R Syntax 2: Conditions, Loops, & Functions

Pilsung Kang
School of Industrial Management Engineering
Korea University

AGENDA

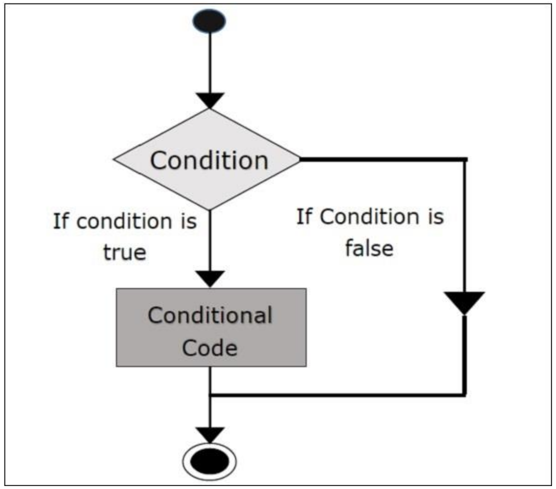
Conditions and Loops

- Understanding conditions and loops are necessary for efficient data analysis
  - ✓ Example of conditions
    - Want to remove instances whose value is greater than 3 standard deviations
    - Want to remove variables with zero variance
    - Want to replace NULL with a constant value
  - ✓ Example of loops
    - Want to make a histogram for each variable in a dataframe
    - Want to compare various machine learning algorithms for the same dataset

Conditions

- if-else condition

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



✓ condition can be a simple logical comparison to a complex function

✓ statement 1: run if the condition is met

✓ statement 2: run if the condition is not met

Conditions

- Condition example 1

```
# Conditions
r <- 1
if (r==4) {
    print("The value of r is 4”)
} else {
    print(“The value of r is not 4")
}
```

✓ Condition: a simple comparison (ask whether r is 4 or not)

✓ Output: a simple statement (print a sentence)

Conditions

- Condition example 1: Caution!

```
# Caution!
r <- 4
if (r==4) {
    print("The value of r is 4")
}
else {
    print("The value of r is not 4")
}
```

✓ must be stated after the right curly bracket in same line

✓ The above code return the error message

```
> # Caution!
> r <- 4
> if (r==4) {
+   print("The valus of r is 4")
+ }
[1] "The valus of r is 4"
> else {
Error: unexpected 'else' in "else"
>   print("The valus of r is not 4")
```

Conditions

- Condition example 2

```r
# Computations are possible in the statements
r <- 3
if (r < 5) {
    cat("The value of squared r is”, r^2)
} else {
    cat(“The value of squared root of r is", sqrt(r))
}
```

✓ Condition: a simple comparison (ask whether r is smaller than 5)

✓ Output: computation result

  - If the condition is met (r is smaller than 5), return the square value of r
  - If the condition is not met, return the squared root of r

Conditions

- Condition example 3

```
# the results of functions can be a condition
carbon <- c(10, 12, 15, 19, 20)
mean(carbon)
median(carbon)

if (mean(carbon) > median(carbon)) {
     print ("Mean > Median")
} else {
     print ("Median <= Mean")
}
```

✓ Condition can be a result of function

✓ In this example, mean of carbon (15.2) is greater than the median of carbon (15)

✓ Hence, the first statement will be printed

Conditions

- Condition example 4: Simple Form

```r
# Simple form
x <- 1
if(x > 0) print("Non-negative number") else print("Negative number")
```

✓ If the statements are simple, the if conditions can be written in one line without curly brackets

Conditions

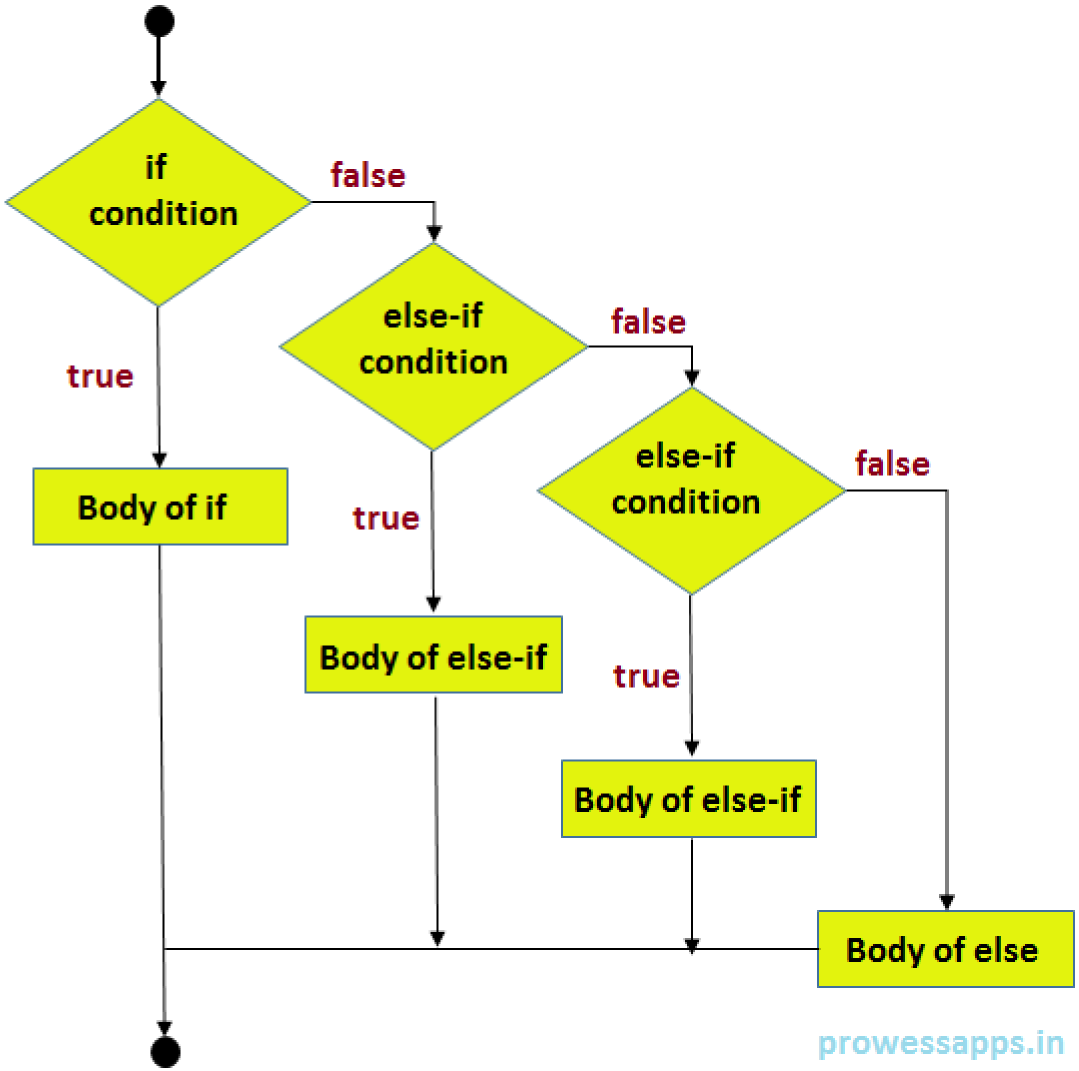
- Condition example 5

```r
# variable initialization with if statement
x <- -2
y <- if(x > 0) 1 else -1
y
```

  ✓ The value of y is initialized by if condition

Conditions

- Condition example 6: if-else ladder



Conditions

- Condition example 6: if-else ladder

```r
# if-else ladder
x <- 0
if (x < 0) {
    print("Negative number")
} else if (x > 0) {
    print("Positive number")
} else print("Zero")
```

Conditions

- Condition example 7: Price calculation
  - ✓ Assume that the tax ratio is different according to the product category

Categories	Products	VAT
A	Book, magazine, newspaper, etc..	8%
B	Vegetable, meat, beverage, etc..	10%
C	Tee-shirt, jean, pant, etc..	20%

```
# Product price calculator w.r.t different category
category <- 'A'
price <- 10 if (category =='A'){
    cat('A vat rate of 8% is applied.','The total price is', price*1.08)
} else if (category =='B'){
    cat('A vat rate of 10% is applied.','The total price is', price*1.10)
} else {
    cat('A vat rate of 20% is applied.','The total price is', price*1.20)
}
```

Conditions

- ifelse: a vectorized condition

  ifelse (condition, statement 1, statement 2)

  ✓ condition: Boolean vector

  ✓ statement 1: run if the condition is met

  ✓ statement 2: run if the condition is not met

```
> x <- 1:10
> y <- ifelse(x%%2 == 0, "even", "odd")
> y
 [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even"
```

Conditions

- Condition example 8: if-else statement

  ifelse (condition, statement 1, statement 2)

  ✓ condition: Boolean vector

  ✓ statement 1: run if the condition is met

  ✓ statement 2: run if the condition is not met

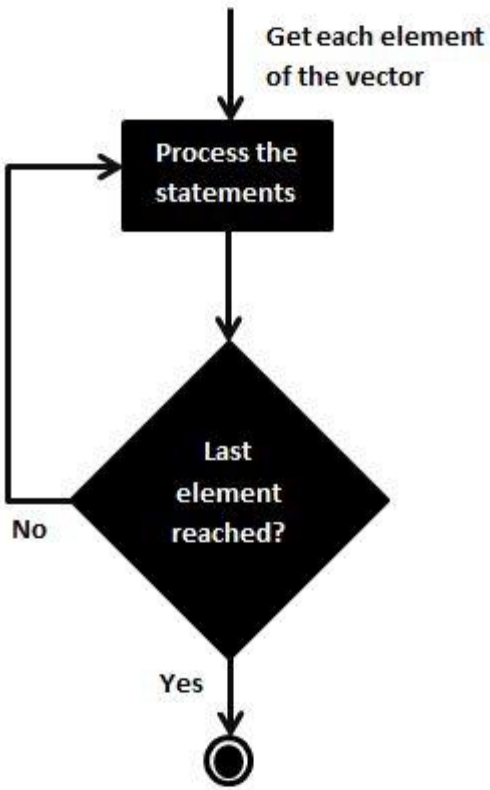
```
# ifelse example
x <- 1:10
y <- ifelse(x%%2 == 0, "even", "odd")
y
```

AGENDA

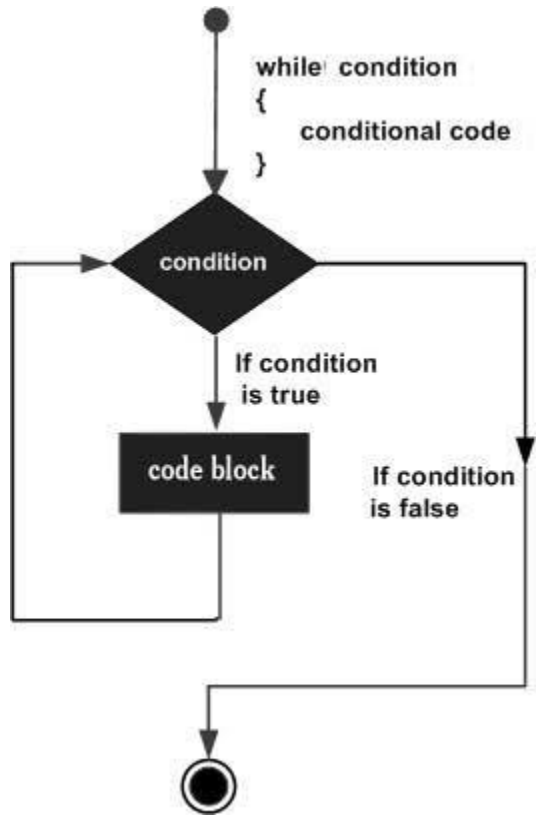
Three Types of Loops

- For loop, While loop, and Repeat-Break loop

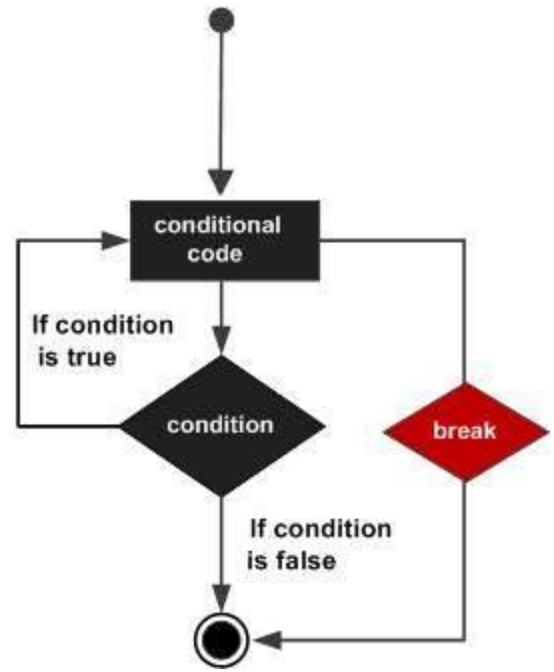
For loop



While loop



Repeat-Break loop



Loops: for

- for loop

```
for (i in x) {
statement
}
```

✓ i: index of loop

✓ x: a set of element for which the loop runs

✓ statement: running part

Loops: for

- for loop example 1

```
# Loop: for statement
n <- c(1:10)
for (i in n) {
    print(i^2)
}
```

✓ Take the integer values from 1 to 10 step by 1

✓ Print the square value of it

```
> for (i in n) {
+   print(i^2)
+ }
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
[1] 49
[1] 64
[1] 81
[1] 100
```

Loops: for

- for loop example 2: for loop with an if statement inside

```
# For loop with an if statement inside
n <- c(1:10)
for (i in n){
     if (i %% 2 == 0) {
          cat(i, "is an even number \n")
     } else {
          cat(i, "is an odd number \n")
     }
}
```

✓ Take the numbers from 1 to 10

- If the number is divided by 2, then print the first statement
- Otherwise, print the second statement

```
1 is an odd number
2 is an even number
3 is an odd number
4 is an even number
5 is an odd number
6 is an even number
7 is an odd number
8 is an even number
9 is an odd number
10 is an even number
```

Loops: for

- for loop example 3: multiple for loops

```
# Multiple for loops
mat <- matrix(data = seq(11, 20, by=1), nrow = 5, ncol =2)
mat
# Create the loop with r and c to iterate over the matrix
for (r in 1:nrow(mat)){
     for (c in 1:ncol(mat)){
     cat("The square of row", r, "and column", c, "is", mat[r,c]^2), "\n")
     }
}
```

```
     [,1] [,2]
[1,]   11   16
[2,]   12   17
[3,]   13   18
[4,]   14   19
[5,]   15   20
```

```
The square of row 1 and column 1 is 121
The square of row 1 and column 2 is 256
The square of row 2 and column 1 is 144
The square of row 2 and column 2 is 289
The square of row 3 and column 1 is 169
The square of row 3 and column 2 is 324
The square of row 4 and column 1 is 196
The square of row 4 and column 2 is 361
The square of row 5 and column 1 is 225
The square of row 5 and column 2 is 400
```

Loops: while

- while loop

```
while (condition) {
statement
}
```

✓ run the statement until the condition is not met

Loops: while

- while loop example 1

```
# While loop
i <- 1
while (i <= 10) {
     i <- i+4
     print(i)
}
```

  ✓ Initialize the variable to 1

  ✓ if i is smaller than or equal to 10, run the statement

```
> i <- 1
> while (i <= 10) {
+   i <- i+4
+   print(i)
+ }
[1] 5
[1] 9
[1] 13
```

Loops: while

- while loop example 2

```
# While loop example 2
# Set variable price
price <- 100
# Loop variable counts the number of loops
loop <- 1
# Set the while statement
while (price > 95){
    # Add a random variation between -10 and 10 to the current price
    price <- price + sample(-10:10, 1)
    # Print the number of loop and price
    cat("The", loop, "-th price is", price, "\n")
    # Count the number of loop
    loop = loop +1
}
```

✓ Initialize the variable price to 100

✓ If the price is greater than 95, add a random variation between -10 and 10 to the current price

✓ It can fall into an infinite loop (loop that never ends)

Loops: while

- while loop example 2

1st trial

```
The 1 -th price is 94
>
```

2nd trial

```
The 2086 -th price is 125
The 2087 -th price is 119
The 2088 -th price is 125
The 2089 -th price is 120
The 2090 -th price is 125
The 2091 -th price is 115
The 2092 -th price is 121
The 2093 -th price is 128
The 2094 -th price is 124
The 2095 -th price is 118
The 2096 -th price is 119
The 2097 -th price is 114
The 2098 -th price is 109
The 2099 -th price is 105
The 2100 -th price is 99
The 2101 -th price is 98
The 2102 -th price is 106
The 2103 -th price is 108
The 2104 -th price is 100
The 2105 -th price is 110
The 2106 -th price is 100
The 2107 -th price is 92
```

3rd trial

```
The 1 -th price is 101
The 2 -th price is 91
>
```

4th trial

```
The 41 -th price is 142
The 42 -th price is 143
The 43 -th price is 148
The 44 -th price is 144
The 45 -th price is 145
The 46 -th price is 140
The 47 -th price is 136
The 48 -th price is 127
The 49 -th price is 131
The 50 -th price is 124
The 51 -th price is 119
The 52 -th price is 115
The 53 -th price is 107
The 54 -th price is 113
The 55 -th price is 119
The 56 -th price is 112
The 57 -th price is 107
The 58 -th price is 98
The 59 -th price is 104
The 60 -th price is 101
The 61 -th price is 100
The 62 -th price is 94
```

Loops: repeat-break

- repeat-break loop

  ```
  repeat {
  statement
  condition break
  }
  ```

  ✓ run the statement first, check the condition, stop if the condition is met

Loops: repeat-break

- repeat-break example 1

```r
# repeat-break example 1
i <- 1
repeat {
    i <- i+4
    print(i)
    if (i > 10) break
}
```

✓ The result is the same as that of the while example 1

```
> repeat {
+   i <- i+4
+   print(i)
+   if (i > 10) break
+ }
[1] 5
[1] 9
[1] 13
```

Loops: repeat-break

- repeat-break example 2: Infinite loop prevention

```r
# repeat-break example 2: Infinite loop prevention
price <- 100
loop = 1
repeat{
    # Add a random variation between -10 and 10 to the current price
    price <- price + sample(-10:10, 1)
    # Print the number of loop and price
    cat("The", loop, "-th price is", price, "\n")
    # Count the number of loop
    loop = loop +1
    # Stop the loop if price > 10 or loop > 10
    if (price > 95 | loop > 10) break
}
```

1st trial

```
The 1 -th price is 92
The 2 -th price is 89
The 3 -th price is 80
The 4 -th price is 79
The 5 -th price is 77
The 6 -th price is 78
The 7 -th price is 83
The 8 -th price is 74
The 9 -th price is 66
The 10 -th price is 61
```

2nd trial

```
The 1 -th price is 109
```

3rd trial

```
The 1 -th price is 94
The 2 -th price is 99
```

AGENDA

Function

- Why functions?
- An incidental advantage of putting code into functions is that the workspace is not then cluttered with objects that are local to the function

```
all()        # returns TRUE if all values are TRUE
any()        # returns TRUE if any values are TRUE
args()       # information on the arguments to a function
cat()        # prints multiple objects, one after the other
cumprod()    # cumulative product
cumsum()     # cumulative sum
diff()       # form vector of first differences
             # N. B. diff(x) has one less element than x
history()    # displays previous commands used
is.factor()  # returns TRUE if the argument is a factor
is.na()      # returns TRUE if the argument is an NA
             # NB also is.logical(), is.matrix(), etc.
length()     # number of elements in a vector or of a list
ls()         # list names of objects in the workspace
```

Function

- Why functions?
- An incidental advantage of putting code into functions is that the workspace is not then cluttered with objects that are local to the function

```
mean()      # mean of the elements of a vector
median()    # median of the elements of a vector
order()     # x[order(x)] sorts x (by default, NAs are last)
print()     # prints a single R object
range()     # minimum and maximum value elements of vector
sort()      # sort elements into order, by default omitting NAs
rev()       # reverse the order of vector elements
str()       # information on an R object
unique()    # form the vector of distinct values
which()     # locates 'TRUE' indices of logical vectors
which.max() # locates (first) maximum of a numeric vector
which.min() # locates (first) minimum of a numeric vector
with()      # do computation using columns of specified data frame
```

Function

- Writing a function

```
function_name <- function(arguments) {
statement 1
statement 2
…
return(object)
}
```

✓ function_name: name that the function is referred to

✓ arguments: inputs that a user should provide to run the function

✓ statements: operations running inside the function

✓ object: function output

Function

- Same operations but different outputs

```
# Same operation but different outputs
distance <- c(148, 182, 173, 166, 109, 141, 166)
mean_and_sd1 <- function(x) {
     avg <- mean(x)
     sdev <- sd(x)
     return(c(mean=avg, SD=sdev))
}
mean_and_sd1(distance)

mean_and_sd2 <- function(x) {
     avg <- mean(x)
     sdev <- sd(x)
     c(mean=avg, SD=sdev)
     return(avg)
}
mean_and_sd2(distance)
```

✓ Both functions take a vector and compute its mean and standard deviation

- First function returns both mean and standard deviation
- Second function only returns the mean

Function

- Function output with return( ) instruction

```
# Return the result with return()
oddcount <- function(x) {
     k <- 0
     print("odd number calculator")
     for (n in 1:x) {
          if (n %% 2 == 1) {
               cat(n, "is an odd number. \n")
               k <- k+1
          }
     }
     return(k)
}
oddcount(10)
```

```
> oddcount(10)
[1] "odd number calculator"
1 is an odd number.
3 is an odd number.
5 is an odd number.
7 is an odd number.
9 is an odd number.
[1] 5
```

Function

- Function output without return( ) instruction but explicitly designate the object

```
# Return the result without return() but explicitly designate the object
oddcount <- function(x) {
     k <- 0
     print("odd number calculator")
     for (n in 1:x) {
          if (n %% 2 == 1) {
               cat(n, "is an odd number. \n")
               k <- k+1
          }
     }
     k
}
oddcount(10)
```

```
> oddcount(10)
[1] "odd number calculator"
1 is an odd number.
3 is an odd number.
5 is an odd number.
7 is an odd number.
9 is an odd number.
[1] 5
```

If return( ) is not used,
the final object inside the function is returned
(not recommended)

Function

- Function output without return( ) instruction and object designation

```r
# Return the result without return() and explicit designation
oddcount <- function(x) {
     k <- 0
     print("odd number calculator")
     for (n in 1:x) {
          if (n %% 2 == 1) {
               cat(n, "is an odd number. \n")
               k <- k+1
          }
     }
}
oddcount(10)
```

```
> oddcount(10)
[1] "odd number calculator"
1 is an odd number.
3 is an odd number.
5 is an odd number.
7 is an odd number.
9 is an odd number.
```

This function returns nothing
because the condition for the last if statement (when n == 10) is not true

Function

- Function arguments: default arguments

```
mean_and_sd3 <- function(x = rnorm(10)) {
     avg <- mean(x)
     sdev <- sd(x)
     return(c(mean=avg, SD=sdev))
}

mean_and_sd3(distance)
mean_and_sd3()
```

✓ If the argument is provided by a user, function statements run with the provided argument

```
> mean_and_sd3(distance)
     mean        SD
155.00000  24.68468
```

✓ If the argument is not provided, default function argument is activated

```
> mean_and_sd3()
      mean         SD
-0.1220926  0.7960788
```

Function

- Function arguments
  - ✓ Each argument has its own name
  - ✓ Name is used to access the corresponding argument within function
  - ✓ Three possible ways to assign the argument
    - Exact name
    - Partially matching names (not recommended)
    - Argument order

```
> addTheLog <- function(first, second) {first + log(second)}
> addTheLog(second=exp(4),first=1)
[1] 5
> addTheLog(s=exp(4),first=1)
[1] 5
> addTheLog(1,exp(4))
[1] 5
```

Function

- Function example 1
  - ✓ Question: from a vector consisting of only 0 and 1, return the indices from which 1 repeatedly appears k times

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	1	1	1	0	0	1	0	1	1	0	1	1	1

- If k = 2, the answer is (1,2,8,11,12)

1	1	1	0	0	1	0	1	1	0	1	1	1

1	1	1	0	0	1	0	1	1	0	1	1	1

1	1	1	0	0	1	0	1	1	0	1	1	1

1	1	1	0	0	1	0	1	1	0	1	1	1

1	1	1	0	0	1	0	1	1	0	1	1	1

Function

- Function example 1
  - ✓ Question: from a vector consisting of only 0 and 1, return the indices from which 1 repeatedly appears k times

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	1	1	1	0	0	1	0	1	1	0	1	1	1

    - If k = 3, the answer is (1,11)

1	**1**	**1**	0	0	1	0	1	1	0	1	1	1

1	1	1	0	0	1	0	1	1	0	**1**	**1**	**1**

    - If = 4, the answer is NULL

Function

- Function example 1

```r
# Function example 1
findrepeats <- function(x, k) {
    n <- length(x)
    repeats <- NULL
    for (i in 1:(n-k+1)) {
        if(all(x[i:(i+k-1)] == 1)) repeats <- c(repeats, i)
    }
    return(repeats)
}
```

✓ This function takes two arguments: x (target vector) and k (number of repeats)

✓ We need to determine the search candidates

- Since we have to check k consecutive numbers, the starting index begins with 1 and ends with (n-k+1)

Starting indices when k = 2

1	1	1	0	0	1	0	1	1	0	1	1	1

Starting indices when k = 3

Function

- Function example 1

```r
# Function example 1
findrepeats <- function(x, k) {
     n <- length(x)
     repeats <- NULL
     for (i in 1:(n-k+1)) {
          if(all(x[i:(i+k-1)] == 1)) repeats <- c(repeats, i)
     }
     return(repeats)
}
```

✓ If statement:

  - if all k consecutive values starting from i^{th} value are 1
  - add the starting index to the variable `repeats`

✓ Example A: i = 2, k= 3 (condition is not satisfied)

1	1	1	0	0	1	0	1	1	0	1	1	1

✓ Example B: i=11, k=3 (condition is satisfied)

1	1	1	0	0	1	0	1	1	0	1	1	1

Function

- Function example 2: Kendall's tau
  - ✓ Raw data: temperature and pressure recorded every hour

Time	10:00	11:00	12:00	13:00	14:00
Temperature	10	15	13	17	20
Pressure	900	920	890	940	920

  - ✓ What to do
    - Determine whether each indicator increases or decreases
    - Return the proportion of the events in which the change directions of the two indicators are the same

Function

- Function example 2: Kendall's tau

```
# Example 2: Kendall's tau
findud <- function(v) {
    vud <- v[-1] - v[-length(v)]
    return(ifelse(vud >0, 1, -1))
}
```

✓ Inner function: determine whether the variable is increased or decreased

- For temperature

Temperature	10	15	13	17	20

`v[-1]`	15	13	17	20
`v[-length(v)]`	10	15	13	17
`vud`	5	-2	4	3
`return(ifelse(vud >0, 1, -1))`	1	-1	1	1

Function

- Function example 2: Kendall's tau

```
# Example 2: Kendall's tau
findud <- function(v) {
    vud <- v[-1] - v[-length(v)]
    return(ifelse(vud >0, 1, -1))
}
```

✓ Inner function: determine whether the variable is increased or decreased

- For pressure

Pressure	900	920	890	940	920

`v[-1]`	920	890	940	920
`v[-length(v)]`	900	920	890	940
`vud`	20	-30	50	-20
`return(ifelse(vud >0, 1, -1))`	1	-1	1	-1

Function

- Function example 2: Kendall's tau

```
udcorr <- function(x,y) {
    ud <- lapply(list(x,y), findud)
    return(mean(ud[[1]] == ud[[2]]))
}

temp <- c(10, 15, 13, 17, 20)
pressure <- c(900, 920, 890, 940, 920)
udcorr(temp,pressure)
```

`ud[[1]]`	1	-1	1	1
`ud[[2]]`	1	-1	1	-1
`ud[[1]] == ud[[2]]`	1	1	1	0

- The final output = 0.75 (3/4)


ANY
questions?