2. 데이터 타입

2.1 변수

데이터 타입 설명에 앞서, 값을 저장하고 추출하는 방법을 알아보기 위해 R에서 사용하는 변수에 대해 알아보자 .

2.1.1 변수 이름

R의 변수명은 알파벳, 숫자,_(언더스코어), .(마침표)로 구성되며, -(하이픈)은 사용할 수 없다.

첫 글자는 알파벳 또는 .으로 시작해야 한다. 만약 .으로 시작한다면 . 뒤에는 숫자가 올 수 없다. 예를 들어 다음은 모두 올바른 변수명이다 .

a, b, a1, a2, .x

다음은 올바르지 않은 변수명이다.

2a, .2 , a-b

R에서는 다른 언어에서 흔히 _를 사용할 만한 상황에서 .를 사용한다. 예를 들어, training_data, validation_data 같은 변수명 대신 data.training, data.validation과 같이 마치 객체의 속성을 접근하는 것처럼 보이는 명명 방식이 종종 사용된다.

2.1.2 변수값 할당

변수에 값을 할당할 때는 <- , <<- 또는 = 연산자를 사용한다. <-와 <<-는 차후에 설명하겠지만 어느 scope에 있는 변수에 값을 할당하는지에 차이가 있다. 그러나 <- 와 =는 대부분의 경우에 바꿔 사용할 수 있고, 따라서 둘 중 어느 것을 사용해야 하는지 다소 논쟁적인 주제다.

cf) <- 와 = 차이

할당 연산자 중 =는 명령의 최상위 수준에서만 사용할 수 있는 반면 <-는 어느 곳에서나 사용할 수 있다. 따라서 함수 호출과 동시에 변수에 값을 할당하는 목적으로는 <- 만 사용할 수 있다. 다음은 1.2.3을 저장한 벡터를 x에 할당한 후, 이 값들의 평균을 구하고 x에 저장된 값을 출력한 예이다.

```
> mean(x<-c(1,2,3))
[1] 2
> x
[1] 1 2 3
```

같은 상황에서 =를 사용하면 x에 값이 저장되지 않는다.

```
> mean(z=c(1,2,3))
Error in mean.default(z = c(1, 2, 3)) :
  argument "x" is missing, with no default
```

이 차이 때문에 =를 사용하면 경우에 따라 표현력에 한계가 있을 수 있다.

2.2 함수 호출 시 인자 지정

R명령의 대부분은 함수 호출로 이뤄진다. 그런데 함수 호출 시 인자를 지정하는 방법이 다른 언어와 다소 차이가 있어 이에 대해 확실히 알아둘 필요가 있다.

R의 함수 인자는 위치 또는 읾을 지정할 수 있다. 예를 들어, 다음과 같은 가상의 함수를 생각해보자.

```
foo(a,b,c=1,d=2)
```

foo 함수는 a,b,c,d 라는 4개의 인자를 받을 수 있으며 c와 d에는 기본값이 지정되어 있으므로 생략이 가능하다. 만약 c 또는 d를 인자로 전달하지 않으면 각각 기본값인 c=1, d=2로 지정된다.

예를 들어, 다음 명령들은 인자 위치에 따라 a=3, b=4, c=1, d=2를 지정하여 함수 foo()를 호출한다.

```
foo(3,4)
foo(3,4,1)
foo(3,4,1,2)
```

다음 명령은 a=3,b=4,c=5를 지정하되 d는 생략했으므로 d=2가 지정되어 foo()를 호출한다.

```
foo(3,4,5)
```

만약 인자의 위치가 기억나지 않거나, 인자 c는 지정하지 않으면서 인자 d를 지정하고 싶은 경우 처럼 프로트타입의 인자 순서와 다르게 인자를 넘겨주고 싶다면 인자 이름을 명시하여 값을 넘겨 줄 수 있다. 또 순서를 지켜 인자를 지정하는 방법과 인자의 이름을 지정하는 방법을 혼용해서 사용할 수도 있다.

다음 명령들은 a=3, b=4, d=5를 지정하며, c를 생략했으므로 기본값 1이 지정되어 함수 foo()를 호출한다.

```
foo(a=3,b=4,d=5)
foo(d=5,a=3,b=4)
foo(3,4,d=5)
```

R에서는 R콘솔에서 명령을하나씩 실행해보면서 개발하는 대회식 개발 방식을 종종사용한다. 이런 방식의 개발을 편하게 할 수 있도록 R에서는 함수의 인자이름을 일부만 명시해도 되도록 허용하고 있다. 인자 이름을 일부만 지정하는 경우를 설명하기 위해 다음 함수를 가정해보자.

```
foo(foobar, foocar=3)
```

foobar에 1을 지정해 이 함수를 호출하는 일반적인 방법은 foo(foobar=1) 또는 foo(1)이다. 그러나 다음에 보인 것처럼 foobar와 foocar를 구분할 수 있을 정도만 인자 이름을 명시해도 된다.

```
foo(foob=1)
foo(fooba=1)
```

두 가지 경우 모두 foobar=1을 인자로 해서 함수 foo()를 호출한다.

2.3 스칼라

이 절에서는 스칼라 데이터 타입에 관해 살펴본다. 스칼라란 단일 차원의 값을 뜻하는 것으로 숫자 1,2,3,...을 예로 들 수 있다. 반면 좌표 평면 위에 있는 점 (1,2)은 2차원 값이므로 이 절에서 설명하는 스칼라에 해당하지 않는다.

R에서 데이터 타입의 기본은 벡터이다. 따라서 스칼라 데이터는 길이가 1인 벡터(즉, 길이가 1인 배열)와 같은 거으로 볼 수 있다.

2.3.1 숫자

정수, 부동소수 등을 숫자형 데이터 타입으로 지원한다. 다음은 a에 3, b에 4.5를 저장한 뒤 c에 a와 b의 합을 저장하고, c의 값을 출력한 예이다. 값을 출력하려면 print()함수를 다른 언어처럼 호출해 쓰거나 마지막 행에 보인 예와 같이 print() 없이 변수명을 그대로 입력하면 된다.

```
> a<-3
> b<-4.5
> c<-a+b
> print(c)
[1] 7.5
> c
[1] 7.5
```

2.3.2 NA

R과 다른 언어의 가장 큰 차이 중 하나가 바로 NA(Not Available) 상수다. NA는 데이터 값이 없음을 뜻한다. 예를 들어, 4명의 시험 점수가 있을 때 3명의 점수는 각각 80,90,75지만 4번째 사람의 점수를 모를 경우 NA를 이용해 4번째 사람의 점수를 표현한다.

```
> one<-80
> two<-90
> three<-75
> four<-NA
> four
[1] NA
```

변수에 NA값이 저장되어 있는지는 is.na()함수로 확인한다.

```
> is.na(four)
[1] TRUE
```

2.3.4 NULL

NULL은 NULL 객체를 뜻하며, 변수가 초기화되지 않았을 때 사용한다. NULL은 NA와 구분해서 생각해야 한다. 어떤 변수에 NULL이 저장되어 있는지는 is.null()을 사용해 판단할 수 있다.

다음은 NULL을 변수에 저장하고 이를 is.null()로 확인하는 예이다.

```
> x<-NULL
> is.null(x)
[1] TRUE
> is.null(1)
[1] FALSE
> is.null(NA)
[1] FALSE
> is.na(NULL)
logical(0)
```

NOTE NULL과 NA의 차이

NA는 결측치, 즉 값이 빠져 있는 경우를 뜻한다. 결측치가 존재하는 이유로는 데이터 입력 중 실수로 값을 입력하지 않은 경우, 값을 어떤 이유로든 관찰되지 못한 경우(예를 들어, 인구 조사에서 특정 가구가 소득을 기재하지 않은 경우), 마지막으로 해당 항목에 적절한 값이 없어서 값이 입력되지 않은 경우(예를 들어, 약품의 냄새를 기록하고 있는 칸에서 특정 약품은 향이 없는 경우)를 둘 수 있다.

반면 NULL은 프로그래밍의 편의를 위해 미정(undefined) 값을 표현하는 데 사용하는 개념이다. is_even 이라는 변수에 a변수의 값이 짝수면 TRUE, 홀수면 FALSE를 저장하는 다음 예를 보자.

```
is_even<-NULL
if(a가 짝수면){
is_even<-TRUE
}
else{
is_even<-FALSE
}
```

위 코드에서 if 조건문이 실행되기 전에는 is_even에 어떤 값을 줘야 할지 알 수없어 NULL로 초기화했다. 그리고 if문을 지나면서 is_even에 적절한 값이 할당되었다. 이처럼 NULL은 변숫값이 아직 미정인 상태를 표현하는 목적으로 사용한다.

2.3.5 문자열

R에는 C등의 언어에서 볼 수 있는 한 개 문자에 대한 데이터 타입(예를 들면, C의 char 데이터 타입)이 없다. 대신 문자열로 모든 것을 표현한다. 문자열은 'this_is_string' 또는 "this_is_string" 과 같이 어느 따옴표로 묶어도 무관하다.

```
> a<-"hello"
> print(a)
[1] "hello"
> a<-'hello'
> print(a)
[1] "hello"
```

2.3.6 진릿값

True, T는 모두 참 값을 의미한다. FALSE, F는 거짓 값을 의미한다. 진릿값에는 &(AND), |(OR), !(NOT) 연산자를 사용할 수 있다.

좀 더 엄밀히 말하면 TRUE와 FALSE는 예약어이고 T,F는 각각 TRUE와 FALSE로 초기화된 전역 변수이다. 따라서 다음과 같이 T에 FALSE를 할당하는 것이 가능하다! 반면 TRUE는 예약어이므로 FALSE를 할당할 수 없다. 이런 이유로 TRUE, FALSE 대신 T,F라는 축약 표현을 사용할 때는 주의가 필요하다 (가능하면 T,F 대신 TRUE, FALSE를 명시적으로 사용하는 것이 좋다.)

```
> T<-FALSE
> TRUE <- FALSE
Error in TRUE <- FALSE : invalid (do_set) left-hand side to assignment
```

AND나 OR 연산자에도 &,|외에도 &&와 ||가 있다. 이들의 차이점은 &,|는 진릿값이 저장된 벡터(배열)끼리 연산할 때 요소별로 계산을 한다는 점이다. 예를 들어, 다음 코드에서는 TRUE, TRUE가 저장된 벡터와 TRUE, FALSE가 저장된 벡터 간에 &(AND) 연산을 수행했다. 그 결과 두 벡터의 첫 번째 요소끼리의 연산 결과는 TRUE && TRUE = TRUE, 두 번째 요소끼리의 연산 결과는 TRUE && FALSE = FALSE 가 되었다.

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

반면 &&는 벡터의 요소 간 계산을 하기 위함이 아닌 TRUE && TRUE 등의 경우와 같이 두 개의 진릿값끼리 연산을 하기 위한 연산자다. 이는 ||와 |의 경우에도 마찬가지다. 예를 들어, 다음 코드를 보면 한 개의 값만 반환됨을 볼 수 있다.

이를 미뤄보면 언제 &, |를 써야 하고 언제 &&, ||를 써야 하는지 명확해진다. &, |는 벡터에 저장된 진릿값 간에 대량으로 논리 연산을 수행할 때 사용한다. &&, ||는 한 개의 진릿값만 필요한 if문 등에서 사용한다.

또 &&, ||는 쇼트 서킷을 지원한다. 따라서 A && B 형태의 코드가 있을 때 A가 TRUE라면 B도 평가하지만, A가 FALSE라면 B를 평가하지 않는다.

언뜻 보기에는 &&나 ||를 &, |보다 많이 사용해야 할 것 같지만, R에서는 벡터나 리스트 내 요소를 한번에 비교하는 연산이 많으므로 오히려 &나 |가 더 유용하다.

2.3.7 팩터

팩터는 범주형 데이터(자료)를 표현하기 위한 데이터 타입이다.

범ㅈ형 데이터란 데이터가 사전에 정해진 특정 유형으로만 분류되는 경우를 뜻한다.

예를 들어, 바으이 크기를 대, 중, 소로 기재하고 있을 때 특정 방의 크기를 '대'라고 적는다면 이 값은 범주형 데이터다. 이와 같이 범주형 변수가 담을 수 있는 값의 목록을 레벨이라고 한다. 따라서 범주형 데이터를 저장하는 데이터 타입인 팩터에는 관측된 값뿐만 아닌 관측 가능한 값의 레벨도 나열해야 한다.

범주형 데이터는 또 다시 명목형과 순서형으로 구분된다.

명목형 데이터(Nominal)는 값들 간에 크기 비교가 불가능한 경우를 뜻한다. 예를 들어, 정치적 경향을 좌파, 우파로 구분하여 저장한 데이터는 명목형이다.

반면 **순서형 데이터**는 대, 중, 소와 같이 값에 순서를 둘 수 있는 경우를 말한다.

범주형 데이터에 상반하는 개념에는 수치형 데이터가 있다. 수치형 데이터는 값을 측정하거나 개수를 세는 경우와 같이 숫자로 나온 값을 의미한다. 예를 들어, 방의 크기를 $30m^2$,$28m^2$ 와 같이 기록하는 경우나 학생들의 성적이 이에 해당한다.

팩터 관련 함수

factor : 팩터 값을 생성한다.

```
factor(
    x, #팩터로 표현하고자 하는 값(주로 문자열 벡터로 지정)
    levels, #값의 레벨
    ordered #TRUE면 순서형, FALSE면 명목형 데이터를 뜻한다.
    )
#반환 값은 팩터형 데이터 값이다.
```

nlevels : 펙터에서 레벨의 개수를 반환한다.

```
nlevels(
    x # 팩터 값
    )
#반환 값은 팩터 값의 레벨 개수다.
```

levels : 팩터에서 레벨의 목록을 반환한다.

```
levels(
    x # 팩터 값
    )
#반환 값은 팩터에서 레벨의 목록이다.
```

is.factor : 주어진 값이 팩터인지 판단한다.

```
is.factor(
    x # R 객체
    )
#반환 값은 x가 팩터면 TRUE, 그렇지 않으면 FALSE이다.
```

ordered : 순서형 팩터를 생성한다.

```
ordered(
    x #팩터로 표현하고자 하는 값(주로 문자열 벡터로 지정)
    )
```

is.ordered : 순서형 팩터인지를 판단한다.

```
is.ordered(
    x #R 객체
    )
#반환 값은 x가 순서형 팩터면 TRUE, 그렇지 않으면 FALSE이다.
```

예를 들어, 성별을 팩터로 만드는 경우를 생각해보자. 성별은 범주형 데이터 중 명목형 데이터에 해당하며, "m"(남성male)과 "f"(여성female) 두 가지 값이 가능하다. 따라서 남성을 저장한 변수 sex를 다음과 같이 생성할 수 있다.

```
> sex <- factor("m",c("m","f"))
> sex
[1] m
Levels: m f
```

위 코드에서 sex에는 "m"이 저장되었고, 이 팩터가 담을 수 있는 값의 레벨은 "m","f"로 제한되었다.

팩터 변수는 nlevels()로 레벨의 수를 알 수 있고, levels()로 레벨의 목록을 볼 수 있다.

```
> nlevels(sex)
[1] 2
> levels(sex)
[1] "m" "f"
```

levles()의 반환 값은 벡터며, 벡터는 다른 언어의 배열처럼 사용할 수 있다. 따라서 각 레벨의 값을 다음과 같이 구할 수 있다. R에서 색인(인덱스)은 0이 아닌 1부터 시작한다는 점에 유의하기 바란다.

```
> levels(sex)[1]
[1] "m"
> levels(sex)[2]
[1] "f"
```

팩터 변수에서 레벨 값을 직접 수정하고자 한다면 levels()에 값을 할당하면 된다. "m"을 "male"로, "f"을 "female"로 바꿔보자.

```
> sex
[1] m
Levels: m f
> levels(sex)<-c("male","female")
> sex
[1] male
Levels: male female
```

여러 개의 값을 팩터로 만들고자 한다면 factor()의 인자 x에 벡터를 지정한다.

```
> factor(c("m","m","f"),c("m","f"))
[1] m m f
Levels: m f
> factor(c("m","m","f"))
[1] m m f
Levels: f m
```

levels 인자를 생략하면 데이터로부터 자동으로 레벨의 목록을 파악한다.

factor()는 기본적으로 데이터에 순서가 없는 명목형 데이터를 만든다.

ordered()를 사용하거나 factor() 호출 시 ordered = TRUE를 지정한다.

```
> ordered("a",c("a","b","c"))
[1] a
Levels: a < b < c
```

앞서와 달리 'Levels'에 순서가 'a<b<c'로 표시되어 있음을 볼 수 있다.

2.4 벡터

벡터는 다른 프로그래밍 언어에서 흔히 접하는 배열의 개념으로, 한 가지 스칼라 데이터 타입의 데이터를 저장할 수 있다. 예를 들어, 숫자만 저장하는 배열, 문자열만 저장하는 배열이 벡터에 해당한다.

R의 벡터는 슬라이스를 제공한다. 슬라이스란 배열의 일부를 잘라낸 뒤 이를 또 다시 배열처럼 다루는 개념을 뜻한다.

또한 벡터의 각 셀에는 이름을 부여할 수 있다. 따라서 벡터에 저장된 요소들을 색인을 사용해서 접근한 ㄴ 것 뿐 아니라 이름을 사용해서도 접근할 수 있다. 이런 특징을 사용하면 데이터를 좀 더 의미있는 형태로 저장할 수 있다.

2.4.1 벡터 생성

벡터는 c()를 사용해 생성하고, names()를 사용해 이름을 부여할 수 있다. 아래에 벡터 관련 함수를 정리했다.

벡터 관련 함수

c: 주어진 값들을 모아 벡터를 생성한다.

```
c(
  ...     #벡터로 모을 R객체들
  )
#반환 값은 벡터다.
```

names : 객체의 이름을 반환한다.

```
names(
    x #이름을 얻어올 R객체
    )
#반환 값은 x와 같은 길이의 문자열 벡터 또는 NULL이다.
```

names<- : 객체에 이름을 저장한다.

```
names (
    x #이름을 저장할 R객체
)<- value #저장할 이름
```

cf) names()는 값을 얻어오는 함수고, names<-() 는 값을 할당하는 함수다. 이 생소한 문법의 할당 함수가 다른 언어와 다른 점 중 하나다.

하나씩 실습해보자,

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

나열하는 인자들은 벡터의 정의대로 한 가지 유형의 스칼라 타입이어야 한다. **만일 서로 다른 타입의 데이터를 섞어서 벡터에 저장하면, 이들 데이터는 한 가지 타입으로 자동 형 변환된다.** 이때 사용되는 형 변환 규칙은 좀 더 표현력이 높은 데이터 타입으로 변환하는 것이다.

cf) 정확한 변환 규칙은 NULL<raw <logical <integer <double <complex <character < list < expression 순서다.

예를 들어, 정수와 부동소수가 섞여 있다면 모두 부동소수로 변환되며, 정수와 문자열이 섞여 있다면 모두 문자열로 변환된다. 예를 들어, 아래 코드에서 숫자형 데이터인 2는 "2"라는 문자열 형태로 자동으로 변환되어 x안에는 문자열 형태의 데이터만 나열된다.

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

그러나 이런 형 변환규칙을 모두 기억할 필요는 없다. 만약 서로 다른 데이터 타입으로 된 데이터를 다루고 싶다면 다음 절에서 다룰 리스트를 사용하고, 벡터에서는 늘 한 가지 데이터 타입만 사용하는 편이 낫다. 벡터는 정첩할 수 없다. 따라서 벡터 안에 벡터를 생성하면 단일 차원의 벡터로 변경된다. 중첩된 구조가 필요하다면 역시 리스트를 사용해야 한다.

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

연속된 숫자를 저장하는 벡터는 자주 사용되기 때문에 1,2,3, ... 같은 값을 저장한 벡터를 손쉽게 생성하는 별도의 문접이 있다. 이에 대해 4장에서 배운다.

벡터의 각 셀에는 names<-() 함수를 사용해 이름을 부여할 수 있다. names()의 반환 값에 원하는 이름을 문자열 벡터로 할당하면 된다.

```
> x<-c(1,3,4)
> names(x)<-c("kim","seo","park")
> x
 kim  seo park
   1    3    4
```

2.4.2 벡터 데이터 접근

벡터의 데이터를 접근하는 데는 색인을 사용하는 방법과 이름을 사용하는 방법이 있다. 또, 벡터에서 특정요소를 제외한 나머지 데이터를 가져오거나, 동시에 여러 셀의 데이터를 접근하는 것 역시 가능하다. 다음 표에 관련된 기본 문법을 정리해뒀다.

벡터 데이터 접근 문법

문법	의미
x[n]	벡터 x의 n번째 요소, n은 숫자 또는 셀의 이름을 뜻하는 문자열
x[-n]	벡터 x에서 n번째 요소를 제외한 나머지, n은 숫자 또는 셀의 이름
x[idx_vector]	벡터 x로부터 idx_vector에 지정된 요소를 얻어옴. 이때 idx_vector는 색인을 표현하는 숫자 벡터 또는 셀의 이름을 표현하는 문자열 벡터
x[start:end]	벡터 x의 start부터 end까지의 값을 반환함. 변환 값은 start 위치의 값과 end 위치의 값을 모두 포함함

다음은 벡터의 길이 관련 함수이다.

length : 객체의 길이를 반환한다.

```
length(
    x #R 객체, 팩터, 배열, 리스트를 지정한다.
    )
#반환 값은 배열의 길이다.
```

nrow : 배열의 행 또는 열의 수를 반환한다,

```
NROW(
    x #R 벡터, 배열 또는 데이터 프레임
)
#반환 값은 행의 수다.
```

벡터는 []안에 색인을 적어 각 요소를 가져올 수 있다. 이때, 색인은 다른 언어와 달리 1부터 시작한다.

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

또한, '-색인' 형태로 음의 색인을 사용해 특정 요소만 제외할 수 있다.

```
> x[-1]
[1] "b" "c"
> x[-3]
[1] "a" "b"
```

여러 위치에 저장된 값을 한 번에 가져오려면 '벡터명[색인 벡터]' 형식을 사용한다.

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

'start : end' 형태의 문법은 start 부터 end 까지의 숫자를 저장한 숫자 벡터를 뜻한다. 따라서 x[start: end]를 사용해 start 부터 end까지의 데이터(start와 end에 위치한 요소 포함)를 볼 수 있다.

```
> x[1:2]
[1] "a" "b"
> x[1:3]
[1] "a" "b" "c"
```

벡터의 각 셀에 names()를 사용해 이름을 부여했다면, 이 이름을 사용해 데이터를 접근할 수 있다.

```
> x<-c(1,3,4)
> names(x)<-c("kim","seo","park")
> x
 kim  seo park
   1    3    4
> x["seo"]
seo
  3
```

벡터에 부여된 이름만 보려면 이름을 부여할 때와 마찬가지로 names()를 사용한다. 다음은 벡터의 두 번째 요소에 부여한 이름이 "seo"임을 보여준다.

```
> names(x)[2]
[1] "seo"
```

벡터의 길이는 length() 또는 NROW()를 통해 알 수 있다. (NROW()가 대문자임에 주의하기 바란다.) 본래 nrow()는 뒤에서 설명할 행렬과 데이터 프레임의 행 수를 알려주는 함수 이지만,

nrow()의 변형인 NROW()는 인자가 벡터인 경우 벡터를 n행 1열의 행렬로 취급해 길이를 반환한다.

따라서 데이터 타입에 무관하게 길이를 알고 싶은 경우 length(), nrow(), NROW()의 구분 없이 항상 NROW()만 사용하면 대부분 문제없이 동작한다.

다음 코드는 세 가지 함수의 사용 예를 보여준다.

```
> x<-c("a","b","c")
> length(x)
[1] 3
> NROW(x) #NROW()는 벡터와 행렬 모두 사용가능
[1] 3
> nrow(x) #nrow()는 행렬만 가능
NULL
```

2.4.4 벡터 연산

벡터는 값을 하나씩 접근해 해당 값을 사용한 계산을 수행하거나, 벡터 전체에 대해 연산을 한 번에 수행하거나, 벡터를 집합처럼 취급해 집합 연산(합집합, 교집합, 차집합)을 계산할 수 있다.

벡터 연산 함수

identical : 객체가 동일한지를 판단한다.

```
identical(
x, #R객체
y  #R객체
    )
#반환 값은 x와 y가 동일하면 TRUE, 그렇지 않으면 FALSE다.
```

다음은 벡터와 관현한 연산자들이다.

벡터 연산자

연산자	의미
value %in% x	벡터 x에 value가 저장되어 있는지 판단함
x + n	벡터 x의 모든 요소에 n을 더한 벡터를 구함. 마찬가지로 *,/,-,= 등의 연산자 적용 가능함

두 벡터가 값은 값을 담고 있는지는 identical() 함수로 알 수 있다.

```
> identical(c(1,2,3),c(1,2,3))
[1] TRUE
> identical(c(1,2,3),c(1,2,100))
[1] FALSE
```

%in% 연산자는 어떤 값이 벡터에 포함되어 있는지를 알려준다.

```
> "a"%in% c("a","b","c")
[1] TRUE
> "d"%in% c("a","b","c")
[1] FALSE
```

벡터 전체 값에 대한 연산을 한 번에 수행하려면 벡터를 마치 하나의 숫자처럼 생각하고 연산을 수행하면 된다. 이와 같이 데이터 전체에 대해 수행하는 연산은 R의 특징 중 하나로, 각 요소별로 연산을 수행하는 경우에 비해 실행 속도가 빠르기 때문에 자주 사용한다.

```
> x<-c(1,2,3,4,5)
> x+1
[1] 2 3 4 5 6
> 10-x
[1] 9 8 7 6 5
```

같은 방법으로 == 또는 != 연산자를 사용해 두 벡터에 저장된 값들을 한 번에 비교할 수 있다.

그러나 흔히 if문 등의 조건문에서는 단 하나의 참 또는 거짓 값을 사용해야 하기에 ==, !=가 아닌 앞서 설명한 identical()을 해야한다는 점을 기억하기 바란다. 흔히 실수하기 쉬운 부분이다.

```
> c(1,2,3) ==c(1,2,100)
[1]  TRUE  TRUE FALSE
> c(1,2,3) !=c(7,2,100)
[1]  TRUE FALSE  TRUE
```

벡터를 집합 set으로 취급해 집합 간 합집합, 교집합, 차집합을 계산할 수 있다.

```
> union(c("a","b","c"),c("d","b","c"))
[1] "a" "b" "c" "d"
> intersect(c("a","b","c"),c("d","b","c"))
[1] "b" "c"
> setdiff(c("a","b","c"),c("d","b","c"))
[1] "a"
```

집합 간 비교에는 setequal()을 사용한다.

```
> setequal(c("a","b","c"),c("d","b","c"))
[1] FALSE
> setequal(c("a","b","c"),c("a","b","c","c"))
[1] TRUE
```

2.4.5 연속된 숫자로 구성된 벡터

다량의 데이터가 있을 때 데이터의 일부를 한 번에 잘라서 처리하기 위해 흔히 숫자 색인 값이 저장된 벡터를 사용한다. 또, 연속된 데이터를 한 번에 잘라낼 경우가 많아 연속된 숫자가 저장된 벡터가 종종 필요하다. R에서는 이를 위한 몇 가지 문법과 함수를 제공하는데 다음 표에 그 내용을 정의했다.

표 2-9 연속된 숫자로 구성된 벡터 관련 함수

seq : 시퀀스를 생성한다.

```
seq(
    from, #시작 값
    to, #끝 값
    by #증가치
    )
#from부터 to까지의 값을 by간격으로 저장한 숫자 벡터를 반환한다.
```

seq_along : 주어진 객체의 길이만큼 시퀀스를 생성한다.

```
seq_along(
    along.with #이 인자 길이만큼 시퀀스를 생성한다.
    )
#반환 값은 along.with의 길이가 N일때, 1부터 N까지의 숫자를 저장한 벡터이다.
```

다음은 시퀀스 생성과 관련한 문법이다.

시퀀스 생성 문법

문법	의미
from:end	from부터 end까지의 숫자를 저장한 벡터를 반환함(from과 end도 반환함)

seq(from, to, by)는 from 부터 end까지의 값을 저장한 벡터를 반환한다. by는 생략가능하며 생략시 자동으로 1로 간주한다.

```
> seq(3,7)
[1] 3 4 5 6 7
> seq(7,3)
[1] 7 6 5 4 3
> seq(3,7,2)
[1] 3 5 7
> seq(3,7,3)
[1] 3 6
```

1씩 증가 또는 감소하는 벡터의 경우 seq()를 사용하지 않고 'start:end' 형태의 축약형으로도 표현 할 수 있다.

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

1부터 주어진 벡터의 길이 N까지의 값을 저장한 색인 벡터가 필요한 경우 NROW()로 길이를 얻어 색인 벡터를 만들 수 있다. 또는 seq_along(x)를 이용해 x의 길이까지의 값을 담은 벡터를 생성 할 수 있다.

```
> x<-c(2,4,6,8,10)
> 1:NROW(x)
[1] 1 2 3 4 5
> seq_along(x)
[1] 1 2 3 4 5
```

2.4.6 반복된 값을 저장한 벡터

반복된 값을 저장한 벡터는 c(1,1,1,1,2,2,2,2) 또는 c(1,2,1,2,1,2,1,2) 처럼 숫자가 반복되는 형태로 나나타는 벡터를 말한다. 이러한 벡터를 색인 벡터로 사용하면 주어진 데이터를 몇 개 분류로 쉽게 나눌 수 있어 종종 사용한다. 반복된 값이 저장된 벡터는 rep()로 생성할 수 있다.

rep : 주어진 값을 반복한다.

```
rep(
    x, #반복할 값이 저장된 벡터
    times, #전체 벡터의 반복 횟수
    each #개별 값의 반복 횟수
    )
#반환 값은 반복된 값이 저장된 x와 같은 타입의 객체다.
```

times와 each의 의미는 다음 예제를 보자.

```
> rep(1:2, times=5)
 [1] 1 2 1 2 1 2 1 2 1 2
> rep(1:2, each=5)
 [1] 1 1 1 1 1 2 2 2 2 2
> rep(1:2, each=5,times=2)
 [1] 1 1 1 1 1 2 2 2 2 2 1 1 1 1 1 2
[17] 2 2 2 2
> rep(1:2, times=2,each=5)
 [1] 1 1 1 1 1 2 2 2 2 2 1 1 1 1 1 2
[17] 2 2 2 2
```

2.5 리스트

자료 구조 책에서 리스트는 배열과 비교할 때 데이터를 중간중간에 삽입하는 데 유리한 구조로 설명한다. 물론 그러한 장점은 동일하지만 R에서 리스트는 데이터를 접근한다는 관점에서 다른 언어의 해시 테이블 또는 딕셔너리로 종종 설명된다. 즉, 리스트는'(키, 값)' 형태의 데이터를 담는 연관 배열이다.

또 다른 리스트의 특징은 벡터와 달리 값이 서로 다른 데이터 타입을 담을 수 있다는 점이다. 따라서 "이름"이라는 키에, "홍길동"이라는 문자열 값을 저장하고, "성적"이라는 키에 95라는 숫자 값을 저장할 수 있다.

2.5.1 리스트 생성

리스트는 list() 함수를 사용해 생성한다.

리스트 생성 함수

list : 리스트 객체를 생성한다.

```
list(
    key1 = value1,
    key2 = value2,
    ...
)
#반환 값은 key1에 value1, key2에 value2 등을 저장한 리스트다.
```

다음은 name에 "foo", height에 70을 저장하는 리스트를 보여준다.

```
> (x<-list(name="foo",height=70))
$name
[1] "foo"

$height
[1] 70
```

이때 각 값이 반드시 스칼라일 필요는 없다. 다음처럼 벡터를 저장할 수도 있다.

```
> (x<-list(name="foo",height=c(1,3,5)))
$name
[1] "foo"

$height
[1] 1 3 5
```

이처럼 리스트에는 다양한 값을 혼합해서 저장할 수 있다.

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


$b
$b$val
[1] 1 2 3 4
```

2.5.2 리스트 데이터 접근

리스트에 저장된 데이터는 색인 또는 키를 사용해 접근할 수 있다.

리스트 데이터 접근 문법

문법	의미
x$key	리스트 x에서 키 값 key에 해당하는 값
x[n]	리스트 x에서 n번째 데이터의 서브리스트
x[[n]]	리스트 x에서 n번째 저장된 값

앞에서 살펴본 것처럼 리스트를 출력해보면 '$키' 형태로 각 키가 나열된다. 데이터는 'x$key ' 형태로 접근한다. 또는 각 요소를 순서대로 x[[n]] 형태로 접근할 수도 있다.

```
> x<-list(name="foo",height=c(1,3,5))
> x$name
[1] "foo"
> x$height
[1] 1 3 5
> x[1]
$name
[1] "foo"

> x[[1]]
[1] "foo"
> x[[2]]
[1] 1 3 5
> x[2]
$height
[1] 1 3 5
```

x[[n]] 과 달리 x[n] 형태는 각 값이 아닌 '(키, 값)'을 담고 있는 서브리스트를 반환한다.

코드에서 볼 수 있듯, x[1]은 (name, "foo")를 담고 있는 리스트이ㅏㄷ.

2.6 행렬

R의 행렬은 벡터와 마찬가지로 행렬에는 한 가지 유형의 스칼라만 저장할 수 있다. 따라서 모든 요소가 숫자인 행렬은 가능하지만, '1열은 숫자, 2열은 문자열'과 같은 형태는 불가하다.

2.6.1 행렬 생성

행렬을 생성하는 함수는 다음과 같다.

행렬 생성 함수

matrix : 행렬을 생성한다.

```
matrix(
    data, #행렬을 생성할 데이터 벡터
    nrow, #행의 수
    ncol, #열의 수
    byrow = FALSE #TRUE로 설정하면 행우선, FALSE일 경우 열 우선으로 데이터를 채운다.
    dimnames = NULL #행렬의 각 차원에 부여할 이름
)
#반환 값은 행렬이다.
```

dimnames : 객체의 각 차원에 대한 이름을 가져온다.

```
dimnames(
    x #R 객체
)
#  반환 값은 객체 x의 각 차원에 대한 이름이다.
```

dimnames <- : 객체의 차원에 이름을 설정한다.

```
dimnames(
    x #R 객체
    )<- value #차원에 부여할 이름
```

rownames: 행렬의 행 이름을 가져온다.

```
rownames(
    x #2차원 이상의 행렬과 유사한 객체
    )
#반환 값은 행 이름이다.
```

행렬은 maxtrix()를 사용해 표현한다. 다음은 1,2,3,4,5,6,7,8,9로 구성된 3X3(3행 3열) 행렬을 만드는 방법을 보여준다.

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

위의 예에서 볼 수 있듯 행렬 생성 시에는 행렬 값을 나열한 뒤 ncol을 사용해 열의 수를 지정하거나 nrow를 사용해 행의 수를 지정한다. 이 예에서는 byrow 값이 생략되어 FALSE이므로 행렬 값이 좌측 열부터 채워졌다(열 우선). 행렬 값을 위쪽 행부터 채우고 싶다면 (행 우선 row major) byrow = TRUE를 지정한다.

```
> 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
```

행렬의 행과 열에 명칭을 부여하고 싶다면 dimnames에 리스트로 행, 열의 이름을 지정한다. 리스트 내 첫 번째 벡터는 행의 이름, 두 번째 벡터는 열의 이름이어야 한다.

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

이미 만들어진 행렬에는 dimnames()를 사용해 이름을 부여할 수 있다.

```
> (x<-matrix(1:9,ncol=3))
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> dimnames(x)<-list(c("r1","r2","r3"),c("c1","c2","c3"))
> x
   c1 c2 c3
r1  1  4  7
r2  2  5  8
r3  3  6  9
```

dimnames()가 행 이름과 열 이름을 한 번에 지정하는 것과 달리 rownames(), colnames()는 각각 행 이름, 열 이름 지정을 위해 특화된 함수다. 다음은 행 이름과 열 이름을 차례로 rownames()와 colnames()를 사용해 부여하는 에다.

```
> (x<-matrix(1:9,ncol=3))
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> rownames(x)
NULL

> rownames(x)<-c("r1","r2","r3")
> x
   [,1] [,2] [,3]
r1    1    4    7
r2    2    5    8
r3    3    6    9
> colnames(x)<-c("c1","c2","c3")
> x
   c1 c2 c3
```

```
r1  1  4  7
r2  2  5  8
r3  3  6  9
```

2.6.2 행렬 데이터 접근

행렬은 색인 또는 행과 열의 이름을 통해 접근할 수 있다.

행렬 데이터 접근 문법

문법	의미
A[ridx, cidx]	행렬A의 ridx행, cidx열에 저장된 값, 이때 ridx값, cidx값에 벡터를 사용해 여러 값을 지정 가능함. ridx나 cidx 중 하나를 생략하면 전체 행 또는 열을 의미함

행렬의 각 요소는 x[ridx, cidx] 형태로 접근한다. 이때 색인은 벡터의 경우와 마찬가지로 1부터 시작한다.

```
> (x<-matrix(c(1,2,3,4,5,6,7,8,9),ncol = 3))
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> x[1,1]
[1] 1
> x[2,1]
[1] 2
```

역시 벡터와 마찬가지로 '-색인' 형태로 음수를 사용해 특정 행이나 열을 제외하거나, 색인에 벡터를 지정해 여러 값을 한 번에 가져올 수 있다. 만약 특정 행이나 열의 전체를 가져오고 싶다면 행이나 열에 아무런 색인도 기재하지 않으면 된다.

다음 코드는 1,2행의 데이터를 가져온다.

```
> x[1:2,]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
#같은 결과를 가져오기 위해 3행을 제외시켜도 된다.
> x[-3,]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
```

색인에는 벡터를 지정할 수 있다. 1행, 3행, 1열, 3열의 값만 추출하는 경우다.

```
> x[c(1,3),c(1,3)]
     [,1] [,2]
[1,]    1    7
[2,]    3    9
```

만약 dimnames를 통해 행과 열에 이름을 부여했다면 그 이름을 직접 사용할 수도 있다. 다음은 r1이라는 이름을 부여한 첫 번째 행의 모든 열을 가져오는 예이다.

```
> (x<-matrix(1:9,nrow=3,dimnames=list(c("r1","r2","r3"),c("c1","c2","c3"))))
   c1 c2 c3
r1  1  4  7
r2  2  5  8
r3  3  6  9
> x["r1",]
c1 c2 c3
 1  4  7
```

2.6.3 행렬 연산

이번 절에서는 행렬과 스칼라 간 그리고 행렬 간의 사칙 연산에 대해 알아본다. 다음은 행렬 연산과 관련한 문법을 보여준다.

행렬 연산자

연산자	의미
A+x	행렬 A의 모든 값에 스칼라 x를 더한다. 이외에도 -, *, / 연산자를 사용할 수 있다.
A + B	행렬 A와 행렬 B의 합을 구한다. 행렬 간의 차는 -연산자를 사용한다.
A %*% B	행렬 A와 행렬 B의 곱을 구한다.

```
> x<-matrix(c(1,2,3,4,5,6,7,8,9),nrow=3)
> x * 2
     [,1] [,2] [,3]
[1,]    2    8   14
[2,]    4   10   16
[3,]    6   12   18
> x/2
     [,1] [,2] [,3]
[1,]  0.5  2.0  3.5
[2,]  1.0  2.5  4.0
[3,]  1.5  3.0  4.5
```

행렬 간의 곱에는 %*%를 사용한다.

```
 x %*% x
     [,1] [,2] [,3]
[1,]   30   66  102
[2,]   36   81  126
[3,]   42   96  150
```

전치행렬은 r()로 구한다.

```
> x<-matrix(c(1,2,3,4,5,6,7,8,9),nrow=3)
> t(x)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

역행렬은 solve()로 계산한다. 다음은 행렬 x의 역행렬을 구한 뒤 x와 곱해 그 결과가 단위행렬이 되는지 확인하는 예다.

```
> (x<-matrix((1:4),ncol=2))
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> solve(x)
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
> x %*% solve(x)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

행렬의 차원은 nrow(), ncol()로 알 수 있으며, 각각 행의 수와 열의 수를 반환한다.

```
> (x<-matrix((1:6),ncol=3))
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> nrow(x)
[1] 2
> ncol(x)
[1] 3
```

dim()은 행렬뿐만 아니라 다양한 데이터 타입(예를 들면, 다음 절에서 설명하는 배열)에 적용가능한 일반(범용) 함수로, 데이터의 차원을 벡터로 반환한다. dim()의 반환 값에 새로운 차원을 지정하면 데이터의 차원을 변경할 수 있다는 점이 특이하다. 다음은 2X3 차원의 행렬을 생성한 뒤 3X2차원으로 바꾼 예이다.

```
> x <- matrix(c(1:6),ncol=3)
> dim(x)
[1] 2 3
> c
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> dim(x)<-c(3,2)
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

2.7 배열

행렬이 2차원 데이터라면 배열은 다차원 데이터이다. 예를 들어 2x3 차원의 데이터를 행렬로 표현한다면 2x3x4 차원의 데이터는 배열로 표현한다.

2.7.1 배열 생성

다음은 배열 생성 함수다.

배열 생성 함수

array : 배열을 생성한다.

```
array(
    data = NA, #데이터를 저장한 벡터
    dim = length(data), #배열의 차원, 이 값을 지정하지 않으면 1차원 배열이 생성된다.
    dimnames = NULL #차원의 이름
    )
#반환 값은 배열이다.
```

예를 들어, 3x4 차원의 배열은 dim(3,4)를 지정하여 생성한다.

```
> array(1:12, dim=c(3,4))
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

이번에는 같은 데이터로 2x2x3 차원의 배열을 만들어보자.

```
> (x<-array(1:12,dim=c(2,2,3)))
, , 1

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

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8

, , 3

     [,1] [,2]
[1,]    9   11
[2,]   10   12
```

위 결과에서 ', , 1', ', , 2', ', , 3' 은 2x2x3 차원에서 가장 마지막에 지정한 차원(즉, 2x2x3에서 x3)을 의미한다.

```
> x[,,3]
     [,1] [,2]
[1,]    9   11
[2,]   10   12
```

2.7.2 배열 데이터 접근

배열의 데이터를 접근하는 방법은 행렬의 경우와 차원의 수만 다를 뿐이지 차이가 없다. 따라서 색인, 이름 등으로 데이털르 접근할 수 있고 dim()을 사용해 데이터의 차원을 알 수 있다.

```
> (x<-array(1:12,dim=c(2,2,3)))
, , 1
```

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

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8

, , 3

     [,1] [,2]
[1,]    9   11
[2,]   10   12

> x[1,1,1]
[1] 1
> x[1,2,3]
[1] 11
> x[,,3]
     [,1] [,2]
[1,]    9   11
[2,]   10   12
> dim(x)
[1] 2 2 3
```

2.8 데이터 프레임

데이터 프레임은 처리할 데이터를 마치 엑셀 스프레드시트와 같이 표 형태로, 정리한 모습을 하고 있다. 데이터 프레임의 각 열에는 관측값의 이름이 저장되고, 각 행에는 매 관측 단위마다 실제 얻어진 값이 저장된다. 예를 들어, 다음 성적 데이터와 같은 모습이 데이터 프레임에 저장되는 데이터의 전형적인 예다.

2.8.1 데이터 프레임 생성

아래에 데이터 프레임 관련 함수를 나열했다.

데이터 프레임 생성 및 관련함수

data.frame : 데이터 프레임을 생성한다.

```
data.frame(
    # value 또는 tag = value로 표현된 데이터 값, '...'은 가변 인자를 의미한다.
    ...,
    #주어진 문자열을 팩터로 저장할 것인지 또는 문자열로 저장할 것인지를 지정하는 인자.
    #기본값은 보통 TRUE이다. 따라서 이 인자를 지정하지 않으면 문자열은 팩터로 저장된다.
    stringAsFactors = default.stringAsFactors()
    )
#반환값은 데이터 프레임이다.
```

str: 임의의 R 객체의 내부 구조(structure)를 보인다.

```
str(
    object #구조를 살펴볼 R객체
)
```

또한, 데이터 프레임에는 앞에서 살펴본 rownames(), colnames(), names ()를 적용할 수 있다.

이들 함수를 사용하면 행과 컬럼(열)에 이름을 부여할 수 있다.

다음은 데이터 프레임에서 사용하는 문법이다.

데이터 프레임 사용 문법

문법	의미
d$colname	데이터 프레임 d에서 컬럼 이름이 colname인 데이터를 접근한다.
d$colname <- y	데이터 프레임 d에서 컬럼이름이 colname인 컬럼에 데이터 y를 저장한다. 만약 colname이 d에 없는 새로운 이름이라면 새로운 컬럼이 추가된다.

데이터 프레임은 data.frame()에 '컬럼이름=데이터' 형태로 데이터를 나열하여 생성한다.

```
> (d<-data.frame(x=c(1,2,3,4,5),y=c(6,7,8,9,10)))
  x  y
1 1  6
2 2  7
3 3  8
4 4  9
5 5 10
```

데이터 프레임의 각 컬럼은 서로 다른 데이터 타입일 수 있다. 다음은 숫자 벡터 x,y에 팩터 z 컬럼을 추가하는 예이다. (stringsAsFactor를 지정하지 않으면 문자열이 팩터로 저장된다.)

```
> (d<-data.frame(x=c(1,2,3,4,5),
                 y=c(6,7,8,9,10),
                 z=c('M','F','M','F','M')))
  x  y z
1 1  6 M
2 2  7 F
3 3  8 M
4 4  9 F
5 5 10 M
```

위에 보인 출력에서는 각 컬럼의 데이터 타입을 보여주지 않아 실제로 어떤 데이터 타입으로 데이터가 저장되는지는 쉽게 알 수 없다. 이 경우 str()을 사용해 구조를 살펴봐야 한다.

```
> str(d)
'data.frame':   5 obs. of  3 variables:
 $ x: num  1 2 3 4 5
 $ y: num  6 7 8 9 10
 $ z: Factor w/ 2 levels "F","M": 2 1 2 1 2
```

이 결과를 보면, z 컬럼이 팩터임을 알 수 있다. 만약 이미 정의된 데이터 프레임에서 컬럼 이름 colname에 데이터 y를 저장하고자 한다면 d$colname <- y문법을 사용한다.

다음은 x컬럼의 값을 1,2,3,4,5 에서 6,7,8,9,10으로 바꾸는 예다.

```
> (d<-data.frame(x=c(1,2,3,4,5),y=c(2,4,6,8,10),z=c('M','F','M','F','M')))
  x  y z
1 1  2 M
2 2  4 F
```

```
3 3  6 M
4 4  8 F
5 5 10 M

> d$x
[1] 1 2 3 4 5

> d$x <- 6:10
> d
   x  y z
1  6  2 M
2  7  4 F
3  8  6 M
4  9  8 F
5 10 10 M
```

위의 데이터 프레임 d에 기존에 없던 컬럼 w를 추가하고자 할 때도 같은 문법을 사용한다.

```
> d$w <- c("A","B","C","D","E")
> d
   x  y z w
1  6  2 M A
2  7  4 F B
3  8  6 M C
4  9  8 F D
5 10 10 M E
```

한 가지 특이할 만한 점은 w컬럼에 저장한 데이터가 벡터이므로 d$w의 데이터 타입 역시 chr(문자열 벡터)이라는 점이다. 이는 data.frame()에 문자열 벡터를 지정할 때 stringAsFactor를 지정하지 않으면 문자열이 팩터로 바뀌는 것과는 다른 점이다.

```
> str(d)
'data.frame':   5 obs. of  4 variables:
 $ x: int  6 7 8 9 10
 $ y: num  2 4 6 8 10
 $ z: Factor w/ 2 levels "F","M": 2 1 2 1 2
 $ w: chr  "A" "B" "C" "D" ...
```

데이터 프레임의 행 이름, 열 이름은 각각 rownames(), colnames() 함수로 지정할 수 있다.

```
> (x<-data.frame(1:3))
  X1.3
1    1
2    2
3    3
> colnames(x)<-c('val')
> x
  val
1   1
2   2
3   3
> rownames(x)<-c('a','b','c')
> x
  val
a   1
```

```
b   2
c   3
```

또는 names()를 사용해도 colnames()와 같은 결과를 얻는다.

```
> names(x)<-c('val_n')
> x
  val_n
a     1
b     2
c     3
```

2.8.2 데이터 프레임 접근

문법	의미
d$colname	데이터 프레임 d의 컬럼 이름 colname에 저장된 데이터
d[m,n,drop=TRUE]	데이터 프레임 d의 m행 n컬럼에 저장된 데이터. m과 n을 벡터로 지정하여 다수의 행과 컬럼을 동시에 가져올 수 있으며 m,n에는 색인 뿐만 아니라 행 이름이나 컬럼 이름을 지정할 수 있다. 만약 m,n 중 하나를 생략하면 모든 행 또는 컬럼의 데이터를 의미한다. d[,n]과 같이 행를 지정하지 않고 특정 컬럼만 가져올 경우 반환 값은 데이터 프레임이 아니라 해당 컬럼의 데이터 타입이 된다. 이러한 형 변환을 원하지 않으면 drop = FALSE를 지정하여 데이터 프레임을 반환하도록 할 수 있다.

데이터 프레임의 각 칼럼은 d$colname과 같이 컬럼이름으로 접근할 수 있으며, 행이나 컬럼의 색인을 사용해서도 데이터에 접근할 수 있다.

```
> d<-data.frame(x=c(1,2,3,4,5),y=c(6,7,8,9,10))
> d
  x  y
1 1  6
2 2  7
3 3  8
4 4  9
5 5 10
> d$x
[1] 1 2 3 4 5
> d[1,]
  x y
1 1 6
> d[1,2]
[1] 6
```

벡터로 색인을 지정하거나 제외할 행 또는 컬럼을 -로 표시할 수 있다.

```
> d[c(1,3),2]
[1] 6 8
> d[-1,-2]
[1] 2 3 4 5
```

또는 컬럼 이름을 지정할 수도 있다.

```
> d[,c("x","y")]
  x  y
1 1  6
2 2  7
3 3  8
4 4  9
5 5 10

> d[,c('x','y')]
  x  y
1 1  6
2 2  7
3 3  8
4 4  9
5 5 10

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

위 코드에서 d[,c("x")]로 x컬럼만 선택했을 때, 데이터 프레임의 일반적인 표 형태 출력이 아니라 벡터처럼 결과가 출력된 것을 볼 수 있다. 이는 컬럼의 차원이 1이 되면 반환 값이 해당 컬럼의 데이터 타입을 따르기 때문이다. 이러한 형 변환을 원치 않는다면 다음과 같이 drop=FALSE 옵션을 지정한다.

```
> d[,c("x"),drop = FALSE]
  x
1 1
2 2
3 3
4 4
5 5
```

주어진 값이 벡터에 존재하는지를 판별하는 %in% 연산자와 데이터 프레임의 컬럼 이름을 반환하는 names()을 이용하면 이용하면 특정 컬럼만 선택하는 작업을 좀 더 손쉽게 할 수 있다. 예를 들어, 다음 코드는 a,b,c 컬럼이 있는 데이터 프레임에서 b,c 컬럼만 선택하는 경우다.

```
> (d<-data.frame(a=1:3,b=4:6,c=7:9))
  a b c
1 1 4 7
2 2 5 8
3 3 6 9
> d[,names(d) %in% c("b","c")]
  b c
1 4 7
2 5 8
3 6 9
```

반대로 !연산자를 사용해 특정 컬럼들만 제외해서 데이터를 선택할 수도 있다.

```
> d[,!names(d) %in% c("b","c")]
[1] 1 2 3
> d[,!names(d) %in% c("b","c"),drop = FALSE]
  a
1 1
2 2
3 3
> d[,!names(d) %in% c("a"),drop = FALSE]
  b c
1 4 7
2 5 8
3 6 9
```

2.8.3 유틸리티 함수

데이터 프레임은 분석할 데이터가 들어 있는 주요 데이터 타입이다. 분석할 데이터는 보통 파일등에서 불러들이므로, 불러들인 데이터가 올바른 데이터 타입으로 저장되어 있는지 확인하는 것이 중요하다. 또, 데이터 프레임에는 분석할 데이터 전체가 저장되므로 데이터양이 많다. 따라서 이러한 데이터를 손쉽게 살펴보는 방법이 필요하다. 이 절에서는 데이터 프레임과 관련하여 참고할 만한 함수들을 설명한다.

head : 객체의 처음 부분을 반환한다.

```
head(
    x, #객체
    n = 6L # 반환할 결과 값의 크기
    )
```

반환 값은 x의 앞부분을 n 만큼 잘라낸 데이터이다.

View : 데이터 뷰어를 호출한다.

```
View(
    x, #객체
    title #뷰어 윈도우의 제목
    )
```

이를 실습해보자.

```
> d<-data.frame(x=1:1000)
> head(d)
  x
1 1
2 2
3 3
4 4
5 5
6 6
> tail(d)
        x
995   995
996   996
997   997
998   998
999   999
1000 1000
```

2.9 타입판별

데이터를 처리하기 위해 여러 함수를 호출하다 보면 결과의 타입이 무엇인지 분명하지 않을 때가 많다. 이 경우 다음 함수들을 사용하여 데이터 타입을 손쉽게 판단할 수 있다.

함수	의미
class(x)	객체 x의 클래스
str(x)	객체 x의 내부 구조
is.factor(x)	주어진 객체 x가 팩터인가?
is.numeric(x)	주어진 객체 x가 숫자를 저장한 벡터인가?
is.character(x)	주어진 객체 x가 문자열을 저장한 벡터인가?
is .matrix(x)	주어진 객체 x가 행렬인가?
is.array(x)	주어진 객체 x가 배열인가?
is.data.frame(x)	주어진 객체 x가 데이터 프레임인가?

다음은 벡터, 행렬, 데이터 프레임에 class()를 적용하여 클래스를 구하는 예시이다.

```
> class(c(1,3))
[1] "numeric"
> class(matrix(1,3))
[1] "matrix"
> class(data.frame(x=c(1,3),y=c(2,6)))
[1] "data.frame"
```

class()는 문자열로 데이터 타입을 반환하는데, 이 예에서는 숫자형 벡터에 numeric, 행렬에 matrix, 데이터 프레임에 data.frame을 반환했다. 특히 class()에 벡터를 인자로 전달할 경우 numeric이 반환되었는데 이는 벡터에 저장된 값이 숫자기 때문이다. 벡터에 저장된 데이터 타입에 따라 이 값은 logical, character, factor 등이 될 수 있다.

데이터 타입은 str()로도 확인해 볼 수 있다. 다음 예에서 벡터와 행렬의 결과가 유사해 보이지만 벡터의 경우 차원이 [1:2, 1] (2차원이고 2행 1열)로 표시되어 있는 점이 다르다.

```
> str(c(1,2))
 num [1:2] 1 2
> str(matrix(c(1,2)))
 num [1:2, 1] 1 2
> str(list(c(1,2)))
List of 1
 $ : num [1:2] 1 2
> str(data.frame(x=c(1,2)))
'data.frame':   2 obs. of  1 variable:
 $ x: num  1 2
```

R의 데이터 타입에는 타입이름이 'typename' 이라 할 때 'is.typename()' 형태의 함수가 존재한다.

이 함수들은 주어진 객체 x가 'typename' 데이터 타입에 해당하는지를 판별하는 용도로 사용한다.

이러한 함수의 예로 팩터인지 여부를 알려주는 is.factor(), 숫자를 저장한 벡터인지를 알려주는 is.numeric(), 문자열을 저장한 벡터인지를 알려주는 is.character() 등을 들 수 있다. 다음에 몇 가지 함수의 사용 예를 보였다.

```
> is.factor(factor(c("m","f")))
[1] TRUE
> is.numeric(1:5)
[1] TRUE
> is.character(c("a","b"))
[1] TRUE
> is.data.frame(data.frame(x=1:5))
[1] TRUE
```

2.10 타입 변환

R의 형 변환은 암시적으로 발생할 수 있고, 때에 따라 형 변환이 전혀 예상치 않은 곳에서 일어난다. 예를 들어, 데이터 프레임 d에 2개 컬럼이 있고 각 컬럼의 데이터 타입이 numeric(숫자를 저장한 벡터)이라고 가정하자. 이때 d[,1]은 첫 번째 컬럼의 데이터를 데이터 프레임이 아닌 numeric(즉, 벡터)으로 반환한다. d에 대한 연산의 결과가 데이터 프레임이 아니라 numeric이라는 점은 사용자를 당황하게 할 수 있는 점이며, 이러한 변환을 피하고 싶다면 drop = FALSE를 지정해야 한다.

또는 파일에서 문자열을 불러들였을 때 data.frame()으로 데이터 프레임을 생성하면서 stringAsFactor = FALSE 를 지정하지 않으면 문자열이 character(문자열 벡터)가 아닌 팩터가 된다.

이와 같은 암시적 형 변환에 대비하기 위해 모든 의심스러운 함수 호출 뒤에는 '2.9 타입 판별'절에서 설명한 str(), class()를 사용해 현재 사용하는 데이터가 올바른 데이터 타입인지 계속 확인할 필요가 있다. 반면 타입을 강제로 변환하고자 할 때도 있을 것이다. 문자열 벡터를 팩터로 변환하는 경우 등이 그 예다. 이러한 변환을 하는 한 가지 방법은 타입 이름이 'typename'이라 할 때 'as.typename()'이라는 함수를 사용하는 것이다. 이를 실습해보자.

또 다른 타입 변환 방법은 factor(), data.frame() 등과 같이 데이터를 생성하는 함수에 다른 타입의 데이터를 인자로 넘겨주는 것이다. 이 경우 필요에 따라 형 변환이 수행된다.

```
> x <- c("a","b","c")
> as.factor(x)
[1] a b c
Levels: a b c
> as.character(as.factor(x))
[1] "a" "b" "c"
```

다음은 행렬을 데이터 프레임으로 변환하는 예다.

```
> x<-x<-matrix(1:9, ncol=3)
> as.data.frame(x)
  V1 V2 V3
1  1  4  7
2  2  5  8
3  3  6  9
```

또는 factor(), matrix(), data.frame() 등과 같은 데이터 생성 함수에 곧바로 다른 타입을 넘겨 형변환을 할 수도 있다. 다음은 행렬을 데이터 프레임으로 변환하기 위해 matrix()의 결과를 data.frame()에 곧바로 넘긴 예다.

```
> (x<-data.frame(matrix(c(1,2,3,4),ncol=2)))
  X1 X2
1  1  3
2  2  4
```

다음은 리스트를 data.frame()에 넘겨 데이터 프레임으로 변환한 예다.

```
> data.frame(list(x=c(1,2),y=c(3,4)))
  x y
1 1 3
2 2 4
```

두 가지 방법이 모두 가능하다면 그 차이는 무엇일까? 'as.typename()'은 표현이 명확하고 간략 하지만 경우에 따라 지원하는 변환의 정도가 약하다. 예를 들어, c("m","f")의 벡터를 as.factor()로 변환하는 경우를 생각해보자. 이 경우 f가 알파벳 순서상 m 보다 앞서므로 as.factor(c("m","f"))의 결과에서 팩터의 레벨은 "f m"으로 정해진다. 팩터의 레벨을 "m f"로 하려면 어떻게 해야 할까? as.factor() 는 변환할 데이터 이상의 인자를 받지 않으므로 as.factor() 를 사용할 때 팩터의 레벨을 "m f"로 지정할 수 있는 방법이 없다. 팩터 레벨의 순서를 "m f "로 하고 싶다면 다음과 같이 factor() 함수를 써야한다.

```
> as.factor(c("m","f"))
[1] m f
Levels: f m
> factor(c("m","f"), levels = c("m","f"))
[1] m f
Levels: m f
```