데이터 처리 및 실습

2022년 1학기

담당교수: 서윤암

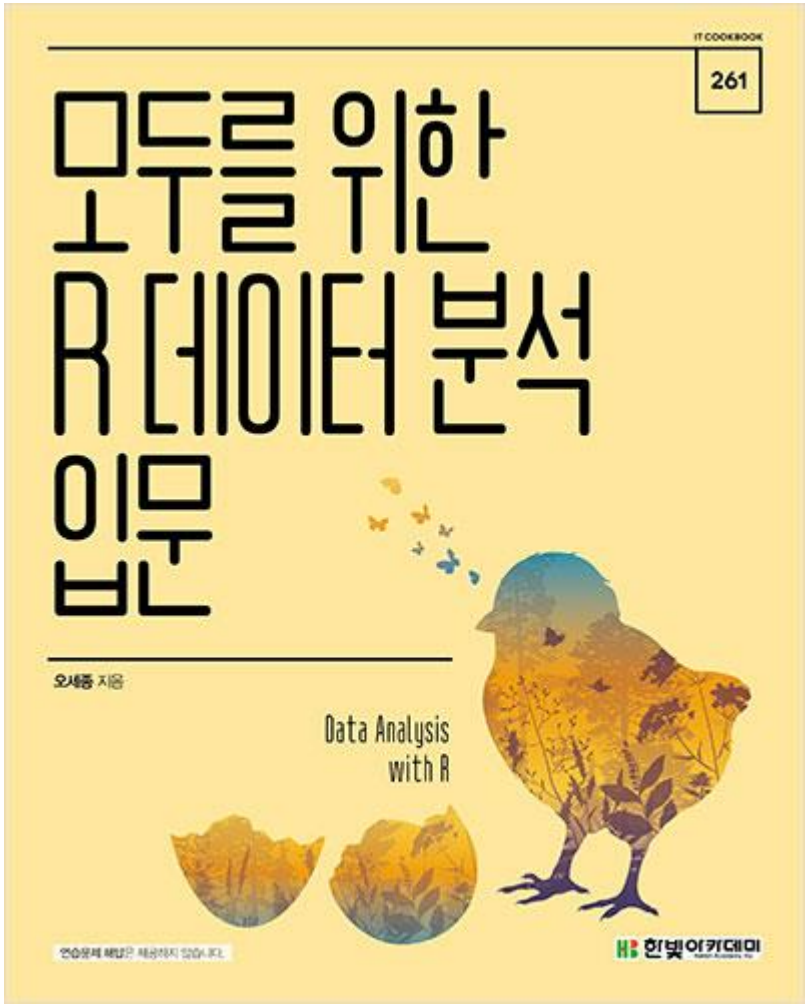
연구실: 자연과학대학 2호관 407

E-mail: seoya@jejunu.ac.kr

▪교재 예제 실습 파일

: 교재에서 사용된 예제 실습 코드는 아래에서 받으실 수 있습니다.

http://www.hanbit.co.kr/src/4459



Chapter 04
Conditional, Repetitive statement and function

Contents

01. Conditional statement

02. Repetitive statement

03. apply()

04. User-defined function (UDF)

05. Finding the location of the data that satisfies the conditions.

Logical operators

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y
|	or	x | y
&	and	x & y

01. Conditional statement

Conditional statements are programming language commands for handling decisions. Specifically, conditionals perform different computations or actions depending on whether a programmer-defined boolean *condition* evaluates to true or false.

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written with the **if** keyword, and it is used to specify a block of code to be executed if a condition is **TRUE**.

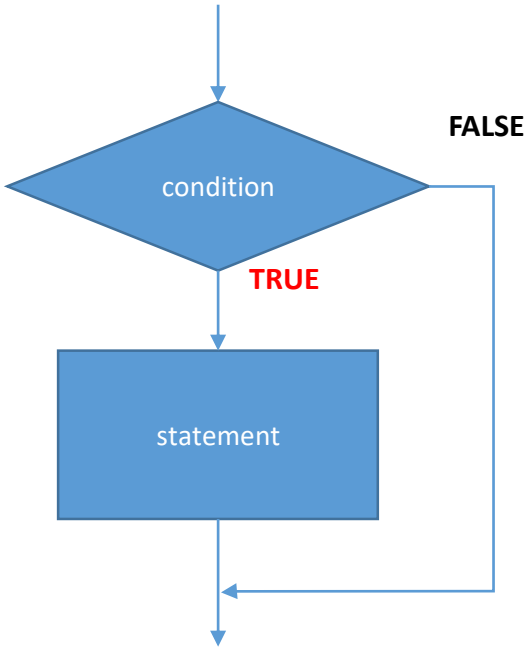
R uses **curly brackets { }** to define the scope in the code.

The syntax of the if statement is:

```
if( condition) {
    statement
}
```

e.g.

```
x <- 5
if(x > 0){
  print("Positive number")
}
```

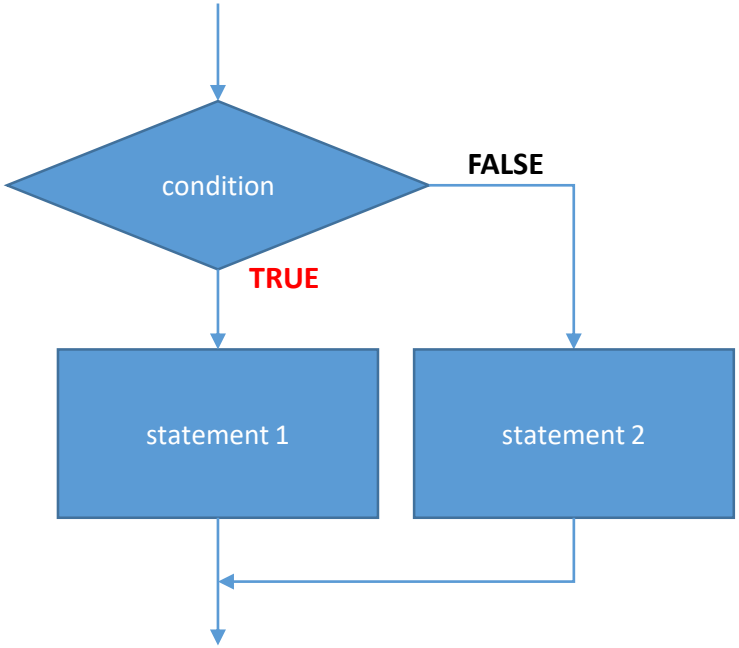


e.g.

```
a <- 33
b <- 200
if (b > a) {
  print("b is greater than a")
}
```

The syntax of **if...else** statement.

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



e.g.

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

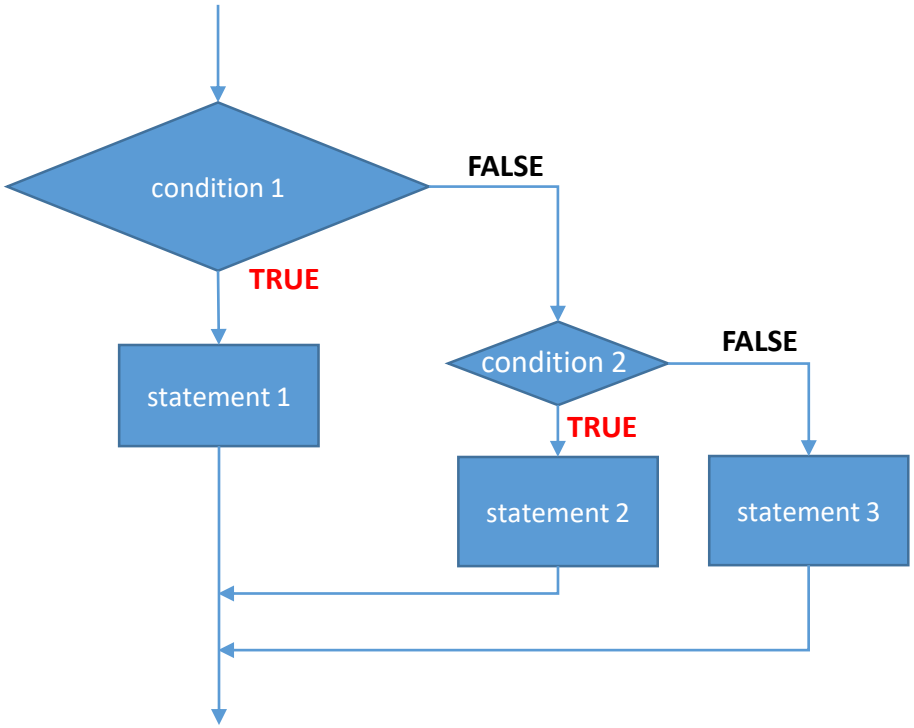
The above conditional can also be written in a single line as follows.

e.g.

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

The **else if** the keyword is R's way of saying "if the previous conditions were not true, then try this condition".

```
if(condition1){
  statement 1
} else if(condition2){
  statement 2
} else{
  statement 3
}
```



Only one statement will get executed depending upon the condition.

e.g.

```
a<- 5
if( a>2){
   print("a")
 }else if(a==5){
   print("c")
 }else{
   print("b")
 }
```

e.g.

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

The **else** keyword catches anything which isn't caught by the preceding conditions.

Nested If Statements

You can also have **if** statements inside **if** statements, this is called nested if statements.

e.g.

```
x <- 41
if (x > 10) {
  print("Above ten")
  if (x > 20) {
    print("and also above 20!")
  } else {
    print("but not above 20.")
  }
} else {
  print("below 10.")
}
```

AND

The **&** symbol (and) is a logical operator and is used to combine conditional statements.

```
e.g.

a <- 200
b <- 33
c <- 500
if (a > b & c > a){
  print("Both conditions are true")
}
```

OR

The | symbol (or) is a logical operator and is used to combine conditional statements.

```
e.g.

a <- 200
b <- 33
c <- 500
if (a > b | a > c){
  print("At least one of the conditions is true")
}
```

code 4-1

```
job.type <- 'A’
if(job.type == 'B') {
        bonus <- 200
} else {
        bonus <- 100
}
print(bonus)
```

code 4-2

```
job.type <- 'B’
bonus <- 100
if(job.type == 'A') {
        bonus <- 200
}
print(bonus)
```

code 4-3

```
score <- 85

if (score > 90)
        { grade <- 'A’
} else if (score > 80) {
        grade <- 'B’
} else if (score > 70) {
        grade <- 'C’
} else if (score > 60) {
        grade <- 'D’
} else {
        grade <- 'F’
}

print(grade)
```

code 4-4

```
a <- 10
b <- 20
if(a>5 & b>5) {          # and
 print (a+b)
}
if(a>5 | b>30) {         # or
 print (a*b)
}
```

code 4-5

```
a <- 10; b <- 20
if (a>b) {
 c <- a
} else {
 c <- b
}
print(c)

a <- 10 ; b <- 20
c <- ifelse(a>b, a, b)
print(c)
```

02. Repetitive statement

Loops are used to repeat a specific block of code.

For Loops

A **for** loop is used for iterating over a sequence.

Structure of the **for loop**:

```
for(i in vector_expression){
   action_command
}
```

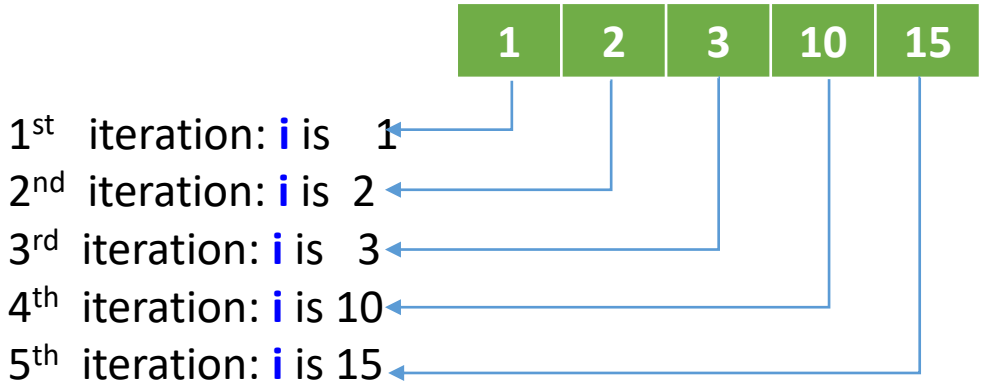
3 main elements:

1. **i** is the loop variable: it is updated at each iteration.
2. **vector_expression**: value attributed to **i** at each iteration (the number of iterations is the **length of vector_expression**).
3. **action_command**: what action to take at each iteration.

Note the usage of **curly brackets {}** to start and end the loop!

As many iterations as elements in the vector:

```
for(i in c(1, 2, 3, 10, 15){
   print(i)
}
```



e.g.

```
for(i in 2:5){
   # prints the value of i at each iteration
   print(i)
   # multiplies i by 2 at each iteration
   y <- i*2
   # prints the value of y at each iteration
   print(y)
}
```

Example of a **for loop** that iterates over a character vector:

e.g.

```
# Character vector
myfruits <- c("apple", "pear", "grape")

# The for loop prints the current element j, and the number of characters of element j

for(j in myfruits){
   print(j)
   print(nchar(j)) # nchar(): Count the Number of Characters (or Bytes or Width)

}
```

code 4-6

```
for(i in 1:5) {
  print('*')
}
```

code 4-6

```
for(i in 1:5) {
  print('*')
}
```

code 4-8 multiplication table

```
for(i in 1:9) {
  cat('2 *', i,'=', 2*i,'\n')          #cat() : Concatenate and Print
}
```

Example of a **for loop** that iterates over each row of a matrix, and prints the minimum value of that row :

e.g.

```
# Matrix of 50 rows, 16 columns
mymat <- matrix(rnorm(800), nrow=50)
# The rnorm in R is a built-in function that generates a vector of normally distributed random numbers.
# For loop over mymat rows
   # 1:nrow(mymat): ranges from 1 to the number of rows in the matrix: 1, 2, 3, 4, ...., 48, 49, 50

for(i in 1:nrow(mymat)){
   # prints the value of i at each iteration
   print(i)
   # prints the minimum value of the ith row of mymat at each iteration
   print(min(mymat[i,]))
}
```

If statement in **For loop**:

You can combine **for loops** and **if statements**:

e.g.

```
# Matrix
mymat <- matrix(rnorm(800),
       nrow=50)

# Loop over rows of mymat and print row if its median value is > 0
for(i in 1:nrow(mymat)){
   # extract the current row
   rowi <- mymat[i,]
   # if median of row is > 0, print row
   if(median(rowi) > 0){
      print(i)
      print(rowi)
   }
}
```

code 4-9 Print only even numbers from 1 to 20.

```
for(i in 1:20) {
  if(i%%2==0) {
    print(i)
  }
}
```

code 4-10 Print the sum from 1 to 100.

```
sum <- 0
for(i in 1:100) {
  sum <- sum + i
}
print(sum)
```

code 4-11

Classification according to petal length in iris data.
Petal.length <= 1.6 ‘L’, Petal.length >= 5.1 ‘H’ , others are ‘M’

```
norow <- nrow(iris)
mylabel <- c( )
for(i in 1:norow) {
        if (iris$Petal.Length[i] <= 1.6) {
                mylabel[i] <- 'L’
        } else if (iris$Petal.Length[i] >= 5.1) {
                mylabel[i] <- 'H’
        } else {
                mylabel[i] <- 'M’
        }
}
print(mylabel)
newds <- data.frame(iris$Petal.Length, mylabel)
head(newds)
```

Nested Loops

You can also have a loop inside of a loop.

e.g. # Print the adjective of each fruit in a list.

```
adj <- c("red","big","tasty")
fruits <- c("apple","banana","cherry")
for(i in adj){
  for(j in fruits){
    print(paste(i, j))    # paste() : Concatenate vectors after converting to the character.
  }
}
```

Q) Complete the multiplication table.

Break

With the **break** statement, we can stop the loop before it has looped through all the items.

```
e.g. # Stop the loop at “cherry”

fruits <- list(“apple”, "banana”, ”cherry”)
for (x in fruits) {
  if (x == "cherry") break
  print(x)
}
```

Next

With the **next** statement, we can skip an iteration without terminating the loop.

```
e.g. # Skip "banana"
fruits <- list("apple", "banana", "cherry")
for (x in fruits) {
  if (x == "banana")  next
  print(x)
}
```

When the loop passes "banana", it will skip it and continue to loop.

code 4-13

```
sum <- 0
for(i in 1:10) {
 sum <- sum + i
 if (i>=5) break
}
sum
```

code 4-14

```
sum <- 0
for(i in 1:10) {
 if (i%%2==0) next
 sum <- sum + i
}
sum
```

while Loops

With the **while** loop, we can execute a set of statements as long as a condition is TRUE

```
while (condition) {
 action_command
}
```

```
e.g. # Print i as long as i is less than 6
i <- 1
while (i < 6) {
  print(i)
  i <- i + 1
}
```

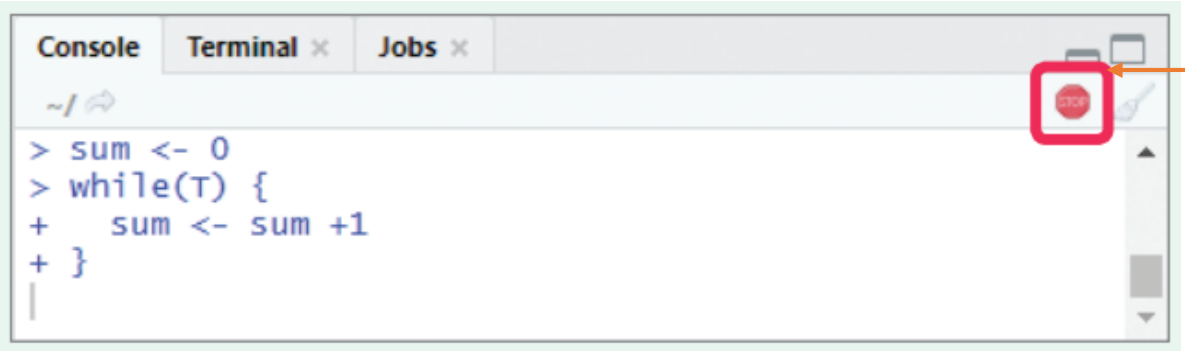
In the example above, the loop will continue to produce numbers ranging from 1 to 5. The loop will stop at 6 because **6 < 6** is FALSE.

The **while** loop requires relevant variables to be ready, in this example, we need to define an indexing variable, **i**, which we set to 1.

code 4-12 Print the sum from 1 to 100.

```
sum <- 0
i <- 1
while(i <=100) {
        sum <- sum + i
        i <- i + 1
}
print(sum)
```

Note: remember to increment i, or else the loop will continue forever.



03. apply()

Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

apply(X, MARGIN, FUN, ..., simplify = TRUE)

X : an array, including a matrix.

MARGIN: 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns.

FUN: the function to be applied

… : optional arguments to FUN

Simplify: a logical indicating whether results should be simplified if possible.

코드 4-15

```
apply(iris[,1:4], 1, mean)    # Apply the function in the row direction.
apply(iris[,1:4], 2, mean)    # Apply the function in the column direction.
```

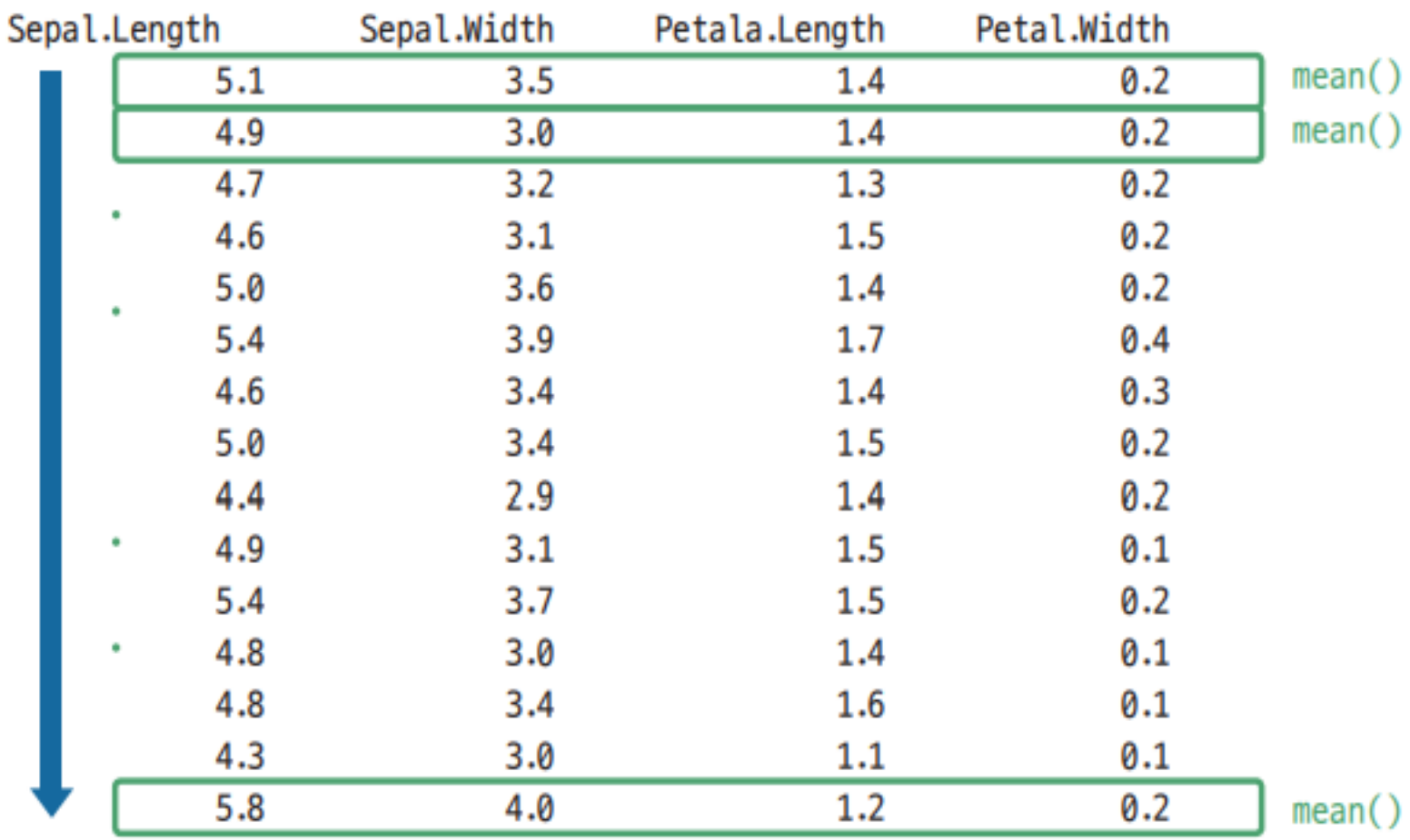


Fig. 4-2 apply(iris[,1:4], 1, mean)

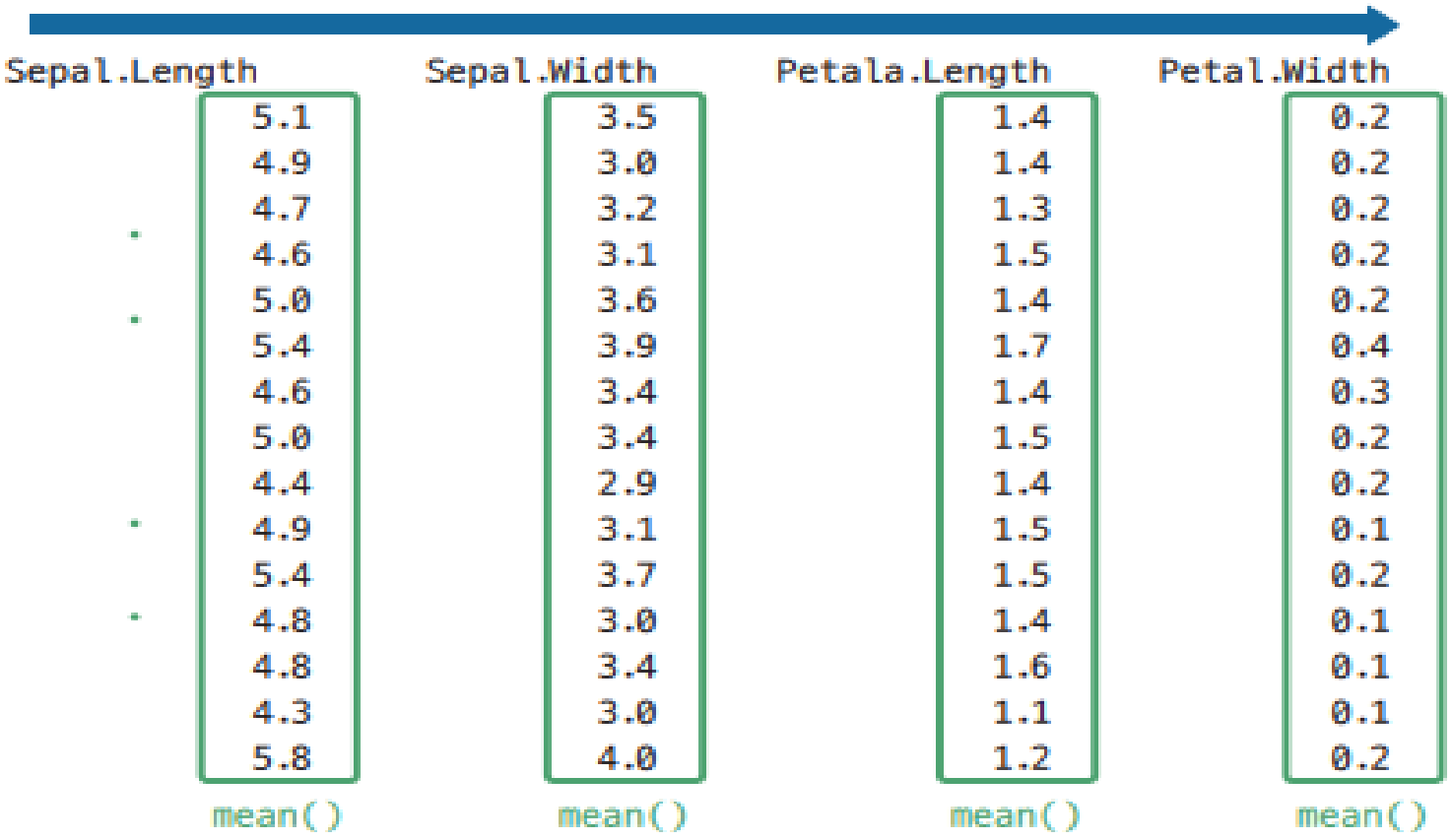


Fig. 4-3 apply(iris[,1:4], 2, mean)

04. User-defined function (UDF)

R Functions

A function is a block of code that only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Creating a Function and Call a Function

```
my_function <- function() { # create a function with the name my_function
  print("Hello World!")
}
my_function()                    # call the function named my_function
```

To call a function, use the function name followed by parenthesis,

like **my_function()**

Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name.

e.g.

```
my_function <- function(fname) {
  paste(fname, "Griffin")
}

my_function("Peter")
my_function("Lois")
my_function("Stewie")
```

Parameters or Arguments?

The terms "parameter" and "argument" can be used for the same thing: information that is passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is a value that is sent to the function when it is called.

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less

```
e.g.

my_function <- function(fname, lname){
  paste(fname, lname)
}

my_function("Peter", "Griffin")
```

If you try to call the function with 1 or 3 arguments, you will get an error.

```
e.g.

my_function <- function(fname, lname){
  paste(fname, lname)
}

my_function("Peter")
```

```
e.g.

my_function <- function(fname, lname){
  paste(fname, lname)
}

my_function("Peter“,”Griffin”, “Jhon”)
```

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without an argument, it uses the default value.

e.g.

```
my_function <- function(country = "Norway"){
  paste("I am from", country)
}

my_function("Sweden")
my_function("India")
my_function() # will get the default value, which is Norway
my_function("USA")
```

Return Values

To let a function return a result, use the **return()** function.

e.g.

```
my_function <- function(x){
  y<- 5 * x
  return (y)
}
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

code 4-19

Returning multiple values

```
myfunc <- function(x,y) {
 val.sum <- x+y
 val.mul <- x*y
 return(list(sum=val.sum, mul=val.mul))
}

result <- myfunc(5,8)
s <- result$sum
m <- result$mul
cat('5+8=', s, '\n')
cat('5*8=', m, '\n')
```

Code 4-20

```
setwd("d:/source")
source("myfunc.R")
# call fuction
a <- mydiv(20,4)
b <- mydiv(30,4)
a+b
mydiv(mydiv(20,2),5)
```

Nested Functions

There are two ways to create a nested function:

- Call a function within another function.
- Write a function within a function.

Call a function within another function.

```
e.g.
Nested_function <- function(x, y){
  a <- x + y
  return(a)
}

Nested_function(Nested_function(2,2), Nested_function(3,3))
```

The function tells x to add y.
The first input Nested_function(2,2) is "x" of the main function.
The second input Nested_function(3,3) is "y" of the main function.
The output is therefore (2+2) + (3+3) = **10**.

Write a function within a function.

```
e.g.

Outer_func <- function(x) {
  Inner_func <- function(y) {
    a <- x + y
    return(a)
  }
  return (Inner_func)
}
output <- Outer_func(3) # To call the Outer_func
output(5)
```

You cannot directly call the function because the **Inner_func** has been defined (nested) inside the **Outer_func.**
We need to call **Outer_func** first in order to call **Inner_func** as a second step.
We need to create a new variable called output and give it a value, which is 3 here.
We then print the output with the desired value of "y", which in this case is 5.
The output is therefore **8** (3 + 5).

05. Finding the location of the data that satisfies the conditions

Which indices are TRUE?

which()

Give the TRUE indices of a logical object, allowing for array indices.

code 4-21

```
score <- c(76, 84, 69, 50, 95, 60, 82, 71, 88, 84)
which(score==69)
which(score>=85)
max(score)
which.max(score)
min(score)
which.min(score)
```

code 4-22

```
score <- c(76, 84, 69, 50, 95, 60, 82, 71, 88, 84)
idx <- which(score<=60)
score[idx] <- 61
score
idx <- which(score>=80)
score.high <- score[idx]
score.high
```

code 4-23

```
idx <- which(iris$Petal.Length>5.0)
idx
iris.big <- iris[idx,]
iris.big
```

code 4-24

```
idx <- which(iris[,1:4]>5.0, arr.ind =TRUE)
idx
```