4. 함수 사용, 비교문, 반복문

함수 정의

R에서 함수를 만들어 사용하기 위한 규칙은 하기와 같다.

함수 생성

❑ 함수 생성 규칙

- function 키워드를 사용하여 생성

```
함수명 <- function ( param1, param2, …. ) {
        함수의 내용
}
```

❑ 함수 구성요소

- 함수명 : 함수의 실제 이름
  (함수명을 갖는 객체로 R환경에 저장됨)
- param : 변수, 함수가 호출될 때 넘기는 값
  (선택사항으로 함수가 인자를 갖지 않을 수도 있음)
- 함수의 내용 : 함수의 실행Logic을 정의한 구문
- 리턴값 : 함수의 결과값을 호출한 위치로 반환하는 값
  (일반적으로 함수 내용에서 마지막 표현식)

예제

❑ 연속된 숫자의 자승(^)을 출력하는 함수 만들기

```
new.function<-function(a){
  for(i in 1:a){
    b<-i^2
    print(b)
  }
}
new.function(6)
```

❑ 실행결과

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```

함수 호출

함수 호출에는 인자있는 함수 호출과 인자없는 함수 호출이 있으며 하기와 같다.

인자값이 있는 함수 호출

❑ 코드

```
new.function<-function(a,b,c) {
  result<-a*b+c
  print(result)
}
new.function(5,3,11)
```

❑ 실행결과

```
> new.function(5,3,11)
[1] 26
```

인자값이 없는 함수 호출

❑ 코드

```
new.function<-function(){
  result<-5*3+11
  print(result)
}
new.function()
```

❑ 실행결과

```
> new.function()
[1] 26
```

함수 호출

함수를 정의할 때 미리 인자 값을 정의함으로써 인자를 넘겨주지 않아도 기본 결과 값을 얻을 수 있는 함수를 호출할 수 있다.

기본값 인수로 함수 호출

❑ 코드

```
new.function<-function(a=5,b=3,c=11) {
  result<-a*b+c
  print(result)
}
new.function()
```

❑ 실행결과

```
> new.function()
[1] 26
```

인자개수가 적을 경우

❑ 코드

```
new.function<-function(a,b,c) {
  print(a)
  print(b)
  print(c)
}
new.function(5, 3)
```

❑ 실행결과

```
> new.function(5, 3)
[1] 5
[1] 3
Error in print(c) : argument "c" is missing, with no default
```

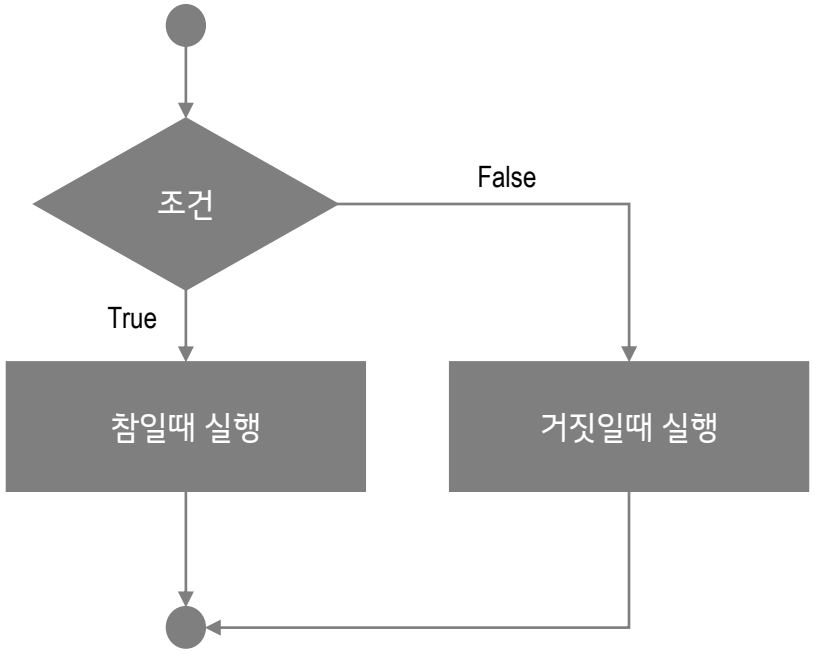
비교문

R에서 특정 조건에 따라 수행하는 문장을 다르게 가져가기 위해서는 비교문이 필요하다.

비교문이란?

❑ **정의**

- 조건의 참/거짓에 따라 다른 명령을 수행하기 위한 구문



조건문의 종류

❑ **if 구문**

- 1개 이상의 문장으로 표시된 boolean형태로 구성

❑ **if … else 구문**

- 조건문의 내용이 거짓일 때 수행하는 형태로 구성

❑ **switch 구문**

- 변수와 일련의 값들에 대한 결과에 따라
  여러 개의 명령을 수행하는 형태로 구성

비교연산자

R에서 사용가능한 비교연산자와 논리연산자는 하기와 같다.

비교연산자

❑ **정의**

- 크고, 작음, 같음, 다름을 판단하는 연산자를 말함

❑ **종류**

> : 초과(왼쪽항이 오른쪽항보다 크면 True)

< : 미만(왼쪽항이 오른쪽항보다 작으면 True)

>= : 이상(왼쪽항이 오른쪽항보다 크거나 같으면 True)

<= : 이하(왼쪽항이 오른쪽항보다 작거나 같으면 True)

== : 같음(두항이 같으면 True, 다르면 False를 출력함)

!= : 다름(두항이 다르면 True, 같으면 False를 출력함)

논리연산자

❑ **정의**

- 논리연산(AND, OR)을 판단하는 연산자를 말함

❑ **종류**

& : AND 연산자

(두 조건을 동시에 만족하면 True,
아닌 경우 False를 출력함)

| : OR연산자

(두 조건중 하나만 만족해도 True,
두 조건 모두 만족하지 않을 경우 False를 출력함)

비교연산자

R에서 사용가능한 비교연산자와 논리연산자 실 사용사례는 하기와 같다.

비교연산자

❑ 코드

```
x<-c(1,2,3,4,5,6)
y<-c(1,2,7,8,9,3)
x>y
x>=y
x==y
x!=y
```

❑ 결과

```
[1] FALSE FALSE FALSE FALSE FALSE  TRUE
[1]  TRUE  TRUE FALSE FALSE FALSE  TRUE
[1]  TRUE  TRUE FALSE FALSE FALSE FALSE
[1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

논리연산자

❑ 코드

```
x>3 & x<5
x>2 | x>5
x[x>3]
sum(x>3 & x<6)
any(x>3)
all(x>3)
```

※ 벡터 간 연산
- AND : &
- OR : |

※ 조건문 간 연산
- AND : &&
- OR : ||

❑ 결과

```
[1] FALSE FALSE FALSE  TRUE FALSE FALSE
[1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE
[1] 4 5 6
[1] 2
[1] TRUE
[1] FALSE
```

비교문

비교문 예제는 하기와 같다.

ifelse문

❑ ifelse(a,b,c) : a가 참이면 b를 출력, 거짓이면 c를 출력

```
no<-scan()
1:10

Ifelse(no%%2==0, '짝수', '홀수')
```

❑ 결과

[1] "짝수"

조건문이 1개인 경우

❑ 입력숫자가 0보다 크면 2배로 출력

```
mf2<-function(x){
  if(x>0){x<-x*2
          return(x)}
}
mf2(3)
mf2(0)
mf2(-3)
```

❑ 결과

[1] 6

비교문

비교문 예제는 하기와 같다.

조건문이 2개인 경우

❑ 입력숫자가 양수면 2배로 출력, 숫자가 0보다 이하면 0출력

```
mf1<-function(x) {
  if(x>0){x<-x*2
          return(x)}
  else{x<-0
       return(x)}
}
mf1(2)
mf1(-1)
mf1(0)
```

❑ 결과

```
[1] 4
[1] 0
[1] 0
```

조건문이 3개 이상인 경우

❑ 입력숫자가 0보다 크면 2배로 출력, 0이면 0출력,
0보다 작으면 -2배로 출력

```
mf2<-function(x){
  if(x>0){x<-x*2
          return(x)}
  else if(x==0){x<-0
                return(x)}
  else if{x<-x*2
          return(x)}
}
mf2(3)
mf2(0)
mf2(-3)
```

❑ 결과

```
[1] 6
[1] 0
[1] -6
```

비교문

비교문 예제는 하기와 같다.

여러가지 조건문

❑ **여러 조건 동시 사용 방법**

- 여러 조건을 동시에 사용하기 위한 비교연산자를 사용함

! : NOT연산자

&& : AND연산자

|| : OR연산자

여러가지 조건문 동시 사용

❑ **코드**

```
myf4<-function(a,b){
  if((a>1) && (b>1)){
        c<-a*b)
        return(c)}
  else{
        c<-a+b
        return©}
}
myf4(2,3)
myf4(2,-1)
```

❑ **결과**

```
[1] 6
[1] 1
```

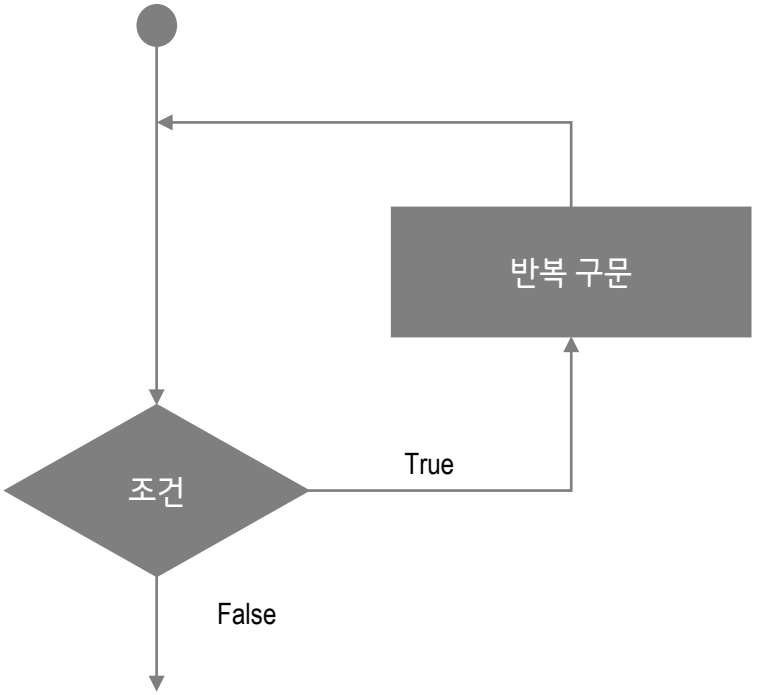
반복문

R에서 특정 명령을 반복해서 수행하기 위해서는 반복문의 정의가 필요하다.

반복문이란?

❑ 정의

- 특정 기능을 여러 번 수행하기 위한 구문



반복문의 종류

❑ **repeat loop**

- 연속된 구문들을 여러 번 수행하고
  루프 변수를 관리하는 코드를 단축시킨다.

❑ **while loop**

- 주어진 조건이 참일 때 하나 이상의 구문을 반복함
- 루프 안의 코드를 수행하기 전에 조건을 검사함

❑ **for loop**

- while 루프와 비슷함
- 루프 안의 코드를 수행한 후 조건을 검사함

❑ **break**

- 루프 구문을 종료하고 루프 다음 구문을 수행함

반복문

R에서 특정 명령을 반복해서 수행하기 위해서는 반복문의 정의가 필요하다.

반복문 형식

❑ **repeat loop**

repeat{반복해서 수행할 문장}

#블록안의 문장을 반복해서 수행함

❑ **while loop**

while(조건){조건이 참일 때 수행할 문장)

#조건이 참일때 블록안의 문장을 수행함

❑ **for loop**

for (i in data){i를 사용한 문장)

#data에 들어 있는 각각의 값을 변수 i에 할당한 후

각각에 대해 블록안의 문장을 수행함

반복문의 예제

❑ **repeat loop**

```
i<-1
repeat{
  print(i)
  if (i>=10){break}
  i<-i+1
```

❑ **while loop**

```
i<-1
while(i<=10){
  print(i)
  i<-i+1
}
```

❑ **for loop**

```
for(i in 1:10){
  print(i)
}
```

R Cheat sheet - function

Function Basics

Functions – objects in their own right
All R functions have three parts:

body()	code inside the function
formals()	list of arguments which controls how you can call the function
environment()	"map" of the location of the function's variables (see "Enclosing Environment")

Every operation is a function call

- +, for, if, [, $, { …
- x + y is the same as `+`(x, y)

Note: the backtick (`), lets you refer to functions or variables that have otherwise reserved or illegal names.

Lexical Scoping

What is Lexical Scoping?

- Looks up value of a symbol. (see "Enclosing Environment")
- **findGlobals()** - lists all the external dependencies of a function

```
f <- function() x + 1
codetools::findGlobals(f)
> '+' 'x'
```

```
environment(f) <- emptyenv()
f()
# error in f(): could not find function "+"
```

- R relies on lexical scoping to find everything, even the + operator.

Function Arguments

Arguments – passed by reference and copied on modify

1. Arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position.
2. Check if an argument was supplied : **missing()**

```
i <- function(a, b) {
  missing(a) -> # return true or false
}
```

3. Lazy evaluation – since x is not used **stop("This is an error!")** never get evaluated.

```
f <- function(x) {
  10
}
f(stop('This is an error!')) -> 10
```

4. Force evaluation

```
f <- function(x) {
  force(x)
  10
}
```

5. Default arguments evaluation

```
f <- function(x = ls()) {
  a <- 1
  x
}
```

f() -> 'a' 'x'	ls() evaluated inside f
f(ls())	ls() evaluated in global environment

Return Values

- **Last expression evaluated or explicit return().**
  Only use explicit return() when returning early.
- **Return ONLY single object.**
  Workaround is to return a list containing any number of objects.
- **Invisible return object value** - not printed out by default when you call the function.

```
f1 <- function() invisible(1)
```

Primitive Functions

What are Primitive Functions?

1. Call C code directly with **.Primitive()** and contain no R code

```
print(sum) :
> function (..., na.rm = FALSE)  .Primitive('sum')
```

2. **formals()**, **body()**, and **environment()** are all NULL
3. Only found in base package
4. More efficient since they operate at a low level

Influx Functions

What are Influx Functions?

1. Function name comes in between its arguments, like + or –
2. All user-created infix functions must start and end with %.

```
`%+%` <- function(a, b) paste0(a, b)
'new' %+% 'string'
```

3. Useful way of providing a default value in case the output of another function is NULL:

```
`%||%` <- function(a, b) if (!is.null(a)) a else b
function_that_might_return_null() %||% default value
```

Replacement Functions

What are Replacement Functions?

1. Act like they modify their arguments in place, and have the special name xxx <-
2. Actually create a modified copy. Can use **pryr::address()** to find the memory address of the underlying object

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:10
second(x) <- 5L
```