R프로그래밍

4장. 기본 표현식

박 혜 승 교 수

4장. 기본 표현식

4.1 할당 표현식
4.2 조건 표현식
4.3 반복 표현식
4.4 마치며

4.1 할당 표현식

4.1 할당 표현식

- **할당 표현식**
  - ✓ 할당은 모든 프로그래밍 언어에서 가장 기본적인 표현식.
  - ✓ 어떤 기호에 값을 할당하거나, 나중에 그 기호에 연결된 값을 참조할 수 있게 하는 역할.
  - ✓ R에서는 할당에 <- 연산자를 사용.

```
> x <- 1
> y <- c(1, 2, 3)
> z <- list(x, y)
```

  - ✓ R에서는 값을 할당하기 전에 기호(객체 이름)와 형식을 미리 선언할 필요가 없음.
  - ✓ 기호가 환경에 없다면 할당을 이용하여 해당 기호를 자동으로 생성.
  - ✓ 이미 객체가 있다면 할당을 이용하여 충돌 없이 해당 객체에 새로운 값을 다시 연결.

4.1 할당 표현식

- **대체 할당 연산자**
  - ✓ f(z)의 결과값을 x 기호에 할당하는 x <- f(z) 처럼 ->를 사용하여 반대 방향으로 할당 가능.

```
> 2 -> x1
```

  - ✓ 다음과 같이 여러 객체가 모두 같은 값을 갖도록 할당 연산자도 연결 가능.

```
> x3 <- x2 <- x1 <- 0
```

  - ✓ 표현식 0은 한 번 평가되고, 똑같은 값이 세 객체에 할당.
  - ✓ 어떻게 동작하는지 더 자세히 알아보기 위해 0 대신 난수 생성 함수를 사용해보자.

```
> x3 <- x2 <- x1 <- rnorm(1)        # rnorm(1) 은 표준 정규 분포에 따라 난수를 만드는 함수.
> c(x1, x2, x3)
[1] 0.7077795 0.7077795 0.7077795
```

  - ✓ 다른 프로그래밍 언어처럼 할당에 = 사용 가능.

```
> x2 = c(1, 2, 3)
```

    - ❖ 둘 다 사용이 가능하고 실제로 결과도 동일.
    - ❖ 구글의 R 스타일 가이드에서는 = 대신에 <-를 사용하길 추천함.

4.1 할당 표현식

- **대체 할당 연산자**
  - ✓ <- 와 = 의 미묘한 차이점.
  - ✓ 먼저 인수 2개를 받아 출력하는 f( ) 함수를 만들어보자.

```
> f <- function(input, data = NULL) {
+     cat("input:\n")
+     print(input)
+     cat("data:\n")
+     print(data)
+ }
> x <- c(1, 2, 3)
> y <- c("some", "text")
> f(input = x)
input:
[1] 1 2 3
data:
NULL
```

- 이 코드에서는 <- 와 = 연산자를 모두 사용하지만, 역할은 서로 다름.
- 처음 두 줄에서 <- 연산자는 할당 연산자로 사용.
- 셋째 줄의 = 는 f( ) 함수의 인수 입력을 지정.

- <- 연산자는 오른쪽의 표현식 c(1, 2, 3)을 평가하고, 평가된 값을 왼쪽 x 라는 기호(변수)의 객체에 할당.
- = 연산자는 할당 연산자가 아니라 함수 인수를 이름과 일치시키는데 사용.

4.1 할당 표현식

- **대체 할당 연산자**
  - ✓ <-와 = 연산자를 할당 연산자로 사용할 때는 서로 바꾸어서 써도 됨.
  - ✓ 앞 코드는 다음 코드와 동일.

```
> x = c(1, 2, 3)
> y = c("some", "text")
> f(input = x)
input:
[1] 1 2 3
data:
NULL
```

  - ✓ 여기서는 = 연산자만 쓰지만, 두 가지 다른 목적으로 사용.
  - ✓ 처음 두 줄에서 = 는 할당을 수행하고, 셋째 줄에서 = 는 명명된 인수를 지정.

4.1 할당 표현식

- **대체 할당 연산자**
  - ✓ 이번에는 = 를 모두 <- 로 변경했을 때 어떤 일이 일어나는지 보자.

```
> x <- c(1, 2, 3)
> y <- c("some", "text")
> f(input <- x)
input:
[1] 1 2 3
data:
NULL
```

  - ✓ 막상 이 코드를 실행하면 앞 코드와 출력 결과가 같음.
  - ✓ 환경을 보면 차이점을 발견할 수 있음.
  - ✓ 즉, 환경에 c(1, 2, 3) 값이 할당된 새로운 input 변수가 생성.

```
> input
[1] 1 2 3
```

  - ✓ 셋째 줄에서 할당 연산 input <- x 는 환경에 새로운 input 객체를 만들고 이 객체는 x 를 결과로 갖음.
  - ✓ input 값은 f( ) 함수의 첫 번째 인수로 제공.
  - ✓ 즉, 이 함수의 첫 번째 인수는 이름과 일치되어 지정되는 것이 아니라 위치와 일치되어 지정되는 것

4.1 할당 표현식

- **대체 할당 연산자**
  - ✓ 함수를 사용하는 전형적인 방법은 다음과 같음.

```
> f(input = x, data = y)
input:
[1] 1 2 3
data:
[1] "some" "text"
```

  - ✓ = 를 모두 <-로 대체한다면 결과는 다음과 같음.

```
> f(input <- x, data <- y)
input:
[1] 1 2 3
data:
[1] "some" "text"
```

4.1 할당 표현식

- **대체 할당 연산자**
  - ✓ = 가 어떤 역할을 하는지 알아보고자 두 인수의 위치를 서로 바꾸어도 결과는 같은 것을 볼 수 있음.

```
> f(data = y, input = x)
input:
[1] 1 2 3
data:
[1] "some" "text"
```

  - ✓ = 대신 <-를 사용할 때는 결과가 뒤바뀐다는 것을 알 수 있음.

```
> f(data <- y, input <- x)
input:
[1] "some" "text"
data:
[1] 1 2 3
```

4.1 할당 표현식

- **대체 할당 연산자**
  - ✓ 앞 코드는 다음 코드를 실행하는 것과 결과가 같다고 볼 수 있음.

```
> data <- y
> input <- x
> f(y, x)
input:
[1] "some" "text"
data:
[1] 1 2 3
```

  - ✓ 이 코드는 f(y, x)를 실행할 뿐 아니라 쓸데없이 현재 환경에 data와 input이라는 추가적인 변수를 생성.

4.1 할당 표현식

- **대체 할당 연산자**
  - ✓ 모호성을 줄이려면 할당 연산자로는 <- 나 = 를 사용.
  - ✓ 함수에서 명명된 인수를 지정하려면 = 만 사용.
  - ✓ 결론적으로 R 코드의 가독성을 높이기 위해 구글 스타일 가이드에서 제안하는 대로 할당에는 <-를 사용하고 명명된 인수 지정에는 = 만 사용하자.

4.1 할당 표현식

- **비표준 이름과 역따옴표 사용하기**
  - ✓ 할당 연산자로 임의의 값을 변수(기호 혹은 이름이 명명된 객체)에 할당 가능.
  - ✓ 직접 할당은 이름 형식이 제한.
  - ✓ 알파벳 a 에서 z, A에서 Z(R은 대·소문자 구분), 언더바(_), 마침표(.)만 사용 가능.
  - ✓ 공백 문자는 들어갈 수 없으며, 언더바(_)로 시작할 수도 없음.
  - ✓ 다음은 사용 가능한 이름.

```
> students <- data.frame()
> us_population <- data.frame()
> sales.2015 <- data.frame()
```

  - ✓ 다음은 규칙을 지키지 않아 사용할 수 없는 이름.

```
> some data <- data.frame()
Error: unexpected symbol in "some data"
> _data <- data.frame()
Error: unexpected input in "_"
> Population(Millions) <- data.frame()
Error in Population(Millions) <- data.frame() :
  object 'Millions' not found
```

❖ some data는 중간에 공백 문자가 포함.
❖ _data는 _로 시작.
❖ Population(Millions)는 함수 호출이라고 할 수 있음.

4.1 할당 표현식

- **비표준 이름과 역따옴표 사용하기**
  - ✓ 유효하지 않은 이름(비표준 이름)을 굳이 사용하고 싶을 때는 역따옴표(`)를 활용.

```
> `some data` <- c(1, 2, 3)
> `_data` <- c(4, 5, 6)
> `Population(Millions)` <- c(city1 = 50, city2 = 60)
```

  - ✓ 이러한 변수를 참조할 때도 마찬가지로 역따옴표를 사용해야 함.
  - ✓ 그렇지 않으면 여전히 유효하지 않은 것으로 간주.

```
> `some data`
[1] 1 2 3
> `_data`
[1] 4 5 6
> `Population(Millions)`
city1 city2
   50    60
```

  - ✓ 마찬가지로 함수와 리스트에도 역따옴표 사용 가능.

```
> `Tom's secret function` <- function(a, d) {
+     (a ^ 2 - d ^ 2) / (a ^ 2 + d ^ 2)
+ }

> l1 <- list(`Group(A)` = rnorm(10), `Group(B)` = rnorm(10))
```

4.1 할당 표현식

- **비표준 이름과 역따옴표 사용하기**
  - ✓ 객체를 직접 참조하는데 이름이 유효하지 않을 때(객체 이름의 규칙을 어겼을 때)도 마찬가지로 역따옴표를 사용해야 이를 참조할 수 있음.

```
> `Tom's secret function`(1, 2)
[1] -0.6
> l1$`Group(A)`
 [1] -0.9926708 -1.3606540  0.8490925  0.3309389  1.5788444 -0.4184110 -1.5443471
0.6233166
 [9]  0.3511112  1.1038816
```

4.1 할당 표현식

- **비표준 이름과 역따옴표 사용하기**
  - ✓ 단 data.frame( )은 예외.

```
> results <- data.frame(`Group(A)` = rnorm(10), `Group(B)` = rnorm(10))
> results
        Group.A    Group.B.
1     1.0912044   0.8001147
2    -1.0405028   1.0828979
3     0.1499670   0.1322296
4    -1.1905868  -0.1043326
5    -1.7572322  -0.6665534
6     0.8183919   0.2597405
7    -0.5373271  -0.4388536
8    -0.7905835   1.2764571
9    -0.7681928   0.6329882
10    0.2049728  -1.2271866
```

  - ✓ data.frame 변수는 객체 이름에서 유효하지 않은 기호들을 모두 마침표로 변경.
  - ✓ 이어지는 코드를 보면 알겠지만, make.names( ) 함수를 사용하는 것과 결과가 동일.

```
> colnames(results)
[1] "Group.A." "Group.B."
```

4.1 할당 표현식

- **비표준 이름과 역따옴표 사용하기**
  - ✓ 실제로 CSV 형식의 테이블 데이터를 불러올 때도 이러한 일이 자주 발생.

```
ID,Category,Population(before),Population(after)
0,A,10,12
1,A,12,13
2,A,13,16
3,B,11,12
4,C,13,12
```

  - ✓ read.csv( ) 함수를 사용해서 CSV 데이터를 읽어 들이면 Population(before)와 Population(after) 변수는 원래 이름을 유지하지 않고 make.names ( ) 함수를 사용하여 R에서 유효한 이름으로 변경.
  - ✓ 어떤 이름을 얻는지 알고자 다음 코드를 실행해 보자.

```
> make.names(c("Population(before)", "Population(after)"))
[1] "Population.before." "Population.after."
```

4.1 할당 표현식

- **비표준 이름과 역따옴표 사용하기**
  - ✓ 때로는 이러한 자동 변경이 불필요.
  - ✓ 이 기능을 비활성화하려면 read.csv( )나 data.frame( ) 함수를 호출할 때 check.names = FALSE를 설정.

```
> results <- data.frame(
+     ID = c(0, 1, 2, 3, 4),
+     Category = c("A", "A", "A", "B", "C"),
+     `Population(before)` = c(10, 12, 13, 11, 13),
+     `Population(after)` = c(12, 13, 16, 12, 12),
+     stringsAsFactors = FALSE,
+     check.names = FALSE)
> results
  ID Category Population(before) Population(after)
1  0        A                 10                12
2  1        A                 12                13
3  2        A                 13                16
4  3        B                 11                12
5  4        C                 13                12
> colnames(results)
[1] "ID"               "Category"         "Population(before)" "Population(after)"
```

4.1 할당 표현식

- **비표준 이름과 역따옴표 사용하기**
  - ✓ stringAsFactors = FALSE는 문자형 벡터를 요인(factors)으로 바꾸는 것을 방지.
  - ✓ check.names = FALSE는 열 이름에 make.names( ) 함수를 적용하는 것을 방지.
  - ✓ 이 두 인수를 사용하면 입력 데이터를 data.frame 객체로 만들 때 원본을 그대로 유지 가능.
  - ✓ 특수 문자가 들어간 열 이름으로 열에 접근할 때는 다음과 같이 역따옴표와 함께 사용해야 함.

```
> results$`Population(before)`
[1] 10 12 13 11 13
```

  - ✓ 요약) 직접적인 할당에 허용되지 않는 특수 문자가 들어간 변수 이름을 생성하거나 참조할 때는 역따옴표를 사용.
  - ✓ 이러한 이름 사용을 권장하는 것은 아님.
  - ✓ 오히려 이것들이 코드 가독성을 저해하고, 오류 원인이 될 수 있음.
  - ✓ 기본 명명 규칙만 허용하는 다른 외부 도구와 함께 사용하기가 더 어려워질 수 있음.
  - ✓ 결론적으로 정말 필요한 경우가 아니라면 되도록 역따옴표는 사용하지 않는 것이 좋음.

4.2 조건 표현식

4.2 조건 표현식

- **조건 표현식**
  - ✓ 프로그램 논리가 완벽히 순차적일 때보다는 특정 조건에 따라 분기를 포함할 때가 더욱 일반적.
  - ✓ 전형적인 프로그래밍 언어의 기본 구조 가운데 하나는 바로 조건 표현식.
  - ✓ R에서는 논리 조건에 따라 논리 흐름을 분기하는 데 if를 사용.

4.2 조건 표현식

- **if 문 사용하기**
  - ✓ 다른 대부분의 프로그래밍 언어와 마찬가지로 if 표현식은 논리적 조건과 함께 사용.
  - ✓ R에서 논리 조건은 단일 요소 논리형 벡터를 생성하는 표현식으로 나타냄.
  - ✓ 예를 들어 양수가 입력되면 1을 반환하고, 그렇지 않으면 아무것도 반환하지 않는 간단한 check_positive() 함수를 작성해 보자.

```
> check_positive <- function(x) {
+     if (x > 0) {
+         return(1)
+     }
+ }
```

  - ✓ 위 함수에서 x > 0 이 따져 보아야 할 조건식이며, 조건이 충족되면 함수는 1을 반환.
  - ✓ 다양하게 입력하여 함수를 확인해 보자.

```
> check_positive(1)
[1] 1
> check_positive(0)
```

4.2 조건 표현식

- **if 문 사용하기**
  - ✓ else if 와 else 분기를 추가하면 양수 입력에서는 1, 음수 입력에서는 -1, 0 에서는 0 을 반환하는 일반적인 sign 함수를 만들 수 있음.

```
> check_sign <- function(x) {
+     if (x > 0) {
+         return(1)
+     } else if (x < 0) {
+         return(-1)
+     } else {
+         return(0)
+     }
+ }
> check_sign(15)
[1] 1
> check_sign(-3.5)
[1] -1
> check_sign(0)
[1] 0
```

4.2 조건 표현식

- **if 문 사용하기**
  - ✓ 모든 함수가 항상 어떤 값을 반환할 필요는 없음.
  - ✓ 다양한 조건에 따라 아무것도 반환하지 않는(더 정확하게는 NULL을 반환하는) 작업 수행 가능.
  - ✓ 다음 함수는 결과값을 항상 명시적으로 반환하는 대신 콘솔 창에 메시지를 출력.
  - ✓ 입력한 숫자 부호에 따라 메시지가 달라짐.

```
> say_sign <- function(x) {
+     if (x > 0) {
+         cat("The number is greater than 0")
+     } else if (x < 0) {
+         cat("The number is less than 0")
+     } else {
+         cat("The number is 0")
+     }
+ }
> say_sign(0)
The number is 0
> say_sign(3)
The number is greater than 0
> say_sign(-9)
The number is less than 0
```

4.2 조건 표현식

- **if 문 사용하기**
  - ✓ if 문 분기를 평가하는 과정
    1. if (cond1) {expr1}에서 cond1을 평가.
    2. cond1이 TRUE이면 해당 표현식 {expr1}을 수행.
       그렇지 않으면 else if (cond2) 분기에서 cond2 조건을 평가.
       이것도 만족하지 않으면 다음 조건으로 넘어 감.
    3. 모든 if 조건과 else if 조건을 만족하지 않으면 else 분기의 표현식을 수행.

4.2 조건 표현식

- **if 문 사용하기**
  - ✓ 가장 간단한 형태는 단순히 if 문만 이용한 분기.

```
if (cond1) {
    # do something
}
```

  - ✓ cond1 조건이 참이 아닌 상황을 다루는 else 문을 추가하면 더욱 형태가 완벽.

```
if (cond1) {
    # do something
} else {
    # do something else
}
```

  - ✓ 하나 이상의 else if 문을 추가하면 더 복잡한 형태도 가능.

```
if (cond1) {
    expr1
} else if (cond2) {
    expr2
} else if (cond3) {
    expr3
} else {
    expr4
}
```

4.2 조건 표현식

- **if 문 사용하기**
  - ✓ 앞의 조건부 분기에서 분기 조건(cond1, cond2, cond3)은 관련이 있을 수도 있고 없을 수도 있음.
  - ✓ 시험 성적에 따라 분기 조건을 만드는 경우가 여기에 완벽하게 어울림.

```
> grade <- function(score) {
+     if (score >= 90) {
+         return("A")
+     } else if (score >= 80) {
+         return("B")
+     } else if (score >= 70) {
+         return("C")
+     } else if (score >= 60) {
+         return("D")
+     } else {
+         return("F")
+     }
+ }
> c(grade(65), grade(59), grade(87), grade(96))
[1] "D" "F" "B" "A"
```

  - ❖ else if 문에 해당하는 분기 조건에서는 이전 조건들이 모두 만족되지 않았다고 가정.
  - ❖ 즉, score >= 80은 실제로는 이전 조건에 따라 점수가 90점 미만이고 동시에 80점 이상인 경우를 의미.
  - ❖ 이 가정을 모두 고려하여 모든 분기 조건을 독립적으로 만들지 않는 이상 분기 순서 변경 불가.

4.2 조건 표현식

- **if 문 사용하기**
  - ✓ 분기 조건의 순서를 다음과 같이 조금 바꾸어 보자.

```
> grade2 <- function(score) {
+     if (score >= 60) {
+         return("D")
+     } else if (score >= 70) {
+         return("C")
+     } else if (score >= 80) {
+         return("B")
+     } else if (score >= 90) {
+         return("A")
+     } else {
+         return("F")
+     }
+ }
> c(grade2(65), grade2(59), grade2(87), grade2(96))
[1] "D" "F" "D" "D"
```

  - ❖ 보다시피 grade2(59)를 제외한 나머지는 모두 결과가 잘못됨.

4.2 조건 표현식

- **if 문 사용하기**
  - ✓ 조건 순서를 재조정하지 않고 이 함수를 올바르게 고치려면 순서에 영향을 받지 않도록 조건들을 다시 만들어야 함.

```
> grade2 <- function(score) {
+     if (score >= 60 && score < 70) {
+         return("D")
+     } else if (score >= 70 && score < 80) {
+         return("C")
+     } else if (score >= 80 && score < 90) {
+         return("B")
+     } else if (score >= 90) {
+         return("A")
+     } else {
+         return("F")
+     }
+ }
> c(grade2(65), grade2(59), grade2(87), grade2(96))
[1] "D" "F" "B" "A"
```

    - ❖ 처음 버전보다 코드가 더 길어짐.
    - ❖ 분기 순서를 잘 따져 보고 분기별 독립성을 고려하는 것이 중요.

4.2 조건 표현식

- **if 조건식 사용하기**
  - ✓ 본질적으로 if 는 원시 함수이기 때문에 이 함수의 반환값은 조건을 만족하는 분기의 표현식 값.
  - ✓ if 도 인라인 표현식으로 사용 가능.
  - ✓ check_positive( ) 함수를 예로 들어 보자.
    - ❖ 조건식에 return( )을 작성하는 대신 함수 본문에서 if 문 안의 값을 반환하여 동일한 결과를 얻을 수 있음.

```
> check_positive <- function(x) {
+     return(if (x > 0) {
+         1
+     })
+ }
```

  - ✓ 사실 표현식 구문은 한 줄로도 표현 가능.

```
> check_positive <- function(x) {
+     return(if (x > 0) 1)
+ }
```

    - ❖ 이 코드에서 함수의 반환값은 함수 본문의 마지막 표현식 값.
  - ✓ 이때 return() 제거 가능.

```
> check_positive <- function(x) {
+     if (x > 0) 1
+ }
```

4.2 조건 표현식

- **if 조건식 사용하기**
  - ✓ check_sign() 함수에도 동일한 원칙이 적용.
  - ✓ 다음과 같이 check_sign() 함수 단순화 가능.

```
> check_sign <- function(x) {
+     if (x > 0) 1 else if (x < 0) -1 else 0
+ }
```

  - ✓ 명시적으로 if 표현식의 값을 얻는 방법을 활용하면 학생 이름과 점수를 고려하여 성적을 구하는 함수도 구현 가능.

```
> say_grade <- function(name, score) {
+     grade <- if (score >= 90) "A"
+     else if (score >= 80) "B"
+     else if (score >= 70) "C"
+     else if (score >= 60) "D"
+     else "F"
+     cat("The grade of", name, "is", grade)
+ }
> say_grade("Betty", 86)
The grade of Betty is B
```

4.2 조건 표현식

- **if 조건식 사용하기**
  - ✓ 조건과 분기가 더욱 복잡해질 때, 서로 다른 분기를 좀 더 명확하게 기술하고자 구문을 사용하고 불필요한 실수를 피하려면 { }를 생략하지 않는 것이 좋음.
  - ✓ 다음은 나쁜 함수의 예.

```
> say_grade <- function(name, score) {
+     if (score >= 90) grade <- "A"
+     cat("Congratulations!\n")
+     else if (score >= 80) grade <- "B"
+     else if (score >= 70) grade <- "C"
+     else if (score >= 60) grade <- "D"
+     else grade <- "F"
+     cat("What a pity!\n")
+     cat("The grade of", name, "is", grade)
+ }
```

    - ❖ 이 함수 작성자는 특정 부분에 어떤 문장을 출력하길 원함.
    - ❖ 조건부 분기를 둘러싸는 괄호가 없으면 해당 분기에 원하는 동작을 추가할 때 작성한 구문에서 오류가 발생할 가능성이 높음.
    - ❖ 콘솔 창에서 앞 코드를 그대로 실행하면 알 수 없는 오류가 수없이 발생하는 것을 볼 수 있음.

4.2 조건 표현식

- **if 조건식 사용하기**

```
> say_grade <- function(name, score) {
+     if (score >= 90) grade <- "A"
+     cat("Congratulations!\n")
+     else if (score >= 80) grade <- "B"
Error: unexpected 'else' in:
"    cat("Congratulations!\n")
    else"
>     else if (score >= 70) grade <- "C"
Error: unexpected 'else' in "    else"
>     else if (score >= 60) grade <- "D"
Error: unexpected 'else' in "    else"
>     else grade <- "F"
Error: unexpected 'else' in "    else"
>     cat("What a pity!\n")
What a pity!
>     cat("The grade of", name, "is", grade)
Error in cat("The grade of", name, "is", grade) : object 'name' not found
> }
Error: unexpected '}' in "}"
> }
Error: unexpected '}' in "}"
```

4.2 조건 표현식

- **if 조건식 사용하기**
  - ✓ 잠재적 위험을 피하는 더 나은 형태는 다음과 같음.

```
> say_grade <- function(name, score) {
+     if (score >= 90) {
+         grade <- "A"
+         cat("Congratulations!\n")
+     } else if (score >= 80) {
+         grade <- "B"
+     }
+     else if (score >= 70) {
+         grade <- "C"
+     }
+     else if (score >= 60) {
+         grade <- "D"
+     } else {
+         grade <- "F"
+         cat("What a pity!\n")
+     }
+     cat("The grade of", name, "is", grade)
+ }
> say_grade("James", 93)
Congratulations!
The grade of James is A
```

4.2 조건 표현식

- **벡터에 if 문 사용하기**
  - ✓ 앞에서 작성한 모든 예제 함수는 단일 입력에서만 동작.
  - ✓ 이들 함수에 벡터를 입력하면 다중 요소 벡터에서 제대로 동작하지 않으므로 경고가 발생.

```
> check_positive(c(1, -1, 0))
[1] 1
Warning message:
In if (x > 0) 1 :
  the condition has length > 1 and only the first element will be used
```

  - ✓ 앞 출력의 결과에서 다중 요소 논리형 벡터를 입력하면 if 문은 첫 번째 요소를 제외한 다른 모든 요소는 무시한다는 것을 알 수 있음.

```
> num <- c(1, 2, 3)
> if (num > 2) {
+     cat("num > 2!")
+ }
Warning message:
In if (num > 2) { :
  the condition has length > 1 and only the first element will be used
```

  - ✓ 앞 코드에서는 첫 번째 요소 1 에 대한 표현식(1 > 2)만 사용된다는 경고가 발생.
  - ✓ 사실 논리형 벡터에는 TRUE 와 FALSE 값이 섞여 있을 수 있기 때문에 논리적 벡터에서 조건식을 적용한다는 의미는 명확하지 않음.

4.2 조건 표현식

- **벡터에 if 문 사용하기**
  - ✓ 일부 논리 함수는 이러한 모호함을 좀 더 분명히 하는 데 유용.
  - ✓ 예를 들어 any( ) 함수는 주어진 벡터 안에 TRUE 가 하나라도 있으면 TRUE 를 반환함

```
> any(c(TRUE, FALSE, FALSE))
[1] TRUE
> any(c(FALSE, FALSE))
[1] FALSE
```

  - ✓ 벡터 안의 값 중에서 하나라도 2 보다 클 때 메시지를 출력하고 싶다면 조건에서 any( ) 함수를 호출.

```
> if (any(num > 2)) {
+     cat("num > 2!")
+ }
num > 2!
```

4.2 조건 표현식

- **벡터에 if 문 사용하기**
  - ✓ 모든 값이 2보다 클 때만 "num > 2!" 메시지를 출력하려면 any( ) 대신 all( ) 함수 사용.

```
> if (all(num > 2)) {
+     cat("num > 2!")
+ } else {
+     cat("Not all values are greater than 2!")
+ }
Not all values are greater than 2!
```

  - ✓ if 문을 사용하여 작업 과정을 분기할 때는 조건이 단일 요소 논리형 벡터인지 확인.
  - ✓ 그렇지 않으면 예상하지 못한 결과 발생 가능.

4.2 조건 표현식

- **벡터에 if 문 사용하기**
  - ✓ 또 다른 예외는 결측값(NA ; Not Available).
  - ✓ 단일 요소 논리형 벡터이지만 이 값은 if 조건에서 아무런 표시 없이 오류를 발생할 수 있음.

```
> check <- function(x) {
+     if (all(x > 0)) {
+         cat("All input values are positive!")
+     } else {
+         cat("Some values are not positive!")
+     }
+ }
```

4.2 조건 표현식

- **벡터에 if 문 사용하기**
  - ✓ check( ) 함수는 결측값이 없는 일반적인 숫자 벡터에서는 완벽하게 동작.
  - ✓ 만약 x 인수가 결측값을 포함하면, 오류로 함수 실행이 종료될 수 있음.

```
> check(c(1, 2, 3))
All input values are positive!
> check(c(1, 2, NA, -1))
Some values are not positive!
> check(c(1, 2, NA))
Error in if (all(x > 0)) { : missing value where TRUE/FALSE needed
```

    - ❖ 이 예제에서 조건을 쓸 때는 결측 값에 주의해야 한다는 사실을 배울 수 있음.
  - ✓ 논리가 복잡하고 입력 데이터가 다양하다면 결측 값을 적절하게 처리하기 힘듦.
  - ✓ any( )나 all( ) 함수는 결측 값을 다루는 데 na.rm 인수를 제공.
  - ✓ 조건 검사를 단순화하는 한 가지 방법은 isTRUE(x)를 사용하는 것.
    - ❖ 이는 내부적으로 identical(TRUE, x)를 호출.
    - ❖ 이 경우 x가 오직 TRUE일 때만 이 조건을 만족하고, 나머지 경우에는 만족하지 않음.

4.2 조건 표현식

- **벡터화된 if: ifelse**
  - ✓ ifelse( ) 함수는 논리형 벡터를 조건으로 받아 다시 벡터를 반환.
  - ✓ 논리형 벡터의 각 요소 값이 TRUE 이면, 두 번째 yes 인수에 해당하는 요소를 선택.
  - ✓ 값이 FALSE 이면 세 번째 no 인수에 해당하는 요소가 선택.
  - ✓ 즉, ifelse( ) 함수는 if 조건문의 벡터화된 버전이라고 볼 수 있음.

```
> ifelse(c(TRUE, FALSE, FALSE), c(1, 2, 3), c(4, 5, 6))
[1] 1 5 6
```

4.2 조건 표현식

- **벡터화된 if: ifelse**
  - ✓ yes와 no 인수는 재활용이 가능하므로 ifelse( )를 사용하여 check_positive( )를 다음과 같이 구현 가능.

```
> check_positive2 <- function(x) {
+     ifelse(x, 1, 0)
+ }
```

  - ✓ if 문을 사용한 check_positive( ) 함수와 ifelse( ) 함수를 사용한 check_positive2( ) 함수의 차이:
    - ❖ check_positive(-1)은 명시적인 반환값을 갖지 않지만, check_positive2(-1)은 0 을 반환.
    - ❖ else 구문이 없는 한 if 조건을 만족할 때만 명시적인 반환값을 갖게 됨.
    - ❖ 이와 반대로 ifelse( )는 yes/no 인수를 모두 지정해야 하기 때문에 항상 결과 벡터를 반환.
  - ✓ ifelse( ) 함수와 if 구문을 단순히 서로 바꾸어 사용한다고 해서 항상 결과가 동일하지 않음.
  - ✓ 예를 들어 조건에 따라 두 요소 벡터를 반환한다고 가정하자.
  - ✓ ifelse( )를 사용하는 경우를 생각해 보자.

```
> ifelse(TRUE, c(1,2), c(2,3))
[1] 1
```

    - ❖ yes 인수의 첫 번째 요소만 반환됨.
    - ❖ yes 인수에 할당된 벡터를 그대로 반환하려면 c(TRUE, TRUE)로 조건을 수정해야 함.
    - ❖ 이 방법은 별로 자연스럽지 않음.

4.2 조건 표현식

- **벡터화된 if: ifelse**
  - ✓ 다음과 같이 if 문을 사용하면 식이 훨씬 자연스러움.

```
> if (TRUE) c(1,2) else c(2,3)
[1] 1 2
```

  - ✓ 벡터화된 입출력을 다룰 때 또 다른 문제점이 있음.
  - ✓ 예를 들어 yes 인수가 수치형 벡터고 no 인수가 문자형 벡터인 경우, 조건에 TRUE와 FALSE 값이 섞여 있으면 출력 값이 모두 한 벡터에 담길 수 있도록 타입 변환을 통해 모든 요소의 타입을 일치시켜야 함.
  - ✓ 결국 문자형 벡터가 만들어짐.

```
> ifelse(c(TRUE, FALSE), c(1, 2), c("a", "b"))
[1] "1" "b"
```

4.2 조건 표현식

- **switch 문 사용하기**
  - ✓ if 문은 조건의 TRUE/FALSE 여부에 따라 분기 선택.
  - ✓ switch 문은 숫자 또는 문자열과 함께 사용하고 해당 입력에 따라 반환할 분기를 선택.
  - ✓ 정수 n을 입력했다고 가정하자.
  - ✓ switch문은 첫 번째 인수 다음에 n 번째 인수 값을 반환하는 식으로 동작.

```
> switch(1, "x", "y")
[1] "x"
> switch(2, "x", "y")
[1] "y"
```

  - ✓ 입력된 정수가 지정된 인수의 개수 범위를 벗어나면 명시적으로 반환되는 값이 없음.(실제로는 보이지 않는 NULL이 반환.)

```
> switch(3, "x", "y")
```

4.2 조건 표현식

- **switch 문 사용하기**
  - ✓ switch( ) 함수에 입력으로 문자열이 주어지는 경우에는 다르게 동작.
  - ✓ 입력과 일치하는 이름을 갖는 인수 값을 결과로 반환.

```
> switch("a", a = 1, b = 2)
[1] 1
> switch("b", a = 1, b = 2)
[1] 2
```

    - ❖ 첫 번째 switch 문은 a와 일치하는 a = 1 인수를 반환.
    - ❖ 두 번째 switch 문은 b와 일치하는 b = 2 인수를 반환.

  - ✓ 입력과 일치하는 인수가 없을 때는 출력되지 않지만 NULL을 반환.

```
> switch("c", a = 1, b = 2)
```

  - ✓ 가능한 모든 경우에 대비하여 다른 입력에 대해(인수 이름을 지정하지 않은) 마지막 인수 추가 가능.

```
> switch("c", a = 1, b = 2, 3)
[1] 3
```

4.2 조건 표현식

- **switch 문 사용하기**
  - ✓ ifelse 함수와 비교했을 때 switch는 if 와 좀 더 유사.
  - ✓ 단일 값(숫자나 문자열)을 입력으로 받으며 어떤 값이든 반환 가능.

```
> switch_test <- function(x) {
+     switch(x,
+         a = c(1, 2, 3),
+         b = list(x = 0, y = 1),
+         c= {
+             cat("You choose c!\n")
+             list(name = "c", value = "something")
+         })
+ }
> switch_test("a")
[1] 1 2 3
> switch_test("b")
$x
[1] 0

$y
[1] 1
```

```
> switch_test("c")
You choose c!
$name
[1] "c"

$value
[1] "something"
```

4.3 반복 표현식

4.3 반복 표현식

- **반복 표현식**
  - ✓ 반복 표현식(또는 루프문)은 어떤 벡터 안의 요소를 순차적으로 반복하거나(for), 어떤 조건을 만족하는지 확인하면서(while) 반복적으로 내부 표현식을 평가.
  - ✓ 이러한 언어 구조는 매번 입력을 변경하면서 동일한 작업을 반복적으로 실행해야 할 때 코드가 중복되는 것을 크게 줄일 수 있음.

4.3 반복 표현식

- **for 루프 사용하기**
  - ✓ for 루프는 주어진 벡터나 리스트를 반복하면서 표현식을 평가.
  - ✓ for 루프의 구문은 다음과 같음.

```
for (var in vector) {
    expr
}
```

  - ✓ 벡터 안의 원소를 차례대로 var 에 할당하여 expr 을 반복적으로 평가.
  - ✓ 벡터에 요소가 n 개 있다면 앞의 반복문은 다음과 같이 동작.

```
var <- vector[[1]]
expr
var <- vector[[2]]
expr
...
var <- vector[[n]]
expr
```

4.3 반복 표현식

- **for 루프 사용하기**
  - ✓ 예를 들어 i 변수를 사용하여 1:3 벡터를 순차적으로 반복하는 루프도 만들 수 있음.
  - ✓ 반복할 때마다 i 값을 화면에 출력.

```
> for (i in 1:3) {
+     cat("The value of i is", i, "\n")
+ }
The value of i is 1
The value of i is 2
The value of i is 3
```

  - ✓ 반복문은 모든 벡터에 적용 가능.

```
> for (word in c("hello","new", "world")) {
+     cat("The current word is", word, "\n")
+ }
The current word is hello
The current word is new
The current word is world
```

4.3 반복 표현식

- **for 루프 사용하기**
  - ✓ 물론 벡터 대신 리스트도 사용 가능.

```
> loop_list <- list(
+     a = c(1, 2, 3),
+     b = c("a", "b", "c", "d"))
> for (item in loop_list) {
+     cat("item:\n length:", length(item),
+     "\n class: ", class(item), "\n")
+ }
item:
 length: 3
 class:  numeric
item:
 length: 4
 class:  character
```

4.3 반복 표현식

- **for 루프 사용하기**
  - ✓ 데이터 프레임도 역시 적용 가능.

```
> df <- data.frame(
+     x = c(1, 2, 3),
+     y = c("A", "B", "C"),
+     stringsAsFactors = FALSE)
> for (col in df) {
+     str(col)
+ }
 num [1:3] 1 2 3
 chr [1:3] "A" "B" "C"
```

  - ✓ 데이터 프레임은 각 요소(열)의 길이가 동일한 리스트라고 할 수 있음.
    - ❖ 언뜻 행렬처럼 보이지만 반드시 각 열이 같은 타입일 필요는 없음.
    - ❖ 각 행은 데이터 레코드를 의미.
  - ✓ 앞 코드에서 루프는 행이 아닌 열을 따라 반복.
    - ❖ for 문을 사용하여 일반 리스트를 반복할 때 하는 동작과 동일.
    - ❖ 대부분은 데이터 프레임을 행마다 반복하고 싶어 함.
    - ❖ 이때도 1부터 데이터 프레임의 행 개수까지 나타내는 정수 벡터를 for 문과 함께 사용하여 수행 가능.

4.3 반복 표현식

- **for 루프 사용하기**
  - ✓ 데이터 프레임에서 행 번호가 주어지면 해당하는 특정 행을 찾아서 작업 수행 가능.
  - ✓ 다음 코드는 행 단위로 데이터 프레임을 반복하고, str( ) 함수를 사용하여 각 행의 구조를 출력.

```
> for (i in 1:nrow(df)) {
+     row <- df[i,]
+     cat("row", i, "\n")
+     str(row)
+     cat("\n")
+ }
row 1
'data.frame': 1 obs. of  2 variables:
 $ x: num 1
 $ y: chr "A"

row 2
'data.frame': 1 obs. of  2 variables:
 $ x: num 2
 $ y: chr "B"

row 3
'data.frame': 1 obs. of  2 variables:
 $ x: num 3
 $ y: chr "C"
```

4.3 반복 표현식

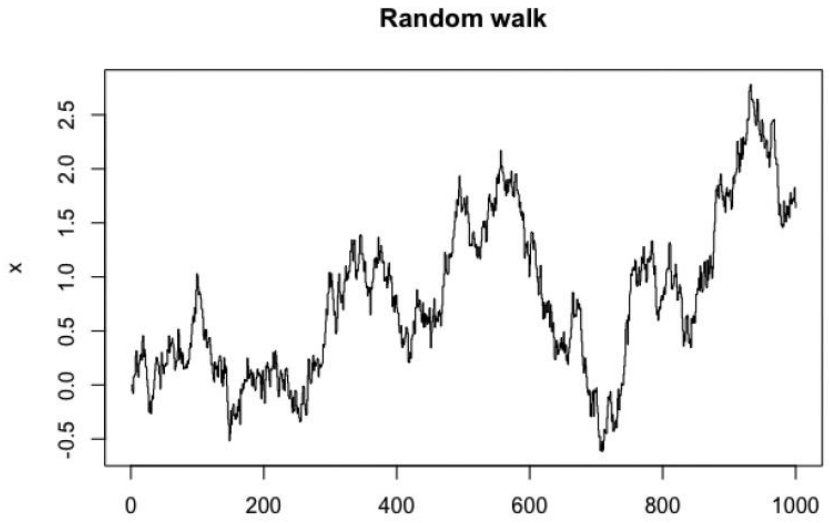
- **for 루프 사용하기**
  - ✓ 행을 따라 데이터 프레임을 반복하는 것이 일반적으로 느리고 코드가 복잡해질 수 있음.
  - ✓ 앞 예제에서 for 루프의 각 반복은 독립적.
  - ✓ 경우에 따라 루프 외부의 변수를 변경해서 특정 상태를 추적하거나 계산하여 누적된 값 유지 가능.
  - ✓ 가장 간단한 예는 1부터 100까지 합을 계산하는 것.

```
> s <- 0
> for (i in 1:100) {
+     s <- s + i
+ }
> s
[1] 5050
```

4.3 반복 표현식

- **for 루프 사용하기**
  - ✓ 다음은 정규 분포에 따라 난수를 샘플링하는 rnorm( ) 함수를 사용하여 랜덤 워크(random walk)를 간단하게 구현한 예제.
    - ❖ rnorm(n, mean=0, sd = 1) 함수에서 n 은 생성할 난수의 개수, mean 은 평균, sd 는 표준편차를 의미.

```
> set.seed(123)
> x <- numeric(1000)    # 길이가 1000이며 모든 원소 값이 0인 벡터 생성.
> for (t in 1:(length(x) - 1)) {
+     x[[t + 1]] <- x[[t]] + rnorm(1, 0, 0.1)
+ }
> plot(x, type = "s", main = "Random walk", xlab = "t")
```



4.3 반복 표현식

- **for 루프 사용하기**
  - ✓ 앞의 두 예제에서 for 루프 안의 결과는 이전 결과에 종속되어 있음.
  - ✓ sum( )이나 cumsum( ) 등 기본 함수를 사용하여 단순화 가능.

```
> sum100 <- sum(1:100)
> random_walk <- cumsum(rnorm(1000, 0, 0.1))
```

  - ✓ 기본 개념은 for 루프와 유사하지만, 벡터화되어 C 언어로 구현하기 때문에 R의 for 루프보다 훨씬 빠름.

4.3 반복 표현식

- **for 루프 흐름 제어하기**
  - ✓ 때로는 for 루프에 개입하는 것도 필요.
  - ✓ 중간에 중단하거나, 현재 반복을 건너뛰거나, 아무 작업도 하지 않고 루프를 완료하도록 선택 가능.
  - ✓ break 를 사용하여 for 문을 종료.

```
> for (i in 1:5) {
+     if (i == 3) break
+     cat("message ", i, "\n")
+ }
message  1
message  2
```

4.3 반복 표현식

- **for 루프 흐름 제어하기**
  - ✓ 예를 들어 어떤 문제의 해를 찾는 데 break 를 사용할 수 있음.
  - ✓ 다음 코드는 (i ^ 2) %% 11 == (i^ 3) %% 17 을 만족하는 1000 에서 1100 사이의 숫자를 검색.
  - ✓ 여기서 ^ 는 거듭제곱 연산자고 %% 는 나누었을 때 남는 나머지를 반환하는 나머지 연산자.

```
> m <- integer()
> for (i in 1000:1100) {
+     if ((i ^ 2) %% 11 == (i ^ 3) %% 17) {
+         m <- c(m, i)
+     }
+ }
> m
[1] 1055 1061 1082 1086 1095
```

```
> m <- integer()
> m
integer(0)
> m <- c(m, 1055)
> m
[1] 1055
> m <- c(m, 1061)
> m
[1] 1055 1061
> m <- c(m, 1082)
> m
[1] 1055 1061 1082
```

  - ✓ 조건을 만족하는 숫자 하나만 필요하다면, 기록을 저장하는 부분을 간단히 break를 사용하여 바꿀 수 있음.

```
> for (i in 1000:1100) {
+     if ((i ^ 2) %% 11 == (i ^ 3) %% 17) break
+ }
> i
[1] 1055
```

  - ✓ 만족하는 해를 하나 찾으면 동시에 for 루프가 중단되고 현재 환경에 i 의 마지막 값이 남아 있기 때문에 조건을 만족하는 해를 알 수 있음.

4.3 반복 표현식

- **for 루프 흐름 제어하기**
  - ✓ 어떤 경우에는 for 루프에서 반복을 건너뛰는 것도 필요.
  - ✓ next 를 사용하여 현재 반복에서 나머지 표현식은 건너뛰고 루프의 다음 반복으로 바로 넘어갈 수 있음.

```
> for (i in 1:5) {
+     if (i == 3) next
+     cat("message ", i, "\n")
+ }
message  1
message  2
message  4
message  5
```

4.3 반복 표현식

- **중첩 for 루프 만들기**
  - ✓ for 루프에는 또 다른 for 루프도 들어올 수 있음.
  - ✓ 예를 들어 벡터 요소의 모든 순열을 출력하고자 함.
  - ✓ 이 문제는 두 단계로 중첩된 for 루프를 작성하여 해결 가능.

```
> x <- c("a", "b", "c")
> combx <- character()
> for (c1 in x) {
+     for (c2 in x) {
+         combx <- c(combx, paste(c1, c2, sep = ",", collapse = ""))
+     } }
> combx
[1] "a,a" "a,b" "a,c" "b,a" "b,b" "b,c" "c,a" "c,b" "c,c"
```

4.3 반복 표현식

- **중첩 for 루프 만들기**
  - ✓ 서로 다른 요소가 포함된 순열이 필요하다면 내부의 for 루프에 다음 조건식을 추가.

```
> combx2 <- character()
> for (c1 in x) {
+     for (c2 in x) {
+         if (c1 == c2) next
+         combx2 <- c(combx2, paste(c1, c2, sep = ",", collapse = ""))
+     } }
> combx2
[1] "a,b" "a,c" "b,a" "b,c" "c,a" "c,b"
```

  - ✓ 이 조건을 반대로 하여 내부 for 루프의 표현식을 다음 코드로 바꾸어도 정확히 같은 결과를 얻음.

```
> if (c1 != c2) {
+     combx2 <- c(combx2, paste(c1, c2, sep = ",", collapse = ""))
+ }
```

4.3 반복 표현식

- **중첩 for 루프 만들기**
  - ✓ 앞 코드는 중첩 루프가 동작하는 방법을 잘 보여 주지만 문제를 해결하는 최선의 방법은 아님.
  - ✓ 일부 내장 함수는 벡터 요소로 조합 또는 순열을 만드는 데 아주 유용.
  - ✓ combn( ) 함수는 원자 벡터와 각 조합에 필요한 요소 개수가 주어졌을 때 벡터 요소를 조합한 행렬 생성.

```
> combn(c("a", "b", "c"), 2)
     [,1] [,2] [,3]
[1,] "a"  "a"  "b"
[2,] "b"  "c"  "c"
```

  - ✓ expand.grid() 함수는 여러 벡터에 있는 요소의 모든 순열을 갖는 데이터 프레임 생성.

```
> expand.grid(n = c(1, 2, 3), x = c("a", "b"))
  n x
1 1 a
2 2 a
3 3 b
4 1 b
5 2 c
6 3 c
```

  - ✓ for 루프도 강력한 기능이지만, 특정 작업을 위해 설계된 함수(lapply( ) 함수)도 존재.
  - ✓ 모든 것을 for 루프에 직접 넣기보다는 내장 함수를 사용하는 것을 고려하면 좋음.

4.3 반복 표현식

- **while 루프 사용하기**
  - ✓ for 루프와 달리 while 루프는 주어진 조건을 만족하는 동안에는 실행을 멈추지 않음.
  - ✓ 예를 들어 다음 while 루프는 x = 0 으로 시작한다. 매번 루프는 x가 x <= 5 인지 확인.
  - ✓ 조건을 만족하면 내부 표현식을 실행.
  - ✓ 조건을 만족하지 않으면 while 루프를 종료.

```
> x <- 0
> while (x <= 5) {
+     cat(x, " ", sep = "")
+     x <- x + 1
+ }
0 1 2 3 4 5
```

4.3 반복 표현식

- **while 루프 사용하기**
  - ✓ x가 더 이상 증가하지 않도록 x <- x + 1 부분을 제거한다면, R 을 강제로 종료할 때까지 계속 실행됨.
  - ✓ while 루프는 정확히 구현하지 않으면 위험할 수 있음.
  - ✓ for 루프와 마찬가지로 흐름 제어 명령문(break와 next)은 while에도 그대로 사용 가능.

```
> x <- 0
> while (TRUE) {
+     x <- x + 1
+     if (x == 4) break
+     else if (x == 2) next
+     else cat(x, '\n')
+ }
1
3
```

4.3 반복 표현식

- **while 루프 사용하기**
  - ✓ 실제로 while 루프는 반복 횟수를 정확히 알 수 없을 때 주로 사용.
  - ✓ 데이터베이스 쿼리의 결과 집합을 여러 부분으로 나누어 행으로 불러올 때 이러한 상황이 많이 발생.

```
res <- dbSendQuery(con, "SELECT * FROM table1 WHERE type = 1")
while (!dbHasCompleted(res)) {
    chunk <- dbFetch(res, 10000)
    process(chunk)
}
```

  - ✓ 먼저 con을 연결하여 table 1에서 type 이 1인 모든 레코드를 수집하는 쿼리를 요청.
  - ✓ 데이터베이스에서 결과 res가 반환되면 여러 그룹으로 나누어 한 번에 하나씩 처리.
  - ✓ 레코드 개수가 얼마나 될지 쿼리를 요청하기 전까지는 알 수 없기 때문에 dbHasCompleted( ) 함수를 써서 모든 데이터를 완전히 가져왔을 때 끝나는 while 루프를 사용.
    - ❖ 이러한 방식으로 거대한 데이터 프레임을 한꺼번에 메모리로 가져오는 것을 방지.
    - ❖ 작은 그룹 단위로 나누어 작업함으로써, 적은 양의 메모리를 사용하여 대량의 데이터 처리 가능.
    - ❖ 물론 이것은 알고리즘 부분인 process() 함수가 데이터를 부분 단위로 처리한다고 전제.

4.3 반복 표현식

- **while 루프 사용하기**
  - ✓ R은 for 루프와 while 루프 외에도 repeat라는 루프 명령을 제공.
  - ✓ repeat 는 while(TRUE) 처럼 명시적인 종료 조건 없이 break가 발생하지 않는 한 계속 반복.

```
> x <- 0
> repeat {
+     x <- x + 1
+     if (x == 4) break
+     else if (x == 2) next
+     else cat(x, '\n')
+ }
1
3
```

4.4 마치며

4.4 마치며

- **마치며**
  - ✓ 이 장에서는 할당, 조건, 반복 표현식을 다뤄보았습니다.
  - ✓ 할당에서는 변수의 명명 규칙과 그 외 관련 내용을 알아보았습니다.
  - ✓ 조건식에서는 if 문을 구문이나 표현식으로 사용하는 방법과 ifelse( )가 if 를 벡터로 처리할 때와 어떻게 다른지 배웠습니다.
  - ✓ 반복문에서는 for 루프와 while 루프의 유사점과 차이점을 알아보았습니다.
  - ✓ 이제 우리는 R 프로그램의 논리 흐름을 제어하는 기본 표현식에 대한 이해를 갖추었습니다.
  - ✓ 다음 장에서는 이전 장에서 배운 것을 사용하여 데이터를 나타내는 기본 객체와 논리를 나타내는 기본 표현으로 수행할 수 있는 작업들이 무엇인지 알아봅니다. 데이터 변환과 통계 분석의 기본 요소로 다양한 범주의 기본 함수도 학습합니다.