R프로그래밍

2장. 기본 객체 알아보기

박 혜 승 교 수

2장. 기본 객체 알아보기

2.1 벡터
2.2 행렬
2.3 배열
2.4 리스트
2.5 데이터 프레임
2.6 함수
2.7 마치며

2.1 벡터

- **벡터(vector)**
  - ✓ 종류(숫자, 참/거짓 값, 문자열)가 같은 기본(primitive)값을 모아 놓은 집합.
  - ✓ 모든 R 객체의 기본 구성 단위 가운데 하나.
  - ✓ R에는 다양한 타입의 벡터가 존재.
    - ❖ 저장하는 요소들이 어떤 종류인지에 따라 구분.

2.1 벡터

- **수치형 벡터 (numeric vector)**
  - ✓ 수치형 벡터는 숫자 값으로 된 벡터.
  - ✓ 수치형 벡터는 가장 많이 사용하는 데이터 타입으로 데이터 분석에서 기본.

```
> 1.5
[1] 1.5
```

  - ✓ 스칼라 값은 가장 단순한 수치형 벡터.
  - ✓ 다른 프로그래밍 언어에는 정수형, 실수형, 문자열 등 여러 종류의 스칼라 타입이 존재하며, 이러한 스칼라 타입은 벡터의 기본 구성 요소.
  - ✓ R에는 스칼라 타입에 딱히 정해진 바가 없음.
  - ✓ 스칼라 숫자만 유일하게 길이가 1인 특수한 수치형 벡터.

2.1 벡터

- **수치형 벡터 (numeric vector)**
  - ✓ 대입 연산자 <- : 오른쪽 값을 왼쪽 변수에 저장.
  - ✓ 예) 이름이 x인 변수에 수치 값 1.5 저장.

```
> x <- 1.5
```

  - ✓ 예) x를 입력하여 어떤 값이 들어 있는지 확인.

```
> x
[1] 1.5
```

2.1 벡터

- **수치형 벡터 (numeric vector)**
  - ✓ 수치형 벡터를 만드는 방법
    - ❖ numeric() 함수: 원하는 길이의 영벡터 생성.

```
> numeric (10)
[1] 0 0 0 0 0 0 0 0 0 0
```

    - ❖ c() 함수: 여러 벡터를 하나로 통합.
      - 길이가 1인 단일 요소 벡터 여러 개를 요소가 여러 개인 다중 요소 벡터 하나로 통합.

```
> c(1, 2, 3, 4, 5)
[1] 1 2 3 4 5
```

      - 단일 요소 벡터와 다중 요소 벡터를 하나로 통합.

```
> c(1, 2, c(3, 4, 5))
[1] 1 2 3 4 5
```

    - ❖ : 연산자: 연속적인 정수로 구성되는 벡터 생성.

```
> 1:5
[1] 1 2 3 4 5
```

      - : 연산자를 사용할 때 주의할 점.

```
> 1 + 1:5
[1] 2 3 4 5 6
```

      - : 연산자가 + 연산자보다 우선하므로 1:5 실행 후, 그 결과에 1을 더함 (연산자 우선순위).

2.1 벡터

- **수치형 벡터 (numeric vector)**
  - ✓ 수치형 벡터를 만드는 방법 (cont.)
    - ❖ seq() 함수
      - 예) 1부터 10까지 2씩 증가하는 벡터를 생성.

```
> seq(1, 10, 2)
[1] 1 3 5 7 9
```

      - 다양한 인수를 가짐.
      - 모든 인수를 활용하여 함수 호출 가능하지만, 보통은 모든 인수를 사용하지 않아도 가능.
      - 어떤 인수는 이미 충분히 좋은 기본값으로 설정되어 있음.
      - 기본값을 변경할 때만 해당 인수를 설정.
      - 예) length.out 인수로 3부터 시작하여 길이가 10인 수치형 벡터 생성.

```
> seq(3, length.out = 10)
[1] 3 4 5 6 7 8 9 10 11 12
```

      - length.out 인수를 사용해서 함수를 호출했으므로, 다른 인수들은 기본값 적용.

2.1 벡터

- **논리형 벡터 (logical vector)**
  - ✓ 논리형 벡터는 TRUE 또는 FALSE 값의 집합.
  - ✓ 논리형 질문에 대한 참/거짓의 답.
  - ✓ 가장 간단한 논리형 벡터는 TRUE 또는 FALSE 그 자체.

```
> TRUE
[1] TRUE
```

  - ✓ 논리형 벡터를 생성하는 가장 일반적인 방법은 R 객체에서 논리형 질문을 만드는 것.

```
> 1 > 2
[1] FALSE
```

2.1 벡터

- **논리형 벡터 (logical vector)**
  - ✓ 동시에 여러가지를 비교하고 싶다면 수치형 벡터 사용.

```
> c(1, 2) > 2
[1] FALSE FALSE
```

    - ❖ R은 이러한 연산 표현식을 "요소별 비교"라고 해석.
    - ❖ 즉, c(1, 2) > 2 == c(1 > 2, 2 > 2)
  - ✓ 길이가 긴 벡터가 길이가 짧은 벡터보다 짧은 배수라면 다중 요소 수치형 벡터끼리 비교 가능.

```
> c(1, 2) > c(2, 1)
[1] FALSE TRUE
```

    - ❖ c(1, 2) > c(2, 1) == c(1 > 2, 2 > 1)

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

    - ❖ 이는 길이가 짧은 벡터를 반복해서 적용하는 원리.
    - ❖ 즉, c(2 > 1, 3 > 2, 2 > -1, 3 > 3)를 의미.
    - ❖ 길이가 긴 벡터에 있는 모든 요소에서 비교가 끝날 때까지는 짧은 벡터 안의 요소들을 지속적으로 재활용.

2.1 벡터

- **논리형 벡터 (logical vector)**
  - ✓ R의 논리 연산자
    - ❖ == 동일, > 큼, >= 크거나 같음, < 작음, <= 작거나 같음.
    - ❖ %in%
      - 왼쪽 벡터에 있는 원소가 오른쪽 벡터 안에 존재하는지 판단.

```
> 1 %in% c(1, 2, 3)
[1] TRUE
> c(1, 4) %in% c(1, 2, 3)
[1] TRUE FALSE
```

      - 수치 비교 연산자와는 달리 %in% 연산자는 길이가 다른 벡터 간에도 원소를 재활용하지 않음.

2.1 벡터

- **문자형 벡터 (character vector)**
  - ✓ 문자열로 구성된 집합
  - ✓ 여기서 문자열은 개별적인 문자 또는 기호가 아니라 this is a string 같은 문자의 집합을 의미
  - ✓ 문자형 벡터를 만드는 데 큰따옴표" "와 작은따옴표' ' 모두 사용 가능

```
> "hello, world!"
[1] "hello, world!"
> 'hello, world!'
[1] "hello, world!"
```

  - ✓ c() 함수로 다중 요소 문자형 벡터 생성 가능

```
> c("Hello", "World")
[1] "Hello" "World"
```

  - ✓ == 연산자로 두 벡터의 같은 위치에 있는 문자열이 서로 동일한지 확인 가능

```
> c("Hello", "World") == c('Hello', 'World')
[1] TRUE TRUE
```

2.1 벡터

- **문자형 벡터 (character vector)**
  - ✓ 큰따옴표나 작은따옴표를 사용하더라도 문자형 벡터가 동일.

```
> c("Hello", "World") == "Hello, World"
[1] FALSE FALSE
```

  - ✓ 두 따옴표가 다르게 동작하는 유일한 경우: 안에 따옴표가 들어간 문자열을 생성할 때.
  - ✓ 큰따옴표가 들어간 문자열을 만들 때 R 인터프리터가 이 따옴표를 문자열을 마무리하는 용도로 해석하지 않게 하려면, 문자열 안에 만든 큰따옴표 앞에 반드시 이스케이프 문자 \ 를 삽입해야 함.
  - ✓ 예) cat() 함수로 원하는 문자를 출력.

```
> cat("Is \"You\" a Chinese name?")
Is "You" a Chinese name?
```

  - ✓ 문자열을 만들 때 작은 따옴표를 사용하면 이 문제를 쉽게 해결 가능.

```
> cat('Is "You" a Chinese name?')
Is "You" a Chinese name?
```

  - ✓ 즉, 큰따옴표로 문자열을 시작할 때는 특별한 이스케이프 문자 없이도 작은따옴표 사용 가능.
  - ✓ 반대로 작은따옴표로 문자열을 시작할 때도 큰 따옴표 사용 가능.

2.1 벡터

- **벡터의 서브-세팅(sub-setting)**
  - ✓ 벡터의 서브-세팅은 어떤 벡터의 몇몇 요소 또는 일부분을 가져오는 것.
  - ✓ 예) 간단한 수치형 벡터를 생성하여 v1 객체에 할당.

```
> v1 <- c(1, 2, 3, 4)
```

  - ✓ 예) v1의 부분 집합(벡터의 두 **번째 요소**)을 얻는 방법.

```
> v1[2]
[1] 2
```

  - ✓ 예) v1의 두 번째**에서** 네 번째**까지**의 요소를 얻는 방법.

```
> v1[2:4]
[1] 2 3 4
```

  - ✓ 예) v1의 세 **번째 요소를 제외한 나머지를** 얻는 방법.

```
> v1[-3]
[1] 1 2 4
```

  - ✓ 원하는 부분 집합을 추출하려면, 벡터 뒤의 대괄호 [ ] 안에 이에 대응하는 수치형 벡터 삽입.

```
> a <- c(1, 3)
> v1[a]
[1] 1 3
```

2.1 벡터

- **벡터의 서브-세팅(sub-setting)**
  - ✓ 앞의 예제들은 모두 원소의 위치를 이용한 방법. 즉, 해당 위치를 지정하는 방식으로 벡터의 부분 집합 확보.
  - ✓ 음수를 활용하여 해당 위치의 원소 제외 가능.
  - ✓ 주의) 양수와 음수 혼합 사용 불가.

```
> v1[c(1, 2, -3)]
Error in v1[c(1, 2, -3)] : only 0's may be mixed with negative subscripts
```

  - ✓ 벡터의 원래 크기를 넘어가는 위치를 사용하여 부분 집합을 구한다면?
  - ✓ 예) v1의 세 번째 요소에서 (존재하지 않는) 여섯 번째까지 부분 집합을 구하는 예제.

```
> v1[3:6]
[1] 3 4 NA NA
```

    - 존재하지 않는 위치에 해당하는 원소는 NA, 즉, 결측 값으로 표기.

2.1 벡터

- **벡터의 서브-세팅(sub-setting)**
  - ✓ 벡터의 부분 집합을 구하는 또 다른 방법은 논리형 벡터를 활용하는 것.
  - ✓ 어떤 요소를 선택할지 결정하는 데 같은 길이의 논리형 벡터 사용 가능.

```
> v1[c(TRUE, FALSE, TRUE, FALSE)]
[1] 1 3
```

  - ✓ 예) 부분 집합을 구할 수도 있지만, 다음과 같이 특정 값을 덮어 쓸 수도 있음.

```
> v1[2] <- 0
```

  - ✓ 이때 v1은 나음과 같음.

```
> v1
[1] 1 0 3 4
```

  - ✓ 예) 서로 다른 위치에 각기 다른 값 할당 가능.

```
> v1[2:4] <- c(0, 1, 3)
```

  - ✓ 이때 v1은 다음과 같음.

```
> v1
[1] 1 0 1 3
```

2.1 벡터

- **벡터의 서브-세팅(sub-setting)**
  - ✓ 논리형 벡터를 활용할 때도 마찬가지로 값을 할당하는 데 사용 가능.

```
> v1[c(TRUE, FALSE, TRUE, FALSE)] <- c(3, 2)
```

  - ✓ 이때 v1은 나음과 같음.

```
> v1
[1] 3 0 2 3
```

  - ✓ 논리형 벡터를 활용할 수 있다는 사실은 결국 논리형 연산을 사용할 수 있다는 것을 의미.
  - ✓ 예) v1에서 2보다 작거나 같은 모든 원소 선택 가능.

```
> v1[v1 <= 2]
[1] 0 2
```

2.1 벡터

- **벡터의 서브-세팅(sub-setting)**
  - ✓ 예) v1에서 $x^2 - x + 1 \geq 0$을 만족하는 모든 원소를 출력.

```
> v1[v1 ^ 2 - v1 + 1 >= 0]
[1] 3 0 2 3
```

  - ✓ 예) $x \leq 2$를 만족하는 모든 원소를 0으로 대체.

```
> v1[v1 <= 2] <- 0
```

  - ✓ 이때 v1은 다음과 같음.

```
> v1
[1] 3 0 0 3
```

  - ✓ 존재하지 않는 요소에 값을 할당하면, 자동으로 해당 원소가 존재하도록 벡터 길이를 늘리고 나머지 원소는 결측 값(NA)로 채움.

```
> v1[10] <- 8
> v1
[1] 3 0 0 3 NA NA NA NA NA 8
```

2.1 벡터

- **이름이 정해진 벡터**
  - ✓ 이름이 정해진 벡터란 수치형 벡터나 논리형 벡터 같은 종류의 새 벡터를 지칭하는 것이 아님
  - ✓ 각 원소에 해당하는 이름이 있는 벡터를 의미.
  - ✓ 벡터를 생성할 때 이름을 붙일 수 있음.

```
> x <- c(a = 1, b = 2, c = 3)
> x
a b c
1 2 3
```

  - ✓ 다음과 같이 이름에 해당하는 문자열을 이용하여 해당 원소 값을 얻을 수 있음.

```
> x["a"]
a
1
```

  - ✓ 문자형 벡터를 사용하면 여러 원소를 얻을 수 있음.

```
> x[c("a", "c")]
a c
1 3
```

2.1 벡터

- **이름이 정해진 벡터**
  - ✓ 문자형 벡터에 중복된 값이 있으면, 결과 역시 중복해서 값을 선택.

```
> x[c("a", "a", "c")]
a a c
1 1 3
```

  - ✓ names() 함수: 벡터 이름 리턴.

```
> names(x)
[1] "a" "b" "c"
```

  - ✓ 벡터의 이름이 정해졌더라도 언제든지 변경 가능
  - ✓ 예) 이름이 다른 문자형 벡터를 사용하여 벡터 이름 변경

```
> names(x) <- c("x", "y", "z")
> x["z"]
z
3
```

2.1 벡터

- **이름이 정해진 벡터**
  - ✓ 이름이 더 이상 필요 없다면 NULL(정의되지 않은 값)을 사용하여 이름 삭제 가능.

```
> names(x) <- NULL
> x
[1] 1 2 3
```

  - ✓ 원하는 이름이 없을 때?
  - ✓ 예) 처음 사용했던 x 벡터로 몇 가지 실험 진행.

```
> x <- c(a = 1, b = 2, c = 3)
> x["d"]
<NA>
  NA
```

    - ❖ 결과는 오류가 아닌 이름과 값이 모두 결측 값인 벡터.
    - ❖ 이름이 있을 때와 없을 때가 섞여 있더라도 선택하는 데 사용된 문자형 벡터 길이와 동일한 길이로 결과 벡터를 얻음.

```
> x[c("a", "d")]
a <NA>
1   NA
```

2.1 벡터

- **원소 추출하기**
  - ✓ 이중 대괄호 [[ ]] 를 이용해 벡터의 원소를 얻을 수 있음.
    - ❖ 대괄호 [ ] 는 벡터의 부분 집합을 얻을 수 있음.
  - ✓ 벡터를 사탕이 들어 있는 상자 10개라고 가정한다면, [ ] 를 이용해 사탕 상자 3개를 선택할 수 있다. 반면, [[ ]] 는 상자를 열어 그 안의 사탕을 얻을 수 있음.
  - ✓ 단순 벡터의 경우, [ ] 와 [[ ]] 모두 한 원소를 가지며, 그 결과는 같음.
  - ✓ 어떤 경우에는 결과가 다른데, 예를 들어 이름이 정해진 벡터를 서브-세팅할 때는 결과가 다름.

```
> x <- c(a = 1, b = 2, c = 3)
> x["a"]
a
1
> x[["a"]]
[1] 1
```

    - ❖ x[“a”] 인수는 사탕 상자 a를 리턴.
    - ❖ x[[“a”]]는 사탕 상자 a의 사탕 1을 리턴.

2.1 벡터

- **원소 추출하기**
  - ✓ 이중 대괄호 [[ ]] 는 한 요소만 추출.
  - ✓ [[ ]] 는 요소가 2개 이상 있는 벡터에는 사용 불가.

```
> x[[c(1, 2)]]
Error in x[[c(1, 2)]] :
attempt to select more than one element in vectorIndex
```

  - ✓ 원하는 위치의 원소만 제외하는 데 사용하는 '음수'도 사용 불가.

```
> x[[-1]]
Error in x[[-1]] :
attempt to select more than one element in get1index <real>
```

  - ✓ 추출하려는 원소의 위치가 범위를 벗어나거나 이름이 없을 때는 코드가 동작하지 않음.

```
> x[["d"]]
Error in x[["d"]] : subscript out of bounds
```

2.1 벡터

- **벡터의 클래스 알아보기**
  - ✓ class( ) 함수: R 객체의 클래스(종류) 리턴

```
> class(c(1, 2, 3))
[1] "numeric"
> class(c(TRUE, TRUE, FALSE))
[1] "logical"
> class(c("Hello", "World"))
[1] "character"
```

  - ✓ 벡터가 원하는 클래스인지 알아보기 위해 is.numeric, is.logical 등과 같은 is.* 형태의 함수 사용.

```
> is.numeric(c(1, 2, 3))
[1] TRUE
> is.numeric(c(TRUE, TRUE, FALSE))
[1] FALSE
> is.numeric(c("Hello", "World"))
[1] FALSE
```

2.1 벡터

- **벡터 변환하기**
  - ✓ 어떤 벡터를 특정 클래스로 강제 변환 가능.
  - ✓ 예) 일부 데이터가 1, 2, 3 숫자를 나타내는 문자열의 경우.

```
> strings <- c("1", "2", "3")
> class(strings)
[1] "character"
```

❖ 이러한 문자열을 그대로 사용하면 수학 연산 불가.

```
> strings + 10
Error in strings + 10 : non-numeric argument to binary operator
```

❖ 이 문자열을 as.numeric( ) 함수를 통해 수치형 벡터로 변환 가능.

```
> numbers <- as.numeric(strings)
> numbers
[1] 1 2 3
> class(numbers)
[1] "numeric"
```

❖ 이제 수학 연산 가능.

```
> numbers + 10
[1] 11 12 13
```

2.1 벡터

- **벡터 변환하기**
  - ✓ as.* ( ) 함수: 주어진 벡터를 다른 클래스의 벡터로 변환.

```
> as.numeric(c("1", "2", "3", "a"))
[1] 1 2 3 NA
Warning message:
NAs introduced by coercion
> as.logical(c(-1, 0, 1, 2))
[1] TRUE FALSE TRUE TRUE
> as.character(c(1, 2, 3))
[1] "1" "2" "3"
> as.character(c(TRUE, FALSE))
[1] "TRUE" "FALSE"
```

  - ✓ 수치형 벡터가 아닌 서로 다른 클래스의 벡터들 사이의 직접적인 산술 연산 불가.

```
> c(2, 3) + as.character(c(1, 2))
Error in c(2, 3) + as.character(c(1, 2)) :
  non-numeric argument to binary operator
```

2.1 벡터

- **수치형 벡터의 산술 연산**
  - ✓ 기본적으로 다음 두 가지 규칙을 따름.
    1. 요소별 방식으로 연산 수행.
    2. 길이가 짧은 벡터를 재활용.

```
> c(1, 2, 3, 4) + 2
[1] 3 4 5 6
> c(1, 2, 3) - c(2, 3, 4)
[1] -1 -1 -1
> c(1, 2, 3) * c(2, 3, 4)
[1] 2 6 12
> c(1, 2, 3) / c(2, 3, 4)
[1] 0.5000000 0.6666667 0.7500000
> c(1, 2, 3) ^ 2
[1] 1 4 9
> c(1, 2, 3) ^ c(2, 3, 4)
[1] 1 8 81
> c(1, 2, 3, 14) %% 2
[1] 1 0 1 0
```

2.1 벡터

- **수치형 벡터의 산술 연산**
  - ✓ 벡터의 이름은 연산 시 고려되지 않음.
  - ✓ 왼쪽 벡터의 이름만 남고, 오른쪽 벡터의 이름은 무시.

```
> c(a = 1, b = 2, c = 3) + c(b = 2, c = 3, d = 4)
a b c
3 5 7
> c(a = 1, b = 2, 3) + c(b = 2, c = 3, d = 4)
a b
3 5 7
```

2.2 행렬

- **행렬 (matrix)**
  - ✓ 행렬은 2차원으로 표현된 벡터.
  - ✓ 벡터에 적용된 원리들은 행렬에도 거의 비슷하게 적용.

- **행렬 만들기**
  - ✓ matrix( ) 함수 호출.
  - ✓ 다음과 같이 벡터를 정의한 후, 행 또는 열의 개수를 설정.

```
> matrix(c(1, 2, 3, 2, 3, 4, 3, 4, 5), ncol = 3)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    3    4
[3,]    3    4    5
```

2.2 행렬

- **행렬 만들기**
  - ✓ 예) ncol = 3 으로 설정해서 해당 벡터를 열이 3개인 행렬 생성 (자동으로 행의 개수는 3).

```
> matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, byrow = FALSE)
     [,1]     [,2]     [,3]
[1,]    1        4        7
[2,]    2        5        8
[3,]    3        6        9
> matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, byrow = TRUE)
     [,1]     [,2]     [,3]
[1,]    1        2        3
[2,]    4        5        6
[3,]    7        8        9
```

2.2 행렬

- **행렬 만들기**
  - ✓ diag( ) 함수: 대각행렬 생성.

```
> diag(1, nrow = 5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
```

2.2 행렬

- **행과 열 이름 정하기**
  - ✓ 가끔 행과 열의 의미가 각각 다를 경우, 이름을 붙이는 것이 유용.

```
> matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, byrow = TRUE, dimnames = list(c("r1",
"r2", "r3"), c("c1", "c2", "c3")))
   c1 c2 c3
r1  1  2  3
r2  4  5  6
r3  7  8  9
```

  - ✓ 행렬을 먼저 만든 이후, rownames( )와 colnames( ) 함수를 사용하여 각각 행과 열의 이름 설정.

```
> m1 <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), ncol = 3)
> rownames(m1) <- c("r1", "r2", "r3")
> colnames(m1) <- c("c1", "c2", "c3")
```

2.2 행렬

- **행렬의 서브-세팅**
  - ✓ 행렬의 서브-세팅: 벡터와 마찬가지로, 행렬에서 원하는 데이터를 추출하는 것.
  - ✓ 행렬은 2차원으로 표현하고 접근 가능한 벡터라고 볼 수 있음.
  - ✓ 2차원으로 행렬 출력 및 접근 가능.
  - ✓ 대괄호 [,] 로 일부 데이터에 접근 가능.
    - ❖ 벡터의 서브-세팅: [ ]
    - ❖ 행렬의 부분 집합을 결정하려면, 각 차원에 벡터 2개를 제공해야 함.
    - ❖ 대괄호의 첫 번째 인수는 행, 두 번째 인수는 열을 선택하는 것을 의미.
    - ❖ 벡터와 마찬가지로, 각 차원에서 다양한 클래스의 벡터 사용 가능.

2.2 행렬

- **행렬의 서브-세팅**
  - ✓ 예) 서브-세팅을 설명하기 위해 행렬 m1을 사용.

```
> m1
    c1 c2 c3
r1   1  4  7
r2   2  5  8
r3   3  6  9
```

  - ✓ 예) 첫 번째 행의 두 번째 열에 해당하는 원소 출력.

```
> m1[1, 2]
[1] 4
```

  - ✓ 원하는 위치에 있는 데이터에 접근 가능.

```
> m1[1:2, 2:3]
    c2 c3
r1   4  7
r2   5  8
```

2.2 행렬

- **행렬의 서브-세팅**
  - ✓ 차원 하나를 비워 놓으면 해당하는 차원의 모든 값이 선택됨.

```
> m1[1,]
c1 c2 c3
 1  4  7
> m1[,2]
r1 r2 r3
 4  5  6
> m1[1:2,]
   c1 c2 c3
r1  1  4  7
r2  2  5  8
> m1[, 2:3]
   c2 c3
r1  4  7
r2  5  8
r3  6  9
```

2.2 행렬

- **행렬의 서브-세팅**
  - ✓ 벡터와 마찬가지로, 음수는 해당 위치를 제외한다는 의미.

```
> m1[-1,]
   c1 c2 c3
r2  2  5  8
r3  3  6  9
> m1[,-2]
   c1 c3
r1  1  7
r2  2  8
r3  3  9
```

2.2 행렬

- **행렬의 서브-세팅**
  - ✓ 행과 열의 이름이 있으면, 문자형 벡터도 사용 가능.

```
> m1[c("r1", "r3"), c("c1", "c3")]
    c1 c3
r1   1  7
r3   3  9
```

  - ✓ 1차원 벡터를 사용하여 행렬의 데이터에 접근 가능.

```
> m1[1]
[1] 1
> m1[9]
[1] 9
> m1[3:7]
[1] 3 4 5 6 7
```

2.2 행렬

- **행렬의 서브-세팅**
  - ✓ 벡터와 마찬가지로, 행렬도 모든 원소가 같은 종류의 객체로 구성.
  - ✓ 어떤 논리식이 주어진다면, 같은 크기의 논리형 행렬을 결과로 얻을 수 있음.

```
> m1 > 3
      c1    c2    c3
r1 FALSE  TRUE  TRUE
r2 FALSE  TRUE  TRUE
r3 FALSE  TRUE  TRUE
```

  - ✓ 벡터의 서브-세팅과 마찬가지로, 다음과 같이 크기가 동일한 논리형 행렬 사용.

```
> m1[m1 > 3]
[1] 4 5 6 7 8 9
```

2.2 행렬

- **행렬 연산자 활용하기**
  - ✓ 벡터에 대한 모든 산술 연산자는 행렬에서도 동일하게 동작.
  - ✓ %*%와 같은 행렬 전용 연산자를 제외하고, 대부분의 연산자는 요소별 연산 수행.

```
> m1 + m1
   c1 c2 c3
r1  2  8 14
r2  4 10 16
r3  6 12 18
> m1 - 2 * m1
   c1 c2 c3
r1 -1 -4 -7
r2 -2 -5 -8
r3 -3 -6 -9
```

```
> m1 * m1
   c1 c2 c3
r1  1 16 49
r2  4 25 64
r3  9 36 81
> m1 / m1
   c1 c2 c3
r1  1  1  1
r2  1  1  1
r3  1  1  1
```

```
> m1 ^ 2
   c1 c2 c3
r1  1 16 49
r2  4 25 64
r3  9 36 81
> m1 %*% m1
   c1 c2  c3
r1 30 66 102
r2 36 81 126
r3 42 96 150
```

2.2 행렬

- **행렬 연산자 활용하기**
  - ✓ t( ) 함수: 전치행렬 생성.

```
> t(m1)
     r1 r2 r3
c1    1  2  3
c2    4  5  6
c3    7  8  9
```

2.3 배열

- **배열 (array)**
  - ✓ 배열은 차원 수가 늘어난 행렬의 확장판.
  - ✓ 배열은 지정된 차원 수(주로 2차원 이상)로 표현하고 접근 가능한 벡터를 의미.

- **배열 만들기**
  - ✓ array( ) 함수: 배열 생성.
  - ✓ 벡터로 데이터를 입력하고, 각 차원에 어떻게 데이터를 배치할지 설정.
  - ✓ 각 차원의 행과 열에 이름 설정 가능.

2.3 배열

- **배열 만들기**
  - ✓ 내부 데이터에 어떻게 접근할 수 있는지 명확하게 확인 가능.
  - ✓ 배열을 만들 때, 각 차원에 이름도 추가 가능.

```
> a1 <- array(c(0, 1, 2, 3, 4, 5, 6, 7, 8, 9), dim = c(1, 5, 2), dimnames =
list(c("r1"), c("c1", "c2", "c3", "c4", "c5"), c("k1", "k2")))
> a1
, , k1

   c1 c2 c3 c4 c5
r1  0  1  2  3  4

, , k2

   c1 c2 c3 c4 c5
r1  5  6  7  8  9
```

2.3 배열

- **배열 만들기**
  - ✓ 이미 만든 배열에 대해 dimnames(x) <- 를 이용하면 여러 문자형 벡터로 된 리스트를 제공하여 각 차원의 이름 설정 가능.

```
> a0 <- array(c(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10), dim = c(1, 5, 2))
> dimnames(a0) <- list(c("r1"), c("c1", "c2", "c3", "c4", "c5"), c("k1", "k2"))
> a0
, , k1

   c1 c2 c3 c4 c5
r1  0  1  2  3  4

, , k2

   c1 c2 c3 c4 c5
r1  5  6  7  8  9
```

2.3 배열

- **배열의 서브-세팅**
  - ✓ 행렬에서 부분 집합 추출하는 것과 동일.
  - ✓ 각 차원에 대한 벡터를 제공하여 배열의 부분 집합 추출 가능.

```
> a1[1,,]
    k1 k2
c1  0  5
c2  1  6
c3  2  7
c4  3  8
c5  4  9
```

2.3 배열

- 배열의 서브-세팅

```
> a1[, 2,]
k1   k2
 1    6
> a1[,,1]
c1   c2   c3   c4   c5
 0    1    2    3    4
> a1[1, 1, 1]
[1] 0
> a1[1, 2:4, 1:2]
     k1   k2
c2    1    6
c3    2    7
c4    3    8
> a1[c("r1"), c("c1", "c3"), "k1"]
c1   c3
 0    2
```

2.3 배열

- **배열의 서브-세팅**
  - ✓ 원소 벡터, 행렬, 배열 모두 성질은 비슷.
  - ✓ 가장 큰 공통점은 동종 데이터 타입(homogeneous data types), 즉, 저장하는 요소의 타입이 동일해야 한다는 것.
  - ✓ R에서는 이기종 데이터 타입(heterogeneous data types)도 존재.
    - ❖ 서로 다른 타입의 요소를 저장할 수 있기 때문에 훨씬 유연.
    - ❖ 메모리 효율성이 떨어지며, 동작 속도가 느림.

2.4 리스트

- **리스트 (list)**
  - ✓ 리스트는 다른 타입의 객체를 비롯하여 또 다른 리스트까지도 포함할 수 있는 일반화된 벡터.
  - ✓ 유연성 측면에서 유용.
  - ✓ 예를 들어 R에 선형 모델을 적용한 결과는 기본적으로 선형 계수(수치형 벡터), 잔차(수치형 벡터), QR 분해(행렬과 기타 다른 객체를 포함하는 리스트) 같은 선형 회귀에서 나올 수 있는 다양한 결과를 포함하는 리스트 객체. 이러한 결과가 모두 한 리스트에 들어 있어 매번 다른 함수를 호출할 필요 없이 정보를 추출하기 매우 편리.

2.4 리스트

- **리스트 만들기**
  - ✓ list( ) 함수: 리스트 생성.
  - ✓ 여러 타입의 객체를 한 리스트에 포함 가능.
  - ✓ 예) 단일 요소 수치형 벡터, 논리형 객체 2개를 포함하는 벡터, 세 값의 문자열을 포함하는 벡터로 구성되는 리스트 생성.

```
> l0 <- list(1, c(TRUE, FALSE), c("a", "b", "c"))
> l0
[[1]]
[1] 1

[[2]]
[1] TRUE FALSE

[[3]]
[1] "a" "b" "c"
```

2.4 리스트

- **리스트 만들기**
  - ✓ 리스트 안에 있는 요소마다 이름 설정 가능.

```
> l1 <- list(x = 1, y = c(TRUE, FALSE), z = c("a", "b", "c"))
> l1
$x
[1] 1

$y
[1] TRUE FALSE

$z
[1] "a" "b" "c"
```

2.4 리스트

- **리스트에서 원소 추출하기**
  - ✓ 가장 일반적인 방법은 달러 기호 $ 를 사용하여, 리스트 요소 값을 이름으로 추출하는 것.

```
> l1 <- list(x = 1, y = c(TRUE, FALSE), z = c("a", "b", "c"))
> l1$x
[1] 1
> l1$y
[1] TRUE FALSE
> l1$z
[1] "a" "b" "c"
> l1$m
NULL
```

  - ✓ 아직 존재하지 않는 m 원소에서 값을 요청하면, NULL 리턴.

2.4 리스트

- **리스트에서 원소 추출하기**
  - ✓ 이중 대괄호 [[n]] 안에 숫자 n을 입력하여, 리스트의 n 번째 원소 값을 추출.
  - ✓ 예) 리스트 l1의 두 번째 구성 요소 추출.

```
> l1[[2]]
[1] TRUE FALSE
```

  - ✓ $ 를 사용할 때와 마찬가지로, [[ ]] 안에 이름을 사용하여 리스트의 구성 요소 추출 가능.

```
> l1[["y"]]
[1] TRUE FALSE
```

2.4 리스트

- **리스트에서 원소 추출하기**
  - ✓ 대부분 연산 전에 어떤 요소를 추출할지 알 수 없으니, [[ ]] 로 값을 추출하는 방법이 더 유연.

```
> member <- "z" # 추출할 객체를 동적으로 결정할 수 있다
> l1[[member]]
[1] "a" "b" "c"
```

2.4 리스트

- **리스트의 서브-세팅**
  - ✓ 리스트에서는 여러 요소를 한꺼번에 추출해야 하는 경우가 많음.
  - ✓ 추출된 여러 구성원은 원래 리스트의 부분 집합으로 된 또 다른 리스트를 구성.
  - ✓ 벡터나 행렬과 마찬가지로, 리스트의 부분 집합을 추출할 때는 대괄호 [ ] 를 사용.
  - ✓ 리스트의 일부 요소를 추출하여, 새로운 리스트에 삽입 가능.

2.4 리스트

- **리스트의 서브-세팅**
  - ✓ 문자형 벡터를 사용하여 이름으로 부분 집합 추출 가능.

```
> l1["x"]
$x
[1] 1
> l1[c("x", "y")]
$x
[1] 1
$y
[1] TRUE FALSE
> l1[1]
```

2.4 리스트

- **리스트의 서브-세팅**
  - ✓ 수치형 벡터를 사용하여 위치로 부분 집합 추출 가능.
  - ✓ 논리형 벡터를 사용하여 어떤 조건을 기준으로 부분 집합 추출 가능.

```
$x
[1] 1
> l1[c(1, 2)]
$x
[1] 1

$y
[1] TRUE FALSE
> l1[c(TRUE, FALSE, TRUE)]
$x
[1] 1

$z
[1] "a" "b" "c"
```

2.4 리스트

- **리스트의 서브-세팅**
  - ✓ 이중대괄호 [[ ]] 는 벡터나 리스트에서 원소 하나를 추출하는 것을 의미.
  - ✓ 대괄호 [ ]는 벡터나 리스트에서 부분 집합을 추출하는 것을 의미.
  - ✓ 벡터를 서브-세팅하면 벡터가 되는 것과 마찬가지로, 리스트를 서브-세팅하면 리스트가 생성.

- **이름이 정해진 리스트**
  - ✓ 리스트의 구성 요소에 이미 이름이 있는지와 상관없이, 원하는 이름을 담은 벡터를 사용하여 간단히 이름을 정하거나 변경 가능.

```
> names(l1) <- c("A","B","C")
> l1
$A
[1] 1

$B
[1] TRUE FALSE

$C
[1] "a" "b" "c"
```

2.4 리스트

- **이름이 정해진 리스트**
  - ✓ 기존 이름 모두 삭제: 리스트의 이름을 NULL로 설정

```
> names(l1) <- NULL
> l1
[[1]]
[1] 1

[[2]]
[1] TRUE FALSE

[[3]]
[1] "a" "b" "c"
```

  - ✓ 이런 식으로 이름을 지우면, 당연히 더 이상 이름으로는 요소에 접근 불가.
  - ✓ 대신, 위치나 논리 조건을 이용해야 함.

2.4 리스트

- **값 할당하기**
  - ✓ 벡터와 마찬가지로, 리스트에 값을 할당하는 방법은 매우 직관적.

```
> l1 <- list(x = 1, y = c(TRUE, FALSE), z = c("a", "b", "c"))
> l1$x <- 0
```

  - ✓ 존재하지 않는 요소에 값을 할당하면, 주어진 이름이나 위치를 가진 새로운 요소가 리스트에 추가.

```
[1] 0

$y
[1] TRUE FALSE

$z
[1] "a" "b" "c"

$m
[1] 4
```

2.4 리스트

- **값 할당하기**
  - ✓ 동시에 여러 값 할당 가능.

```
> l1[c("y", "z")] <- list(y = "new value for y", z = c(1, 2))
> l1
$x
[1] 0

$y
[1] "new value for y"

$z
[1] 1 2

$m
[1] 4
```

2.4 리스트

- **값 할당하기**
  - ✓ 어떤 구성 요소를 삭제하고 싶다면, 해당 요소에 NULL을 할당.

```
> l1$x <- NULL
> l1
$y
[1] "new value for y"

$z
[1] 1 2

$m
[1] 4
```

  - ✓ 다음과 같이 여러 구성 요소를 한꺼번에 삭제 가능.

```
> l1[c("z", "m")] <- NULL
> l1
$y
[1] "new value for y"
```

2.4 리스트

- **기타 함수**
  - ✓ 어떤 객체가 리스트인지 확실히 알고 싶다면, is.list( ) 함수 사용.

```
> l2 <- list(a = c(1, 2, 3), b = c("x", "y", "z", "w"))
> is.list(l2)
[1] TRUE
> is.list(l2$a)
[1] FALSE
```

  - ✓ as.list( ) 함수를 사용하여 벡터를 리스트로 변환 가능.

```
> l3 <- as.list(c(a = 1, b = 2, c = 3))
> l3
$a
[1] 1

$b
[1] 2

$c
[1] 3
```

2.4 리스트

- **기타 함수**
  - ✓ 기본적으로 모든 리스트 안의 요소를 호환 가능한 타입의 벡터로 변환해주는 unlist( ) 함수를 호출하여, 손쉽게 리스트를 벡터로 강제 변환 가능.

```
> l4 <- list(a = 1, b = 2, c = 3)
> unlist(l4)
a b c
1 2 3
```

  - ✓ 숫자와 문자열이 섞여 있는 리스트에 대해 unlist( ) 함수를 호출할 때는 모든 요소가 다 함께 변환될 수 있는 가장 가까운 타입으로 자동 변환.

```
> l4 <- list(a = 1, b = 2, c = "hello")
> unlist(l4)
    a     b       c
  "1"   "2" "hello"
```

  - ✓ l4$a와 l4$b는 숫자라서 문자로 변환할 수 있지만, l4$c는 문자이기 때문에 숫자로 변환 불가.
  - ✓ 이 경우, 모든 요소가 호환되는 가장 가까운 타입은 문자형 벡터.

2.5 데이터 프레임

- **데이터 프레임 (data frame)**
  - ✓ 데이터 프레임은 행과 열이 여러 개 있는 데이터 집합
  - ✓ 행렬처럼 보이지만, 반드시 각 열이 동일한 타입일 필요는 없음.
  - ✓ 가장 보편적으로 사용되는 데이터셋 형식.
  - ✓ 각 행은 '데이터 레코드'를 의미하며, 다양한 타입의 여러 열로 구성.
  - ✓ 예)

이름	성별	나이	전공
Ken	남자	24	금융학
Ashley	여자	25	통계학
Jennifer	여자	23	컴퓨터 과학

2.5 데이터 프레임

- **데이터 프레임 만들기**
  - ✓ data.frame( ) 함수: 데이터 프레임 생성.
  - ✓ 각 열에 맞는 타입의 벡터를 사용하여 설정.

```
> persons <- data.frame(Name = c("Ken", "Ashley", "Jennifer"),
+     Gender = c("Male", "Female", "Female"),
+     Age = c(24, 25, 23),
+     Major = c("Finance", "Statistics", "Computer Science"))
> persons
      Name Gender Age              Major
1      Ken   Male  24            Finance
2   Ashley Female  25         Statistics
3 Jennifer Female  23 Computer Science
```

  - ✓ 데이터 프레임을 만드는 방법은 리스트와 동일.
  - ✓ 근본적으로 데이터 프레임은 테이블의 열을 나타내며, 같은 수의 원소를 갖는 벡터로 구성된 리스트이기 때문.

2.5 데이터 프레임

- **데이터 프레임 만들기**
  - ✓ 원시 데이터에서 데이터 프레임 생성하는 방법.
    - ❖ data.frame( ) 직접 호출.
    - ❖ as.data.frame( )을 호출하여 주어진 리스트를 데이터 프레임으로 변환.

```
> l1 <- list(x = c(1, 2, 3), y = c("a", "b", "c"))
> data.frame(l1)
  x y
1 1 a
2 2 b
3 3 c
> as.data.frame(l1)
  x y
1 1 a
2 2 b
3 3 c
```

2.5 데이터 프레임

- **데이터 프레임 만들기**
  - ✓ 행렬을 통해 데이터 프레임 생성

```
> m1 <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, byrow = FALSE)
> data.frame(m1)
  X1 X2 X3
1  1  4  7
2  2  5  8
3  3  6  9
> as.data.frame(m1)
  V1 V2 V3
1  1  4  7
2  2  5  8
3  3  6  9
```

  - ✓ 행렬을 변환할 때는 새로운 데이터 프레임에 열 이름을 자동 지정.
  - ✓ 행렬에 이미 열이나 행 이름이 있다면, 변환 이후에도 보존.

2.5 데이터 프레임

- **행과 열 이름 정하기**
  - ✓ 데이터 프레임은 리스트이면서 동시에 행렬과 비슷한 형태.
  - ✓ 데이터 프레임에서 데이터에 접근하는 방식은 이 두 가지 성질을 모두 포함.

```
> df1 <- data.frame(id = 1:5, x = c(0, 2, 1, -1, -3), y = c(0.5, 0.2, 0.1, 0.5, 0.9))
> df1
  id  x   y
1  1  0 0.5
2  2  2 0.2
3  3  1 0.1
4  4 -1 0.5
5  5 -3 0.9
```

  - ✓ 행렬과 마찬가지로, 행과 열 이름 변경 가능.

```
> colnames(df1) <- c("id", "level", "score")
> rownames(df1) <- letters[1:5]
> df1
  id level score
a  1     0   0.5
b  2     2   0.2
c  3     1   0.1
d  4    -1   0.5
e  5    -3   0.9
```

2.5 데이터 프레임

- **데이터 프레임의 서브-세팅**
  - ✓ 리스트처럼 서브-세팅하기
    - ❖ 데이터 프레임을 벡터의 리스트로 간주하여 리스트 표기법을 따라 값을 추출하거나 부분 집합을 추출.
    - ❖ 예) $ 기호를 사용하여 이름으로 열 값 추출 또는 [[ ]]에 위치 정보를 이용하여 데이터 접근.

```
> df1$id
[1] 1 2 3 4 5
> df1[[1]]
[1] 1 2 3 4 5
```

2.5 데이터 프레임

- **데이터 프레임의 서브-세팅**
  - ✓ 리스트처럼 서브-세팅하기 (cont.)
    - ❖ [ ] 에 수치형 벡터를 사용하여 위치별로 열 추출.
    - ❖ 문자형 벡터를 사용하여 이름으로 열 추출.
    - ❖ 논리형 벡터를 사용하여 TRUE와 FALSE로 열 추출.

```
> df1[1]
  id
a  1
b  2
c  3
d  4
e  5
```

```
> df1[1:2]
  id level
a  1     0
b  2     2
c  3     1
d  4    -1
e  5    -3
> df1["level"]
  level
a     0
b     2
c     1
d    -1
e    -3
```

```
> df1[c("id", "score")]
  id score
a  1   0.5
b  2   0.2
c  3   0.1
d  4   0.5
e  5   0.9
> df1[c(TRUE, FALSE, TRUE)]
  id score
a  1   0.5
b  2   0.2
c  3   0.1
d  4   0.5
e  5   0.9
```

2.5 데이터 프레임

- **데이터 프레임의 서브-세팅**
  - ✓ 행렬처럼 서브-세팅하기
    - ❖ 리스트 사용법은 행 선택을 지원하지 않음.
    - ❖ 행렬 표기법은 좀 더 유연한 기능 제공.
    - ❖ 2차원 접근 방법을 사용하면, 열과 행 모두 선택 가능.
    - ❖ [행, 열] 표기법을 사용하면, 행과 열을 선택하는 벡터를 지정하여 부분 집합 추출 가능.
    - ❖ 예) 다음과 같이 열 선택자 지정 가능.

```
> df1[, "level"]
[1] 0 2 1 -1 -3
> df1[, c("id", "level")]
   id level
a   1     0
b   2     2
c   3     1
d   4    -1
e   5    -3
> df1[, 1:2]
   id level
a   1     0
b   2     2
c   3     1
d   4    -1
e   5    -3
```

2.5 데이터 프레임

- **데이터 프레임의 서브-세팅**
  - ✓ 행렬처럼 서브-세팅하기 (cont.)
    - ❖ 주의) 행렬 표기법을 사용하면, 출력이 자동으로 단순화.
    - ❖ 즉, 한 열만 선택하면 결과는 데이터 프레임이 아니라, 해당 열의 값.
    - ❖ 결과를 항상 데이터 프레임으로 유지하려면, 열이 하나만 있을 때는 다음과 같이 두 표기법을 함께 사용해야 함.

```
> df1[1:4,]["id"]
  id
a  1
b  2
c  3
d  4
```

    - ❖ 여기서 첫 번째 [ ] 안의 그룹은 첫 번째 행 4개와 모든 열이 선택된 행렬 방식으로 데이터 프레임의 부분 집합 추출.
    - ❖ 두 번째 [ ] 안의 그룹은 id 열만 선택하는 리스트 방식으로 데이터 프레임의 부분 집합 추출.
    - ❖ 결과 객체는 데이터 프레임의 형태.

2.5 데이터 프레임

- **데이터 프레임의 서브-세팅**
  - ✓ 행렬처럼 서브-세팅하기 (cont.)
    - ❖ 결과가 자동으로 단순화되는 것을 피하는 또 다른 방법: drop = FALSE 지정.

```
> df1[1:4, "id", drop = FALSE]
  id
a  1
b  2
c  3
d  4
```

    - ❖ 데이터 프레임의 부분 집합을 데이터 프레임으로 출력하려면 항상 drop = FALSE 로 설정.
    - ❖ 그렇지 않으면, 열 하나만 선택하는 입력에 대해 데이터 프레임이 아닌 벡터가 결과로 리턴.

2.5 데이터 프레임

- **데이터 프레임의 서브-세팅**
  - ✓ 데이터 필터링
    - ❖ 다음 코드는 score >= 0.5 라는 조건을 만족하는 df1의 행을 선택하고, id와 level 열을 선택.

```
> df1$score >= 0.5
[1] TRUE FALSE FALSE TRUE TRUE
> df1[df1$score >= 0.5, c("id", "level")]
   id level
a   1     0
d   4    -1
e   5    -3
```

2.5 데이터 프레임

- **데이터 프레임의 서브-세팅**
  - ✓ 데이터 필터링 (cont.)
    - ❖ 다음 코드는 이름이 a, d, e인 행들과 id, score 열을 선택.

```
> rownames(df1) %in% c("a", "d", "e")
[1] TRUE FALSE FALSE TRUE TRUE
> df1[rownames(df1) %in% c("a", "d", "e"), c("id", "score")]
   id    score
a   1      0.5
d   4      0.5
e   5      0.9
```

    - ❖ 기본적으로 행을 선택할 때는 논리형 벡터, 열을 선택할 때는 문자형 벡터를 사용하는 행렬 표기법

2.5 데이터 프레임

- **값 설정하기**
  - ✓ 데이터 프레임의 부분 집합에 해당하는 값을 설정할 때도 리스트 방식과 행렬 방식 사용 가능.
  - ✓ 리스트처럼 값 설정하기
    - ❖ $와 <- 기호를 사용하여 새로운 값을 리스트 구성 요소로 할당 가능.

```
> df1$score <- c(0.6, 0.3, 0.2, 0.4, 0.8)
> df1
  id level score
a  1     0   0.6
b  2     2   0.3
c  3     1   0.2
d  4    -1   0.4
e  5    -3   0.8
```

2.5 데이터 프레임

- **값 설정하기**
  - ✓ 리스트처럼 값 설정하기 (cont.)
    - ❖ [ ]는 한 표현식에서 여러 열을 동시에 변경 가능.
    - ❖ [[ ]]는 한 번에 한 열만 수정 가능.

```
> df1["score"] <- c(0.8, 0.5, 0.2, 0.4, 0.8)
> df1
   id level score
a   1     0   0.8
b   2     2   0.5
c   3     1   0.2
d   4    -1   0.4
e   5    -3   0.8
```

2.5 데이터 프레임

- **값 설정하기**
  - ✓ 리스트처럼 값 설정하기 (cont.)

```
> df1[["score"]] <- c(0.4, 0.5, 0.2, 0.8, 0.4)
> df1
   id  level  score
a   1      0    0.4
b   2      2    0.5
c   3      1    0.2
d   4     -1    0.8
e   5     -3    0.4
> df1[c("level", "score")] <- list(level = c(1, 2, 1, 0, 0), score = c(0.1, 0.2, 0.3,
0.4, 0.5))
> df1
   id  level  score
a   1      1    0.1
b   2      2    0.2
c   3      1    0.3
d   4      0    0.4
e   5      0    0.5
```

2.5 데이터 프레임

- **값 설정하기**
  - ✓ 행렬처럼 값 설정하기
    - ❖ 좀 더 유연한 방식이 필요하다면 행렬 표기법 사용.

```
> df1[1:3, "level"] <- c(-1, 0, 1)
> df1
   id level score
a   1    -1   0.1
b   2     0   0.2
c   3     1   0.3
d   4     0   0.4
e   5     0   0.5
> df1[1:2, c("level", "score")] <- list(level = c(0, 0), score = c(0.9, 1.0))
> df1
   id level score
a   1     0   0.9
b   2     0   1.0
c   3     1   0.3
d   4     0   0.4
e   5     0   0.5
```

2.5 데이터 프레임

- **요인 (factor)**
  - ✓ 데이터 프레임의 기본 성질 가운데 꼭 기억해야 할 한가지는 바로 메모리를 최대한 효율적으로 사용하려고 한다는 것.
  - ✓ 이 성질 때문에 가끔 예상치 못한 문제가 발생.
  - ✓ 예) 문자형 벡터를 사용하여 데이터 프레임의 열을 만들 때, 문자형 벡터는 자동으로 요인 형태로 변환되고 반복되는 부분이 최대한 없도록 메모리는 최소로 사용하려고 함.
  - ✓ 실제로 요인 타입은 제한된 가능한 값을 표현하는 '레벨'이라는 미리 지정된 가능한 값의 집합에서 나온 '정수형 벡터'라고 볼 수 있음.

2.5 데이터 프레임

이름	성별	나이	전공
Ken	남자	24	금융학
Ashley	여자	25	통계학
Jennifer	여자	23	컴퓨터 과학

- **요인 (factor)**
  - ✓ 예) persons 데이터 프레임에 str( ) 함수를 호출.

```
> str(persons)
'data.frame': 3 obs. of 4 variables:
 $ Name  : Factor w/ 3 levels "Ashley","Jennifer",..: 3 1 2
 $ Gender: Factor w/ 2 levels "Female","Male": 2 1 1
 $ Age   : num 24 25 23
 $ Major : Factor w/ 3 levels "Computer Science",..: 2 3 1
```

  - ✓ 이름, 성별, 전공이 문자형 벡터가 아니라 요인 객체라는 사실을 분명히 확인할 수 있음.
  - ✓ Gender는 Female 또는 Male 둘 중 하나이기 때문에 요인으로 표시되는 것이 합리적.
  - ✓ 이 두 값을 표시하는 데 정수 2개만 사용하는 것이 반복되는 모든 값을 저장하는 문자형 벡터를 사용하는 것보다 훨씬 효율적.
  - ✓ 가능한 값에 제한이 없을 때는 이것이 문제가 될 수 있음.

2.5 데이터 프레임

이름	성별	나이	전공
Ken	남자	24	금융학
Ashley	여자	25	통계학
Jennifer	여자	23	컴퓨터 과학

- **요인 (factor)**
  - ✓ 예) persons 데이터 프레임에 새로운 이름을 입력할 때 발생하는 일.

```
> persons[1, "Name"] <- "John"
Warning message:
In `[<-.factor`(`*tmp*`, iseq, value = "John") :
 invalid factor level, NA generated
> persons
         Name    Gender   Age              Major
1        <NA>      Male    24            Finance
2       Ashley    Female    25         Statistics
3    Jennifer    Female    23    Computer Science
```

  - ✓ 경고 메시지가 발생하는 것은 데이터 프레임을 처음 만들 때 Name 열에는 John이라는 단어가 없었기 때문에 첫 번째 사람의 이름을 없는 값으로 설정할 수 없어 발생한 것.
  - ✓ Gender에 Unknown이라고 설정해도 같은 일이 발생.
  - ✓ 정확히 동일한 이유 때문.
  - ✓ 데이터 프레임을 정의할 때 문자형 벡터로 처음 열을 만들면서 기본적으로 열은 요인이 됨.
  - ✓ 문자형 벡터로 만들어진 열은 해당 문자형 벡터의 고유한 값으로 사전을 생성하고, 이 사전 안에서만 값을 가져와야 하는 요인이 됨.

2.5 데이터 프레임

이름	성별	나이	전공
Ken	남자	24	금융학
Ashley	여자	25	통계학
Jennifer	여자	23	컴퓨터 과학

- **요인 (factor)**
  - ✓ 메모리가 비싸지 않은 요즘에 이러한 특징은 가끔 도움이 되지 않고 오히려 귀찮을 수 있음.
  - ✓ 이러한 성질을 피하는 가장 간단한 방법은 data.frame() 함수로 데이터 프레임을 만들 때 stringsAsFactors = FALSE를 설정하는 것.

```
> persons <- data.frame(Name = c("Ken", "Ashley", "Jennifer"),
+     Gender = factor(c("Male", "Female", "Female")),
+     Age = c(24, 25, 23),
+     Major = c("Finance", "Statistics", "Computer Science"),
+     stringsAsFactors = FALSE)
> str(persons)
'data.frame': 3 obs. of  4 variables:
 $ Name  : chr  "Ken" "Ashley" "Jennifer"
 $ Gender: Factor w/ 2 levels "Female","Male": 2 1 1
 $ Age   : num  24 25 23
 $ Major : chr  "Finance" "Statistics" "Computer Science"
```

  - ✓ 정말 요소 객체로 설정하고 싶다면 나중에 특정 열을 factor() 함수를 활용하여 명시적으로 요소로 설정할 수 있음.
  - ✓ 이전에 살펴보았듯이, Gender 열이 이 경우에 아주 적합함.

2.5 데이터 프레임

이름	성별	나이	전공
Ken	남자	24	금융학
Ashley	여자	25	통계학
Jennifer	여자	23	컴퓨터 과학

- **데이터 프레임에 유용한 함수**
  - ✓ summary( ) 함수: 데이터 프레임의 각 열에 대한 기본 통계량을 나타내는 테이블 제공.

```
> summary(persons)
     Name       Gender       Age            Major
Length:3           Female:2   Min.   :23.0   Length:3
Class :character   Male  :1   1st Qu.:23.5   Class :character
Mode  :character               Median :24.0   Mode  :character
                               Mean   :24.0
                               3rd Qu.:24.5
                               Max.   :25.0
```

  - ✓ Gender 요인은 각 값 혹은 레벨이 있는 행의 개수를 측정하여 표시.
  - ✓ 수치형 벡터는 열에 있는 수치 값들의 중요한 분위수를 표시.
  - ✓ 다른 유형은 열의 길이, 클래스, 모드를 표시.

이름	성별	나이	전공
Ken	남자	24	금융학
Ashley	여자	25	통계학
Jennifer	여자	23	컴퓨터 과학

2.5 데이터 프레임

- **데이터 프레임에 유용한 함수**
  - ✓ rbind( ), cbind( ) 함수: 데이터 프레임을 행, 열로 바인딩.
  - ✓ 데이터 프레임에 행을 추가하려면, rbind( ) 함수를 사용하여 새로운 레코드 추가.

```
> rbind(persons, data.frame(Name = "John", Gender = "Male", Age = 25, Major =
"Statistics"))
        Name  Gender  Age            Major
1        Ken    Male   24          Finance
2      Ashley  Female   25       Statistics
3   Jennifer  Female   23 Computer Science
4       John    Male   25       Statistics
```

  - ✓ 사람마다 등록은 했는지(Registered), 프로젝트 개수는 몇 개인지(Projects)를 나타내는 새로운 열을 추가하고 싶다면, cbind( ) 함수를 사용함.

```
> cbind(persons, Registered = c(TRUE, TRUE, FALSE), Projects = c(3, 2, 3))
        Name  Gender  Age            Major Registered Projects
1        Ken    Male   24          Finance       TRUE        3
2      Ashley  Female   25       Statistics       TRUE        2
3   Jennifer  Female   23 Computer Science      FALSE        3
```

  - ✓ rbind( ), cbind( ) 함수를 사용하면, 원래 데이터는 변경되지 않고, 주어진 행과 열이 추가된 새로운 데이터 프레임을 만드는 것.

2.5 데이터 프레임

- **데이터 프레임에 유용한 함수**
  - ✓ expand.grid( ) 함수: 각 열에 가능한 모든 값의 조합을 포함하는 데이터 프레임 생성.

```
> expand.grid(type = c("A", "B"), class = c("M", "L", "XL"))
  type class
1    A     M
2    B     M
3    A     L
4    B     L
5    A    XL
6    B    XL
```

2.5 데이터 프레임

- **데이터 읽고 쓰기**
  - ✓ R은 파일에서 테이블을 읽거나 파일에 데이터 프레임을 저장하는 여러 가지 함수를 제공.
  - ✓ 파일에 테이블을 저장한다면, 행과 열의 배치를 지정하는 일반적인 규칙을 따름.
  - ✓ 대부분은 read.table()이나 read.csv() 같은 함수를 호출.
  - ✓ 가장 대중적인 데이터 형식은 CSV(Comma-Separated Values).
    - ❖ 기본적으로 서로 다른 열에 해당하는 값을 쉼표로 구분.
    - ❖ 첫 번째 행은 기본적으로 헤더로 간주.
    - ❖ 예)개인 정보를 담은 데이터를 다음과 같이 CSV 형식으로 표현 가능.

persons.csv

```
Name,Gender,Age,Major
Ken,Male,24,Finance
Ashley,Female,25,Statistics
Jennifer,Female,23,Computer Science
```

2.5 데이터 프레임

- **데이터 읽고 쓰기**
  - ✓ 가장 대중적인 데이터 형식은 CSV(Comma-Separated Values). (cont.)
    - ❖ 파일에 있는 데이터를 R 환경으로 읽어 오려면 read.csv(file) 호출.
      - 여기서 file은 데이터 파일의 경로를 의미.
    - ❖ 데이터 파일을 확실히 찾을 수 있게 data 폴더를 작업 디렉터리 아래에 만들고 파일을 옮김.
    - ❖ getwd() 함수: 작업 디렉터리를 리턴.

```
> read.csv("data/persons.csv")
      Name   Gender  Age              Major
1      Ken     Male   24            Finance
2  Ashley   Female   25         Statistics
3 Jennifer   Female   23 Computer Science
```

    - ❖ 어떤 데이터 프레임을 CSV 파일에 저장하려면, 다음과 같이 몇 가지 추가적인 인수와 함께 write.csv(file) 호출.

```
> write.csv(persons, "data/persons.csv", row.names = FALSE, quote = FALSE)
```

      - row.names = FALSE 는 행 이름을 저장하지 않는다는 의미.
      - quote = FALSE 는 문자열에 따옴표를 생략한다는 의미.

2.6 함수

- **함수 (function)**
  - ✓ 함수란 호출이 가능한 객체.
  - ✓ 기본적으로 입력(매개변수 또는 인수)을 받아 출력 값을 반환하는 내부 논리가 존재하는 시스템.
  - ✓ 사실 R 환경에서 우리가 사용하는 모든 것은 객체이며, 실행하는 모든 것은 함수.
  - ✓ R에서는 모든 함수 역시 객체.
  - ✓ <-와 + 또한 인수 2개를 취하는 함수.
  - ✓ 이진 연산자라고 하는 것들 역시 본질적으로는 함수.
  - ✓ 인터렉티브한 데이터 분석을 할 때는 기본 내장된 함수와 패키지 수천 개에서 제공하는 함수만으로도 충분하므로 스스로 함수를 만들 필요가 거의 없음.
  - ✓ 데이터 조작이나 분석에서 어떤 로직이나 프로세스를 반복해야 할 때, 기존 함수가 특정 작업의 요구 사항이나 특별한 데이터 형식을 만족하도록 설계한 것이 아니기에 이러한 함수는 사용자 목적을 완전히 충족시키지 못할 수 있음. → 특정 요구 사항에 맞는 함수를 직접 작성해야 함.

2.6 함수

- **함수 만들기**
  - ✓ 예) 수치 값 x와 y 를 각각 입력 받아 더하는 add 함수 만들기.

```
> add <- function(x, y) {
+     x + y
+ }
```

  - ✓ (x, y)는 해당 함수의 입력 인수.
  - ✓ {x + y}는 x, y처럼 사용 가능한 심벌들로 구성된 표현식을 포함하는 함수의 몸통 부분.
  - ✓ return() 함수를 명시적으로 호출하지 않는 이상, 마지막 표현식의 값이 함수의 반환 값을 결정.
  - ✓ 마지막으로 이렇게 만든 함수를 add에 할당하여 이후 add 함수를 호출할 수 있게 됨.
  - ✓ R에서 함수는 객체. add 객체가 어떤 함수인지 보려면 콘솔 창에 add 입력.

```
> add
function(x, y) {
x + y
}
```

2.6 함수

- **함수 호출하기**
  - ✓ 함수 호출에는 함수명(인수 1, 인수2, ...) 같은 형식의 구문이 필요.

```
> add(2, 3)
[1] 5
```

  - ✓ 함수 호출 구조.
    - ❖ R은 먼저 현재 환경에 add 함수의 정의 여부 확인.
    - ❖ add가 앞서 작성한 함수를 참조.
    - ❖ x가 2, y가 3인 로컬 환경 확인.
    - ❖ 함수 안의 표현식은 이 인수 값을 고려하여 평가.
    - ❖ 마지막으로 add 함수는 표현식의 결과값인 5를 반환.

2.6 함수

- **동적 타이핑**
  - ✓ R의 함수는 입출력의 형식, 즉 타입에 엄격하지 않기 때문에 좀 더 유연.
  - ✓ 입력 유형은 호출하기 전에는 미리 고정되어 있지 않음.
  - ✓ 본래 함수가 스칼라 값에서 동작하도록 설계했다 하더라도, 자동으로 일반화되어 + 연산자와 연동되는 모든 벡터에서도 연산을 수행.
  - ✓ 예) 함수 변경 없이 다음 코드를 실행 가능.

```
> add(c(2, 3), 4)
[1] 6 7
```

  - ✓ 사실 스칼라 값 역시 R에서는 벡터로 다루므로, 엄밀히 말해 앞 예제는 실제로 동적 타이핑의 유연성을 보여 주지는 않음.

```
> add(as.Date("2014-06-01"), 1)
[1] "2014-06-02"
```

    - ❖ 이 함수는 별도의 타입 확인 없이 인수 2개를 그대로 표현식에 입력.
    - ❖ as.Date() 함수: 날짜를 표현하는 Date 객체 생성.
    - ❖ add 함수는 추가 변경 없이도 Date 객체와 완벽하게 호환.
    - ❖ 다음과 같이 두 인수에서 + 연산자가 잘 호환되지 않는다면 함수 호출은 실패.

```
> add(list(a = 1), list(a = 2))
Error in x + y : non-numeric argument to binary operator
```

2.6 함수

- **함수 일반화**
  - ✓ 함수란 특정 문제를 푸는 논리나 프로세스 집합을 잘 정의하여 추상화한 것.
  - ✓ 개발자라면 보통 광범위한 경우에 적용이 가능하도록 함수를 일반화.
  - ✓ R처럼 타이핑이 강하지 않은 프로그래밍 언어에서는 쉽게 함수 일반화 가능.
  - ✓ add( )를 좀 더 일반화하여 다양한 기본 산술 연산을 처리할 수 있게 calc( ) 같은 또 다른 함수를 정의할 수 있음. 이 새로운 함수는 두 벡터 x, y와 사용자가 원하는 연산을 입력 받는 문자형 벡터 type, 이렇게 세 가지 인수를 받음. 다음 코드는 뒤에서 다룰 흐름 제어를 활용하여 구현.

```
> calc <- function(x, y, type) {
+     if (type == "add") {
+         x + y
+     } else if (type == "minus") {
+         x - y
+     } else if (type == "multiply") {
+         x * y
+     } else if (type == "divide") {
+         x / y
+     } else {
+         stop("Unknown type of operation")
+     }
+ }
```

2.6 함수

- **함수 일반화**
  - ✓ 함수를 정의했으니 적당한 값을 넣어 함수를 호출해보자.

```
> calc(2, 3, "minus")
[1] -1
```

  - ✓ 자동으로 수치형 벡터를 사용한 연산 가능.

```
> calc(c(2, 5), c(3, 6), "divide")
[1] 0.6666667 0.8333333
```

  - ✓ + 연산자가 잘 통하는 다른 유형의 벡터에서도 사용할 수 있게 일반화.

```
> calc(as.Date("2014-06-01"), 3, "add")
[1] "2014-06-04"
```

  - ✓ 적합하지 않은 값을 인수에 적용한다면?

```
> calc(1, 2, "what")
Error in calc(1, 2, "what") : Unknown type of operation
```

  - ✓ 가장 마지막의 else 구문 안의 표현식 수행
  - ✓ stop( ) 함수 호출은 오류 메시지와 함께 전체 함수 연산 즉시 중단.

2.6 함수

- **함수 일반화**
  - ✓ 적합하지 않은 인수에 대한 모든 가능한 경우를 고려했는가?

```
> calc(1, 2, c("add", "minus"))
[1] 3
Warning message:
In if (type == "add") { :
 the condition has length > 1 and only the first element will be used
```

    - ❖ type 인수가 다중 요소 벡터로 주어질 경우는 고려하지 않음.
    - ❖ 벡터를 다른 벡터와 비교하면 다중 요소 논리형 벡터가 발생.
    - ❖ if 조건문을 모호하게 만드는 다중 요소 논리형 벡터가 발생.
    - ❖ if (c(TRUE, FALSE))가 무엇을 의미하는지 알 수 없음.

2.6 함수

- **함수 일반화**
  - ✓ 적합하지 않은 인수에 대한 모든 가능한 경우를 고려했는가? (cont.)
    - ❖ 이 문제를 해결하기 위해서는 벡터 길이가 1인지 여부만 확인하면 된다.

```
> calc <- function(x, y, type) {
+     if (length(type) > 1L) stop("Only a single type is accepted")
+     if (type == "add") {
+         x + y
+     } else if (type == "minus") {
+         x - y
+     } else if (type == "multiply") {
+         x * y
+     } else if (type == "divide") {
+         x / y
+     } else {
+         stop("Unknown type of operation")
+     }
+ }
```

    - ❖ 인수를 미리 확인하여 예외 처리되는 것을 확인할 수 있음.

```
> calc(1, 2, c("add", "minus"))
Error in calc(1, 2, c("add", "minus")) : Only a single type is accepted
```

2.6 함수

- **함수 인수의 기본값**
  - ✓ 어떤 함수는 광범위한 입력을 허용하고 다양한 요구를 만족시키는 것을 볼 때 아주 유연하다고 할 수 있음.
  - ✓ 대부분 유연성이 높아질수록 인수 개수 역시 증가.
  - ✓ 유연한 함수를 사용할 때마다 인수 수십 개를 매번 지정해야 한다면 코드가 아주 지저분해질 것.
  - ✓ 적절한 인수의 기본값을 사용할 수 있다면 함수를 호출하는 코드는 한결 간단해질 것.
  - ✓ arg = value 같은 식으로 인수의 기본값을 설정.
  - ✓ 해당 인수는 선택적 인수.
  - ✓ 예) 선택적 인수를 사용하여 함수 구현.

```
> increase <- function(x, y = 1) {
+     x + y
+ }
```

2.6 함수

- **함수 인수의 기본값**
  - ✓ 새로 만든 increase() 함수는 x만으로도 호출 가능.
  - ✓ 명시적으로 값을 지정하지 않는 이상, y는 자동으로 1.

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

  - ✓ R 함수는 대부분 인수를 여러 개 갖고, 그중 일부는 기본값으로 설정.
  - ✓ 인수의 기본값은 다수 사용자의 성향에 크게 의존하므로 이 기본값을 결정하는 것이 까다로울 수 있음.

2.7 마치며

- **마치며**
  - ✓ 벡터는 동일한 유형의 요소만 저장할 수 있는 동종 데이터 유형.
  - ✓ 반대로 리스트와 데이터 프레임은 다른 유형의 요소를 저장하는 것이 가능하다는 점에서 더 융통성이 있다고 볼 수 있음.
  - ✓ 이 데이터 구조에서 부분 집합을 추출하거나 개별 요소를 추출하는 방법을 배움.
  - ✓ 마지막으로 함수 생성과 호출도 살펴봄.
  - ✓ 이제 게임 규칙을 어느 정도 알았으니 놀이터에 익숙해져야 함.