



국립공주대학교

KONG JU NATIONAL UNIVERSITY

## 04-조건문,반복문,함수


R프로그래밍

소프트웨어학과  
김정은



# 목차

---

- 조건문 
- 반복문
- apply() 함수
- 사용자 정의 함수
- 조건에 맞는 데이터의 위치 찾기

# if-else문

---

- 조건문(conditional statement)에 따라 특정 명령을 실행을 하도록 하는 프로그래밍 명령문
- 조건에 따라 실행할 명령문을 달리해야 하는 경우에 사용
- if-else문의 기본 문법

```
if(비교 조건) {  
    조건이 참일 때 실행할 명령문(들)  
} else {  
    조건이 거짓 일 때 실행할 명령문(들)  
}
```

# 기본 if-else문

---

## 코드 4-1

```
job.type <- 'A'
if(job.type == 'B') {
  bonus <- 200    # 직무 유형이 B일 때 실행
} else {
  bonus <- 100    # 직무 유형이 B가 아닌 나머지 경우 실행 } print(bonus)
```

```
> job.type <- 'A'
> if(job.type == 'B') {
+   bonus <- 200          # 직무 유형이 B일 때 실행
+ } else {
+   bonus <- 100          # 직무 유형이 B가 아닌 나머지 경우 실행
+ }
> print(bonus)
[1] 100
```

# else가 생략된 if문

---

## 코드 4-2

```
job.type <- 'B'
bonus <- 100
if(job.type == 'A') {
  bonus <- 200 # 직무 유형이 A일 때 실행
}
print(bonus)
```

```
> job.type <- 'B'
> bonus <- 100
> if(job.type == 'A') {
+   bonus <- 200 # 직무 유형이 A일 때 실행
+ }
> print(bonus)
[1] 100
```

# 다중 if-else문

---

## 코드 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)
```

```
> score <- 85
...(중간 생략)
> print(grade)
[1] "B"
```

# 여기서 잠깐! 코드블록

---

- 1. if와 else 다음에 있는 중괄호 { }는 프로그래밍에서 코드블록이라고 부름
- 2. 여러 명령문을 하나로 묶어주는 역할

```
> a <- 10  
> if(a<5) {  
+ print(a)  
+ } else {  
+ print(a*10)  
+ print(a/10)  
+ }  
[1] 100  
[1] 1
```

# 조건문에서 논리 연산자의 사용

- if문에 논리 연산자를 사용하면 복잡한 조건문을 서술할 수 있음
- 대표적인 논리연산자는 &(and)와 |(or)

## 코드 4-4

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

```
> a <- 10
> b <- 20
> if(a>5 & b>5) {      # and 사용
+   print (a+b)
+ }
[1] 30
> if(a>5 | b>30) {      # or 사용
+   print (a*b)
+ }
[1] 200
```



# ifelse문

---

- 조건에 따라 둘 중 하나의 값 또는 변수를 선택할 때 사용
- ifelse문의 문법

코드 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)
```

```
> a <- 10
> b <- 20
...(중간 생략)
> print(c)
[1] 20
>
> a <- 10
> b <- 20
>
> c <- ifelse(a>b, a, b)
> print(c)
[1] 20
```

# 여기서 잠깐! 코드블록

---


- 1. if-else문에서 발생할 수 있는 오류
- 2. else는 반드시 if문의 코드블록이 끝나는 부분에 있는 }와 같은 줄에 작성해야 함

```
job.type <- 'A'
if (job.type == 'B') {
  bonus <- 200
}
else {                # 에러 발생, 윗 줄로 옮겨야 한다.
  bonus <- 100
}
```

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

# 목차

---

- 조건문
- **반복문** 
- apply() 함수
- 사용자 정의 함수
- 조건에 맞는 데이터의 위치 찾기

# for문

---

- 반복문(repetitive statement)은 정해진 동작을 반복적으로 수행할 때 사용하는 명령문
- 동일 명령문을 여러 번 반복해서 실행할 때 사용
- for문의 문법

```
for (반복 변수 in 반복 범위) {  
    반복할 명령문(들)  
}
```

# 기본 for문

---

## 코드 4-6

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

```
> for(i in 1:5) {  
+   print('*')  
+ }  
[1] "*"  
[1] "*"  
[1] "*"  
[1] "*"  
[1] "*"
```

# 반복 범위에 따른 반복 변수의 값 변화

---

## 코드 4-7

```
for(i in 6:10) {  
  print(i)  
}
```

```
> for(i in 6:10) {  
+   print(i)  
+ }  
[1] 6  
[1] 7  
[1] 8  
[1] 9  
[1] 10
```

# 반복 변수를 이용한 구구단 출력

---

## 코드 4-8

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

```
> for(i in 1:9) {  
+   cat('2 *', i,'=', 2*i,'\n')  
+ }  
  
2 * 1 = 2  
2 * 2 = 4  
2 * 3 = 6  
2 * 4 = 8  
2 * 5 = 10  
2 * 6 = 12  
2 * 7 = 14  
2 * 8 = 16  
2 * 9 = 18
```

# for문 안에서 if문의 사용

---

## 코드 4-9

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

```
> for(i in 1:20) {  
+   if(i%%2==0) {      # 짝수인지 확인  
+     print(i)  
+   }  
+ }  
[1] 2  
[1] 4  
[1] 6  
[1] 8  
[1] 10  
[1] 12  
[1] 14  
[1] 16  
[1] 18  
[1] 20
```



# 1~100 사이의 숫자의 합 출력

---

## 코드 4-10

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

```
> sum <- 0
> for(i in 1:100) {
+   sum <- sum + i      # sum에 i 값을 누적
+ }
> print(sum)
[1] 5050
```

# iris에서 꽃잎의 길이에 따른 분류 작업

---

## 코드 4-11

```
norow <- nrow(iris)           # iris의 행의 수
mylabel <- c()                 # 비어있는 벡터 선언
for(i in 1:norow) {
  if (iris$Petal.Length[i] <= 1.6) { # 꽃잎의 길이에 따라 레이블 결정
    mylabel[i] <- 'L'
  } else if (iris$Petal.Length[i] >= 5.1) {
    mylabel[i] <- 'H'
  } else {
    mylabel[i] <- 'M'
  }
}
print(mylabel)                 # 레이블 출력
newds <- data.frame(iris$Petal.Length, mylabel) # 꽃잎의 길이와 레이블 결합
head(newds)                    # 새로운 데이터셋 내용 출력
```

# iris에서 꽃잎의 길이에 따른 분류 작업 (계속)

---

```
> norow <- nrow(iris)                # iris의 행의 수
> mylabel <- c()                      # 비어있는 벡터 선언
> for(i in 1:norow) {
+   if (iris$Petal.Length[i] <= 1.6) {    # 꽃잎의 길이에 따라 레이블 결정
+     mylabel[i] <- 'L'
+   } else if (iris$Petal.Length[i] >= 5.1) {
+     mylabel[i] <- 'H'
+   } else {
+     mylabel[i] <- 'M'
+   }
+ }
> print(mylabel)                      # 레이블 출력
[1] "L" "L" "L" "L" "L" "M" "L" "L" "L" "L" "L" "L" "L" "L" "L" "L"
[18] "L" "M" "L" "M" "L" "L" "M" "M" "L" "L" "L" "L" "L" "L" "L" "L"
[35] "L" "L" "L" "L" "L" "L" "L" "L" "L" "L" "M" "L" "L" "L" "L" "M"
[52] "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M"
```

# iris에서 꽃잎의 길이에 따른 분류 작업 (계속)

---

```
[69] "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "H" "M"
[86] "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "H" "H"
[103] "H" "H" "H" "H" "M" "H" "H" "H" "H" "H" "H" "M" "H" "H" "H" "H" "H"
[120] "M" "H" "M" "H" "M" "H" "H" "M" "M" "H" "H" "H" "H" "H" "H" "H" "H"
[137] "H" "H" "M" "H" "H" "H" "H" "H" "H" "H" "M" "H" "H" "H"

> newds <- data.frame(iris$Petal.Length, mylabel) # 꽃잎의 길이와 레이블 결합
> head(newds)                                     # 새로운 데이터셋 내용 출력
```

	iris.Petal.Length	mylabel
1	1.4	L
2	1.4	L
3	1.3	L
4	1.5	L
5	1.4	L
6	1.7	M

# while문

---

- while문은 어떤 조건이 만족하는 동안 코드블록을 수행하고, 해당 조건이 거짓일 경우 반복을 종료하는 명령문

```
while (비교조건) {  
    반복할 명령문(들)  
}
```

# While문 (계속)

---

## 코드 4-12

```
sum <- 0
i <- 1
while(i <=100) {
  sum <- sum + i # sum에 i 값을 누적
  i <- i + 1    # i 값을 1 증가시킴
}
print(sum)
```

```
> sum <- 0
> i <- 1
> while(i <=100) {
+   sum <- sum + i          # sum에 i 값을 누적
+   i <- i + 1              # i 값을 1 증가시킴
+ }
> print(sum)
[1] 5050
```

# break

---

- 특정 조건을 만족하면 반복문을 벗어남

코드 4-13

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

```
> sum <- 0
> for(i in 1:10) {
+   sum <- sum + i
+   if (i>=5) break
+ }
> sum
[1] 15
```

# next

---

- 특정 조건이 성립하면 뒤는 무시하고 처음으로 돌아감

## 코드 4-14


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

```
> sum <- 0
> for(i in 1:10) {
+   if (i%%2==0) next
+   sum <- sum + i
+ }
> sum
[1] 25
```



# 목차

---

- 조건문
- 반복문
- **apply() 함수** 
- 사용자 정의 함수
- 조건에 맞는 데이터의 위치 찾기

# apply() 함수의 개념

---

- 반복 작업이 필요한 경우에는 반복문을 적용
- 반복 작업의 대상이 매트릭스나 데이터프레임의 행(row) 또는 열(column)인 경우는 for문이나 while문 대신에 apply() 함수를 이용할 수 있음
- apply() 함수의 문법

```
apply(데이터셋, 행/열방향 지정, 적용 함수)
```

# apply() 함수의 적용


---

## 코드 4-15

```
apply(iris[,1:4], 1, mean)  # row 방향으로 함수 적용  
apply(iris[,1:4], 2, mean)  # col 방향으로 함수 적용
```

```
> apply(iris[,1:4], 1, mean)      # row 방향으로 함수 적용  
[1] 2.550 2.375 2.350 2.350 2.550 2.850 2.425 2.525 2.225  
[10] 2.400 2.700 2.500 2.325 2.125 2.800 3.000 2.750 2.575  
[19] 2.875 2.675 2.675 2.675 2.350 2.650 2.575 2.450 2.600  
[28] 2.600 2.550 2.425 2.425 2.675 2.725 2.825 2.425 2.400  
[37] 2.625 2.500 2.225 2.550 2.525 2.100 2.275 2.675 2.800  
...(중간 생략)  
[136] 4.775 4.425 4.200 3.900 4.375 4.450 4.350 3.875 4.550  
[145] 4.550 4.300 3.925 4.175 4.325 3.950
```

# apply() 함수의 적용 (계속)

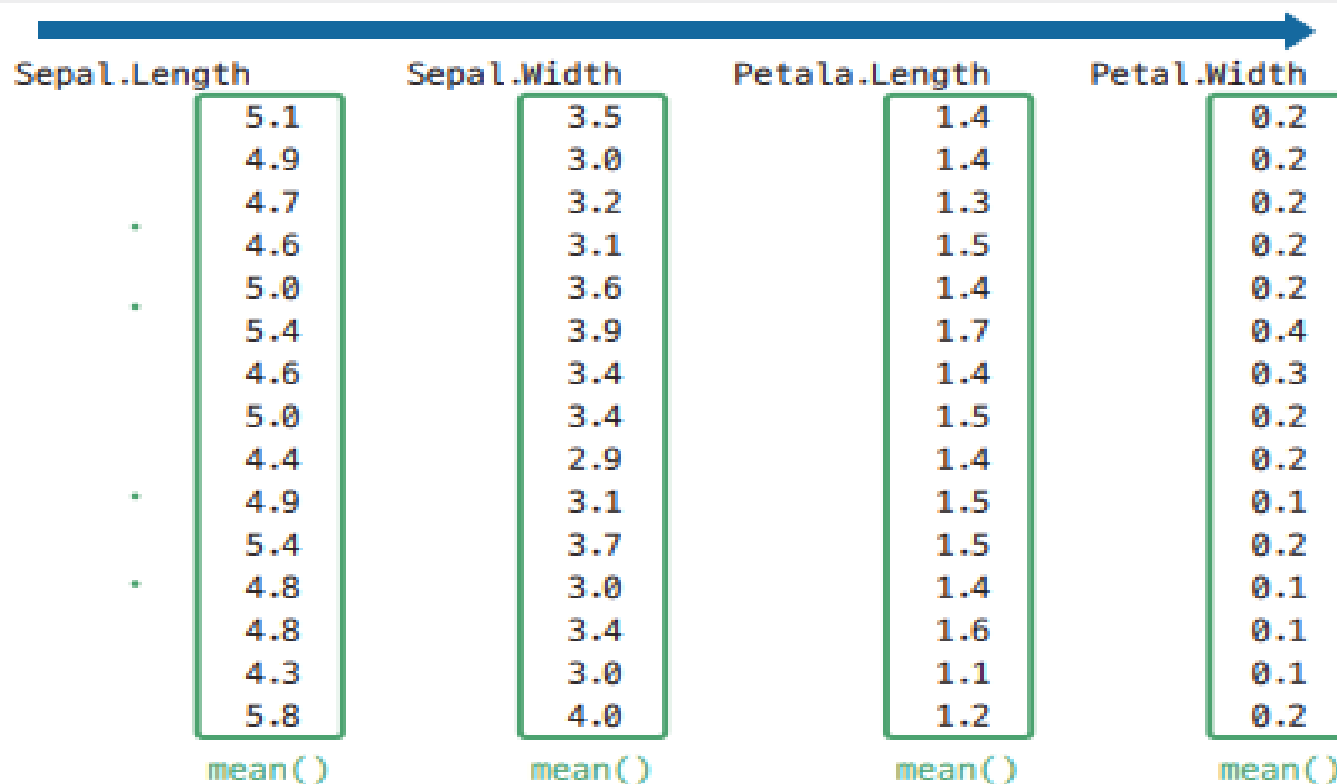


Sepal.Length	Sepal.Width	Petala.Length	Petal.Width	
5.1	3.5	1.4	0.2	mean()
4.9	3.0	1.4	0.2	mean()
4.7	3.2	1.3	0.2	
• 4.6	3.1	1.5	0.2	
• 5.0	3.6	1.4	0.2	
5.4	3.9	1.7	0.4	
4.6	3.4	1.4	0.3	
5.0	3.4	1.5	0.2	
4.4	2.9	1.4	0.2	
• 4.9	3.1	1.5	0.1	
5.4	3.7	1.5	0.2	
• 4.8	3.0	1.4	0.1	
4.8	3.4	1.6	0.1	
4.3	3.0	1.1	0.1	
5.8	4.0	1.2	0.2	mean()

# apply() 함수의 적용 (계속)

```
> apply(iris[,1:4], 2, mean) # col 방향으로 함수 적용
```


```
Sepal.Length Sepal.Width Petal.Length Petal.Width  
5.843333     3.057333     3.758000     1.199333
```



Sepal.Length	Sepal.Width	Petala.Length	Petal.Width
5.1	3.5	1.4	0.2
4.9	3.0	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5.0	3.6	1.4	0.2
5.4	3.9	1.7	0.4
4.6	3.4	1.4	0.3
5.0	3.4	1.5	0.2
4.4	2.9	1.4	0.2
4.9	3.1	1.5	0.1
5.4	3.7	1.5	0.2
4.8	3.0	1.4	0.1
4.8	3.4	1.6	0.1
4.3	3.0	1.1	0.1
5.8	4.0	1.2	0.2
mean()	mean()	mean()	mean()

# 목차

---

- 조건문
- 반복문
- apply() 함수
- **사용자 정의 함수** 
- 조건에 맞는 데이터의 위치 찾기

# 사용자 정의 함수 만들기

---

- R은 사용자들도 자신만의 함수를 만들어 사용할 수 있는 기능을 제공하는데, 이를 사용자 정의 함수라고 함
- 사용자 정의 함수 문법

```
함수명 <- function(매개변수 목록) {  
  실행할 명령문(들)  
  return(함수의 실행 결과)  
}
```

# 사용자 정의 함수를 만들고 사용하기

---

코드 4-16

```
mymax <- function(x,y) {  
  num.max <- x  
  if (y > x) {  
    num.max <- y  
  }  
  return(num.max)  
}
```

코드 4-17

```
mymax(10,15)  
a <- mymax(20,15)  
b <- mymax(31,45)  
print(a+b)
```

```
> mymax(10,15)  
[1] 15  
> a <- mymax(20,15)  
> b <- mymax(31,45)  
> print(a+b)  
[1] 65
```



# 사용자 정의 함수의 매개변수에 초기값 설정하기

---

## 코드 4-18

```
mydiv <- function(x,y=2) {  
  result <- x/y  
  return(result)  
}
```

```
mydiv(x=10,y=3)    # 매개변수 이름과 매개변수값을 쌍으로 입력  
mydiv(10,3)        # 매개변수값만 입력  
mydiv(10)           # x에 대한 값만 입력(y 값이 생략됨)
```

```
> mydiv <- function(x,y=2) {  
+   result <- x/y  
+   return(result)  
+ }  
>  
> mydiv(x=10,y=3)  
[1] 3.333333  
> mydiv(10,3)  
[1] 3.333333  
> mydiv(10)  
[1] 5
```

# 함수가 반환하는 결과값이 여러 개일 때의 처리

## 코드 4-19

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

```
> 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$sum           # 5, 8의 합  
> m <- result$mul           # 5, 8의 곱  
> cat('5+8=', s, '\n')  
5+8= 13  
> cat('5*8=', m, '\n')  
5*8= 40
```

# 사용자 정의 함수의 저장 및 호출

## 코드 4-20

```
setwd("d:/source")      # myfunc.R이 저장된 폴더  
source("myfunc.R")      # myfunc.R 안에 있는 함수 실행
```

# 함수 사용

```
a <- mydiv(20,4)         # 함수 호출
```

```
b <- mydiv(30,4)         # 함수 호출
```

```
a+b
```

```
mydiv(mydiv(20,2),5) # 함수 호출
```

```
> setwd("d:/source")    # myfunc.R이 저장된 폴더  
> source("myfunc.R")    # myfunc.R 안에 있는 함수 실행
```

# 함수 사용

```
> a <- mydiv(20,4)       # 함수 호출
```

```
> b <- mydiv(30,4)       # 함수 호출
```

```
> a+b
```


```
[1] 12.5
```

```
> mydiv(mydiv(20,2),5)   # 함수 호출
```

```
[1] 2
```

# 목차

---

- 조건문
- 반복문
- apply() 함수
- 사용자 정의 함수
- 조건에 맞는 데이터의 위치 찾기 

# 조건에 맞는 데이터의 위치 찾기

---

- 데이터 분석을 하다보면 자신이 원하는 데이터가 벡터나 매트릭스, 데이터 프레임 안에서 어디에 위치하고 있는지를 알기 원하는 때가 있음
- 예를 들어, 50명의 학생 성적이 저장된 벡터가 있는데 가장 성적이 좋은 학생은 몇 번째에 있는지를 알고 싶은 경우
- 이런 경우 편리하게 사용할 수 있는 함수가 `which()`, `which.max()`, `which.min()` 함수

# 조건에 맞는 데이터의 위치 찾기 (계속)

## 코드 4-21

```
score <- c(76, 84, 69, 50, 95, 60, 82, 71, 88, 84)
which(score==69)      # 성적이 69인 학생은 몇 번째에 있나
which(score>=85)      # 성적이 85 이상인 학생은 몇 번째에 있나
max(score)           # 최고 점수는 몇 점인가
which.max(score)      # 최고 점수는 몇 번째에 있나
min(score)           # 최저 점수는 몇 점인가
which.min(score)      # 최저 점수는 몇 번째에 있나
```

```
> score <- c(76, 84, 69, 50, 95, 60, 82, 71, 88, 84)
> which(score==69)      # 성적이 69인 학생은 몇 번째에 있나
[1] 3
> which(score>=85)      # 성적이 85 이상인 학생은 몇 번째에 있나
[1] 5 9
> max(score)           # 최고 점수는 몇 점인가
[1] 95
> which.max(score)      # 최고 점수는 몇 번째에 있나
[1] 5
> min(score)           # 최저 점수는 몇 점인가
[1] 50
> which.min(score)      # 최저 점수는 몇 번째에 있나
[1] 4
```

---

Thank You  
Any Questions?