ChatGPT

R의 데이터 타입과 자료 구조

R 언어에서 데이터를 다룰 때는 데이터가 **어떤 유형(type)**이며 **어떤 구조(structure)**로 저장되어 있는지 이해하는 것이 중요합니다. 데이터 유형과 구조에 대한 이해가 부족하면 코드가 예상과 다르게 동작하거나 초보자가 혼란을 느끼기 쉽습니다. 여기서는 R의 기본 데이터 **유형**과 **자료 구조**를 초보자도 이해하기 쉽게 설명하고, 각 예시와 함께 `typeof()`, `class()`, `str()` 함수를 사용하여 객체의 타입을 확인하는 방법도 다루겠습니다.

목차:

각 섹션에서는 개념 설명과 함께 간단한 R 코드 예제를 제공합니다. 예제는 **이름 목록**, **온도 값 목록**, **참/거짓 플래그** 등 일상적인 시나리오를 사용하여 이해를 돕겠습니다. 또한 곳곳에 가벼운 해설과 초보자가 흔히 저지르는 실수를 피하는 팁도 포함하였습니다.

1. 기본 데이터 유형 (원자형 데이터 타입)

원자형(atomic) 데이터 유형은 더 이상 쪼갤 수 없는 기본적인 데이터 타입을 말합니다. R에는 대표적으로 다섯 가지의 기본 원자형 데이터 타입이 있습니다: **숫자형(numeric)**, **문자형(character)**, **논리형(logical)**, **복소수형(complex)**, **Raw 타입(raw)** [1]. (R 문서에 따라 integer도 별도로 언급되지만, 여기서는 numeric에 정수형을 포함하여 설명합니다.) 이 섹션에서는 각 데이터 유형의 의미와 사용 예시를 살펴봅니다.

Numeric (숫자형)과 Integer (정수형)

숫자형(numeric)은 수치 데이터를 나타내는 타입으로, R에서는 기본적으로 **실수(double)** 형태로 저장됩니다 [2]. 예를 들어 3이나 3.14 같은 값을 저장하면 R 내부에서는 이를 double 타입(배정밀도 실수)으로 취급합니다. 한편 **정수**

(integer)는 숫자형의 한 종류로, 소수점이 없는 정수 값을 저장하는 타입입니다. R에서 정수로 값을 저장하려면 숫자 뒤에 대문자 `L`을 붙입니다.

- **예시:**

```
x_num <- 5   # 숫자 5를 numeric 타입(double)으로 저장
x_int <- 5L  # 숫자 5를 integer 타입(정수)으로 저장
typeof(x_num)  # "double" (실수로 저장됨)
typeof(x_int)  # "integer" (정수로 저장됨)
```

위 코드에서 `x_num`은 숫자 5를 저장했지만 `typeof(x_num)`를 해보면 **"double"**이 출력됩니다. 반면 `x_int`는 `5L`로 저장하여 정수형이 되었고, `typeof(x_int)` 결과는 **"integer"**입니다[3]. R에서는 숫자를 기본적으로 실수로 취급하므로, 정수 연산이 필요한 경우 `L`을 붙이거나 `as.integer()` 함수를 사용해 변환해야 합니다.

숫자형 데이터는 보통 **나이, 온도, 거리, 통계치** 등 연속적인 양적 데이터를 표현하는 데 사용됩니다. 예를 들어 일주일간의 기온을 numeric 벡터로 저장할 수 있습니다:

```
temps <- c(22.5, 23.0, 21.5, 20.0, 19.5, 18.0, 17.5)  # 일주일 기온 (numeric 벡터)
mean(temps)  # 평균 계산
```

만약 정수로 된 데이터 (예: 학생 5명의 시험 점수)를 다룬다면 정수형으로 저장할 수도 있지만, 대부분의 연산에서 실수형과 정수형은 크게 구분하지 않고 사용할 수 있습니다. **정수형과 실수형의 차이를 크게 신경 쓰지 않아도 되지만**, 큰 데이터에서 메모리를 최적화하거나 소수점이 없어야 하는 경우에만 정수형을 명시적으로 사용하는 편입니다.

참고: **특수 숫자 값**으로 `Inf`(무한대)와 `NaN`(숫자가 아님)이 있습니다. 예를 들어 `1/0`은 `Inf`로, `0/0`은 `NaN`으로 표현됩니다. 이런 값들이 나오면 연산 결과에 문제가 없는지 확인하는 것이 좋습니다.

Character (문자형)

문자형(character)은 글자나 문자열 데이터를 표현하는 타입입니다. 하나의 문자 `"A"`부터 `"Hello, world!"` 같은 문자열까지 모두 character로 취급됩니다. 문자형 데이터는 주로 **이름, 주소, 범주형 값("Low"/"Medium"/"High" 등)**을 다루는 데 사용됩니다. R에서 문자형 리터럴은 **따옴표**로 감싸서 표현합니다.

- **예시:**

```
name <- "홍길동"        # 하나의 문자열 (문자형 스칼라 값)
fruits <- c("Apple", "Banana", "Cherry")  # 문자열 벡터
typeof(name)    # "character"
length(fruits)  # 3 (fruits 벡터에는 3개의 요소가 있음)
```

위 예에서 `name` 변수는 `"홍길동"`이라는 문자열을 담고 있으며, `typeof(name)`은 **"character"**를 반환합니다[4]. `fruits` 변수는 문자형 원소 3개로 이루어진 벡터입니다.

문자열은 따옴표로 감싸야 하며, `"` (쌍따옴표)와 `'` (홑따옴표) 중 어느 것으로 감싸도 됩니다. 문자열 벡터는 여러 텍스트 값을 한데 모아 관리할 때 사용합니다. 예를 들어 `fruits` 벡터는 과일 이름 목록을 저장한 것입니다.

초보자가 흔히 하는 실수 중 하나는 **숫자를 문자로 착각하거나 그 반대**를 하는 경우입니다. `"42"`는 문자열 `"42"`일 뿐 숫자 42와는 다릅니다. 만약 `c(42, "42")`처럼 숫자와 문자열을 같은 벡터에 넣으면, R은 한 벡터 내 원소 타입을 통일하기 위해 숫자 42를 문자열 `"42"`로 자동 변환(coerce)합니다 [5]. 즉, 서로 다른 타입을 같은 벡터에 섞으면 모두 **문자형으로 변환**되어 버리니 주의해야 합니다 (자세한 내용은 뒤의 벡터 섹션에서 다룹니다).

Logical (논리형)

논리형(logical)은 참(`TRUE`)과 거짓(`FALSE`) 값을 표현하는 데이터 타입입니다. 논리형 값은 주로 **조건식의 결과**나 **플래그(flag)**로 활용됩니다. 예를 들어 어떤 값이 기준을 초과하는지 여부(TRUE/FALSE), 사용자가 로그인했는지 여부(TRUE/FALSE) 등을 나타낼 때 사용합니다.

- **예시:**

```
is_raining <- FALSE     # 비가 오는지 여부 (논리형 변수)
flags <- c(TRUE, TRUE, FALSE, TRUE)  # 논리형 벡터
typeof(is_raining)  # "logical"
```

위에서 `is_raining`은 비가 오는지에 대한 상태를 논리값으로 저장했습니다. `typeof(is_raining)` 결과는 **"logical"**입니다. `flags` 벡터는 여러 개의 논리값을 담고 있습니다.

논리형 값은 일반적으로 비교 연산이나 조건문에서 생성됩니다. 예를 들어 `5 > 3`은 TRUE를, `10 == 12`는 FALSE를 반환합니다. 또한 `&` (그리고), `|` (또는) 같은 논리 연산자를 사용해 여러 논리값을 결합할 수도 있습니다.

> **Note:** R에서 TRUE와 FALSE는 대문자로 작성해야 합니다. `T`와 `F`라는 축약형도 존재하지만, 이들은 단순한 변수로 간주될 수 있으므로 사용을 권장하지 않습니다 [6]. 잘못하면 `T <- FALSE`와 같은 식으로 `T` 값이 바뀌어 버리는 혼란이 생길 수 있으니 항상 `TRUE`/`FALSE` 전체를 쓰는 습관을 들이세요.

Complex (복소수형)

복소수형(complex)은 수학에서의 복소수를 표현하는 데이터 타입입니다. 복소수란 실수부와 허수부를 갖는 수로, R에서는 `a + bi` 형태로 작성합니다 (여기서 `i`는 $\sqrt{-1}$, 즉 허수 단위입니다). 복소수형 데이터는 통계 일반 사용자보다는 **공학적/수학적 계산**이나 **푸리에 변환** 같은 전문 분야에서 가끔 사용됩니다. 일반적인 데이터 분석에서는 등장 빈도가 낮습니다 [7].

- **예시:**

```
z <- 2 + 3i    # 복소수 2+3i 생성
typeof(z)      # "complex"
Re(z); Im(z)   # 실수부와 허수부 추출 (각각 2와 3)
```

위 코드에서 `z`는 복소수 2+3i를 저장한 변수이며, `typeof(z)` 결과 **"complex"**로 표시됩니다. `Re(z)`는 실수부(Real part)를, `Im(z)`는 허수부(Imaginary part)를 반환하는 함수입니다. 복소수는 벡터 연산이나 행렬 연산에서도 지원되며, 예를 들어 복소수 벡터를 만들어 계산할 수도 있습니다.

복소수형은 초보자에게는 다소 생소할 수 있지만 알아만 두면 됩니다. 필요할 때가 아니면 잘 쓰지 않으므로, R에서 이런 타입도 지원한다 정도로 이해하고 넘어가셔도 좋습니다 [7].

Raw (Raw 타입)

Raw 타입(raw)은 데이터를 **바이트(byte) 단위의 이진 형태**로 저장하기 위한 특수한 타입입니다. 보통 문자를 ASCII/UTF-8 등의 **바이너리 인코딩**으로 다루거나 바이너리 파일을 읽고 쓸 때 사용됩니다. 일반적인 데이터 분석에서는 거의 사용하지 않으므로, 초보자라면 "이런 것도 있구나" 정도로만 인지하면 됩니다 [7].

Raw 벡터를 만드는 한 가지 방법은 문자열을 raw 바이트로 변환하는 `charToRaw()` 함수를 쓰는 것입니다. 예를 들어:

- **예시:**

```
raw_vec <- charToRaw("ABC")  # 문자열 "ABC"를 raw 바이트로 변환
raw_vec      # 41 42 43 (16진수로 0x41 = 'A', 0x42 = 'B', 0x43 = 'C')
typeof(raw_vec) # "raw"
```

위 코드에서 `"ABC"`라는 문자열을 `charToRaw()`로 변환하면 `41 42 43`과 같이 출력되는데, 이는 'A', 'B', 'C'의 ASCII 코드값을 16진수로 나타낸 것입니다. `typeof(raw_vec)` 결과 **"raw"**인 것을 확인할 수 있습니다. `class(raw_vec)` 또한 `"raw"`로 표시됩니다. Raw 타입은 내용이 사람에게 읽히는 형태가 아니며, 주로 바이너리 데이터를 직접 다룰 때 쓰입니다.

요약하면, **숫자형, 문자형, 논리형, 복소수형, Raw 타입**이 R의 주요 원자형 데이터 타입입니다 [1]. 이들은 **한 개의 값**(혹은 스칼라 값)을 가질 수도 있고, 같은 유형의 값 여러 개로 구성된 **벡터** 형태로 존재할 수도 있습니다. 다음으로 이러한 기본 타입들을 활용해 구성되는 R의 자료 구조들을 살펴보겠습니다.

2. 자료 구조 (데이터 구조)

R에서는 원자형 타입들을 조합하여 다양한 **자료 구조(data structure)**를 형성합니다. 주요 자료 구조로는 **벡터(vector)**, **팩터(factor)**, **리스트(list)**, **행렬(matrix)**, **배열(array)**, **데이터 프레임(data frame)** 등이 있습니다 [8] [9]. 이들 중 일부는 모든 원소가 **동일한 타입**을 가져야 하고 (예: 벡터, 행렬, 배열), 일부는 **다양한 타입**을 포함할 수 있습니다 (예: 리스트, 데이터 프레임). 각 구조가 어떻게 생겼고 언제 사용하는지 하나씩 살펴보겠습니다.

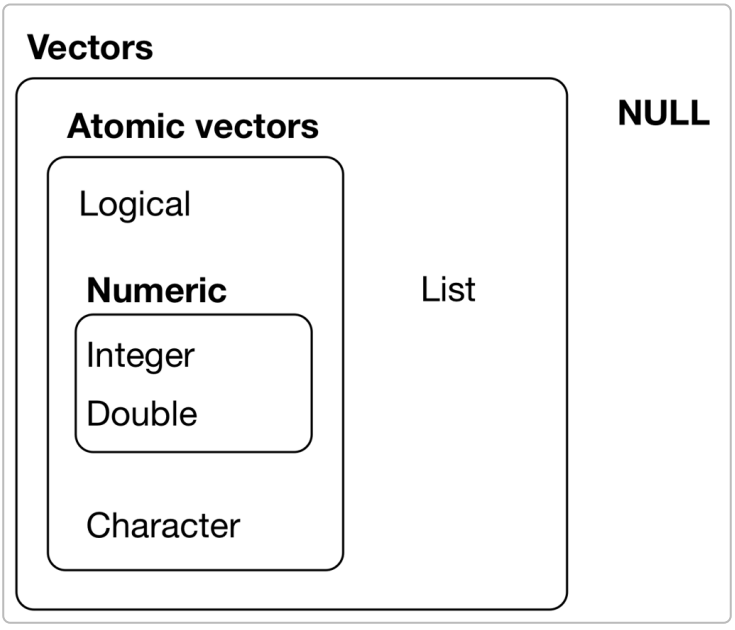

그림: R의 벡터 자료 구조 계층. R에서는 **벡터**가 기본 자료 구조이며, 그 중 **원자형 벡터(Atomic vectors)**는 원소들이 모두 같은 타입인 벡터를 말합니다. 원자형 벡터에는 **논리형, 숫자형(정수/실수), 문자형, 복소수형, Raw형**이 포함됩니다 [1]. **리스트(List)**는 벡터의 한 종류지만 원소들이 서로 다른 타입일 수 있어 별도로 구분되며, `NULL`은 값이 아예 없는 특수한 객체로 사용됩니다 (NULL은 길이 0인 벡터처럼 동작합니다).

Vector (벡터)

벡터(vector)는 R에서 **가장 기본적이고 중요한 자료 구조**입니다 [10]. 벡터는 **동일한 데이터 타입**의 값들을 1차원으로 쭉 나열한 구조로, 다른 언어의 배열(array)과 유사합니다 (R에서는 다차원 배열을 별도로 array라고 부르므로 혼동하지 마세요). 벡터는 R에서 데이터 연산의 기본 단위이기 때문에, R 프로그래밍의 많은 부분이 벡터를 중심으로 이루어집니다.

벡터를 생성하는 방법은 여러 가지가 있습니다:

- `c()` **함수 사용 (combine):** 가장 흔한 방법으로, `c()` 안에 값을 콤마로 구분해 나열하면 그 값들을 원소로 갖는 벡터가 생성됩니다.
- **콜론 연산자 사용:** `start:end` 형태로 시작값부터 끝값까지 1씩 증가하는 수열 벡터를 만들 수 있습니다.
- **시퀀스 함수 사용:** `seq()` 함수로 특정 간격의 등차수열 벡터를 생성할 수 있습니다.
- **벡터 생성 함수:** `numeric()`, `character()`, `logical()` 등 함수를 사용해 해당 타입의 지정 길이를 가진 벡터(초기값은 기본 NA 또는 0/"" 등)도 만들 수 있습니다 [11] [12].
- **예시:**

```
# 1) c() 함수를 사용한 벡터 생성
nums <- c(10, 20, 30)      # 숫자형 벡터
chars <- c("안녕", "Hello") # 문자형 벡터 (두 개의 문자열)
bools <- c(TRUE, FALSE, TRUE) # 논리형 벡터

# 2) 콜론을 사용한 벡터 생성
seq1 <- 1:5   # 1, 2, 3, 4, 5로 이루어진 벡터

# 3) seq() 함수를 사용한 벡터 생성
seq2 <- seq(from=1, to=10, by=2)  # 1, 3, 5, 7, 9 (1부터 2씩 증가하여 10이하까지)

# 4) 특정 타입의 벡터 초기화
empty_num <- numeric(3)   # 0이 세 개 들어있는 numeric 벡터 (0 0 0)
empty_char <- character(4) # ""(빈 문자열) 네 개로 이루어진 문자형 벡터
```

위 예시에서 `nums`는 세 개의 숫자로 이루어진 numeric 벡터입니다. `chars`는 한글 `"안녕"`과 영어 `"Hello"` 두 문자열로 이루어진 문자형 벡터이고, `bools`는 TRUE/FALSE 값으로 이루어진 논리형 벡터입니다. `seq1`은 1부터 5까지 연속된 정수 벡터를, `seq2`는 1부터 2씩 증가하는 홀수 벡터를 생성했습니다. 또한 `numeric(3)`은 길이 3의 숫자형 벡터를 0으로 채워서 만들고, `character(4)`는 길이 4의 문자형 벡터를 빈 문자열로 채워 생성합니다 [12] [13].

벡터의 중요한 성질은 **모든 원소가 동일한 타입**이어야 한다는 것입니다. 만약 서로 다른 타입을 한 벡터에 넣으면, R은 자동으로 **형 변환(coercion)**을 수행하여 모든 원소를 가장 **일반적인 타입**으로 바꿉니다 [5]. 일반적으로 논리 < 정수 < 실수 < 복소수 < 문자형 순서로 변환이 일어납니다 [14]. 예를 들어:

```r
mix1 <- c(TRUE, 2)     # TRUE가 1로 변환되어 numeric 벡터가 됨 (1, 2)
mix2 <- c(1.5, "3.2")  # 숫자 1.5가 문자열 "1.5"로 변환되어 문자형 벡터 ("1.5", "3.2")
mix3 <- c(FALSE, 2+3i) # FALSE->0+0i로 변환되어 복소수형 벡터 (0+0i, 2+3i)
```

`mix1`의 경우 TRUE (논리형)가 1 (숫자형)로 변환되어 최종 벡터는 숫자형이 되었습니다. `mix2`에서는 숫자 1.5가 문자 "1.5"로 변환되어 최종 벡터는 문자형이 됩니다. `mix3`에서는 FALSE가 0+0i로 변환되어 복소수형 벡터가 되었습니다. 이처럼 한 벡터 안에 타입을 혼용하면 R이 강제로 변환을 해버리므로, **가능하면 한 벡터에는 한 가지 타입의 데이터만 담는 것이 원칙**입니다.

벡터는 **길이(length)** 속성을 가지며, `length()` 함수를 통해 벡터의 원소 개수를 알 수 있습니다. 또한 `typeof()`, `class()`, `str()` 함수로 벡터의 내부 정보를 확인할 수 있습니다[15]. 예를 들어 위 `chars` 벡터에 대해:

```r
typeof(chars)  # "character" (원소들의 타입)
length(chars)  # 2 (원소 개수)
class(chars)   # "character" (특별한 클래스를 갖지 않은 기본 문자형 벡터)
str(chars)     # chr [1:2] "안녕" "Hello" (구조 출력)
```

`str(chars)`의 출력은 `chr [1:2] "안녕" "Hello"`와 같이 나타납니다[16]. 이는 문자형(chr) 벡터이고 [1:2] 범위(2개의 원소)를 가지며 각 원소를 일부 보여준다는 의미입니다. 이처럼 `str()` 함수는 객체의 유형과 내용을 요약해서 보여주므로 디버깅이나 데이터 확인 시 매우 유용합니다.

> **팁: 벡터에 이름 부여하기** – 벡터의 각 원소에 이름(label)을 줄 수도 있습니다. 예를 들어 `scores <- c(Alice=90, Bob=75, Carol=85)`처럼 하면 `scores` 벡터의 각 값에 "Alice", "Bob", "Carol"이라는 이름이 붙습니다. 이름은 `names(scores)`로 확인할 수 있습니다. 이는 데이터 프레임의 열 이름과 비슷한 개념으로, 결과 해석을 쉽게 해줍니다.

Factor (팩터)

팩터(factor)는 R에서 범주형(categorical) 데이터를 표현하는 특수한 구조입니다. 팩터는 겉보기에는 문자형 벡터와 매우 유사하지만, **내부적으로는 정수형 값을 가지며 각 정수에 레이블(label)**이 붙어 있는 형태입니다[17]. 팩터는 미리 정의된 범주(levels) 값만을 가질 수 있으며, 통계분석이나 그래픽 함수에서 범주형 자료로 인식되기 때문에 유용합니다.

예를 들어 **설문조사 응답**("만족", "보통", "불만족")이나 **날씨 상태**("맑음", "흐림", "비")처럼 몇 개의 범주로 분류되는 데이터를 팩터로 표현할 수 있습니다. 팩터를 사용하면 해당 값들이 단순 문자열이 아니라 범주형임을 R에게 알려주는 효과가 있습니다. 특히 회귀분석이나 분산분석 같은 통계 모델에서 팩터는 범주형 변수로 처리됩니다.

- **예시:**

```r
# 설문조사 만족도 결과를 팩터로 저장
survey <- factor(c("Good", "Bad", "Bad", "Good", "Ok"),
                 levels = c("Bad", "Ok", "Good"))
survey   # [1] Good Bad  Bad  Good Ok
         # Levels: Bad Ok Good
typeof(survey)  # "integer"
```

```
class(survey)  # "factor"
str(survey)    # Factor w/ 3 levels "Bad","Ok","Good": 3 1 1 3 2
```

위 코드에서 문자 벡터 `c("Good", "Bad", "Bad", "Good", "Ok")`를 팩터로 변환했습니다. `levels` 인수를 통해 가능한 범주 값을 "Bad", "Ok", "Good"으로 미리 정해주었습니다. `survey`를 출력하면 각 값이 보이지만(`Good, Bad, ...`), `Levels:` 아래에 팩터가 가지는 모든 범주(level) 목록이 표시됩니다. `typeof(survey)`를 보면 **"integer"**인데, 이는 팩터가 내부적으로 숫자 코드로 저장되기 때문입니다[17]. 반면 `class(survey)`는 **"factor"**로, 팩터임을 나타냅니다. `str(survey)` 출력을 보면 `"Factor w/ 3 levels "Bad","Ok","Good": 3 1 1 3 2"`처럼 보이는데, 이는 팩터가 3개의 레벨을 가지며 각 요소가 해당 레벨의 코드값을 갖고 있음을 보여줍니다 (예: "Good"은 코드 3, "Bad"는 코드 1, "Ok"는 코드 2로 저장됨).

팩터를 사용할 때 몇 가지 유의사항이 있습니다:

- 팩터는 **텍스트처럼 보여도 내부는 숫자**이므로, 문자열 연산을 직접 하면 예상과 다른 결과가 나올 수 있습니다. 예를 들어 팩터 값들을 `paste()`로 붙이면 레이블이 아닌 숫자 코드가 붙는 상황이 생길 수 있습니다. 이러한 경우 팩터를 **문자형으로 변환(**`as.character()`**)**해서 처리해야 합니다.
- 팩터에는 정의된 수준(level) 외의 값은 할당할 수 없습니다. 예를 들어 위 `survey` 팩터에는 "Great"이라는 값이 레벨에 정의되어 있지 않으므로 나중에 `survey[6] <- "Great"` 이런 것은 허용되지 않습니다 (잘못된 factor 레벨이라는 오류가 납니다). 새로운 값을 넣으려면 미리 레벨을 추가하거나 팩터를 다시 만들어야 합니다.
- 팩터에는 레벨 간 **순서(order)**가 있을 수도, 없을 수도 있습니다. 순서가 있는 팩터(ordered factor)는 "낮음 < 보통 < 높음"처럼 순서를 가지는 범주를 표현하며, `factor(..., ordered=TRUE)` 혹은 `ordered()` 함수를 사용해 만들 수 있습니다. 순서형 팩터는 크기 비교가 가능하다는 차이가 있지만, 기본적인 특성은 팩터와 같습니다.

일반적으로 **데이터를 파일에서 불러올 때 문자형 열이 자동으로 팩터로 변환**되는 기능이 있었으나 (R 4.0 이전 기본 동작), 최근 R 버전에서는 기본적으로 문자열을 팩터로 자동 변환하지 않습니다. 하지만 구버전이나 특정 상황에서 팩터로 바뀌는 경우가 있으니, 문자 데이터를 다룰 때 의도치 않게 팩터가 되지 않았는지 `str()`로 확인하는 습관을 들이는 것이 좋습니다.

List (리스트)

리스트(list)는 R에서 **서로 다른 타입의 객체들을 담을 수 있는 컨테이너** 자료 구조입니다. 리스트는 일종의 **재귀적 벡터(recursive vector)**라고 불리기도 하는데, 이는 리스트의 원소로 또 다른 리스트를 포함할 수도 있기 때문입니다[18]. 리스트는 Python의 리스트나 JSON 구조와 비슷한 면이 있으며, **여러 데이터를 한 번에 묶어서 반환**하거나 **복잡한 객체**를 표현하는 데 많이 사용됩니다.

리스트의 중요한 특징은 **동일한 자료형을 가질 필요가 없다**는 것입니다[18]. 하나의 리스트 안에 숫자, 문자, 논리값, 심지어 벡터나 행렬, 데이터 프레임까지 모두 넣을 수 있습니다. 이 때문에 리스트는 굉장히 유연한 구조이며, 함수의 리턴 값으로 여러 결과를 한꺼번에 돌려줄 때 주로 쓰입니다.

- **예시:**

```
# 다양한 타입의 데이터를 하나의 리스트에 담기
person <- list(
  name = "John Doe",
  age = 29,
  scores = c(90, 85, 92),
```

```
  passed = TRUE
)
str(person)
```

위 코드에서 `person`이라는 리스트를 생성했습니다. 이 리스트에는 `name` (문자형), `age` (숫자형), `scores` (숫자형 벡터), `passed` (논리형) 요소가 들어 있습니다. `str(person)`의 출력은 다음과 같습니다 (내용 요약):

```
List of 4
 $ name  : chr "John Doe"
 $ age   : num 29
 $ scores: num [1:3] 90 85 92
 $ passed: logi TRUE
```

이를 통해 리스트의 구조를 확인할 수 있습니다. `person` 리스트는 4개의 항목을 가지며 (`List of 4`), 각 항목의 이름과 타입, 값의 일부를 보여줍니다.

리스트를 생성할 때는 위 예시처럼 `list()` 함수를 사용하며, 각 원소에 이름을 부여할 수도 있습니다 (`name=...` 처럼). 리스트의 원소에 접근하는 방법은 **인덱스**나 **이름**을 사용하는 두 가지가 있습니다:

- `my_list[[i]]` : i번째 원소에 접근. (겹대괄호 `[[ ]]`는 리스트 원소 그 자체를 가져옴)
- `my_list$name` : 이름이 `name`인 원소에 접근 (편리한 문자 접근 방법).
- `my_list[i]` : i번째 원소를 **하나짜리 리스트**로 가져옴. (대괄호 하나 `[ ]`는 결과를 여전히 리스트 형태로 유지)

예를 들어 `person$name` 또는 `person[[1]]`은 `"John Doe"`를 반환하지만, `person[1]`은 길이 1의 리스트 (첫 번째 원소만 가진 리스트)를 반환합니다. 초보자는 대괄호 하나와 둘의 차이를 헷갈리기 쉬우므로, **리스트에서 특정 원소의 실제 값에 접근하려면 항상 `[[ ]]`나 `$` 표기법을 써야** 함을 기억하세요.

또한 빈 리스트를 미리 만들어 놓고 나중에 채우는 것도 가능합니다: `empty_list <- vector("list", length=5)`처럼 하면 5개 항목을 담을 빈 리스트가 생성됩니다 [19] [20]. 리스트는 데이터 프레임의 기초가 되는 구조이기도 한데, 다음에 설명할 데이터 프레임은 사실 리스트의 특수한 형태입니다 [21].

Matrix (행렬)

행렬(matrix)은 R에서 2차원으로 데이터를 배치한 구조로, 수학의 행렬 개념과 유사합니다. 사실 R에서 행렬은 **"차원을 가진 벡터"**로 이해할 수 있습니다 [22] [23]. 즉, 행렬은 벡터에 행(row)과 열(column)의 **차원 속성**(`dim`)을 부여한 것이며, 별도의 새로운 자료형이라기보다는 벡터의 확장판입니다 [23]. 행렬의 중요한 특징은 **모든 원소가 동일한 데이터 타입**이어야 한다는 점으로, 이 점은 벡터와 같습니다.

행렬을 생성하는 방법은 여러 가지가 있습니다:

- `matrix()` **함수 사용:** 데이터 (또는 데이터 생성식)와 `nrow`(행 수), `ncol`(열 수) 등을 지정하여 행렬을 생성합니다.
- **벡터에** `dim` **속성 지정:** 이미 있는 벡터에 `dim(x) <- c(nrow, ncol)`처럼 차원 속성을 부여하여 행렬로 변환할 수 있습니다 [24] [25].
- `cbind()` / `rbind()` **사용:** 여러 벡터를 열로 묶거나 행으로 묶어서 행렬을 만들 수 있습니다 [26] [27]. 예를 들어 `cbind(x, y)`는 벡터 `x`, `y`를 열 단위로 합쳐서 행렬로 만듭니다.

- **예시:**

```
# 1) matrix() 함수 사용
m1 <- matrix(1:6, nrow=2, ncol=3)
# 2) 벡터에 dim<- 사용
v <- 1:6
dim(v) <- c(2, 3)  # v를 2x3 행렬로 변환
# 3) cbind()/rbind() 사용
a <- 1:3
b <- 4:6
m2 <- cbind(a, b)  # 3x2 행렬 (a와 b를 열로 결합)
```

`m1`과 `v`는 똑같은 값을 가진 2x3 행렬입니다. `1:6`은 1부터 6까지의 벡터인데, `matrix(1:6, nrow=2, ncol=3)`는 이를 2행 3열로 배치한 행렬로 만들어 줍니다. 똑같은 작업을 `v` 벡터에 `dim(v) <- c(2,3)`로 해주면 `v` 자체가 행렬 구조로 바뀝니다[24]. `m2`는 `a`와 `b` 두 개의 3원소 벡터를 열 방향으로 붙여 3행 2열 행렬을 만든 것입니다 (`m2` 내용은 첫 번째 열이 1,2,3; 두 번째 열이 4,5,6).

행렬은 기본적으로 **열(column) 우선 순서**로 채워집니다[28]. `m1 <- matrix(1:6, nrow=2, ncol=3)`의 결과를 예로 보면, `1:6` 벡터는 `c(1,2,3,4,5,6)`인데 이를 2행 3열로 채울 때 **먼저 첫 번째 열에 1,2가 들어가고**, 다음 열에 3,4, 다음 열에 5,6이 들어갑니다. 만약 행 우선으로 채우고 싶다면 `matrix(..., byrow=TRUE)` 옵션을 주면 됩니다[29].

행렬의 타입은 그 원소의 타입에 따라 결정됩니다. 예를 들어 정수로 이루어진 행렬은 내부적으로 정수형 벡터이며, 문자로 이루어진 행렬은 문자형 벡터입니다. 행렬인지 여부는 `class()`로 확인할 수 있는데, 행렬은 클래스가 `"matrix"`로 표시됩니다. 흥미로운 점은 `typeof()`를 해보면 행렬도 결국 벡터이기 때문에 **원소의 타입**을 알려준다는 것입니다.

예를 들어 `m1`이 전부 정수로 이루어졌다면 `class(m1)`은 `"matrix"` (또는 `c("matrix","array")`)로 나오지만, `typeof(m1)`은 `"integer"`로 나옵니다[30][31]. 즉, R은 이 객체가 행렬 클래스의 객체이긴 하나 본질적으로는 정수들의 모음이라는 것을 보여줍니다.

- **예시 (행렬의 `class()`와 `typeof()`):**

```
class(m1)  # "matrix" "array"
typeof(m1) # "integer"
```

`class(m1)` 결과는 `"matrix"` (그리고 `"array"`)로, 이 객체가 행렬임을 나타냅니다. `typeof(m1)` 결과는 `"integer"`로, 행렬의 내부 저장이 정수형 벡터라는 뜻입니다. 만약 실수 행렬이라면 `typeof()`가 `"double"`로 나오겠죠[32]. 이처럼 `class()`와 `typeof()`는 객체를 바라보는 관점이 다르며, 이에 대해서는 뒤에서 더 설명하겠습니다.

행렬의 원소에 접근하려면 **인덱스 두 개**를 사용합니다: `matrix[row, col]` 형식으로 `[행번호, 열번호]`를 지정하면 됩니다. 예를 들어 `m1[2, 3]`은 m1 행렬의 2행 3열 원소를 가리킵니다. 콤마 앞 인덱스를 생략하면 해당 열 전체, 뒤 인덱스를 생략하면 해당 행 전체를 의미합니다. (`m1[1, ]`는 1행 전체, `m1[, 2]`는 2열 전체를 벡터로 반환.) 행렬은 2차원이므로 `dim(m1)`으로 차원을 확인할 수 있고, `nrow(m1)`, `ncol(m1)`로 행 수와 열 수를 각각 알아낼 수 있습니다.

Array (배열)

배열(array)은 행렬을 일반화한 개념으로, **2차원보다 높은 차원**을 갖는 다차원 배열 구조를 말합니다. R에서 배열은 사실 벡터에 차원 속성을 부여한다는 점에서 행렬과 동일한 방식으로 동작합니다. 다만 배열은 3차원, 4차원 … n차원까지 확장될 수 있다는 차이가 있습니다 [33] .

배열을 만드는 방법은 `array()` 함수를 사용하는 것이 일반적입니다. `array(data, dim = c(X, Y, Z, ...))` 형식으로 쓰며, `data`는 채울 데이터를 지정하고 `dim`에는 각 차원의 크기를 벡터로 지정합니다. 만약 `data`의 길이가 지정한 차원 전체 크기보다 작으면, 데이터가 반복 recycling되어 배열을 채웁니다.

- **예시:**

```
# 2x2x2 삼차원 배열 만들기
arr <- array(1:8, dim = c(2, 2, 2))
arr
```

`arr`은 2 x 2 x 2 크기의 3차원 배열입니다. `1:8` 벡터 (1,2,...,8)를 3차원으로 채웠기 때문에, 결과는 2x2 행렬이 두 개 쌓인 형태가 됩니다. 출력 결과는 매트릭스 2개가 층으로 표시되어, 예컨대 다음과 같이 보일 것입니다:

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

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

여기서 `, , 1` 부분은 3차원 배열의 세 번째 차원 인덱스가 1인 “첫 번째 면”을 나타내고, `, , 2`는 두 번째 면을 나타냅니다. 각 면은 2x2 행렬이며, 첫 면에는 1,2,3,4가, 둘째 면에는 5,6,7,8이 채워진 모습입니다. (행렬과 마찬가지로 기본은 열 우선 채우기이므로 1-2가 첫 열, 3-4가 둘째 열로 들어갔습니다.)

배열 역시 행렬과 마찬가지로 모든 원소가 같은 타입이어야 합니다. 배열의 `class()`를 찍어 보면 `"array"`라고 나옵니다. 2차원 배열은 행렬과 동일하게 취급되므로 `class(matrix_obj)`를 하면 `"matrix"`와 `"array"` 두 값을 모두 보여주지만, 3차원 이상의 배열은 `"array"`로만 표시됩니다. 내부적으로는 이것도 벡터이므로 `typeof()`는 원소 타입을 나타냅니다 (예: numeric 배열이면 "integer"/"double", 문자 배열이면 "character"). 배열의 원소 접근은 대괄호 안에 위치를 차원별로 콤마로 구분하여 지정합니다. 예를 들어 `arr[2, 1, 2]`는 3차원 배열 `arr`의 [2행, 1열, 2번째 면]의 값을 가리킵니다.

배열은 고차원 데이터를 다루거나 이미지처럼 행렬 여러 개 층으로 이루어진 데이터를 처리할 때 사용됩니다. 하지만 2차원 이상의 데이터는 보통 **데이터 프레임** 또는 **리스트**로 관리하는 경우가 많아서, 배열은 상대적으로 사용 빈도가 낮습니다. 필요할 때 만드는 법만 알아 두고, 기본 데이터 분석에서는 행렬(2차원)까지만 자주 쓰인다고 생각하면 됩니다.

Data Frame (데이터 프레임)

데이터 프레임(data frame)은 R에서 **표 형식의 데이터(테이블)**를 다루기 위한 기본 구조입니다. 데이터 프레임은 열(column)들로 이루어져 있으며, 각 열은 **벡터**로 취급됩니다. 데이터 프레임은 엑셀의 시트나 SQL의 테이블과 유사하게 행과 열로 데이터를 구성하며, **각 열마다 데이터 타입이 동일**하지만 **열마다 서로 다른 타입**을 가질 수 있다는 특징이 있습니다 [21]. 즉, 어떤 열은 숫자형, 다른 열은 문자형, 또 다른 열은 팩터일 수 있습니다. 이러한 유연성 덕분에 데이터 프레임은 통계 분석에서 가장 흔히 쓰이는 자료 구조입니다.

데이터 프레임을 생성하는 방법은 보통 두 가지입니다:

- **데이터를 불러와서 생성:** CSV 파일이나 Excel 파일을 `read.csv()`, `read.table()` 등으로 읽으면 자동으로 데이터 프레임이 만들어집니다. 이 방식이 가장 일반적입니다.
- **직접 코드로 생성:** `data.frame()` 함수를 사용하여 벡터들을 열로 묶어 데이터 프레임 객체를 만들 수 있습니다.
- **예시:**

```
# 벡터를 묶어서 데이터 프레임 생성
names <- c("철수", "영희", "길동")
ages  <- c(25, 30, 19)
passed <- c(TRUE, TRUE, FALSE)
df <- data.frame(Name = names, Age = ages, Passed = passed)
print(df)
str(df)
```

`names`, `ages`, `passed` 세 개의 벡터를 준비한 뒤, `data.frame()`으로 묶어 하나의 데이터 프레임 `df`를 만들었습니다. `df`를 출력하면 다음과 같이 표시됩니다:

```
  Name Age Passed
1 철수  25   TRUE
2 영희  30   TRUE
3 길동  19  FALSE
```

각 열에 이름이 붙어서 표처럼 보입니다. `str(df)`로 구조를 확인해보면:

```
'data.frame': 3 obs. of  3 variables:
 $ Name   : chr  "철수" "영희" "길동"
 $ Age    : num  25 30 19
 $ Passed: logi  TRUE TRUE FALSE
```

라고 나옵니다. 이 출력은 `df`가 3개의 관측치(obs., 행)와 3개의 변수(열)를 가진 데이터 프레임이며, 각 열의 이름과 유형을 보여줍니다. Name 열은 문자형 (chr)이고, Age 열은 numeric (num), Passed 열은 logical (logi)임을 알 수 있습니다. 데이터 프레임의 `class(df)`는 당연히 `"data.frame"`입니다.

데이터 프레임은 내부적으로 **리스트의 특수한 형태**입니다 [21]. 사실 데이터 프레임은 각 열이 리스트의 원소로 저장된 리스트이며, 모든 원소(열)의 길이가 동일하도록 제약이 걸린 리스트라고 이해할 수 있습니다 [21]. 확인해보면 `is.list(df)`가 TRUE를 반환하고, `typeof(df)`는 "list"로 표시됩니다 [34] [35]. 앞서 생성한 `df`에 대해 `typeof(df)`를 하면 **"list"**가 나오는데, 이는 데이터 프레임이 메모리 상에서 리스트로 구현되어 있기 때문입니다 [34]. 하지만 `class(df)`는 "data.frame"으로, R이 이를 표처럼 다루도록 특별 취급하는 객체임을 나타냅니다.

데이터 프레임을 다룰 때 유용한 함수로는 `nrow()` (행 수), `ncol()` (열 수), `names()` 또는 `colnames()` (열 이름들), `head()` (처음 몇 행 미리보기), `tail()` (마지막 몇 행) 등이 있습니다 [36]. 또한 데이터 프레임도 리스트이므로 `df$ColumnName` 또는 `df[["ColumnName"]]`으로 특정 열벡터에 접근할 수 있고, `df[i, j]`로 i번째 행, j번째 열 원소에 접근할 수 있습니다.

주의: 데이터 프레임에서 문자열이 팩터로 자동 변환되는 문제는 앞서 팩터 부분에서 언급했듯이 R 4.0부터는 기본적으로 발생하지 않습니다. 그러나 직접 `data.frame()`을 사용할 때 오래된 R 코드에서는 `stringsAsFactors=FALSE` 옵션을 주곤 했는데, 현재 버전에서는 기본이 FALSE라 굳이 지정할 필요는 없습니다. 그래도 기존 코드나 자료를 참고할 때 이 옵션을 볼 수 있으니 알아두세요.

또한 데이터 프레임을 다루다 보면 **요소별 연산**에 주의해야 합니다. 행렬은 숫자형이면 `m1 * 2` 같은 연산이 바로 통하지만, 데이터 프레임은 리스트처럼 동작하므로 `df * 2` 같은 연산이 바로 되지 않습니다. 데이터 프레임의 특정 열에 연산을 적용하려면 `df$Age * 2` 이런 식으로 벡터를 꺼내서 해야 합니다. 혹은 `dplyr` 패키지 같은 도구를 사용하여 처리합니다. 초보 단계에서는 헷갈릴 수 있으니, 데이터 프레임은 기본적으로 **리스트의 집합**으로 동작한다는 점을 유념하세요.

以上が、Rにおける基本データ型と主なデータ構造の説明です (한국어 설명 중간에 일본어?)

---

[1] [2] [3] [7] [9] 20 Vectors | R for Data Science
https://r4ds.had.co.nz/vectors.html

[4] [12] [13] [15] [16] [18] [19] [20] [23] [25] [27] [29] [30] [31] [32] Programming with R: Data Types and Structures
https://swcarpentry.github.io/r-novice-inflammation/13-supp-data-structures.html

[5] [6] [8] [10] [11] [14] [17] [21] [22] [24] [26] [28] [36] Understanding basic data types in R
https://resbaz.github.io/2014-r-materials/lessons/01-intro_r/data-structures.html

[33] Understanding vectors, matrices and arrays? : r/Rlanguage - Reddit
https://www.reddit.com/r/Rlanguage/comments/zogef0/understanding_vectors_matrices_and_arrays/

[34] [35] What is the difference between class and typeof function in R?
https://www.tutorialspoint.com/what-is-the-difference-between-class-and-typeof-function-in-r