loop over a Matrix

- ***for*** Loop over a Matrix: A matrix has 2-dimension, rows and columns. To iterate over a matrix, we have to define two for loops, namely one for the rows and another for the column.

Example 3:

```
# Create a matrix
  mat <- matrix(data = seq(10, 15, by=1), nrow = 3, ncol =2)
# Create the loop with r and c to iterate over the matrix
  for (r in 1:nrow(mat))
      for (c in 1:ncol(mat))
          print(paste("Row", r, "and column",c, "have values of", mat[r,c]))
[1] "Row 1 and column 1 have values of 10" [1] "Row 1 and column 2 have values of 13"
[1] "Row 2 and column 1 have values of 11" [1] "Row 2 and column 2 have values of 14"
[1] "Row 3 and column 1 have values of 12" [1] "Row 3 and column 2 have values of 15"
```


CN1

Slide 27

CN1 Chris Najim, 3/13/2020

❑ Looping – *for* Loop over a Dataframe

- Example: ***for*** Loop over a list of Dataframe in a Function

```
> m <- data.frame("ID"=c("a","b","c"),"Value"=c(0.23,0.54,0.71))
> n <- data.frame("ID"=c("c","d"),"Value"=c(0.91,0.52))
> o <- data.frame("ID"=c("f","g","h","i"),"Value"=c(0.15,0.49,0.89,1.89))
```

```
Datalist<-list(m,n,o)
stackData<-function(alist){
    result=data.frame()
        for (i in 1:length(alist)) {
         result=rbind(result,alist[[i]])
        }
   return(result)
  }
```

```
> stackData(Datalist)
  ID Value
1  a  0.23
2  b  0.54
3  c  0.71
4  c  0.91
5  d  0.52
6  f  0.15
7  g  0.49
8  h  0.89
9  i  1.89
```

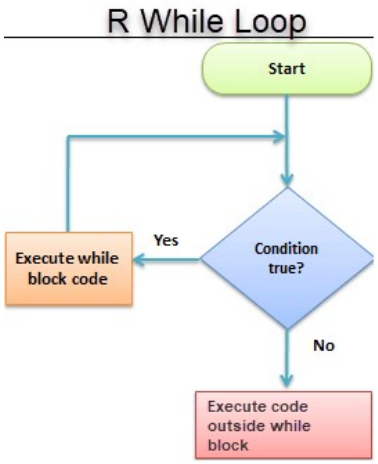
❑ Looping – *while* Loop Statement

- ***while*** Statement: The ***while*** loop does not make use of a loop variable.

  It has the following form:

  1. condition is evaluated. If it is TRUE, then statement is evaluated until statement evaluates to FALSE
  2. write a closing condition at some point otherwise the loop will go on indefinitely
  3. the while loop syntax

```
while (condition) {
    Expression
}
```



❑ Looping – *while* Loop Example

- Example:

```
x<-0
test<-1
while(test>0) {
        x<-x+1
        print(x^2)
        test<-(x<6)
}
```

Output:

[1] 1 [1] 4 [1] 9 [1] 16 [1] 25 [1] 36

❑ Looping – *while* Loop Example

- Example: You bought a stock at price of 50 dollars. If the price goes below 45, we want to short it. Otherwise, we keep it in our portfolio. The price can fluctuate between -10 to +10 around 50 after each loop. You can write the code as follow:

```
set.seed(456)
stock <- 50                          # Set variable stock
price <- 50                          # Set variable price
loop <- 1                            # Loop variable counts the number of loops
while (price > 45) {                 # Set the while statement
      price <- stock + sample(-10:10, 1)   # Create a random price between 40 and 60
            loop = loop +1           # Count the number of loop
      print(loop)                    # Print the number of loop
 }
cat('it took',loop,'loop before we short the price. The lowest price is',price)
```

❑ Looping – *repeat* Loop Statement

- The ***repeat*** Statement:

  The repeat statement causes repeated evaluation of the body until a break is requested

  The syntax of repeat statement

  - When using repeat, statements must be a block statement

  - Within the block, you need to both perform some computation and test whether or not to break from the loop

❑ Looping – *repeat* Loop Example

- The ***repeat*** Statement:

  Example: we can rewrite the previous code by using repeat instead of while

```
x<-0
repeat {
        x<-x+1
       print(x^2)
       if(x==6) break
        }
 [1] 1
 [1] 4
 [1] 9
 [1] 16
 [1] 25
 [1] 36
```

Q & A

Thank You

Contact Information:

Your comments and questions are valued and encouraged.

Mei Najim
E-mail: mnajim@uchicago.edu

LinkedIn: https://www.linkedin.com/in/meinajim/