ISOM 3390: Business Programming in R

Instructor: Bingjie Qian (PhD candidate)

Summer 2020, HKUST

Topic 3: Control Structures

3.1 Overview of Control Structures

R programs are made up of **statements**. Statements are the basic units of instructions that the R interpreter parses and processes.

R statements are separated with a `;` or a new line.

A semicolon always indicates the end of a statement.

```
x = 1:100; mean(x)
```

```
## [1] 50.5
```

A new line sometimes does not indicate the end of a statement. Statements are processed when they are syntactically complete.

```
course <-          # this line of code is syntactically incomplete
  c('R programming', 'Business analytics')
course
```

```
## [1] "R programming"      "Business analytics"
```

```
# try the code in the console
```

Statements can be grouped together using braces { and } to form a larger unit of execution, called a **block**. A block is processed as a whole. R only prints the output of the last statement.

```
{
x <- 1:100
mean(x)
course <- c('R programming', 'Business analytics')
course
}
```

```
## [1] "R programming"      "Business analytics"
```

In general, R executes statements sequentially from the first to the last statement, but sometimes we may want to control the flow of a program.

Control Structures

Control structures allow us to control the flow of a program. Controls structures in R are very similar to those of other programming languages.

Two common control structures in R:

- **Conditional Statements (Conditionals)**: execute different sets of statements, depending on whether a condition is true or false.
- **Iterative Statements (Loops)**: execute a set of statements repeatedly.

Most control structures are not used in interactive sessions (in the console), rather they are usually used for writing functions or longer expressions.

3.2 Conditional Statements

Conditional statements allow us to execute different sets of statements, depending on whether a condition is true or false.

3.2.1 if Statement

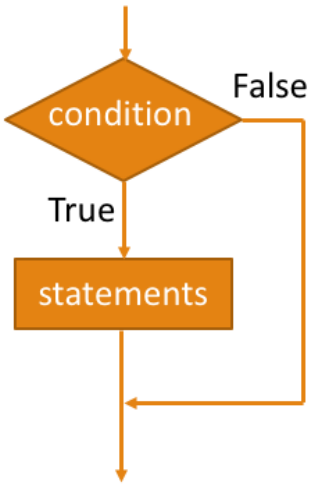

Fig. 1 Flowchart of if Statement

Syntax:

```
if (condition){
  statements
}
```

- If the `condition` is `TRUE`, the `statements` get executed.
- If the `condition` is `FALSE`, nothing happens.

```
x <- 6
if (x > 0) {
```

```
  print("A positive number")
}

## [1] "A positive number"
```

3.2.2 if else Statement

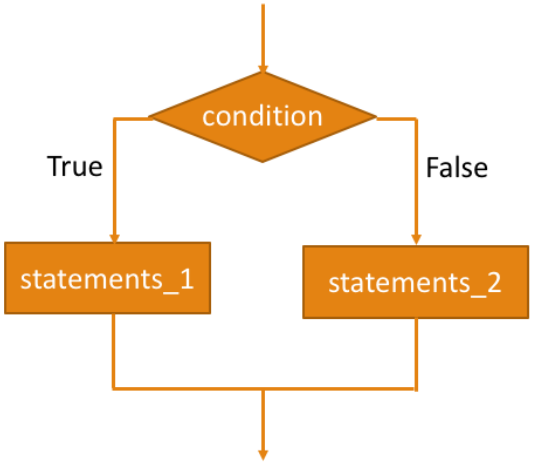

Fig. 2 Flowchart of if else Statement

Syntax:

```
if (condition) {
  statements_1
} else {
  statements_2
}
```

- If the `condition` is `TRUE`, `statements_1` get executed.
- If the `condition` is `FALSE`, `statements_2` get executed.

```
x <- 6
if (x > 100) {
  print("A large number")
} else {
  print("A small number")
}

## [1] "A small number"
```

3.2.3 if else Ladder

The `if else` ladder allows us to choose among more than 2 alternatives. Only one of the alternatives will be executed.

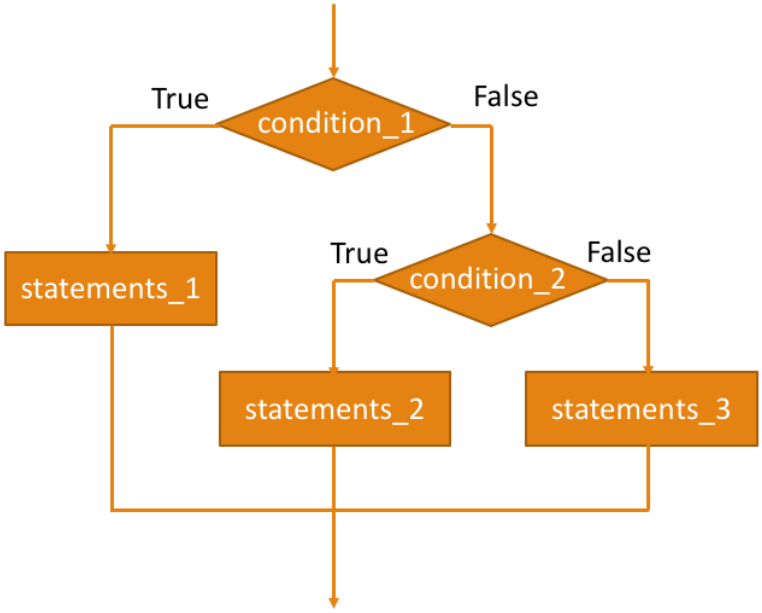

Fig. 3 Flowchart of if else Ladder

Syntax:

```
if (condition_1) {
  statements_1
} else if (condition_2) {
  statements_2
} else {
  statements_3
}
```

- `else` can only be used once at the end. The `else` clause gets executed if none of the above conditions are `TRUE`.
- The keywords `else if` and `else` must be in the same line as the closing brace `}`.

```
x <- 6
if (x > 0) {
  print("A positive number")
} else if (x < 0) {
  print("A negative number")
} else {
  print("It's zero!")
}

## [1] "A positive number"
```

Conditional statements can be written in various forms.

- Single-line statements don't need braces:

```r
x <- 6
if (x > 0) y <- "Positive" else y <- "Not positive"
y
```

```
## [1] "Positive"
```

- The statement can return a value:

```r
x <- 6
y <- if (x > 0) "Positive" else "Not positive"
y
```

```
## [1] "Positive"
```

[Task 1: if else Statement]

Inspect the outputs of the two code snippets below and explain how they work.

```r
x <- 1:10
if (length(x)) "not empty" else "empty"
```

```
## [1] "not empty"
```

```r
x <- numeric()
if (length(x)) "not empty" else "empty"
```

```
## [1] "empty"
```

[End of Task 1]

3.2.4 `switch()`: Select One from Several Alternatives

Syntax:

```r
switch(EXPR, ...)
```

- `switch(EXPR, ...)` evaluates `EXPR` and chooses one of the alternatives accordingly.
- `EXPR` is the expression to evaluate.
- `...` is the alternatives. If `EXPR` is a string, the alternatives will be named, perhaps except for one alternative to deal with the unmatched case.

```r
cars$speed
```

```
##  [1]  4  4  7  7  8  9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14 14
15 15
## [26] 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22 23 24 24 24
24 25
```

```r
type <- "MedianSpeed"
switch(type, MeanSpeed = mean(cars$speed), MedianSpeed = median(cars$speed),
"I don't understand")
```

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

When calling `switch()`, you can leave the right hand side of = empty:

```
switch("b", a = 1, b = , c = 2, d = , e = 3, "nothing")
```

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

```
switch("d", a = 1, b = , c = 2, d = , e = 3, "nothing")
```

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

```
switch("f", a = 1, b = , c = 2, d = , e = 3, "nothing")
```

```
## [1] "nothing"
```

As you can see, when an empty result is found in a match, `switch()` will match the input to the next value that is not empty. This is an interesting feature that you can use if multiple inputs have the same output.

The above code snippet is equivalent to the following one:

```
switch("b", a = 1, b = 2, c = 2, d = 3, e = 3, "nothing")
```

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

```
switch("d", a = 1, b = 2, c = 2, d = 3, e = 3, "nothing")
```

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

```
switch("f", a = 1, b = 2, c = 2, d = 3, e = 3, "nothing")
```

```
## [1] "nothing"
```

[Task 2: `switch()`]

Rewrite the following code snippet with `switch()` (`stop()` stops execution and throws an error):

```
x = "a"   # can be other values
if (x == "a") {
  "option 1"
} else if (x == "b") {
  "option 2"
} else if (x == "c") {
  "option 3"
} else {
  stop("Invalid x value")
}
```

```
## [1] "option 1"
```

[End of Task 2]

3.2.5 `ifelse()`: Conditional Element Selection

The function `ifelse()` performs **conditional element selection**.

Syntax:

```
ifelse(test, yes, no)
```

- `test` is a logial vector of conditions, `yes` and `no` are two vectors we want to select elements from.
- `ifelse(test, yes, no)` returns a vector which is filled with elements selected from either `yes` or `no`, depending on whether the element of `test` is `TRUE` or `FALSE`.
- If `yes` or `no` are too short, their elements are **recycled**.

```
set.seed(0)
x <- rnorm(5, 0, 2)
ifelse(x > 0, x, -x)   # `x > 0` is a logical vector of conditions

## [1] 2.5259086 0.6524667 2.6595985 2.5448586 0.8292829

ifelse(x > 0, x, 0)    # the `no` vector (0) is recycled

## [1] 2.5259086 0.0000000 2.6595985 2.5448586 0.8292829
```

3.3 Iterative Statements

Iterative statements (loops) allow us to execute a set of statements repeatedly.

3.3.1 `for`: Definite Loops

`for` is used for **definite loops** when the number of times to repeat is clear in advance.

Syntax:

```
for (counter in sequence){
  statements
}
```

- `counter` is a variable that iterates over the `seqeunce` (or a vector or a list).
- `statements` (called the *body* of the loop) are repeatedly run through iterations.

```
x <- c("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` loops can be nested (arbitrarily deeply).

```
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
```

Here, `seq_len()` is very useful in loops. Another useful function is `seq_along()`.

```
x <- c(1, 50, 20, 12)
seq_len(length(x))
```

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

```
seq_along(x)
```

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

```
1:length(x)
```

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

```
# for a vector with length >= 1, the above three give the same result
# but for a vector of length 0, there is a difference:
x <- c()
seq_len(length(x))
```

```
## integer(0)
```

```
seq_along(x)
```

```
## integer(0)
```

```
1:length(x)    # returns a vector of length 2 in decreasing order
```

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

[Task 3: Grade Assignement]

Run the following code snippet to generate a sequence of random numbers.

- The function `runif(n, min = 0, max = 1)` draws random numbers from a uniform distribution (`?runif` to learn more).
- The function `round()` rounds the values in its first argument to the specified number of decimal places (`?round` to learn more).
- `set.seed(0)` is used here to ensure the same set of random numbers are generated no matter when and where you run this code (otherwise, `runif()` is likely to generate different random numbers across different calls).

```
set.seed(0)
scores <- round(runif(50, 45, 100), 1)
scores
```

```
##  [1] 94.3 59.6 65.5 76.5 95.0 56.1 94.4 97.0 81.3 79.6 48.4 56.3 54.7 82.8
66.1
## [16] 87.3 72.4 84.5 99.6 65.9 87.8 96.4 56.7 80.8 51.9 59.7 66.2 45.7 66.0
92.8
## [31] 63.7 71.5 78.0 72.1 55.2 90.5 81.8 88.7 50.9 84.8 67.6 90.2 80.6 88.1
75.4
## [46] 74.1 88.4 46.3 71.2 85.3
```

(a) Suppose that these numbers represent students' scores of a course. Write a **`for` loop** to convert these numbers to letter grades. The letter grade will be "A" if a score is at least 90. The letter grade will be "B" if a score is between 80 and 90. The letter grade will be "F" if a score is lower than "80". Create a variable called `grades` to store the result.

Tips:

- Create an empty character vector to fill things up later;
- A `for` loop is needed to traverse these 50 scores;
- The `if` statement may need to use its `else if` and `else` clause.

(b) Use `ifelse()` to do the same thing with a single line of code.

[End of Task 3]

3.3.2 `while`: Conditional Loops

`while` is used for **conditional loops** when we know the condition to stop the iterations.

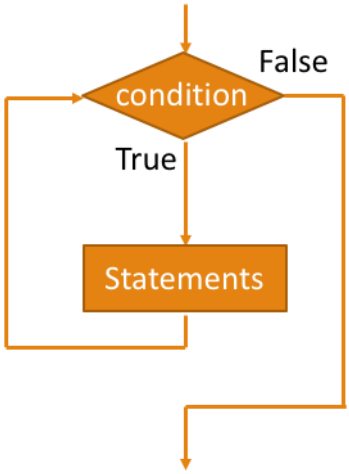

Fig. 4 Flowchart of while Loops

Syntax:

```
while (condition) {
  statements
}
```

- R will run the `statements` repeatedly while the `condition` is TRUE.
- In other words, R will run the `statements` repeatedly until the `condition` truns FALSE.

```
i <- 1
x <- c("a", "b", "c", "d")
while (i <= length(x)) {
  print(x[i])
  i <- i+1
}

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

Compared with `for` loops, `while` loops are more general, in that every `for` loops could be replaced with a `while` loop (but not vice versa).

[Task 4: Least Common Multiple]

The least common multiple of a set of integers is the smallest number that is divisible by these integers. For example, the least common multiple of 2, 3 and 4 is 12.

Suppose that an arbitraty number of values are stored in a vector named `find.lcm`. Use conditionals and/or loops to find the least common multiple for values in this vector.

For example, for `find.lcm <- c(4, 5, 7)`, this code should yield 140; for `find.lcm <- c(3, 5, 7, 23)`, this code should return 2415; and `find.lcm <- 2:14`, this code should yield 360360.

[End of Task 4]

3.3.3 `repeat`: Infinite Loops

Syntax:

```
repeat {
  statements
}
```

- `statements` will be executed repeatedly until we press the ESCAPE button or quit R.
- We can break out of a `repeat` loop by including a `break` statement via conditionals. Then `repeat` does the same thing as `while` does.
- Infinite loops should generally be *avoided*, even if they are theoretically correct.

```
i <- 1
x <- c("a", "b", "c", "d")
repeat {
  if (i > length(x))   # try: delete these two lines and run
    break
  print(x[i])
  i<-i+1
}
```

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

3.3.4 `next` and `break`: Control Loops Explicitly

- `next` skips the rest of the current iteration and starts the next iteration.

```
for(i in 1:8) {
  if(i <= 5) next    # skip the first 5 iterations
  print(i)
}
```

```
## [1] 6
## [1] 7
## [1] 8
```

```
print(paste("no. of iterations:", i))
```

```
## [1] "no. of iterations: 8"
```

- `break` terminates the loop immediately.

```
for(i in 1:8) {
  if(i >= 5) break   # jump out of the loop when i >= 5
  print(i)
}
```

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

```
print(paste("no. of iterations:", i))
```

```
## [1] "no. of iterations: 5"
```

3.3.5 Vectorized Operations

Vectorized operations are always prefered to explicit loops when implementing element-wise operations, and should be leveraged whenever possible.

```
a = 1:5
b = 11:15

# vectorization:
c <- a + b
c
```

```
## [1] 12 14 16 18 20
```

```
# explicit loop:
c <- vector("numeric", length(a))
for (i in seq_along(a)) c[i] <- a[i] + b[i]
c
```

```
## [1] 12 14 16 18 20
```

Benefits of vectorized operations:

- Clarity: the syntax is about what we're doing.
- Abstraction: the syntax hides how the computer does it.
- Concision: we write less.
- Generality: same syntax works for numbers, vectors, arrays, ...
- Speed: modifying big vectors over and over is slow in R; work gets done by optimized low-level code.

3.4 Summary

Control structures allow us to control the flow of an R program.

Common control structures in R:

- **Conditional Statements (Conditionals)**: execute different sets of statements, depending on whether a condition is true or false.
  - `if/else`: test a condition and act on it
  - `switch()`: select one from several alternatives
  - `ifelse()`: conditional element selection
- **Iterative Statements (Loops)**: execute a set of statements repeatedly.
  - `for`: execute a loop for a fixed number of times
  - `while`: repeat a loop while a condition is true
  - `repeat`: run an infinite loop (must break out of it to stop)
  - `next`: skip the current iteration of a loop;
  - `break`: break out of a loop

[Task 5: Groupwise data summarization]

We're going to look at a data set on 97 men who have prostate cancer (from the book The Elements of Statistical Learning).

Download the data **pros.dat** from Canvas into the directory where you put your in-class exercise R script.

In this data, there are 9 variables measured on these 97 men:

- `lpsa`: log PSA score
- `lcavol`: log cancer volume
- `lweight`: log prostate weight
- `age`: age of patient
- `lbph`: log of the amount of benign prostatic hyperplasia
- `svi`: seminal vesicle invasion
- `lcp`: log of capsular penetration
- `gleason`: Gleason score
- `pgg45`: percent of Gleason scores 4 or 5

Use the `read.table()` function to load the data into your R session, and store it as a data frame `pros.dat`.

Before reading the data, you may need to set your *working directory* to the directory where you put the data and in-class exercise R script.

```
pros.dat <- read.table("pros.dat")
str(pros.dat)
```

```
## 'data.frame':    97 obs. of  9 variables:
##  $ lcavol : num  -0.58 -0.994 -0.511 -1.204 0.751 ...
##  $ lweight: num  2.77 3.32 2.69 3.28 3.43 ...
##  $ age    : int  50 58 74 58 62 50 64 58 47 63 ...
##  $ lbph   : num  -1.39 -1.39 -1.39 -1.39 -1.39 ...
##  $ svi    : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ lcp    : num  -1.39 -1.39 -1.39 -1.39 -1.39 ...
##  $ gleason: int  6 6 7 6 6 6 6 6 6 6 ...
##  $ pgg45  : int  0 0 20 0 0 0 0 0 0 0 ...
##  $ lpsa   : num  -0.431 -0.163 -0.163 -0.163 0.372 ...
```

```
head(pros.dat)
```

```
##       lcavol  lweight age      lbph svi       lcp gleason pgg45       lpsa
## 1 -0.5798185 2.769459  50 -1.386294   0 -1.386294       6     0 -0.4307829
## 2 -0.9942523 3.319626  58 -1.386294   0 -1.386294       6     0 -0.1625189
## 3 -0.5108256 2.691243  74 -1.386294   0 -1.386294       7    20 -0.1625189
## 4 -1.2039728 3.282789  58 -1.386294   0 -1.386294       6     0 -0.1625189
## 5  0.7514161 3.432373  62 -1.386294   0 -1.386294       6     0  0.3715636
## 6 -1.0498221 3.228826  50 -1.386294   0 -1.386294       6     0  0.7654678
```

We randomly assign the 97 observations into 10 groups using the function `sample()`:

```
set.seed(0)
grouping <- sample(1:10, size = 97, replace = T)
grouping   # each number represents the group the corresponding observation
is assgined to
```

```
##  [1]  9  4  7  1  2  7  2  3  1  5  5 10  6 10  7  9  5  5  9  9  5  5  2
10  9
## [26]  1  4  3  6 10 10  6  4  4 10  9  7  6  9  8  9  7  8  6 10  7  3 10
6  8
## [51]  2  2  6  6  1  3  3  8  6  7  6  8  7  1  4  8  9  9  7  4  7  6  1
5  6
## [76]  1  9  7  7  3  6  2 10 10  7  3  2 10  1 10 10  8 10  5  7  8  5
```

`sample()` generates a sample of a fixed size (i.e., 97) from `1:10` by sampling with replacement. (`?sample` for more information).

(a) Write a `for` loop (possibly nested) to caclualate the groupwise sum of each column. Store the result in a **10 by 9** (10 groups and 9 columns) matrix called `groupwise.sum`. Your code should be as concise as possible.

Tips:

- You may need a nested loop with the outer loop iterating over columns and the inner loop iterating over groups;
- Use `grouping` with Boolen indexing to find observations assigned into one group;

- Use matrix indexing to fill results in `groupwise.sum`.

(b) Convert the `pros.dat` data frame to a matrix called `pros.mat` using the following code:

```
pros.mat <- as.matrix(pros.dat)
```

Then write a `for` loop to do the same thing. Now you only need a flat loop with `colSums()`.

(c) Figure out how to use a function named `rowsum()` with `pros.mat` to achieve the same goal.

[End of Task 5]