

# Loops and Functions



**Feng Li**

**feng.li@cufe.edu.cn**

**School of Statistics and Mathematics  
Central University of Finance and Economics**

# Today we are going to learn...

1 Control-flow constructs

2 Functions

3 Loops

## The if condition

- Comparison

```
x == y      all.equal()      identical()
x != y
x > y
x < y
x >= y
x <= y
%in%
```

- What would expect when  $x$  and  $y$  are vectors, matrices,...
- The if condition

```
if (condition)
{
    do something
}
else
{
    do something else
}
```

## Lab 1.3

- 1 February 29, known as a leap day in the calendar, is a date that occurs in most years that are evenly divisible by 4, such as 2004, 2008, 2012 and 2016. Years that are evenly divisible by 100 do not contain a leap day, with the exception of years that are evenly divisible by 400, which do contain a leap day; thus 1900 did not contain a leap day while 2000 did.
- 2 Write a function called `is.leapday` to check if a given year has February 29 [Hint: you may need `?%%.]`.
- 3 Test your function for some years.
- 4 What can you do to improve for the function in terms of error tolerance?
- 5 If I want to check which year has a leap day for a sequence of given years. Modify your function to implement it.

```
isLeapday <- function(year)
{
  mod4 <- year%%4
  mod100 <- year%%100
  mod400 <- year%%400

  LeapdayIndex = ((mod4 == 0 & mod100 != 0) | mod400 == 0)
  ## if((mod4 == 0 & mod100 != 0) | mod400 == 0)
  ## {
  ##   out <- TRUE
  ## }
  ## else
  ## {
  ##   out <- FALSE
  ## }
  ## return(out)
  return(LeapdayIndex)
}
```

```
whichLeapday <- function(year)
{
  if(!is.numeric(year))
  {
    stop("You must specify a numerical input.")
  }

  mod4 <- year%%4
  mod100 <- year%%100
  mod400 <- year%%400

  leapdayIndex <- ((mod4 == 0 & mod100 != 0) | mod400 == 0)

  out <- year[leapdayIndex]

  return(out)
}
```

# Functions

- Create a function object

```
myFun = function (par)
{
  out = max(par1) - min(par2)
  return(out)
}
```

- Load the function: `source()`
- Execute your function

# Functions

- The input parameters type
- Validate the inputs
- Error catching
- Return types



## Lab 1.1

- ① Write a function `mySummary` where the input argument is `x` can be any vector and the output should contain the basic summary (mean, variance, length, max and minimum values, type) of the vector you have supplied to the function.
- ② Test your function with some vectors (that you make up by yourself).
- ③ What will happen if your input is not a vector (e.g. a data frame `weekPlanNew`) in our previous example?

## Lab 1.2

- ❶ The roots for the quadratic equation  $ax^2 + bx + c = 0$  are of the form

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

- ❷ Write a function named `quaroot` to solve the roots of given quadratic equation with `a`, `b`, `c`, as input arguments. [Hint: you may need the `sqrt()` function]
- ❸ Test your function on the following equations

$$x^2 + 4x - 1 = 0$$

$$-2x^2 + 2x = 0$$

$$3x^2 - 9x + 1 = 0$$

$$x^2 - 4 = 0$$

- ❹ Test your function with the equation  $5x^2 + 2x + 1 = 0$ . What are the results? Why? [Hint: check  $b^2 - 4ac$ ]
- ❺ Modify your function and return NA if  $b^2 - 4ac < 0$ .

```
quaroot <- function(a, b, c)
{
  x1 <- (-b+sqrt(b^2-4*a*c))/(2*a)
  x2 <- (-b-sqrt(b^2-4*a*c))/(2*a)

  out <- c(x1, x2)
  return(out)
}
```

```
quaroot <- function(a, b, c)
{
  d <- b^2-4*a*c

  if(d<0)
  {
    x1 <- NA
    x2 <- NA
  }
  else
  {
    x1 <- (-b+sqrt(d))/(2*a)
    x2 <- (-b-sqrt(d))/(2*a)
  }

  out <- c(x1, x2)
  return(out)
}
```

# The for loop

```
B = matrix(1:10,2,5)
C = matrix(100:109,2,5)
A = matrix(NA,2,5)
for(i in 1:n)
{
  A[i] = B[i] + C[i]
}
```

# The while loop

```
B = matrix(1:10,2,5)
C = matrix(100:109,2,5)
A = matrix(NA,2,5)
```

```
i = 0
while(i != 10)
{
  i = i + 1
  A[i] = B[i] + C[i]
}
```

## `apply()` type loops I

- Calculate row sums for a matrix with a loop.
- Apply `sum()` function to each row of the matrix.
- `apply()` to an array with higher dimension.
- Apply your own function to each row of the matrix.
- `lapply()` Apply a function to a list
- `mapply()` Apply a function to multiple list or vector arguments.
- The ... arguments in a function.
- Supply more arguments to `apply()` type functions
- The advantage of `()apply`.
  - Easy construct
  - Less coding
  - `()apply` type loops is essentially a more efficient version loop.

# Write efficient loops

- Avoid loops as much as possible.
- Use `()apply` type loop if possible.
- Think a lot about under- and over-flow
- Allocate the memory space before looping. This is a much slower loop.

```
B = matrix(1:10,2,5)
C = matrix(100:109,2,5)
A = NULL
for(i in 1:n)
{
  A[i] = B[i] + C[i]
}
```



## Suggested reading

- R-intro (2015): **Chapter 9, 10**
- Jones (2009): **Chapter 6.4**