R프로그래밍

5장. 기본 객체 활용하기

박 혜 승 교 수

5장. 기본 객체 활용하기

5.1 객체 함수 사용하기

5.1 객체 함수 사용하기

- 객체 타입 알아보기

  ✓ R에는 여러가지 타입의 객체가 존재.

  ✓ 우리가 다룰 객체를 사용자가 직접 정의한다고 가정하자.입력 객체의 타입에 따라 서로 다른 방식으로 동작하는 함수가 존재한다. 입력 객체가 원자 벡터(수치형·문자형·논리형 벡터)일 때는 첫 번째 요소를 반환하지만, 입력 객체가 데이터와 인덱스의 리스트일 때는 사용자 정의 요소를 반환하는 take_it 함수를 만들어야 한다고 가정하자. 예를 들어 입력이 c(1, 2, 3) 같은 수치형 벡터일 때, 함수는 첫 번째 요소인 1을 반환해야 한다. 입력이 c("a", "b", "c") 이면 이 함수는 a를 반환해야 한다. 그러나 입력이 리스트(data = c("a", "b", "c"), index = 3)라면 함수는 세 번째 요소(index = 3), 즉 c를 반환해야 함.

```
> take_it <- function(x) {
+     if (is.atomic(x)) {
+         x[[1]]
+     } else if (is.list(x)) {
+         x$data[[x$index]]
+     } else {
+         stop("Not supported input type")
+     }
+ }
```

```
> take_it(c(1, 2, 3))
[1] 1
> take_it(list(data = c("a", "b", "c"), index = 3))
[1] "c"
```

5.1 객체 함수 사용하기

- 객체 타입 알아보기

✓ 지원하지 않는 타입의 입력이라면, 함수는 오류 메시지를 발생시키고 중지.
✓ 예를 들어 take_it 함수는 함수를 입력으로 받을 수 없음.
✓ 이 경우에 mean 함수를 인수로 전달하면 그것은 else 조건으로 넘어가서 멈춤.

```
> take_it(mean)
Error in take_it(mean) : Not supported input type
```

✓ 입력이 리스트이지만, data와 index를 포함하지 않으면 어떻게 될까?

```
> take_it(list(input = c("a", "b", "c")))
NULL
```

✓ x$data가 NULL이고, NULL에서 어떤 값을 추출하면 NULL이므로 결과는 NULL.

```
> NULL[[1]]
NULL
> NULL[[NULL]]
NULL
```

5.1 객체 함수 사용하기

- 객체 타입 알아보기

✓ 리스트에 data는 있는데, index가 없다면 다음 오류가 발생.

```
> take_it(list(data = c("a", "b", "c")))
Error in x$data[[x$index]] :
  attempt to select less than one element in get1index
```

✓ x$index가 NULL이 되고 벡터에서 NULL을 사용하여 값을 추출하면 오류가 발생하기 때문.

```
> c("a", "b", "c")[[NULL]]
Error in c("a", "b", "c")[[NULL]] :
  attempt to select less than one element in get1index
```

✓ 앞 코드는 NULL[[2]]가 NULL을 반환하는 첫 번째 경우와 조금 비슷함.

```
> take_it(list(index = 2))
NULL
```

✓ NULL이 계산에 포함되는 이러한 예외 경우에 익숙하지 않다면 이전 예제에서 보여 주는 오류 메시지에 정보가 매우 부족하다고 느낄 수 있다. 코드가 더 복잡할 때 이러한 오류가 발생하면 짧은 시간 안에 정확한 원인을 찾기가 힘들 것이다. 좋은 해결책 중 하나는 함수를 구현할 때 입력을 직접 확인하고 인수에 대한 가정을 반영하는 것이다.

5.1 객체 함수 사용하기

- 객체 타입 알아보기

✓ 앞의 잘못된 사례를 해결하고자 다음 구현에서 각 인수의 타입이 원하는 것인지 고려해 보자.

```
> take_it <- function(x) {
+     if (is.atomic(x)) {
+         x[[1]]
+     } else if (is.list(x)) {
+         x$data[[x$index]]
+     } else {
+         stop("Not supported input type")
+     }
+ }
```

```
> take_it2 <- function(x) {
+     if (is.atomic(x)) {
+         x[[1]]
+     } else if (is.list(x)) {
+         if (!is.null(x$data) && is.atomic(x$data)) {
+             if (is.numeric(x$index) && length(x) == 1) {
+                 x$data[[x$index]]
+             } else {
+                 stop("Invalid index")
+             }
+         } else {
+             stop("Invalid data")
+         }
+     } else {
+         stop("Not supported input type")
+     }
+ }
```

5.1 객체 함수 사용하기

- 객체 타입 알아보기

  ✓ 앞 코드에서는 x가 리스트일 때, 먼저 x$data가 null이 아니고 원자 벡터인지 확인.
  ✓ 다음으로 x$index가 단일 요소 수치형 벡터인지 혹은 스칼라로 올바르게 지정되었는지 확인.
  ✓ 조건 중 하나라도 만족하지 않으면 함수가 중지되면서 유용한 오류 메시지가 표시되어 사용자에게 입력 내용에 오류가 있음을 전달.

  ✓ 기본으로 내장된 검사 함수에는 기발한 기능 제공.
  ✓ 예를 들어 is.atomic(NULL)은 TRUE를 반환.
  ✓ x 리스트에 data 요소가 없으면 if (is.atomic(x$data))의 TRUE 분기가 여전히 수행될 수 있으며, 이 또한 결국 NULL이라는 결과로 이어짐.
  ✓ 이러한 인수 검사로 이 코드는 여러 상황에 좀 더 강력해졌으며, 가정을 위반하더라도 더 유용한 오류 메시지 전달 가능.

```
> take_it2(list(data = c("a", "b", "c")))
Error in take_it2(list(data = c("a", "b", "c"))) : Invalid index
> take_it2(list(index = 2))
Error in take_it2(list(index = 2)) : Invalid data
```

  ✓ 이 함수를 구현하는 또 다른 방법은 객체 지향 프로그래밍과 관련한 후반부에 등장하는 S3 디스패치를 사용하는 것.

5.1 객체 함수 사용하기

- 객체 타입 알아보기

◆ 객체의 클래스와 타입 조사하기

✓ is.* 함수 외에도 class()와 typeof() 함수를 사용하여 take_it 함수 구현 가능.
✓ 다음 코드는 여러 가지 타입의 객체에 대한 class()와 typeof()의 결과를 보여 줌.
✓ x 객체에 대해 class()와 typeof() 함수를 호출한 후 str() 함수를 사용하여 객체 구조 확인.
✓ 수치형 벡터, 정수형 벡터, 문자형 벡터의 경우는 각각 다음과 같음.

```
> x <- c(1, 2, 3)
> class(x)
[1] "numeric"
> typeof(x)
[1] "double"
> str(x)
 num [1:3] 1 2 3
```

```
> x <- 1:3
> class(x)
[1] "integer"
> typeof(x)
[1] "integer"
> str(x)
 int [1:3] 1 2 3
```

```
> x <- c("a", "b", "c")
> class(x)
[1] "character"
> typeof(x)
[1] "character"
> str(x)
 chr [1:3] "a" "b" "c"
```

5.1 객체 함수 사용하기

- 객체 타입 알아보기

◆ **객체의 클래스와 타입 조사하기 (cont.)**

✓ 리스트, 데이터 프레임의 경우는 각각 다음과 같음.

```
> x <- list(a = c(1, 2), b = c(TRUE, FALSE))
> class(x)
[1] "list"
> typeof(x)
[1] "list"
> str(x)
List of 2
 $ a: num [1:2] 1 2
 $ b: logi [1:2] TRUE FALSE
```

```
> x <- data.frame(a = c(1, 2), b = c(TRUE, FALSE))
> class(x)
[1] "data.frame"
> typeof(x)
[1] "list"
> str(x)
'data.frame':    2 obs. of  2 variables:
 $ a: num  1 2
 $ b: logi  TRUE FALSE
```

5.1 객체 함수 사용하기

- 객체 타입 알아보기

◆ **객체의 클래스와 타입 조사하기 (cont.)**

✓ typeof() 함수는 객체 하위 레벨의 내부 타입을 반환.

✓ class() 함수는 객체 상위 레벨의 클래스를 반환.

✓ data.frame은 본질적으로 길이가 같은 요소들로 된 리스트 객체.

✓ 데이터 프레임은 데이터 프레임에 관련한 함수가 이를 인식할 수 있게 data.frame이라는 클래스를 갖지만, typeof() 함수는 여전히 이것이 내부적으로는 리스트라는 사실을 알려 줌.

5.1 객체 함수 사용하기

- 데이터 차원 조사하기

✓ 행렬, 배열, 데이터 프레임은 차원과 관련한 속성을 갖고 있음.

◆ 행렬의 데이터 차원 구하기

✓ R에서 벡터는 1차원 데이터 구조를 갖음.

```
> vec <- c(1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5, 6)
> class(vec)
[1] "numeric"
> typeof(vec)
[1] "double"
```

5.1 객체 함수 사용하기

- 데이터 차원 조사하기

◆ **행렬의 데이터 차원 구하기 (cont.)**

✓ 2차원 데이터는 dim(), nrow(), ncol() 함수 활용 가능.

```
> sample_matrix <- matrix(vec, ncol = 4)
> sample_matrix
     [,1]   [,2]    [,3]     [,4]
[1,]    1      2       3        4
[2,]    2      3       4        5
[3,]    3      4       5        6
> class(sample_matrix)
[1] "matrix"
> typeof(sample_matrix)
[1] "double"
> dim(sample_matrix)
[1] 3 4
> nrow(sample_matrix)
[1] 3
> ncol(sample_matrix)
[1] 4
```

5.1 객체 함수 사용하기

- 데이터 차원 조사하기

◆ **행렬의 데이터 차원 구하기 (cont.)**

✓ 앞 코드에서 첫 표현식은 수치형 벡터 vec에서 4열로 구성된 행렬 생성.
✓ 이 행렬은 matrix라는 클래스를 가지며, typeof() 결과는 vec에서 double을 그대로 상속.
✓ 행렬은 차원을 갖는 데이터 구조이므로 dim() 함수는 행렬의 차원을 벡터 형태로 반환.
✓ nrow()와 ncol() 함수는 각각 행과 열의 개수를 반환.
✓ 두 함수의 결과는 각각 dim() 함수 결과의 첫 번째, 두 번째 요소와 값이 같음.

5.1 객체 함수 사용하기

- 데이터 차원 조사하기

◆ **배열의 데이터 차원 구하기**

✓ 차원이 더 높은 데이터라면 일반적으로 배열로 표현.
✓ 예를 들어 vec은 3차원으로 표현 가능.
✓ 즉, 한 요소에 접근하려면 3차원에 해당하는 위치 정보가 3개 필요.

```
> sample_array <- array(vec, dim = c(2, 3, 2))
> sample_array
, , 1

     [,1]    [,2]    [,3]
[1,]    1       3       3
[2,]    2       2       4

, , 2

     [,1]    [,2]    [,3]
[1,]    3       5       5
[2,]    4       4       6
```

```
> class(sample_array)
[1] "array"
> typeof(sample_array)
[1] "double"
> dim(sample_array)
[1] 2 3 2
> nrow(sample_array)
[1] 2
> ncol(sample_array)
[1] 3
```

5.1 객체 함수 사용하기

- 데이터 차원 조사하기

◆ 배열의 데이터 차원 구하기 (cont.)

✓ 배열 역시 array 클래스를 갖지만, vec에서 물려받은 double이라는 데이터 타입을 유지.

✓ dim() 함수의 결과 벡터 길이는 데이터의 차원 수와 같음.

◆ 데이터 프레임의 데이터 차원 구하기

✓ 행렬이 벡터에 차원 속성을 추가하여 생성되는 반면, 데이터 프레임은 각 요소의 길이가 같은 리스트로부터 생성.

✓ dim(), nrow(), ncol() 함수 모두 데이터 프레임에서도 동일하게 활용 가능.

```
> sample_data_frame <- data.frame(a = c(1, 2, 3), b = c(2, 3, 4))
> class(sample_data_frame)
[1] "data.frame"
> typeof(sample_data_frame)
[1] "list"
> dim(sample_data_frame)
[1] 3 2
> nrow(sample_data_frame)
[1] 3
> ncol(sample_data_frame)
[1] 2
```

5.1 객체 함수 사용하기

- 데이터 차원 조사하기

◆ **데이터 구조 재조정하기**

✓ dim(x) <- y 구문은 x의 차원을 y로 변경하겠다는 의미.

✓ 평범한 벡터에 이 표현식을 적용하면 특정 차원을 갖는 행렬로 변환 가능.

```
> sample_data <- vec
> dim(sample_data) <- c(3, 4)
> sample_data
     [,1]   [,2]   [,3]   [,4]
[1,]    1      2      3      4
[2,]    2      3      4      5
[3,]    3      4      5      6
> class(sample_data)
[1] "matrix"
> typeof(sample_data)
[1] "double"
```

✓ 객체의 클래스가 numeric에서 matrix로 변환.

✓ 객체의 타입은 여전히 double로 유지.

5.1 객체 함수 사용하기

- 데이터 차원 조사하기

◆ **데이터 구조 재조정하기 (cont.)**

✓ 행렬은 다음 표현식으로 구조를 재조정할 수 있음.

```
> dim(sample_data) <- c(4, 3)
> sample_data
     [,1]    [,2]    [,3]
[1,]    1       3       5
[2,]    2       4       4
[3,]    3       3       5
[4,]    2       4       6
```

5.1 객체 함수 사용하기

- 데이터 차원 조사하기

◆ **데이터 구조 재조정하기 (cont.)**

✓ 벡터나 행렬, 배열의 차원을 재조정하는 것은 객체를 표현하는 방법과 접근하는 방법이 달라지는 것일 뿐, 메모리에 저장된 데이터가 변경되는 것은 아님. 따라서 행렬을 다음과 같이 배열로 재조정 가능.

```
> dim(sample_data) <- c(3, 2, 2)
> sample_data
, , 1

     [,1]   [,2]
[1,]    1      2
[2,]    2      3
[3,]    3      4
, , 2
     [,1]   [,2]
[1,]    3      4
[2,]    4      5
[3,]    5      6
> class(sample_data)
[1] "array"
```

5.1 객체 함수 사용하기

- 데이터 차원 조사하기

◆ **데이터 구조 재조정하기 (cont.)**

✓ dim(x) <- y는 prod(y)가 length(x)와 같을 때만 동작.

✓ 즉, 모든 차원 크기의 곱(3*2*2)이 데이터 요소의 길이와 같아야 함.

✓ 그렇지 않으면 다음과 같이 오류 발생.

```
> dim(sample_data) <- c(2, 3, 4)
Error in dim(sample_data) <- c(2, 3, 4) :
  dims [product 24] do not match the length of object [12]
```

5.1 객체 함수 사용하기

- 데이터 차원 조사하기

◆ **차원 반복하기**

✓ 데이터 프레임은 주로 레코드를 모아 놓은 것이며, 각 행은 레코드를 의미.

✓ 데이터 프레임에 저장된 모든 레코드를 반복하는 작업이 일반적.

✓ 다음 데이터 프레임을 살펴보자.

```
> sample_data_frame
  a b
1 1 2
2 2 3
3 3 4
```

5.1 객체 함수 사용하기

- 데이터 차원 조사하기

◆ **차원 반복하기 (cont.)**

✓ 이 데이터 프레임에 대해 `1:nrow(x)`에 따라 `for` 루프를 사용하여 변수 값들을 출력하는 작업을 행을 따라 반복 가능.

```
> sample_data_frame
  a b
1 1 2
2 2 3
3 3 4
```

```
> for (i in 1:nrow(sample_data_frame)) {
+     # sample text:
+     # row #1, a: 1, b: 2
+     cat("row #", i, ", ",
+         "a: ", sample_data_frame[i, "a"],
+         ", b: ", sample_data_frame[i, "b"],
+         "\n", sep = "")
+ }
row #1, a: 1, b: 2
row #2, a: 2, b: 3
row #3, a: 3, b: 4
```

5.2 논리 함수 사용하기

5.2 논리 함수 사용하기

- 논리 연산자

[표 5-1] R의 논리 연산자

기호	설명	예제	결과
&	벡터화한 AND	c(T, T) & c(T, F)	c(TRUE, FALSE)
|	벡터화한 OR	c(T, T) | c(T, F)	c(TRUE, TRUE)
&&	일변량 AND	c(T, T) && c(F, T)	FALSE
||	일변량 OR	c(T, T) || c(F, T)	TRUE
!	벡터화한 NOT	!c(T, F)	c(FALSE, TRUE)
%in%	벡터화한 IN	c(1, 2) %in% c(1, 3, 4, 5)	c(TRUE, FALSE)

5.2 논리 함수 사용하기

- 논리 연산자

  - ✓ 조건문에서 &&와 ||은 단일 요소 논리형 벡터를 산출하는 데 필요한 논리 연산을 수행할 때 자주 사용.
  - ✓ &&를 사용할 때 발생할 수 있는 잠재적 위험은 다중 요소 벡터로 작업하면 양쪽 벡터의 첫 번째 요소를 제외한 모든 요소를 자동으로 무시한다는 것.
  - ✓ 다음 코드는 조건문에서 && 또는 &를 사용할 때 어떤 차이점이 있는지 보여준다.

```
> test_direction <- function(x, y, z) {
+     if (x < y & y < z) 1
+     else if (x > y & y > z) -1
+     else 0
+ }
```

  - ✓ x, y, z의 값이 단조 증가하면 1을 반환.
  - ✓ x, y, z의 값이 단조 감소하면 -1을 반환.
  - ✓ 이 함수에서는 &를 사용하여 벡터화한 AND 연산을 수행.

5.2 논리 함수 사용하기

- 논리 연산자

✓ 함수의 인수가 스칼라 값이라면 다음과 같이 완벽히 동작.

```
> test_direction(1, 2, 3)
[1] 1
```

✓ &는 벡터화한 연산자이기 때문에 인수 가운데 하나가 여러 요소를 갖는 벡터일 때는 다중 요소 벡터를 반환.

✓ if는 단일 요소 논리형 벡터만 적용 가능하기 때문에 다중 요소 벡터가 온다면 경고 발생.

```
> test_direction(c(1, 2), c(2, 3), c(3, 4))
[1] 1
Warning message:
In if (x < y & y < z) 1 else if (x > y & y > z) -1 else 0 :
  the condition has length > 1 and only the first element will be used
```

5.2 논리 함수 사용하기

- 논리 연산자

✓ test_direction 함수에 있는 &를 &&로 대체하여 test_direction2 함수를 만든다면?

```
> test_direction2 <- function(x, y, z) {
+     if (x < y && y < z) 1
+     else if (x > y && y > z) -1
+     else 0
+ }
```

✓ 이제 두 함수는 서로 다른 성질을 보이겠지만, 여전히 스칼라 입력에서는 같은 결과를 보임.

```
> test_direction2(1, 2, 3)
[1] 1
```

✓ 다중 요소 입력에서는 각 입력 벡터의 두 번째 요소부터 조용히 무시되며, 어떤 경고도 발생시키지 않음.

```
> test_direction2(c(1, 2), c(2, 3), c(3, 4))
[1] 1
```

✓ 결국 &와 && 가운데 무엇을 사용하는 것이 맞을까?

✓ 입력 벡터의 모든 위치의 요소들이 단조성을 갖는지 알아보고 싶을 때는 모두 적당하지 않음.

✓ 다음 절에서 다룰 논리 집계 함수를 사용!

5.2 논리 함수 사용하기

- 논리 함수

◆ 논리형 벡터 집계하기

✓ any()와 all() 함수는 자주 사용되는 논리 집계 함수.
✓ any()는 입력 벡터 내 하나 이상의 요소가 참일 때 TRUE를 반환하고, 아니면 FALSE를 반환.
✓ all()는 입력 벡터의 모든 요소가 참일 때만 결과가 TRUE고, 그렇지 않으면 FALSE를 반환.

```
> x <- c(-2, -3, 2, 3, 1, 0, 0, 1, 2)
> any(x > 1)
[1] TRUE
> all(x <= 1)
[1] FALSE
```

✓ 두 함수의 공통점은 다중 요소 논리형 벡터가 아닌 TRUE 아니면 FALSE라는 단일 값만 반환.
✓ 이전 절의 요구 사항을 모두 만족하는 함수를 만들 때는 all() 함수와 &연산자를 if 조건문과 함께 사용 가능.

```
> test_all_direction <- function(x, y, z) {
+     if (all(x < y & y < z)) 1
+     else if (all(x > y & y > z)) -1
+     else 0
+ }
```

5.2 논리 함수 사용하기

- 논리 함수

◆ 논리형 벡터 집계하기 (cont.)

✓ test_all_direction() 함수도 스칼라 값을 입력하면 test_direction(), test_direction2() 함수와 정확히 같은 결과를 갖음.

```
> test_all_direction(1, 2, 3)
[1] 1
```

✓ 벡터 입력에 대해 이 함수는 c(1, 2, 3)과 c(2, 3, 4)가 모두 단조성을 갖는지 테스트.

```
> test_all_direction(c(1, 2), c(2, 3), c(3, 4))
[1] 1
```

✓ 다음 코드는 두 번째 요소들이 단조성을 갖지 않는 반대의 경우(c(2, 4, 4))를 보여줌.

```
> test_all_direction(c(1, 2), c(2, 4), c(3, 4))
[1] 0
```

5.2 논리 함수 사용하기

- 논리 함수

◆ 논리형 벡터 집계하기 (cont.)

✓ any() 함수나 && 연산자를 사용하면 이 함수는 여러 가지로 다르게 변형할 수 있음.

```
> test_any_direction <- function(x, y, z) {
+     if (any(x < y & y < z)) 1
+     else if (any(x > y & y > z)) -1
+     else 0
+ }
> test_all_direction2 <- function(x, y, z) {
+     if (all(x < y) && all(y < z)) 1
+     else if (all(x > y) && all(y > z)) -1
+     else 0
+ }
> test_any_direction2 <- function(x, y, z) {
+     if (any(x < y) && any(y < z)) 1
+     else if (any(x > y) && any(y > z)) -1
+     else 0
+ }
```

5.2 논리 함수 사용하기

- 논리 함수

◆ **TRUE인 요소 파악하기**

✓ which() 함수는 논리형 벡터에서 TRUE인 요소의 위치(혹은 인덱스)를 알려줌.

```
> x
[1] -2 -3  2  3  1  0  0  1  2
> abs(x) >= 1.5
[1] TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE
> which(abs(x) >= 1.5)
[1] 1 2 3 4 9
```

5.2 논리 함수 사용하기

- 논리 함수

◆ **TRUE인 요소 파악하기 (cont.)**

✓ abs(x) >= 1.5가 논리형 벡터를 결과로 돌려주면 which() 함수가 이 벡터에서 값이 TRUE인 요소들의 위치를 반환하는 것을 분명히 볼 수 있음.

✓ 이 메커니즘은 논리 조건을 이용하여 벡터나 리스트의 요소를 필터링하는 방식과 아주 유사.

```
> x[x >= 1.5]
[1] 2 3 2
```

✓ 이 예제에서 x >= 1.5라는 조건은 논리형 벡터를 돌려줌.

✓ TRUE 값에 해당하는 x의 요소들을 선택.

✓ 모든 요소가 FALSE 값을 갖는 논리형 벡터 사용 가능.

✓ 이때는 어떤 요소도 선택되지 않고 길이가 0인 수치형 벡터를 반환.

```
> x[x >= 100]
numeric(0)
```

5.2 논리 함수 사용하기

- 결측 값 다루기

  ✓ ab실제 데이터에는 NA로 표시되는 결측 값이 자주 등장.

  ✓ 다음 수치형 벡터가 간단한 예.

```
> x <- c(-2, -3, NA, 2, 3, 1, NA, 0, 1, NA, 2)
```

  ✓ 결측 값으로 산술 연산을 하면 결과 역시 결측 값.

```
> x + 2
 [1] 0 -1 NA  4  5  3 NA  2  3 NA  4
```

  ✓ 논리형 벡터에서도 TRUE나 FALSE 값 뿐만 아니라 참과 거짓이 불분명한 상황을 고려하기 위해 NA 값을 허용.

```
> x > 2
 [1] FALSE FALSE    NA FALSE  TRUE FALSE    NA FALSE FALSE    NA FALSE
```

5.2 논리 함수 사용하기

- 결측 값 다루기

  ✓ 결론적으로 any()나 all() 등 논리 집계 함수 역시 결측 값을 고려.

```
> x
[1] -2 -3 NA  2  3  1 NA  0  1 NA  2
> any(x > 2)
[1] TRUE
> any(x < -2)
[1] TRUE
> any(x < -3)
[1] NA
```

  ✓ 결측 값이 들어 있는 논리형 벡터를 처리할 때 any() 함수는 입력 벡터의 요소 중 하나라도 TRUE이면 TRUE 반환.

5.2 논리 함수 사용하기

- 결측 값 다루기

  ✓ 결측 값이 있는 입력 벡터에서 TRUE인 요소가 하나도 없으면 함수 결과는 NA.
  ✓ 입력 벡터에 FALSE만 있으면 함수 결과는 FALSE.
  ✓ 이 논리를 확인할 수 있게 다음 코드를 실행해 보자.

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

5.2 논리 함수 사용하기

- 결측 값 다루기

✓ 벡터 안의 결측 값을 모두 무시하고 싶다면 함수를 호출할 때 na.rm = TRUE를 활용.

```
> any(x < -3, na.rm = TRUE)
[1] FALSE
```

✓ all() 함수에서는 비슷하면서도 다소 반대되는 논리가 적용.

```
> x
[1] -2 -3 NA  2  3  1 NA  0  1 NA  2
> all(x > -3)
[1] FALSE
> all(x >= -3)
[1] NA
> all(x < 4)
[1] NA
```

5.2 논리 함수 사용하기

- 결측 값 다루기

  ✓ 입력 벡터의 요소 중에 하나라도 FALSE이면 함수 결과는 FALSE.
  ✓ 결측 값이 있는 입력 벡터에서 어떤 요소도 FALSE가 아니면 함수는 NA를 반환.
  ✓ 입력 벡터에 TRUE만 있으면 결과는 TRUE.
  ✓ 이 논리를 확인할 수 있게 다음 코드를 실행해 보자.

```
> all(c(TRUE, FALSE, NA))
[1] FALSE
> all(c(TRUE, TRUE, NA))
[1] NA
> all(c(TRUE, TRUE))
[1] TRUE
```

  ✓ 마찬가지로 na.rm = TRUE를 사용하여 모든 결측 값을 고려하지 않을 수도 있음.

```
> all(x >= -3, na.rm = TRUE)
[1] TRUE
```

5.2 논리 함수 사용하기

- 결측 값 다루기

✓ 논리 집계 함수 외에 결측 값이 있을 때도 데이터 필터링 역시 다르게 동작.

✓ 예를 들어 다음 코드 결과에는 x >= 0으로 형성된 논리형 벡터의 해당 위치에 결측 값이 남아 있는 것을 볼 수 있음.

```
> x
[1] -2 -3 NA  2  3  1 NA  0  1 NA  2
> x[x >= 0]
[1] NA  2  3  1 NA  0  1 NA  2
```

✓ 반대로 which() 함수는 입력된 논리형 벡터에 있는 결측 값을 보존하지 않음.

```
> which(x >= 0)
[1] 4  5  6  8  9 11
```

✓ 다음 예제에서 보듯이 이렇게 얻은 인덱스로 필터링한 결과에는 결측 값이 없음.

```
> x[which(x >= 0)]
[1] 2 3 1 0 1 2
```

5.2 논리 함수 사용하기

- 논리적 강제 변환

✓ 논리형으로 입력 받아야 하는 일부 함수는 수치형 벡터 같은 비논리형 벡터도 허용.
✓ 함수 동작은 논리형 벡터를 사용할 때와 크게 다르지 않음.
✓ 이는 비논리형 벡터가 논리형 벡터로 강제 변환되기 때문임.
✓ 예를 들어 if 조건에 수치형 벡터를 넣으면 다음과 같이 강제 변환.

```
> if (2) 3
[1] 3
> if (0) 0 else 1
[1] 1
```

✓ R에서는 0이 아닌 모든 값은 TRUE로 강제 변환되고, 0만 FALSE로 변환.
✓ 문자열 값은 논리 값으로 변환이 불가능.

```
> if ("a") 1 else 2
Error in if ("a") 1 else 2 : argument is not interpretable as logical
```

5.3 수학 함수 사용하기

5.3 수학 함수 사용하기

- 기본 함수

[표 5-2] 기본 수학 함수

기호	예	값
$\sqrt{x}$	sqrt(2)	1.4142136
e^x	exp(1)	2.7182818
ln(x)	log(1)	0
$\log_{10}(x)$	log10(10)	1
$\log_2(x)$	log2(8)	3

5.3 수학 함수 사용하기

- 기본 함수

  ✓ sqrt() 함수는 실수를 입력으로 받음.
  ✓ 다만 입력으로 음수가 주어지면 결과는 NaN 값.

```
> sqrt(-1)
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
```

5.3 수학 함수 사용하기

- 기본 함수

✓ R에서 숫자 값은 유한한 값, 무한대(Inf 및 -Inf), NaN 값이 될 수 있음.
✓ 다음 코드는 무한대 값을 생성.
✓ 우선 양의 무한대를 만들어 보자.

```
> 1 / 0
[1] Inf
```

✓ 다음 코드는 음의 무한대 값을 생성.

```
> log(0)
[1] -Inf
```

✓ 숫자 값이 유한인지, 무한인지, NaN인지 확인하는 여러 테스트 함수가 존재.

```
> is.finite(1 / 0)
[1] FALSE
> is.infinite(log(0))
[1] TRUE
```

5.3 수학 함수 사용하기

- 기본 함수

  ✓ is.infinite()를 사용하여 숫자 값이 Inf인지 -Inf인지를 어떻게 확인할 수 있을까?
  ✓ R에서는 무한대 값에서도 부등식 사용 가능.

```
> 1 / 0 < 0
[1] FALSE
> 1 / 0 > 0
[1] TRUE
> log(0) < 0
[1] TRUE
> log(0) > 0
[1] FALSE
```

5.3 수학 함수 사용하기

- 기본 함수

✓ is.infinite()를 사용하여 숫자를 테스트하고 동시에 요소들을 0과 비교 가능.

```
> is.pos.infinite <- function(x) {
+     is.infinite(x) & x > 0
+ }
> is.neg.infinite <- function(x) {
+     is.infinite(x) & x < 0
+ }
> is.pos.infinite(1/0)
[1] TRUE
> is.neg.infinite(log(0))
[1] TRUE
```

✓ 입력 값이 log 함수의 범위, 즉 x > 0을 벗어나면 NaN을 경고와 함께 반환.

```
> log(-1)
[1] NaN
Warning message:
In log(-1) : NaNs produced
```

5.3 수학 함수 사용하기

- 숫자 반올림 함수

  ✓ 다음과 같이 숫자를 반올림할 때는 여러 가지 반올림 함수를 사용.

[표 5-3] 반올림 함수

기호	예	값
[x] log	ceiling(10.6)	11
[x] log	floor(9.5)	9
truncate	trunc(1.5)	1
round	round(pi,3)	3.142
유효 숫자	signif(pi, 3)	3.14

5.3 수학 함수 사용하기

- 삼각 함수

[표 5-4] 삼각 함수

기호	예	값
sin(x)	sin(0)	0
cos(x)	cos(0)	1
tan(x)	tan(0)	0
arcsin(x)	asin(1)	1.5707963
arccos(x)	acos(1)	0
arctan(x)	atan(1)	0.7853982

5.3 수학 함수 사용하기

- 삼각 함수

  ✓ R에서도 π의 숫자 버전을 제공.

  ```
  > pi
  [1] 3.141593
  ```

  ✓ 수학에서는 방정식 sin(π) = 0 을 엄격히 적용.

  ✓ 부동소수점의 정밀도 문제 때문에 R을 포함한 일반적인 수치 계산 소프트웨어에서는 동일한 수식이 정확히 0으로 연결되지는 않음.

  ```
  > sin(pi)
  [1] 1.224647e-16
  ```

  ✓ 가까운 숫자들의 크기를 비교하려면 all.equal() 함수를 대신 사용.

  ✓ sin(pi) == 0 이 FALSE를 반환하는 반면, all.equal(sin(pi), 0)은 기본 허용 오차가 1.5e-8일 때 TRUE를 반환.

5.3 수학 함수 사용하기

- 삼각 함수

  ✓ 입력이 π의 배수이면 정밀한 계산을 위해 다음 세 가지 함수를 제공.

[표 5-5] π 배수 삼각 함수

기호	예	값
sin(πx)	sinpi(1)	0
cos(πx)	cospi(0)	1
tan(πx)	tanpi(1)	0

5.3 수학 함수 사용하기

- 쌍곡선 함수

[표 5-6] 쌍곡선 함수

기호	예	값
sinh(x)	sinh(1)	1.1752012
cosh(x)	cosh(1)	1.5430806
tanh(x)	tanh(1)	0.7615942
arcsinh(x)	asinh(1)	0.8813736
arccosh(x)	acosh(1)	0
arctanh(x)	atanh(0)	0

5.3 수학 함수 사용하기

- 극한 함수

  ✓ 일반적으로 숫자들의 최댓값 또는 최솟값 계산을 자주 사용.
  ✓ 다음은 max()와 min() 함수의 간단한 사용법을 보여줌.

  [표 5-7] 극한 함수

기호	예	값
max(...)	max(1, 2, 3)	3
min(...)	min(1, 2, 3)	1

5.3 수학 함수 사용하기

- 극한 함수

✓ 이 두 함수는 인수에 다중 스칼라 뿐만 아니라 벡터를 입력해도 동작.

```
> max(c(1, 2, 3))
[1] 3
```

✓ 여러 벡터를 입력했을 때도 잘 동작.

```
> max(c(1, 2, 3),
+     c(2, 1, 2),
+     c(1, 3, 4))
[1] 4
> min(c(1, 2, 3),
+     c(2, 1, 2),
+     c(1, 3, 4))
[1] 1
```

5.3 수학 함수 사용하기

- 극한 함수

✓ max() 함수는 입력된 모든 벡터 값 중에서 최댓값을 반환.
✓ min() 함수는 입력된 모든 벡터 값 중에서 최솟값을 반환.
✓ 모든 벡터에서 각 위치의 최댓값 또는 최솟값을 구하려면 어떻게 해야 할까? pmax() 함수!

```
> pmax(c(1, 2, 3),
+      c(2, 1, 2),
+      c(1, 3, 4))
[1] 2 3 4
```

✓ 이것은 기본적으로 위치가 1인 모든 숫자에서 최댓값을 찾은 후 다음 위치 2에 해당하는 숫자 중 최댓값을 탐색.

```
> x <- list(c(1, 2, 3),
+      c(2, 1, 2),
+      c(1, 3, 4))
> c(max(x[[1]][[1]], x[[2]][[1]], x[[3]][[1]]),
+      max(x[[1]][[2]], x[[2]][[2]], x[[3]][[2]]),
+      max(x[[1]][[3]], x[[2]][[3]], x[[3]][[3]]))
[1] 2 3 4
```

✓ 이것을 병렬 최댓값이라고 함.

5.3 수학 함수 사용하기

- 극한 함수

✓ pmin() 함수는 병렬 최솟값을 찾음.

```
> pmin(c(1, 2, 3),
+      c(2, 1, 2),
+      c(1, 3, 4))
[1] 1 1 2
```

✓ 이 두 함수는 바닥 함수나 천장 함수처럼 특정 함수를 사용하여 벡터화된 함수를 신속하게 작성하는 데 매우 유용.

✓ 예를 들어 spread() 조각 함수는 입력 값이 -5보다 더 작으면 결과값은 -5가 되고, 입력 값이 -5에서 5 사이면 입력된 값을 그대로 출력. 입력 값이 5보다 크면 결과값은 5.

✓ 가장 단순한 구현에서는 조각별 구간을 분기하는 if를 사용.

```
> spread <- function(x) {
+     if (x < -5) -5
+     else if (x > 5) 5
+     else x
+ }
```

5.3 수학 함수 사용하기

- 극한 함수

✓ spread() 함수는 스칼라 입력에서는 동작하지만, 자동으로 벡터화되지는 않음.

```
> spread(1)
[1] 1
> spread(seq(-8, 8))
[1] -5
Warning message:
In if (x < -5) -5 else if (x > 5) 5 else x :
  the condition has length > 1 and only the first element will be used
```

✓ 한 가지 방법은 pmin()과 pmax() 함수를 사용하는 것.

✓ spread2() 함수는 자동으로 벡터화.

```
> spread2 <- function(x) {
+     pmin(5, pmax(-5, x))
+ }
> spread2(seq(-8, 8))
[1] -5 -5 -5 -5 -4 -3 -2 -1  0  1  2  3  4  5  5  5  5
```

5.3 수학 함수 사용하기

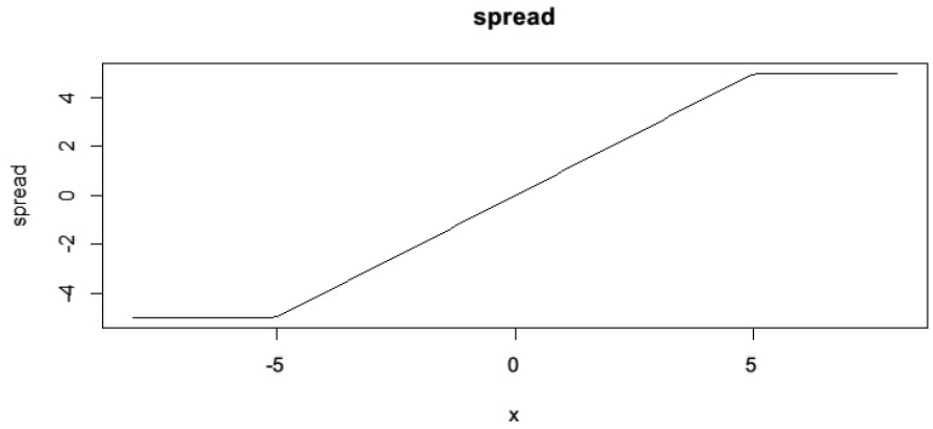
- 극한 함수

✓ 또 다른 방법은 ifelse()를 사용하는 것.

```
> spread3 <- function(x) {
+     ifelse(x < -5, -5, ifelse(x > 5, 5, x))
+ }
> spread3(seq(-8, 8))
 [1] -5 -5 -5 -5 -4 -3 -2 -1  0  1  2  3  4  5  5  5  5
```

✓ 앞의 두 함수 spread2()와 spread3()은 결과가 같음.

[그림 5-1] spread2( )와 spread3( ) 조각 함수의 입력에 따른 결과 그래프



5.4 수치 해석 활용하기

5.4 수치 해석 활용하기

- 근 구하기
  - ✓ 다음 방정식의 해를 찾아보자.
    $x^2 + x - 2 = 0$
  - ✓ 수동으로 해를 찾으려면 앞의 방정식을 다음과 같이 인수분해.
    $(x + 2)(x - 1) = 0$
  - ✓ 이 방정식의 근은 x1 = -2와 x2 = 1

5.4 수치 해석 활용하기

- 근 구하기

✓ polyroot() 함수는 다항식의 근을 반환.

$$p(x) = z_1 + z_2 x + \cdots + z_n x^{n-1}$$

✓ 앞 문제에서 방정식에 있는 0차항 계수부터 최고차항의 계수까지 다항식 계수 벡터를 지정하자.

✓ 이때 차수가 증가하는 방향으로 계수를 지정하기 위해 계수 벡터는 c(-2, 1, 1).

```
> polyroot(c(-2, 1, 1))
[1] 1-0i -2+0i
```

✓ polyroot() 함수는 항상 각 요소 값이 a + bi의 복소수 형태인 복소수 벡터를 반환.

5.4 수치 해석 활용하기

- 근 구하기

✓ 함수 결과가 반드시 실수 형태인 실근이라면 Re() 함수를 사용하여 복소근의 실수 부분만 추출 가능.

```
> Re(polyroot(c(-2, 1, 1)))
[1] 1 -2
```

✓ 한편으로 이것은 polyroot() 함수가 다항식의 복소근을 찾을 수 있음을 의미.

✓ 다음은 가장 간단한 예.

$$x^2 + 1 = 0$$

✓ 복소근을 찾으려면 다항식 계수 벡터를 지정해야 함.

```
> polyroot(c(1, 0, 1))
[1] 0+1i 0-1i
```

5.4 수치 해석 활용하기

- 근 구하기

✓ 약간 더 복잡한 예제는 다음 방정식의 근을 찾는 것.

$$x^3 - x^2 - 2x - 1 = 0$$

```
> r <- polyroot(c(-1, -2, -1, 1))
> r
[1] -0.5739495+0.3689894i -0.5739495-0.3689894i  2.1478990-0.0000000i
```

✓ 모든 복소근을 찾음.

✓ 이를 확인하기 위해 x를 r로 변경.

```
> r ^ 3 - r^ 2 - 2 * r - 1
[1] 8.881784e-16+1.110223e-16i 8.881784e-16+2.220446e-16i 8.881784e-16-4.188101e-16i
```

✓ 몇 가지 수치 계산과 관련한 특성 때문에 완벽하게 0은 아니지만 0에 매우 가까운 값을 출력.

✓ 오차의 8자리 숫자까지만 신경을 쓴다면 round() 함수를 사용하여 이 근이 유효하다는 것을 알 수 있음.

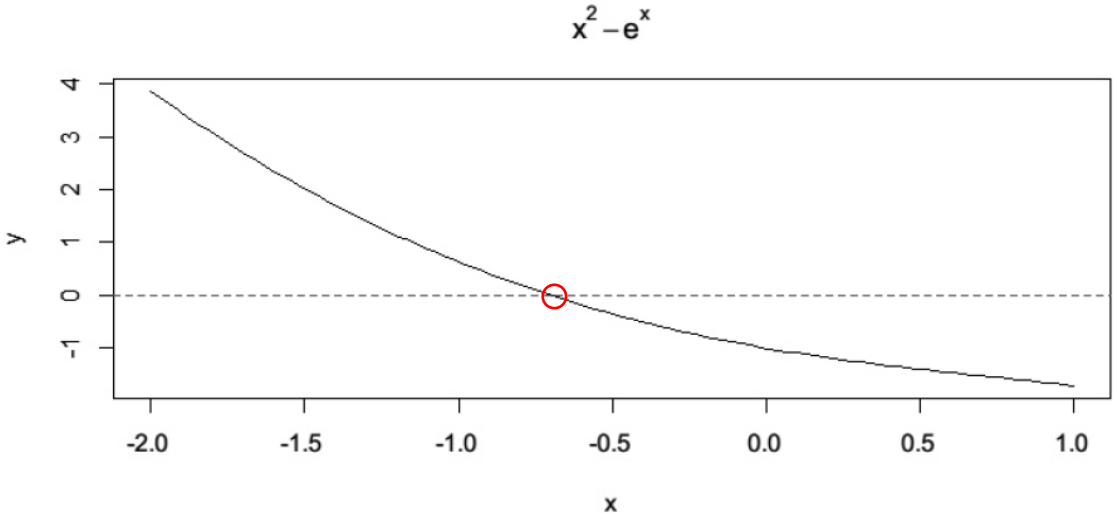
```
> round(r ^ 3 - r ^ 2 - 2 * r - 1, 8)
[1] 0+0i 0+0i 0+0i
```

5.4 수치 해석 활용하기

- 근 구하기

  ✓ uniroot() 함수는 f(x) = 0에 대한 하나의 근을 찾는 데 유용.
  ✓ 간단한 예제는 특정 범위 안에서 다음 방정식의 근을 찾는 것.
  $x^2 - e^x = 0$
  ✓ 다음 범위 내에서 [그림 5-2]는 이 방정식에 대한 그래프.
  $x \in [-2,1]$

[그림 5-2] $x^2 - e^x = 0$ 의 그래프



5.4 수치 해석 활용하기

- 근 구하기

  ✓ 함수의 곡선은 근이 [-1.0, -0.5]에 있음을 보여줌.

  ✓ 방정식과 이 간격에서 uniroot() 함수는 대략적인 근의 위치(x), 그 지점의 함수 값(y), 소요되는 반복 횟수(iter) 및 추정된 근의 정확도를 포함하는 리스트를 반환.

```
> uniroot(function(x) x ^ 2 - exp(x), c(-2, 1))
$root
[1] -0.7034583

$f.root
[1] -1.738305e-05

$iter
[1] 6

$init.it
[1] NA

$estim.prec
[1] 6.103516e-05
```

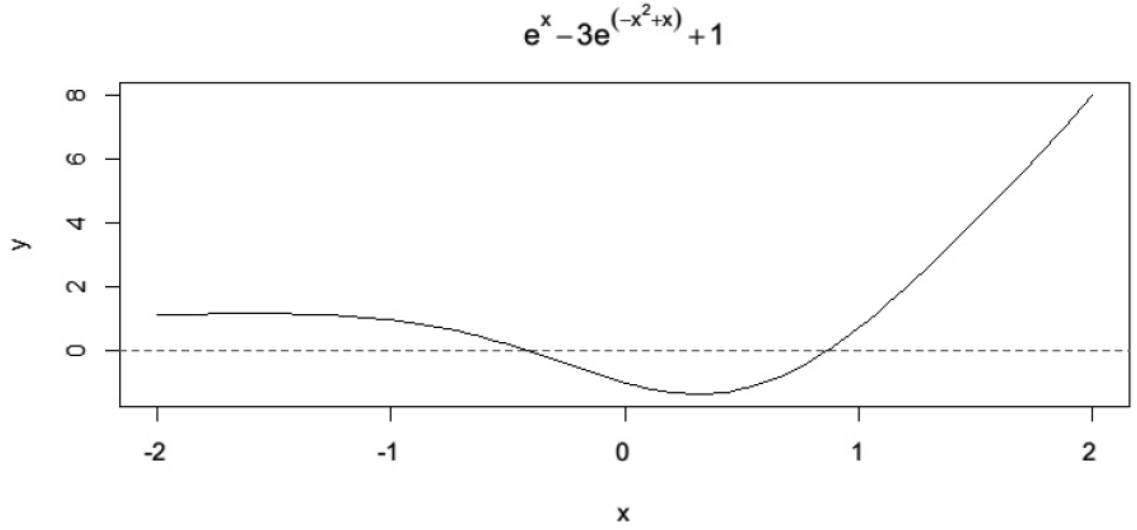
5.4 수치 해석 활용하기

- 근 구하기

  ✓ 좀 더 복잡한 예제는 다음 방정식의 근을 찾는 것.
  $e^x - 3e^{-x^2+x} + 1$

  ✓ 다음 범위 내에서 [그림 5-3]은 이 방정식에 대한 그래프.
  $x \in [-2,2]$

[그림 5-3] $e^x - 3e^{-x^2+x} + 1$ 의 그래프



5.4 수치 해석 활용하기

- 근 구하기

  ✓ 이 방정식의 근은 -2와 2 사이에 있는 것이 분명.

  ✓ uniroot() 함수는 한 번에 하나의 근만 구할 수 있다 보니 함수가 검색 범위에서 단조성을 보일 때 최고의 함수.

  ✓ [-2, 2] 범위에서 근을 찾으려고 하면 다음 오류가 발생.

```
> f <- function(x) exp(x) - 3 * exp(-x ^ 2 + x) + 1
> uniroot(f, c(-2, 2))
Error in uniroot(f, c(-2, 2)) :
  f() values at end points not of opposite sign
```

  ✓ 해당 범위의 양 끝에서 이 함수가 서로 다른 부호의 함수 값을 갖고 있어야 함.

  ✓ 이 범위를 2개 부분으로 나누어 각각의 근을 구할 수 있음.

```
> uniroot(f, c(-2, 0))$root
[1] -0.4180424
> uniroot(f, c(0, 2))$root
[1] 0.8643009
```

5.4 수치 해석 활용하기

- 근 구하기

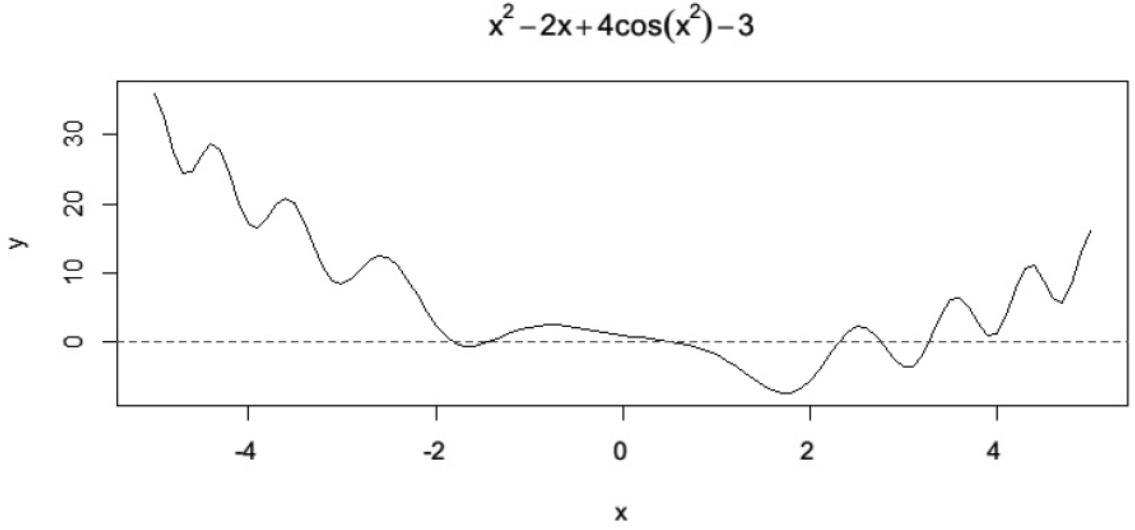
✓ 훨씬 더 복잡한 방정식을 다뤄보자.

$x^2 - 2x + 4\cos(x^2) - 3 = 0$

✓ 다음 범위 내에서 [그림 5-4]는 이 방정식에 대한 그래프.

$x \in [-5,5]$

[그림 5-4] $x^2 - 2x + 4\cos(x^2) - 3 = 0$ 의 그래프



5.4 수치 해석 활용하기

- **근 구하기**

  ✓ 이 그래프에서 볼 수 있듯이 주어진 방정식을 만족하는 근이 여러 개 존재.
  ✓ 다음은 그중 [0, 1] 범위에 있는 근을 찾는 코드.

```
> uniroot(function(x) x ^ 2 - 2 * x + 4 * cos(x ^ 2) - 3, c(0, 1))$root
[1] 0.5593558
```

  ✓ 앞 예제처럼 함수 이름을 정하지 않고 uniroot()에 방정식 함수를 직접 전달 가능.
  ✓ 이것을 **임의의 함수**라고 함.

5.4 수치 해석 활용하기

- 미적분

미분

✓ D() 함수는 주어진 변수에 대한 함수의 미분식을 심벌릭하게 계산.

✓ 예를 들어 dx^2/dx 를 유도해 보자.

```
> D(quote (x ^ 2), "x")
2 * x
```

✓ $dsin(x)\cos(xy)/dx$ 를 유도해 보자.

```
> D(quote(sin(x) * cos(x * y)), "x")
cos(x) * cos(x * y) - sin(x) * (sin(x * y) * y)
```

✓ quote() 함수 덕분에 표현식 안의 기호를 직접 평가하는 대신 심벌릭하게 접근이 가능.

5.4 수치 해석 활용하기

- 미적분

✓ 미분 결과 자체는 평가하지 않는 표현식이기에 eval()을 호출하여 필요한 모든 기호의 값이 주어진다면 이를 평가 가능.

```
> z <- D(quote(sin(x) * cos(x * y)), "x")
> z
cos(x) * cos(x * y) - sin(x) * (sin(x * y) * y)
> eval(z, list(x = 1, y = 2))
[1] -1.75514
```

✓ quote()로 표현식 객체를 만들고, eval()은 기호 값을 지정하여 표현식을 평가.

✓ 이렇게 만든 표현식 객체는 R에 메타프로그래밍이라는 강력한 기능을 가져다 줌.

5.4 수치 해석 활용하기

- 미적분

적분

✓ 적분은 심벌릭 계산이 아니기 때문에 표현식을 작성할 필요는 없지만, 관련 함수는 제공.

✓ 예를 들어 다음 수식은 분명히 적분과 관련한 문제.

$$\int_0^{\frac{\pi}{2}} \sin(x)\, dx$$

✓ 기본적으로 0에서 pi / 2 사이의 사인 곡선 아래 영역을 계산.

✓ R은 이러한 문제를 아주 유연하게 해결할 수 있는 내장 함수 integrate()를 제공.

```
> result <- integrate(function(x) sin(x), 0, pi / 2)
> result
1 with absolute error < 1.1e-14
```

5.4 수치 해석 활용하기

- 미적분

  ✓ 결과는 수치 값 하나로 보이지만 다른 정보를 담고 있음.
  ✓ 사실 이것은 다음 정보를 담은 리스트 객체.

```
> str(result)
List of 5
 $ value       : num 1
 $ abs.error   : num 1.11e-14
 $ subdivisions: int 1
 $ message     : chr "OK"
 $ call        : language integrate(f = function(x) sin(x), lower = 0, upper = pi/2)
 - attr(*, "class")= chr "integrate"
```

5.5 통계 함수 사용하기

5.5 통계 함수 사용하기

- 통계 함수 사용하기

  ✓ R은 랜덤 샘플링부터 통계 검정까지 다양한 종류의 통계 연산과 모델링 함수를 제공.
  ✓ 이것으로 높은 생산성을 얻을 수 있으며, 비슷한 종류의 함수들은 같은 인터페이스를 사용.

5.5 통계 함수 사용하기

- 벡터에서 샘플링하기

✓ 통계에서는 모집단을 조사할 때 보통은 랜덤 샘플링(무작위 표본)을 먼저 수행.
✓ sample() 함수는 주어진 벡터나 리스트에서 무작위 샘플(표본)을 추출.
✓ 기본적으로 sample() 함수는 표본을 비복원 추출.
✓ 예를 들어 다음 코드는 수치형 벡터에서 표본 5개를 비복원 추출.

```
> sample(1:6, size = 5)
[1] 2 3 6 5 4
```

✓ replace = TRUE 를 이용하면 복원 추출.

```
> sample(1:6, size = 5, replace = TRUE)
[1] 5 2 1 6 1
```

✓ sample() 함수는 수치형 벡터 외에 다른 타입의 벡터에도 동일하게 사용 가능.

```
> sample(letters, size = 3)
[1] "o" "y" "g"
```

비복원추출 : 크기가 n인 표본을 뽑기 위해, n개의 원소를 하나씩 뽑는다. 원소를 하나 뽑을 때마다 복원하지 않고 다음 원소를 뽑는다. n개의 원소를 다 뽑으면 모집단에 돌려놓는다.

복원추출 : 크기가 n인 표본을 뽑기 위해, n개의 원소를 하나씩 뽑는다. 원소를 하나 뽑을 때마다 복원하고 다음 원소를 뽑는다. n개의 원소를 다 뽑으면 모집단에 돌려놓는다.

5.5 통계 함수 사용하기

- 벡터에서 샘플링하기

  ✓ 리스트 객체에도 동일하게 사용 가능.

```
> sample(list(a = 1, b = c(2, 3), c = c(3, 4, 5)), size = 2)
$c
[1] 3 4 5

$a
[1] 1
```

5.5 통계 함수 사용하기

- 벡터에서 샘플링하기

✓ sample() 함수는 대괄호[ ]를 사용한 서브세팅을 지원하는 모든 객체에서 표본 추출 가능.
✓ 또한, 가중 표본 추출을 지원.
✓ 즉, 다음과 같이 각 요소에 대한 확률 지정 가능.

```
> grades <- sample(c("A", "B", "C"), size = 20, replace = TRUE,
+     prob = c(0.25, 0.5, 0.25))
> grades
[1] "B" "B" "A" "A" "C" "B" "B" "B" "A" "C" "B" "C" "A" "A" "C" "B" "C" "A" "B" "B"
```

✓ 각 값이 몇 번씩 추출되었는지 알아보려면 table() 함수를 사용.

```
> table(grades)
grades
A B C
6 9 5
```

5.5 통계 함수 사용하기

- **랜덤 분포 이용하기**
  - ✓ 수치 해석 시뮬레이션에서는 주어진 벡터가 아닌 랜덤 분포에서 샘플을 추출할 때가 더 자주 있음
  - ✓ R은 자주 사용되는 확률 분포를 활용할 수 있는 다양한 내장 함수를 제공함
  - ✓ R에서는 통계 분포에 따라 난수를 생성하기가 매우 쉬움
  - ✓ 가장 일반적으로 사용되는 분포는 균등 분포와 정규 분포임

5.5 통계 함수 사용하기

- 랜덤 분포 이용하기

  ✓ 통계적으로 볼 때 주어진 범위 안에서 균등 분포에서 값을 도출할 확률은 어떤 값이든 동일.

  ✓ runif(n) 함수는 [0, 1] 사이의 균등 분포에서 난수를 n개 생성.

```
> runif(5)
[1] 0.6189534 0.4805088 0.9238931 0.2406727 0.7764816
```

  ✓ min과 max를 설정하여 난수를 생성하는 범위 조절 가능.

```
> runif(5, min = -1, max = 1)
[1] -0.009318575  0.960457928 -0.861250625  0.111857928 -0.621252949
```

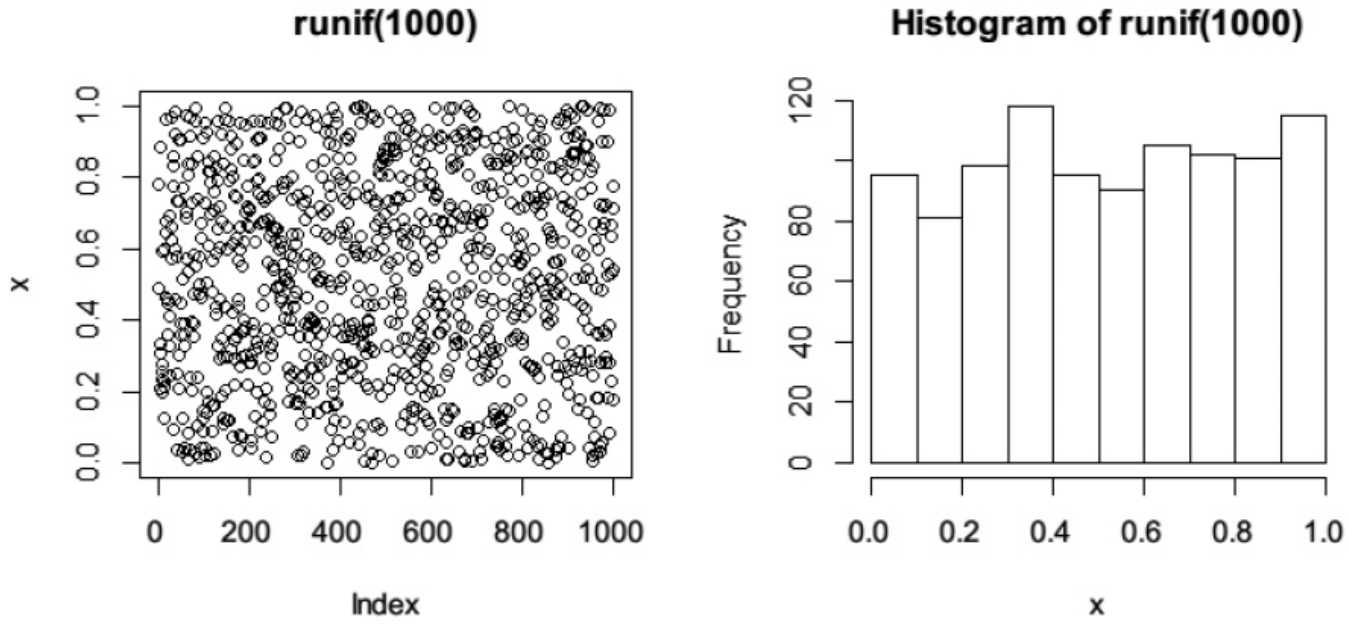
5.5 통계 함수 사용하기

- 랜덤 분포 이용하기

  ✓ runif(1000)으로 난수 1000개를 만들어 그래프를 그린다면 [그림 5-5]와 같은 산점도를 얻을 수 있음.

  ✓ 생성된 난수가 0에서 1까지 모든 간격에 거쳐 거의 균등하게 분포(균등 분포).

[그림 5-5] runif(1000)으로 생성된 난수를 사용한 산점도와 히스토그램



5.5 통계 함수 사용하기

- 랜덤 분포 이용하기

  ✓ rnorm() 함수는 표준 정규 분포에 따라 난수를 생성.

```
> rnorm(5)
[1] -0.53461665 -0.82351673 -0.26303398 -0.06960184  1.99180191
```

  ✓ 난수 생성 함수가 동일한 인터페이스를 공유.

  ✓ runif()와 rnorm() 함수의 첫 번째 인수는 생성할 난수의 개수 n.

  ✓ 나머지 인수는 랜덤 분포 자체의 관련 인수.

  ✓ 정규 분포에 대한 나머지 인수는 평균 mean과 표준 편차 sd.

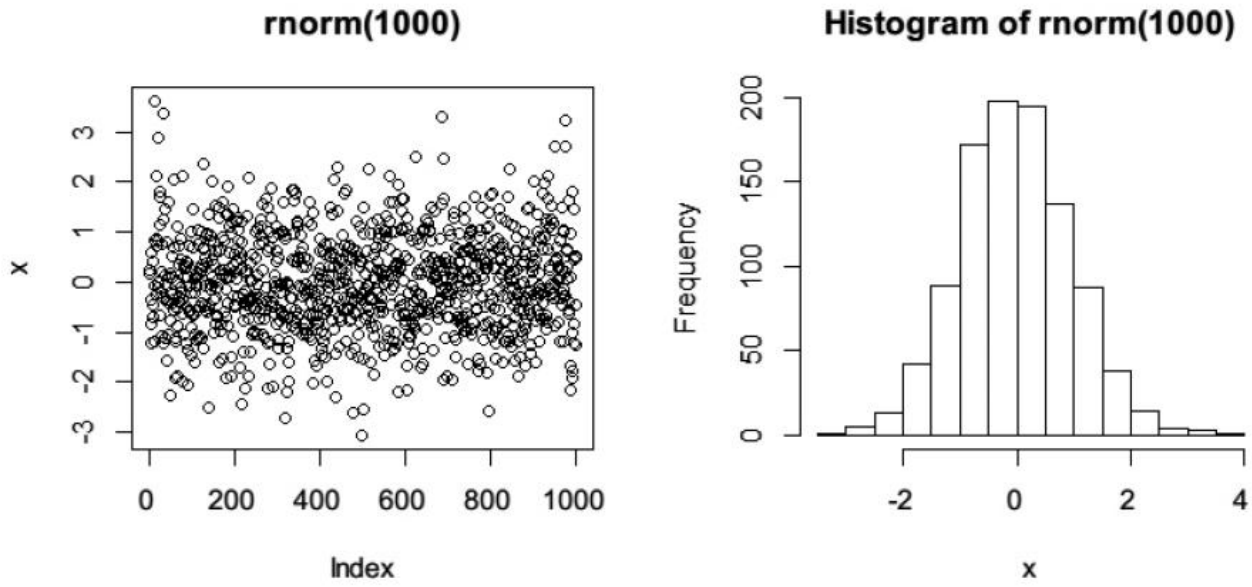
```
> rnorm(5, mean = 2, sd = 0.5)
[1] 1.435445 1.453391 1.796017 2.293780 2.410556
```

5.5 통계 함수 사용하기

- 랜덤 분포 이용하기

  ✓ rnorm(1000)으로 [그림 5-6]과 같은 그래프를 얻을 수 있음.
  ✓ 점들이 균등하게 분배되지 않고 평균값 부근에 집중.

[그림 5-6] rnorm(1000)으로 생성된 난수를 사용한 산점도와 히스토그램



5.5 통계 함수 사용하기

- 랜덤 분포 이용하기

  - ✓ 통계 분포는 특정 수식으로 표현함
  - ✓ 이론적인 수식에 접근하려고 R은 랜덤 분포에 대한 내장 함수들을 제공.
  - ✓ 균등 분포의 경우, 확률 밀도 함수 dunif(), 누적 밀도 함수 punif(), 사분위수 함수 qunif(), 난수 생성 함수 runif()를 제공.
  - ✓ 정규 분포의 경우, 관련 함수로 dnorm(), pnorm(), qnorm() 함수를 제공.

5.5 통계 함수 사용하기

- 요약 통계 계산

  - ✓ 주어진 데이터셋을 한눈에 파악하려면 요약 통계가 필요.
  - ✓ R은 수치형 벡터에서 평균, 중간값, 표준 편차, 분산, 최댓값, 최솟값, 범위, 사분위 값을 포함하는 요약 통계를 계산하는 함수를 제공.
  - ✓ 다중 수치형 벡터는 공분산 행렬과 상관 행렬 계산 가능.

5.5 통계 함수 사용하기

- 요약 통계 계산

✓ 다음 코드는 내장 함수를 사용하여 이러한 요약 통계를 계산하는 방법을 보여준다.
✓ 먼저 표준 정규 분포에서 길이가 50인 난수 벡터를 생성.

```
> x <- rnorm(50)
```

✓ mean() 함수는 x의 평균을 계산.

```
> mean(x)
[1] -0.1051295
```

✓ 이는 다음 코드와 동일.

```
> sum(x) / length(x)
[1] -0.1051295
```

✓ mean() 함수는 평균을 계산할 때 입력 데이터의 양 끝에서 일정 비율의 데이터는 버림.

```
> mean(x, trim = 0.05)
[1] -0.141455
```

✓ 다른 값들과 멀리 떨어져 있는 특이 값들이 x에 포함된다면 위 방법(trim)을 이용하여 입력 값에서 특이 값들을 지울 수 있기 때문에, 이렇게 구한 평균값이 더 강인(robust)하다고 할 수 있음.

5.5 통계 함수 사용하기

- 요약 통계 계산

  ✓ 표본 데이터 위치를 나타내는 또 다른 측정 방법은 표본 중간 값.
  ✓ 표본 데이터에서 관측 값의 절반은 중간 값보다 크고, 나머지 절반은 중간 값보다 작음.
  ✓ 데이터에 극단 값이 있더라도 중간 값은 강인한 측정치.
  ✓ median(x) 함수는 x에 대한 표본 중간 값을 반환.

```
> median(x)
[1] -0.2312157
```

5.5 통계 함수 사용하기

- 요약 통계 계산

  ✓ 평균이나 중간 값 같은 위치 추정과 더불어 변이 추정 역시 중요.

  ✓ sd() 함수는 표준 편차를 계산하여 반환.

```
> sd(x)
[1] 0.8477752
```

  ✓ var() 함수는 분산을 계산하여 반환.

```
> var(x)
[1] 0.7187228
```

5.5 통계 함수 사용하기

- 요약 통계 계산

✓ 데이터에서 극단 값을 얻는 간단한 방법은 min(), max() 함수를 사용하는 것.

```
> c(min = min(x), max = max(x))
      min       max
-1.753655  2.587579
```

✓ range() 함수는 최소값, 최대값 두 가지(범위)를 한 번에 얻을 때 사용.

```
> range(x)
[1] -1.753655  2.587579
```

5.5 통계 함수 사용하기

- 요약 통계 계산

✓ 가끔 데이터가 일정한 분포를 따르지 않을 때가 있음.
✓ 불규칙성 때문에 위치와 변이 추정에 어려움을 겪고 잘못된 결과를 얻을 수 있음.
✓ 이럴 경우에는 quantile(x) 함수를 통해 데이터의 사분위 값을 살펴볼 필요가 있음.

```
> quantile(x)
        0%         25%         50%         75%        100%
-1.7536547  -0.6774037  -0.2312157   0.2974412   2.5875789
```

✓ 더 많은 분위 값을 얻기 위해서는 probs 인수를 활용할 수 있음.

```
> quantile(x, probs = seq(0, 1, 0.1))
         0%         10%         20%         30%         40%         50%         60%
         70%         80%         90%        100%
-1.75365470 -1.11623175 -0.89118655 -0.50463051 -0.41223992 -0.23121569 0.009806393
 0.177344522  0.55051014  0.96860771  2.58757888
```

✓ 데이터가 불규칙한 분포를 따른다면 분위 값 사이의 차이가 작아졌다 커졌다 할 것.

5.5 통계 함수 사용하기

- 요약 통계 계산

✓ summary() 함수는 많이 사용되는 사분위 값, 중간 값, 평균값을 포함하는 요약 통계를 출력.

```
> summary(x)
   Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-1.7540   -0.6774   -0.2312   -0.1051    0.2974    2.5880
```

✓ 최소값과 최대값은 각각 0% 분위 값과 100% 분위 값을 의미.

✓ summary() 함수는 여러 종류의 객체에 다양한 방식으로 적용 가능.

✓ 예를 들어 summary() 함수를 데이터 프레임 객체에 적용 가능.

```
> df <- data.frame(score = round(rnorm(100, 80, 10)),
+    grade = sample(letters[1:3], 100, replace = TRUE))
> summary(df)
          score   grade
 Min.   : 60.00    a:34
 1st Qu.: 73.00    b:38
 Median : 79.00    c:28
 Mean   : 79.65
 3rd Qu.: 86.00
 Max.   :107.00
```

5.5 통계 함수 사용하기

- 요약 통계 계산

공분산 행렬과 상관 행렬 계산하기

✓ 지금까지는 주어진 단일 벡터에 대해 가장 많이 사용되는 요약 통계를 알아보았음.

✓ 벡터가 2개 이상 주어졌을 때는 공분산 행렬과 상관 행렬 계산 가능.

✓ 다음 코드는 x와 상관관계에 있는 또 다른 y 벡터를 생성.

```
> y <- 2 * x + 0.5 * rnorm(length(x))
```

✓ cov(x, y) 함수는 x와 y 사이의 공분산 계산.

```
> cov(x, y)
[1] 1.419859
```

✓ cor(x, y) 함수는 상관 계수를 계산.

```
> cor(x, y)
[1] 0.9625964
```

✓ 이 두 함수 모두 벡터 2개 이상에서도 동작.

✓ 벡터 2개 이상에서 공분산과 상관 행렬을 계산하려면 행렬이나 데이터 프레임으로 입력해야 함.

5.5 통계 함수 사용하기

- 요약 통계 계산

  ✓ 다음 코드에서는 x와 크기가 같은 랜덤 벡터 z를 생성.
  ✓ z는 균등 분포를 따르고 x나 y으로부터 독립적.
  ✓ cbind() 함수를 사용하여 열 3개를 갖는 행렬을 만들고, 이 벡터 사이의 공분산 행렬을 계산.

```
> z <- runif(length(x))
> m1 <- cbind(x, y, z)
> cov(m1)
          x           y          z
x 0.7187228  1.41985899 0.04229950
y 1.4198590  3.02719645 0.07299981
z 0.0422995  0.07299981 0.08005535
```

5.5 통계 함수 사용하기

- 요약 통계 계산

✓ 비슷한 방법으로 상관 행렬을 직접 계산 가능.

```
> cor(m1)
          x         y         z
x 1.0000000 0.9625964 0.1763434
y 0.9625964 1.0000000 0.1482881
z 0.1763434 0.1482881 1.0000000
```

✓ y에 랜덤 노이즈가 섞여 있기는 하지만 x와 선형 관계가 있기 때문에 둘 사이에는 높은 상관성이 있을 것이라고 기대.

✓ z는 그렇지 않음.

✓ 상관 행렬을 보면, 우리 기대와 크게 다르지 않음.

✓ 책의 범주를 벗어나지만, 통계적으로 어떤 결론을 도출하려면 철저한 통계 검정을 수행해야 함.

5.5 apply 계열 함수 활용하기

5.6 apply 계열 함수 활용하기

- apply 계열 함수 활용하기

  ✓ 표현식을 반복적으로 평가하는 경우, for 루프는 제일 마지막에 선택하는 옵션.
  ✓ 각 반복이 서로 독립적일 때 훨씬 깔끔하고 읽기 쉬운 다른 방법들이 있기 때문.
  ✓ 다음 코드는 for 루프를 사용하여 len 벡터로 길이를 지정하고, 독립된 정규 분포 랜덤 벡터 3개를 갖는 리스트를 생성.

```
> len <- c(3, 4, 5)
> # 먼저 환경에 리스트를 정의한다
> x <- list()
> # 그리고 for 루프를 사용하여 각각의 길이를 갖는 랜덤 벡터를 생성한다
> for (i in 1:3) {
+     x[[i]] <- rnorm(len[i])
+ }
> x
[[1]]
[1] 1.4572245  0.1434679  -0.4228897

[[2]]
[1] -1.4202269 -0.7162066 -1.6006179 -1.2985130

[[3]]
[1] -0.6318412  1.6784430  0.1155478  0.2905479 -0.7363817
```

5.6 apply 계열 함수 활용하기

- apply 계열 함수 활용하기

✓ 앞 코드가 간단해 보이지만, lapply() 함수를 사용한 다음 코드와 비교했을 때는 상대적으로 매우 장황.

```
> lapply(len, rnorm)
[[1]]
[1] -2.0125198  0.5049327  0.8281116

[[2]]
[1] 0.3358507 -1.0591244  1.5677167 -0.3701466

[[3]]
[1] 1.77903836  0.55140201  1.19031065  0.33060223 -0.06465223
```

✓ lapply() 함수를 활용한 코드가 훨씬 간단.

✓ len 벡터의 각 요소에 대해 rnorm() 함수를 적용하고, 각 결과를 리스트로 반환.

✓ apply 계열 함수는 함수를 인수로 받는 **고차 함수**.

5.6 apply 계열 함수 활용하기

- lapply

  - ✓ lapply() 함수는 벡터(또는 리스트) 하나와 함수 하나를 입력으로 받아, 주어진 벡터의 각 요소를 입력 함수에 적용하여 모든 결과를 포함하는 리스트를 반환.
  - ✓ 각 반복 연산이 서로 독립적일 때 유용한 함수.
  - ✓ 이럴 경우에는 반복자를 명시적으로 설정할 필요가 없음.
  - ✓ 학생 정보를 가진 students 리스트가 있다고 가정하자.

```
> students <- list(
+     a1 = list(name = "James", age = 25,
+         gender = "M", interest = c("reading", "writing")),
+     a2 = list(name = "Jenny", age = 23,
+         gender = "F", interest = c("cooking")),
+     a3 = list(name = "David", age = 24,
+         gender = "M", interest = c("running", "basketball")))
```

  - ✓ 위 리스트 요소를 이용해 다음 형태의 문자형 벡터를 생성하려고 한다.

```
James, 25 year-old man, loves reading, writing
```

5.6 apply 계열 함수 활용하기

- lapply

✓ sprintf() 함수는 타입별 표시 기호(예를 들어 문자열은 %s, 정수는 %d)가 있는 자리를 해당 입력 인수로 대체하여 원하는 문자열을 만드는 데 유용.

```
> sprintf("Hello, %s! Your number is %d.", "Tom", 3)
[1] "Hello, Tom! Your number is 3."
```

✓ students 객체를 사용한 반복 연산이 서로 독립적.

✓ 즉, James에 대한 연산은 Jenny를 비롯한 다른 학생과는 아무런 관련이 없음.

```
> lapply(students, function(s) {
+     type <- switch(s$gender, "M" = "man", "F" = "woman")
+     interest <- paste(s$interest, collapse = ", ")
+     sprintf("%s, %d year-old %s, loves %s.", s$name, s$age, type, interest)
+ })
$a1
[1] "James, 25 year-old man, loves reading, writing."

$a2
[1] "Jenny, 23 year-old woman, loves cooking."

$a3
[1] "David, 24 year-old man, loves running, basketball."
```

5.6 apply 계열 함수 활용하기

- lapply

  ✓ 앞선 예제에서는 객체로 할당되지 않은 익명 함수 function(s)를 사용.
  - ❖ 다시 말해 이 함수는 임시이며 이름이 따로 없음.
  - ❖ 이 함수를 어떤 기호로 선언할 수도 있는데, 이 기호가 결국 함수 이름이 되고 lapply()에서 사용 가능.

  ✓ students의 각 s 요소에 대해 이 함수는 학생 종류와 관심 사항을 쉼표로 연결.

5.6 apply 계열 함수 활용하기

- sapply

  - ✓ 결과를 담는 객체로 리스트가 항상 최선은 아님.
  - ✓ 때로는 결과를 단순한 벡터나 행렬 형태로 담고 싶을 때도 있음.
  - ✓ sapply() 함수는 구조에 따라 결과를 더 단순한 형태로 생성.
  - ✓ 1:10 벡터의 각 요소에 제곱을 계산하고자 한다고 가정하자.
  - ✓ lapply()를 사용하면 제곱 값을 갖는 리스트를 얻을 수 있을 것.
  - ✓ 결과로 나오는 리스트는 각 요소가 숫자 하나를 의미하는 단일 벡터인 것에 비해서 너무 쓸데없이 덩치만 큰 느낌.
  - ✓ 이때 이 결과를 하나의 벡터로 저장하고 싶다면 sapply() 사용.

```
> sapply(1:10, function(i) i ^ 2)
[1]   1   4   9  16  25  36  49  64  81 100
```

  - ✓ 입력 함수가 각 요소에 대해 다중 요소 벡터를 결과로 반환한다면, sapply() 함수는 각 결과 벡터를 열 벡터로 하는 행렬을 출력.

```
> sapply(1:10, function(i) c(i, i ^ 2))
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    3    4    5    6    7    8    9    10
[2,]    1    4    9   16   25   36   49   64   81   100
```

5.6 apply 계열 함수 활용하기

- vapply

✓ sapply()는 다루기 쉽고 편하지만, 이 점이 반대로 독이 될 수도 있음.

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

✓ x 안의 숫자들을 제곱한 값을 원소로 하는 수치형 벡터를 얻고자 할 때 sapply()는 자동으로 결과 구조를 데이터에 맞게 조정.

```
> sapply(x, function(x) x ^ 2)
     [,1] [,2] [,3]
[1,]    1    4    1
[2,]    4    9    9
```

✓ 입력 데이터에 변형이 있다면 sapply()는 이것까지 허용하여 예측하지 못한 결과를 초래.

✓ 예를 들어 x의 마지막 요소에 잘못해서 숫자 하나가 더 들어갔다고 하자.

```
> x1 <- list(c(1, 2), c(2, 3), c(1, 3, 3))
```

✓ 이때 sapply() 함수는 단순한 행렬이 아닌 다음 리스트를 결과로 출력.

```
> sapply(x1, function(x) x ^ 2)
[[1]]
[1] 1 4

[[2]]
[1] 4 9

[[3]]
[1] 1 9 9
```

5.6 apply 계열 함수 활용하기

- vapply

  - ✓ vapply() 함수를 먼저 사용한다면 이 실수를 바로 잡아낼 수 있음
  - ✓ vapply() 함수에는 요소별 반복에서 얻는 결과 형태를 지정하는 추가 인수가 존재.
  - ✓ 다음 코드에서 numeric(2)는 결과 형태를 지정하는 부분.
    - ❖ 각 요소를 반복 적용할 때마다 요소 2개를 갖는 수치형 벡터가 결과로 나와야 한다는 것을 의미.
    - ❖ 이 형식에서 벗어난다면 결국 오류가 발생하며 실행이 종료.

```
> vapply(x1, function(x) x ^ 2, numeric(2))
Error in vapply(x1, function(x) x^2, numeric(2)) :
  values must be length 2,
  but FUN(X[[3]]) result is length 3
```

  - ✓ 원래 입력을 이용하면 sapply()를 사용했을 때와 정확히 일치하는 결과 출력.

```
> vapply(x, function(x) x ^ 2, numeric(2))
     [,1] [,2] [,3]
[1,]    1    4    1
[2,]    4    9    9
```

  - ✓ 결론적으로 vapply()는 추가적인 형식 검사를 실행함으로써 sapply()의 안전한 버전.
  - ✓ 결과 형식을 미리 정할 수 있다면 vapply() 함수를 사용하는 것을 권장.

5.6 apply 계열 함수 활용하기

- mapply

  ✓ lapply()와 sapply() 함수는 한 벡터의 요소에 따라 반복적.
  ✓ 반면, mapply() 함수는 여러 벡터에서 반복적.
  ✓ 즉, mapply()는 sapply()의 다변량 버전.

```
> mapply(function(a, b, c) a * b + b * c + a * c,
+     a = c(1, 2, 3), b = c(5, 6, 7), c = c(-1, -2, -3))
[1] -1 -4 -9
```

  ✓ 이 반복 함수에서는 스칼라값 뿐만 아니라 다중 요소 벡터까지도 결과가 될 수 있음.
  ✓ mapply() 역시 sapply()처럼 자동으로 결과를 단순화.

```
> df <- data.frame(x = c(1, 2, 3), y = c(3, 4, 5))
> df
  x y
1 1 3
2 2 4
3 3 5
> mapply(function(xi, yi) c(xi, yi, xi + yi), df$x, df$y)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    3    4    5
[3,]    4    6    8
```

5.6 apply 계열 함수 활용하기

- mapply

  ✓ Map() 함수는 lapply()의 다변량 버전으로, 결과는 언제나 리스트 형태.

```
> Map(function(xi, yi) c(xi, yi, xi + yi), df$x, df$y)
[[1]]
[1] 1 3 4

[[2]]
[1] 2 4 6

[[3]]
[1] 3 5 8
```

5.6 apply 계열 함수 활용하기

- apply
  - ✓ apply() 함수는 주어진 행렬이나 배열의 특정 영역 또는 차원에 입력 함수를 적용.
  - ✓ 예를 들어 각 행(즉, 첫 번째 차원)의 합을 계산하고자 할 때, MARGIN = 1로 설정하여 행을 따라 반복되면서 sum 함수를 적용.

```
> mat <- matrix(c(1, 2, 3, 4), nrow = 2)
> mat
     [,1]    [,2]
[1,]    1       3
[2,]    2       4
> apply(mat, 1, sum)
[1] 4 6
```

5.6 apply 계열 함수 활용하기

- apply

  ✓ 각 열(두 번째 차원)의 합을 계산하려면, MARGIN = 2로 설정하여 열을 따라 반복되면서 sum 함수를 적용.

```
> apply(mat, 2, sum)
[1] 3 7
```

  ✓ apply 함수는 배열 입력과 행렬 출력도 지원.

```
> mat2 <- matrix(1:16, nrow = 4)
> mat2
     [,1]   [,2]   [,3]   [,4]
[1,]    1      5      9     13
[2,]    2      6     10     14
[3,]    3      7     11     15
[4,]    4      8     12     16
```

5.6 apply 계열 함수 활용하기

- apply

✓ 다음 코드는 각 열의 최대값, 최소값을 보여 주는 행렬을 생성

```
> apply(mat2, 2, function(col) c(min = min(col), max = max(col)))
     [,1]  [,2]  [,3]  [,4]
min     1     5     9    13
max     4     8    12    16
```

✓ 다음 코드는 각 행의 최대값, 최소값을 보여 주는 행렬을 생성.

```
> apply(mat2, 1, function(col) c(min = min(col), max = max(col)))
    [,1]  [,2]  [,3]  [,4]
min    1     2     3     4
max   13    14    15    16
```

5.7 마치며

5.7 마치며

- **마치며**

  - ✓ 이 장에서는 내장 함수를 사용한 예제로 기본 객체를 활용하는 방법을 알아보았습니다.
  - ✓ 실제로 이것은 R 언어의 어휘력이 됩니다.
  - ✓ 객체 타입을 검사하고 알아보며, 데이터 차원에 접근하고 재배치하는 몇 가지 기본 함수를 학습하였습니다.
  - ✓ 데이터를 필터링하는 여러 가지 논리 연산자와 함수도 배웠습니다.
  - ✓ 수치형 데이터 구조와 함께 작업하는 데 필요한 기본적인 수학 함수, 근을 찾고 미적분을 계산할 수 있는 내장된 수치 해석 함수, 랜덤 샘플링을 수행하고 데이터의 요약 통계를 작성하는 몇 가지 통계 함수도 배웠습니다.
  - ✓ 또 반복으로 더 쉽게 결과를 수집할 수 있는 apply 계통의 함수들도 살펴보았습니다.
  - ✓ 중요한 데이터 범주 가운데 또 다른 하나는 문자형 벡터로 표현되는 문자열입니다.