의료정보학: R을 이용한 자료분석

R 설치하고 맛보기

이윤석

2021. 의예과 2년 1학기

1 설치 및 첫 실행

Linux 사용자가 아니라면 R 소프트웨어의 설치용 바이너리를 실행함으로써 R을 설치할 수 있다. R의 공식 홈페이지[1]에서 "설치" 관련 링크를 따라가서 최신 버전을 골라서 내려받을 수 있다. 설치 과정에서 특별히 주의할 만한 사항은 없으며, R의 출력문을 우리글로 번역하면 여전히 매끄럽지 않으므로 훗날에 닥칠 용어의 혼란을 피하기 위해서 메시지가 "영어"로 표시되도록 선택지를 골라서 설치하는 것이 좋다.

```
R version 3.5.3 (2019-03-11) -- "Great Truth"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[R.app GUI 1.70 (7632) x86_64-apple-darwin15.6.0]

[History restored from /Users/aadm/.Rapp.history]

>
```

이 글이 쓰이는 시점에서 최신의 R 소프트웨어 판본은 4.0대에 들어 있

[1] https://www.r-project.org/

지만 R의 상하위 호환성은 우수하기 때문에 얼마간의 다른 판본을 쓰더라도 서로간 차이를 발생하지 않을 것이다. 화면에 보이는 R 시동 화면은 판본 3.5.3이다. 이 화면에서 보이는 R의 대화형 입출력 행들을 콘솔로 부른다. 스크립트 창을 별개로 열기 전에는 사용자는 오로지 콘솔을 통해서 R과 대화한다. 여기까지 아무 에러 없이 시동했다면, 보이는 화면처럼 R이 명령 대기 상태에 놓이고, R의 대기 프롬프트는 오른꺽쇠(“>”)이다. 콘솔에서는 오른꺽쇠 다음에 명령을 입력한 뒤에 엔터키를 입력하는 방식으로 명령을 전달한다.

우리의 첫 R 명령문은 1부터 10까지의 자연수의 합을 구하라는 명령문이다.

```
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
1 + 2 + 3 + 4 + 5+6 +7 + 8 + 9 + 10
```

첫 행에서는 덧셈 기호 좌우로 공백을 한 칸씩 두었고 둘째 행에서는 제멋대로 쳤지만 결과는 55로 같다. 55 앞에 써 있는 “[1]”이라는 표시의 의미를 지금은 무시하자. 기호 좌우의 공백을 제멋대로 쓴 것이 통했었는데 제멋대로 쓰는 데에도 한계가 있다.

```
> 1 + 2 + 3 + 4 + 5+6 +7 + 8 + 9 + 1 0
Error: unexpected numeric constant in "1 + 2 + 3 + 4 + 5+6 +7 + 8 + 9 +
    1 0"
```

하나의 숫자 10글 반드시 모아서 10으로 써야 하지 사이에 공백을 두어서는 위와 같은 에러가 발생한다. 앞으로도 수많은 에러를 만날 터인데, 초보자가 만들어내는 프로그래밍 에러의 태반은 오타에서 기인함을 기억하자. 고등학교 시절에 배운 수학함수들을 머릿속에 떠올려 R에 넣어 보아라.

```
exp(1)
log(10)
sin(30)
sqrt(2)
```

결과가 뜻하는 바는 차차 알기로 하고 이번에는 제법 복잡한 수식을 계산

해 보겠다. 중고교 수학에서 근의 공식으로 알려진 수식이다.

$$when \quad ax^2 + bx + c = 0,$$
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$x^2 + 2x + 1 = 0$, 즉 a = 1, b = 2, c = 1로 설정하고, 이론적으로는 근이 두 개지만 제곱근 항을 0으로 만들어서 근을 하나로 줄였다. 그 근을 x에 담았다가 출력하는 코드이다.

```
> a <- 1
> b <- 2
> c <- 1
> x <- (-b + sqrt(b^2 - 4 * a * c)) / (2 * a)
> print(x)
[1] -1
```

위에서 a, b, c, x를 변수(variables)로 부른다. "이름을 가진 저장공간"으로 생각하면 편하다. 구문 `a <- 1`는 "a라는 이름의 변수를 만들어서 1을 보관한다"라는 뜻이다. 이어지는 `b <- 2`, `c <- 1`에서도 관례에 따랐고 x의 경우에는 사용자가 값을 직접 입력하지 않은 채 주어진 변수들로 일정한 계산을 하게끔 한 뒤 그 결과를 넣었다는 점만 다르다.

변수명은 알파벳이나 마침표로 시작하고 알파벳과 숫자(그리고 기호인 마침표[.] 또는 밑줄[_])로 이루어져야 한다. 알파벳의 대소문자를 다른 문자로 인식하므로 변수 a와 변수 A는 다른 변수이다. 안에 든 게 무엇이건 역따옴표(backquote; 키보드의 숫자 1키의 왼쪽에 놓인 키)로 둘러싼 이름은 무엇이든 허용된다[2]. 다음은 허용되는 변수명이다[3].

```
myVariable, URno, this_is_not_a_pipe., Lucky7, bmw.x7
```

다음은 규칙을 어겼으므로 허용되지 않는 변수명이다.

[2]한글 변수명을 써야 하는 불가피한 경우가 아니라면 고려하지도 말라.

[3]잘 알려지지 않은 불허 규칙이 있다. 마침표로 시작한 변수명이 숫자로 이어져서는 안 된다는 것이다. 즉 `.april`은 허용되나 `.4pril`은 허용되지 않는다.

```
7isMyfavorite, Room-No-17, smile+laugh, lotto6/45
```

다음은 R이 내부적으로 특별한 의미를 부여한 낱말들이어서 ("예약어 (reserved word)") 변수명으로 사용할 수 없다.

```
if else repeat while function for in next break
TRUE FALSE NULL Inf NaN NA
NA_integer_ NA_real_ NA_complex_ NA_character_
```

여러 프로그래밍 언어들과는 달리 R의 변수 유형은 유연하다. 별도로 선언하는 절차 없이도 변수가 생성되며, 어떤 값이 배정되는지에 따라 그때그때 자동으로 유형을 달리한다. 숫자를 받으면 숫자 변수가 되고 문자열을 받으면 문자열 변수가 되었다가 참거짓 값을 받으면 논리 변수가 된다.

```
> (flexie <- 365.25)
[1] 365.25
> class(flexie)
[1] "numeric"
> (flexie <- "Wouldn't it be loverly?")
[1] "Wouldn't it be loverly?"
> class(flexie)
[1] "character"
> (flexie <- TRUE)
[1] TRUE
> class(flexie)
[1] "logical"
```

변수 flexie가 처음에는 숫자였다가 문자열이 되었다가 논리 변수가 되는 과정에서 아무런 제약도 없고 아무런 경고도 발생하지 않았다. 이 점은 가끔 프로그래머의 논리흐름을 흐리게 하는 장애물이 되므로 긴 R 코드가 뜻하는 대로 작동하지 않을 때 중간 중간에 변수의 유형이 예기치 않게 변화하지 않았는지 점검하는 것도 중요하다.

여기까지 맛보기를 마치면서 R을 종료한다. 시스템 네임스페이스에 올라와 있는 내용을 점검하고 종료하였다.

```
> ls()
```

오늘 사용했던 변수 a, b, c, flexie, x가 알파벳 순서대로 나열되었고 `quit()` 함수로 R 종료를 명령했더니 작업공간 이미지를 저장할지 여부를 묻는다. “n”를 눌러서 종료한다. 처음부터 `quit("no")`로 종료하면 결과가 같다.

```
> q()
Save workspace image? [y/n/c]:
```

2 두 번째 R 세션 실행

R.app을 클릭해서 R을 구동하거나 스크립트 파일을 먼저 여는 방식으로 R을 구동할 수도 있다(그림 1). 이 부분만은 운영체제 간 차이가 있으며 중요치 않다. 스크립트 파일이라는 것은 실행 가능한 R 코드를 담고 있는 텍스트파일로서 확장자가 `.R`인 파일이다. R이 기본으로 보유하고 있는 자산은 기본

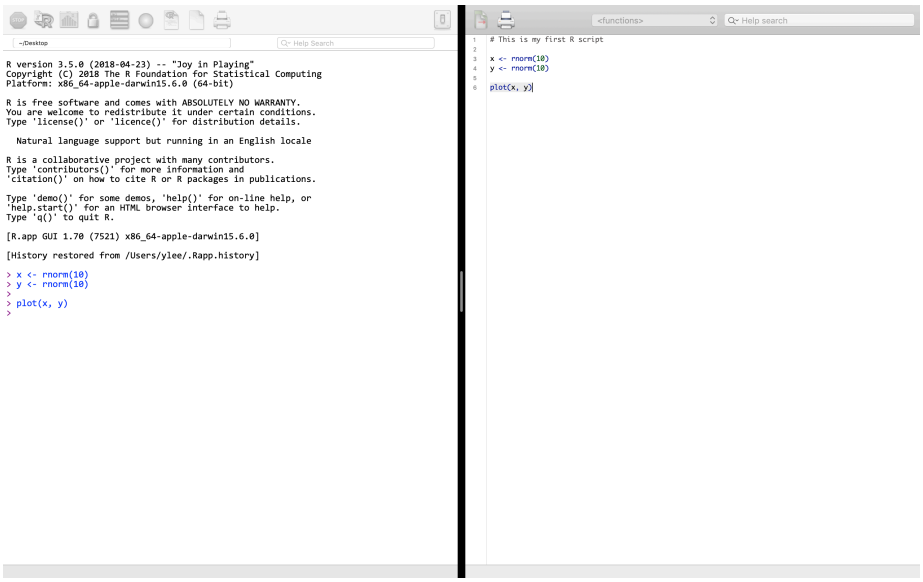

그림 1: 스크립트 파일과 함께 구동된 R. 오른쪽이 스크립트, 왼쪽이 R이다. 운영체제마다 모습이 다를 수 있다.

(base) 패키지로 불리지만 R을 설치할 때 덤으로 설치되는 패키지는 base를

포함해서 서른 종이다. 다음의 결과는 사용자마다, 시기마다 다를 것이므로 다르다고 염려하지 말라.

```
> rownames(installed.packages())
 [1] "base"       "boot"       "class"      "cluster"    "codetools"
     "compiler"
 [7] "datasets"   "foreign"    "graphics"   "grDevices"  "grid"
     "KernSmooth"
[13] "lattice"    "MASS"       "Matrix"     "methods"    "mgcv"       "nlme"
[19] "nnet"       "parallel"   "rpart"      "spatial"    "splines"    "stats"
[25] "stats4"     "survival"   "tcltk"      "tools"      "translations" "utils"
```

이들까지 포괄해서 R의 기본 패키지라고 보아도 무방하지만 이들 중 일부 패키지를 쓰려면 메모리에 장착한다(= "네임스페이스를 활성화한다"). 라이브러리 MASS를 장착하려면,

```
library(MASS)
```

의 꼴로 명령을 내리면 된다. MASS 패키지에 관한 전반적인 안내문은 ??MASS 명령으로 얻을 수 있으며 CRAN 사이트에서는[4] 알파벳 순서대로 색인할 수 있다.

설치는 되어 있지만 메모리에 올라오지 않은[5] 패키지 속 함수를 쓸 수 있는 방법이 하나 있다. 아예 명령문에 패키지 이름을 네임스페이스로 지정하는 방법이다. 가령 nlme가 보유하고 있는 선형혼합모형 함수인 lme()를 써서 nlme가 보유하고 있는 자료철인 Orthodont을 분석할 때 nlme를 장착하지 않고도 다음 명령들만으로 충분하다.

```
> nlme::lme(distance ~ age * Sex, random = ~ 1 | Subject,
data = nlme::Orthodont) #
```

여러 패키지가 동시에 메모리에 올라온다고 해도 실제로 줄어드는 R 자원은 아주 적지만, 혼란을 피하고자 할 때 장착하지 않은 채 패키지를 사용하는 것도 슬기로운 습관이다. namespace::function_name() 형태로 쓴다.

[4] http://cran.r-project.org

[5] 이 표현을 자주 만나게 될 것이다. "installed but NOT loaded"

이렇게 표기하는 습관은 특정 함수가 속해 있는 네임스페이스를 기억할 수 있다는 장점도 함께 지니고 있다.

R이 가진 객체 중에서 가장 독특한 유형을 들자면, 벡터(vector)일 것이다. 기하학에서 배웠던 벡터의 개념과는 무관하다. R이 말하는 벡터는, 다수의 동질적인 자료 값을 포괄하는 자료구조를 뜻하며 반대의 뜻을 가진 스칼라는 실제로 R에는 존재하지 않는다. 앞 절에서 다루었던 a, b, c, x 등은 (정의상) 모두 스칼라처럼 보였지만 R은 스칼라조차도 요소가 하나뿐인 벡터로 취급한다. 요소가 하나뿐이라는 말은 벡터의 길이가 1이라는 뜻이다.

```
> a <- 3.14
> a
[1] 3.14
> is.vector(a)
[1] TRUE
> length(a)
[1] 1
```

다수의 값이 아니라 단일값을 가진 a도 스칼라가 아니라 벡터라고 했었다. a를 출력할 때 행의 맨 앞에 [1]이 먼저 등장하는 이유도 그래서이다: "벡터 a의 첫 요소임"이라는 뜻이다. 여러 개의 요소로 이루어진 벡터는 가령 다음처럼 생겼다.

```
> (a <- c(1, 2, 3, 5, 7, 11, 13, 17, 19, 23))
 [1]  1  2  3  5  7 11 13 17 19 23
> (x <- c("Call", "me", "Ishmael", ".") )
[1] "Call"    "me"      "Ishmael" "."
> class(a)
[1] "numeric"
> class(x)
[1] "character"
> is.vector(a)
[1] TRUE
> is.vector(x)
[1] TRUE
```

a는 1부터 23까지의 소수로 이루어진 길이 10인 숫자 벡터(numeric vector)

이고 x는 마침표를 포함해서 네 개의 낱말로 이루어진 문자 벡터(character vector)이다. 위에 등장한 함수 `class()`와 `length()`는 벡터 객체에 대해서 행할 수 있는 "방법(method)" 또는 번역하지 않은 채 "메소드"로 불린다. 전자는 벡터의 유형을 밝히고 후자는 벡터의 길이를 밝힌다. 출력물의 표시 `[1]`을 통해서 이들의 결과물조차도 벡터임을 알 수 있을 것이다. 오늘 다루든 아니든 R의 모든 객체는 벡터이다.

벡터를 입력하는 방법은 다양하며 일반적으로는 고유한 값을 직접 입력하게 되는데 단지 숫자를 나열하기만 해서는 되지 않으며, 위에서 나왔던 `c()` 함수를 쓴다. `c()` 함수는 인수로 받은 객체들을 하나의 벡터 객체로 통합해 준다. 만약 유형이 다른 값이 섞여 있다면 논리형 < 숫자형 < 문자형의 우세도에 따라 강제변환이 이루어진다. 즉,

```
> c(1, 3, 5, 8, TRUE)
[1] 1 3 5 8 1
> c(1, 3, 5, 8, TRUE, "fair coin")
[1] "1"       "3"       "5"       "8"       "TRUE"    "fair coin"
```

이다. 논리형이 숫자로 변환될 때 TRUE는 1, FALSE는 0으로 바뀐다[6].

사용자가 직접 입력하는 벡터가 아니라 일정한 규칙에 따라 벡터를 생산하는 경우도 흔하다. R은 세 가지 함수를 구비하고 있다. 첫째 from:to의 꼴로 순열을 만들어 내는 콜론 함수로서, 응용의 폭이 넓다.

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> 19:11
[1] 19 18 17 16 15 14 13 12 11
> 1:5 * 2 - 1
[1] 1 3 5 7 9
```

둘째는 `seq(a, b)`의 형태로 호출하는 범용의 순열 함수이다. a에는 시작값을 넣고 b에는 끝값을 넣는데 옵션으로 `by` 또는 `length`를 넣을 수 있다. `by =` 는 숫자 사이의 간격으로서 옵션명을 생략할 수 있고 `length =` 는 생

[6] 숫자를 논리형으로 변환하는 경우에는 0을 제외한 모든 값이 TRUE로, 0만은 FALSE로 처리된다.

산되는 벡터의 길이를 지칭하는 뜻으로서 생략할 수 없다. 아래 예의 `length = 3`은 1과 10 사이들 세 개의 숫자로 균등분할하겠다는 뜻이다.

```
> seq(1, 10, by = 2)
[1] 1 3 5 7 9
> seq(1, 10, 2)
[1] 1 3 5 7 9
> seq(1, 10, length = 3)
[1]  1.0  5.5 10.0
```

마지막 자동 벡터 생산기는 `rep()`으로서 일정한 주기로 반복하는 벡터를 만든다. 대표적인 쓰임은 다음과 같다.

```
> rep(c(1, 2, 3), 4)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3
> rep(c(100, 1000), each = 3)
[1]  100  100  100 1000 1000 1000
> (group_name <- rep(c("control", "study"), each = 10))
 [1] "control" "control" "control" "control" "control" "control"
     "control" "control"
 [9] "control" "control" "study" "study" "study" "study" "study" "study"
[17] "study" "study"  "study"  "study"
> (dosage <- rep(rep(c("Standard", "Mega"), each = 5), 2))
 [1] "Standard" "Standard" "Standard" "Standard" "Standard" "Mega" "Mega"
 [8] "Mega"   "Mega"    "Mega"    "Standard" "Standard" "Standard"
     "Standard"
[15] "Standard" "Mega" "Mega"    "Mega"    "Mega"    "Mega"
```

입력된 요인변수는 요인(factor)이 아니라 문자(character)이므로 후처리가 필요하다.

```
> class(dosage)
[1] "character"
> dosage <- factor(dosage, c("Standard", "Mega"))
> class(dosage)
[1] "factor"
```

반면에 `gl()` 함수는 규칙적으로 반복하는 요인 벡터를 생성한다. RCT 연구 결과를 입력할 때 요인으로 변수명을 반복할 때 쓰임이 좋다.

```
> # Repeating 2-level factor by 10 each
> gl(2, 10, labels = c("control", "study"))
 [1] control control control control control control control control
     control control
[11] study study  study  study  study  study  study  study  study  study
Levels: control study
> # Repeating 2-level factor by 5 each totalled 20
> gl(2, 5, 20, labels = c("Standard", "Mega"))
 [1] Standard Standard Standard Standard Standard Mega Mega Mega  Mega
[10] Mega   Standard Standard Standard Standard Standard Mega Mega Mega
[19] Mega   Mega
Levels: Standard Mega
```

프로그래밍에 익숙지 않은 사용자가 이해하기 가장 어려운 자료형이 요인일 것이다. 요인은 레벨(level)과 이름(label)으로 이루어진 명목척도이며 가장 간단한 요인형은 이분형(二分型)이다. "Male/Female," "Boy/Girl," "Success/Failure" 등이 모두 이분형 요인으로서 내부적으로는 1과 2으로 코딩되면서 이름만 다를 뿐이다. 다음은 outcome이라는 요인형 벡터를 정의하고 입력하는 과정으로서 이름으로 "Success/Failure" 중 하나를 가지도록 한다.

```
> k <- c("Success", "Failure", "Success", "Success")
> outcome <- factor(k)
> outcome
[1] Success Failure Success Success
Levels: Failure Success
```

요인 벡터를 만들 때 이름을 지정하지 않으면 입력된 값들의 고유값(unique value)이 자동으로 전달된다. 요인의 레벨은 알파벳 순서가 디폴트이므로 다른 방식을 원한다면 `factor()` 함수 내에 `levels =` 옵션을 쓰면 된다.

```
> (outcome <- factor(k, levels = c("Success", "Failure")))
[1] Success Failure Success Success
```

```
Levels: Success Failure
```

손수 레벨을 지정해야 하는 당장의 용도는 보이지 않지만 분산분석 등에서 요인 변수의 계수를 해석하거나 대비값을 생성할 때 반드시 알고 있어야 하는 사항이다. 세 개 이상의 레벨을 가진 요인은 위의 이분형 요인을 자연스럽게 확장해서 만들면 된다.

```
> monitor <- rep(c("TEE", "doppler", "ET.nitrogen", "oximetry"), each =
    3) #
> monitor <- factor(monitor, levels = c("TEE", "doppler",
    "ET.nitrogen", "oximetry")) #
> monitor
 [1] TEE         TEE         TEE         doppler     doppler     doppler
 [7] ET.nitrogen ET.nitrogen ET.nitrogen oximetry    oximetry    oximetry
Levels: TEE doppler ET.nitrogen oximetry
```

이렇게 요인을 만든 뒤에 `as.numeric()` 함수를 쓰면 "요인"의 껍질을 벗길 수 있다.

```
> as.numeric(monitor)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4
> as.numeric(outcome)
[1] 1 2 1 1
```

이로써 알게 되는 눈에 보이지 않았던 내부값으로, 레벨의 순서에 따라 1, 2, 3, 4, ...의 방식으로 요인 내부가 코딩 되어 있다는 점이다. 하지만 요인으로 셈을 할 수도 대소관계를 따질 수도 없다.

```
> outcome[1]
[1] Success
Levels: Success Failure
> outcome[2]
[1] Failure
Levels: Success Failure
> outcome[1] + outcome[2]
[1] NA
Warning message:
```

```
In Ops.factor(outcome[1], outcome[2]) : '+' not meaningful for factors
> outcome[1] < outcome[2]
[1] NA
Warning message:
In Ops.factor(outcome[1], outcome[2]) : '<' not meaningful for factors
> # Natural response to boolean operation
> 1 < 2
[1] TRUE
```

요인은 요인일 뿐으로 이름만 가치를 가지는 명목척도이다. 다만 요인형 자료에도 "순서"의 의미는 부여할 수 있다. 이른바 "순서척도(ordinal scale)"이다. 순서척도를 가진 요인을 생성할 때는 `factor()` 함수의 옵션으로 `ordered = TRUE`를 지정한다.

```
> m <- rep(c("unhappy", "neutral", "happy", "very happy"), 2)
> happy_score <- factor(m, levels = c("unhappy", "neutral", "happy",
    "very happy"), ordered = TRUE) #
> happy_score
[1] unhappy  neutral  happy    very happy unhappy neutral  happy
[8] very happy
Levels: unhappy < neutral < happy < very happy
> happy_score[1] < happy_score[2]
[1] TRUE
```

임상에서 측정되는 수많은 척도들, 통증과 만족도를 평가하는 NRS, VRS 등이 모두 순서척도이다[7].

숫자 벡터 x에 1부터 360까지의 숫자를 넣고 (그것이 각도라고 생각하면서) 360개 각도에 대한 사인함수값을 계산할 때 R 사용자는 `sin()` 함수를 360번 시행할 필요가 없다. R이 제공하는 여러 함수들이 벡터를 입력 받아 벡터로 결과를 내기 때문이다. 지면에 720개의 숫자 공해를 만들 수는 없으므로 일단 45도 간격으로 길이 8인 벡터를 만든 후에 360개 숫자는 그래프를 그려서 시연했다. `sin()` 함수가 요구하는 인수는 각도(degree)가 아니라

[7] 오랜 논쟁을 겪은, 통증을 평가하는 지표 중 VAS는, 제대로 측정되었다면 간격척도로 처리할 수 있다.

각도의 라디안(radian)이므로 180으로 나눈 뒤 pi를 곱했다[8].

```
> (x <- seq(1, 360, 45))
[1]   1  46  91 136 181 226 271 316
> (y <- sin(x / 180 * pi))
[1]  0.01745241  0.71933980  0.99984770  0.69465837 -0.01745241 -0.71933980
[7] -0.99984770 -0.69465837
> x <- 1:360
> y <- sin(x / 180 * pi)
> plot(x, y, type = "l")
```

결과 그래프는 각자의 컴퓨터에서 확인할 수 있다.

2.1 복잡한 자료형

벡터와 벡터는 횡(row)과 종(column)으로 결합할 수 있다. 횡으로 붙이려면 rbind() 함수를 쓰고 종으로 붙이려면 cbind() 함수를 쓴다. 이렇게 결합된 벡터들은 매트릭스(matrix)를 이룬다.

```
> v1 <- 1:5
> v2 <- 11:15
> v3 <- 11:15/3
> v1
[1] 1 2 3 4 5
> v2
[1] 11 12 13 14 15
> v3
[1] 3.666667 4.000000 4.333333 4.666667 5.000000
> cbind(v1, v2, v3)
     v1 v2       v3
[1,]  1 11 3.666667
[2,]  2 12 4.000000
[3,]  3 13 4.333333
```

[8] pi는 R 내장상수(built-in contstant)지만 실체는 "변수"이므로 사용자의 조작으로 인해서 값이 바뀔 수 있다. 내장값을 되찾는 방법이 없는 것은 아니지만 혼란을 초래할 것이므로 pi와 엇비슷한 이름을 가진 변수는 만들지 않는 게 상책이다. pi 외의 R 내장상수로는 LETTERS, letters, month.abb, month.name가 있다.

```
[4,] 4 14 4.666667
[5,] 5 15 5.000000
> (m <- rbind(v1, v2, v3))
       [,1] [,2]     [,3]     [,4] [,5]
v1  1.000000   2  3.000000 4.000000   5
v2 11.000000 12 13.000000 14.000000 15
v3  3.666667   4  4.333333 4.666667   5
> class(m)
[1] "matrix"
```

횡으로 결합된 매트릭스가 낯선 방식으로 출력되었다. 이름이 어떻게 붙어 있든 매트릭스에서 열 단위로 하나의 변수를 이루는 것으로 상정하기 때문이다. 매트릭스의 열에도 이름을 붙일 수 있다.

```
> colnames(m) <- c("c1", "c2", "c3", "c4", "c5")
> m
         c1 c2       c3       c4 c5
v1  1.000000 2 3.000000 4.000000  5
v2 11.000000 12 13.000000 14.000000 15
v3  3.666667 4 4.333333 4.666667  5
```

매트릭스 자료에 직접 시행할 수 있는 함수로 `apply()`가 있다. 매트릭스의 횡과 열 단위로 특정 함수를 연산하게 하는 것으로 활용도가 높다.

```
> apply(m, 1, sum)
     v1      v2      v3
15.00000 65.00000 21.66667
> apply(m, 2, function(x) mean(x))
     c1      c2      c3      c4      c5
5.222222 6.000000 6.777778 7.555556 8.333333
```

두 번째 인수에 1을 지시하면 행 단위의 연산, 2를 지시하면 열 단위의 연산이 실행될 것이다. 행 단위의 연산 결과는 행의 개수(m에서 3)만큼, 열 단위의 연산 결과는 열의 개수(m에서 5)만큼 나온다.

벡터를 붙이지 않고 원소 값을 나열하는 방식으로도 매트릭스를 만들 수 있다. 나열할 때 디폴트 순서는 1열, 2열, 3열의 값을 차례로 (한 벡터 안에)

나열하고 옵션으로 열의 수(ncol =)를 지정한다. 이걸 뒤집어서 행 단위로 입력하고자 하면 byrow = TRUE로 설정하면 된다.

```
> matrix(c(5, 4, 2, 3, 8, 1, 9, 6, 6), ncol = 3)
     [,1] [,2] [,3]
[1,]    5    3    9
[2,]    4    8    6
[3,]    2    1    6
> (mm <- matrix(c(5, 4, 2, 3, 8, 1, 9, 6, 6), ncol = 3, byrow = TRUE)) #
     [,1] [,2] [,3]
[1,]    5    4    2
[2,]    3    8    1
[3,]    9    6    6
```

벡터에서 특정 위치의 값을 추출하기 위해서 브래킷 연산자(일명 extractor)를 썼던 것을 확장해서 매트릭스의 원소 값을 추출할 수 있다. 브래킷 내부에 [row no, col no] 형식으로 추출한다.

```
> mm[3, 1]
[1] 9
```

매트릭스가 2차원 벡터라면 매트릭스와 매트릭스를 묶어서 3차원 벡터로도 만들 수 있다. 배열이라고 부르며 array() 함수가 담당한다.

```
> m1 <- matrix(1:4, ncol = 2)
> m2 <- matrix(9:6, ncol = 2)
> m3 <- matrix(c(52, 5, 28, 99), ncol = 2)
> (ar <- array(c(m1, m2, m3), dim = c(2, 2, 3)))
, , 1

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

, , 2

     [,1] [,2]
```

```
[1,]    9    7
[2,]    8    6

, , 3

     [,1] [,2]
[1,]   52   28
[2,]    5   99
```

4차원을 넘어서 다차원 벡터를 생성하는 것도 어렵지 않지만 실세계에 대응할 만한 예를 찾기 어려워서 예시를 생략했다. 요소 추출 때도 브래킷 연산자를 써서 `[row no, col no, outer no]`로 인덱싱 할 수 있다. `ar[1, 2, 3]`의 값이 무엇일지 생각해 보라.

행렬에서 행과 열의 이름의 기본값은 일련번호이다. 여기에 의미 있는 이름을 붙일 수 있다. 행 이름은 `rownames()`로, 열 이름은 `colnames()`를 사용하면 된다.

```
> (m <- matrix(c(41, 590, 232, 6105), ncol = 2))
     [,1] [,2]
[1,]   41  232
[2,]  590 6105
> colnames(m) <- c("Death", "Alive"); rownames(m) <- c("Steroid Use",
    "No")
> m
           Death Alive
Steroid Use   41   232
No           590  6105
```

`dimnames()`로 두 가지를 한꺼번에 처리할 수 있다. 다만 인수를 리스트 형식으로 전달하여야 한다.

```
> dimnames(m) <- list(c("Death", "Alive"), c("Steroid Use", "No"))
```

2.2 더욱 복잡한 자료형

벡터와 매트릭스, 배열은 모두 동일한 유형의 객체를 요소로 가진다. 이종 객체를 하나로 뭉칠 수 있는 자료형은 리스트(list)와 데이터프레임(data.frame)이다[9]. 리스트 객체는 실물 장부와 가장 비슷하다.

```
> myGift <- list(name = "Lego Construction kit", price = 75000, unit =
    "KRW", on.hand = TRUE) #
> print(myGift)
$name
[1] "Lego Construction kit"

$price
[1] 75000

$unit
[1] "KRW"

$on.hand
[1] TRUE
```

myGift 리스트 객체 속에는 문자, 숫자, 논리값이 혼재해 있고 서로를 강제로 변환하지 않았다. 잊었을까 염려에 다시 확인하자면 이 리스트에 속한 값들을 벡터 통합 함수 c()에 집어 넣으면 모조리 문자로 강제변환됐었다.

```
c(name = "Lego Construction kit", price = 75000, unit = "KRW", on.hand
    = TRUE) #
                name                price                 unit
"Lego Construction kit"          "75000"                "KRW"
             on.hand
              "TRUE"
```

[9]벡터와 리스트를 대비하는 표현이다. 실제로는 리스트도 기술적으로는 벡터이다. 잊어도 그만인 이론이다.

2.3 조건문

프로그래밍 언어가 지시하는 일련의 명령들을 조건에 따라 분기하는 것이 조건문의 기능이다. "`if (condition) expr`"이 기본 형태이며 조건(condition)이 참이면 구문(expr)이 실행되고 거짓이면 실행되지 않는다. if 문은 else 문과 짝을 맺을 수도 있어서 else 이후의 구문은 조건이 거짓일 때만 실행된다. 실행할 구문이 여러 줄로 구성되었다면 대괄호로 묶는다.

`if.. else` 구문은 스칼라, 즉 단일값에 반응해서 단일값을 결과로 내어주는 반면`ifelse()` 함수는 벡터 함수여서 벡터수신하면 벡터로 반응한다.

```
> Malory <- "boy"
> if (Malory == "boy")
+  print("Malory is a boy.") else print("Malory is not a boy.")
[1] "Malory is a boy."
```

`if`와 `else`는 프로그램의 흐름을 분기할 때 쓰며 `ifelse(`*`condition, expr1, expr2`*`)`는 조건문에 따라 서로 다른 벡터를 반환하는 함수이다. 참이면 expr1, 거짓이면 expr2를 되돌린다.

```
> ifelse(Malory == "boy", "Good boy.", "Good girl")
[1] "Good boy."
> Malory <- "girl"
> ifelse(Malory == "boy", "Good boy.", "Good girl")
[1] "Good girl"
> (Malory <- rep(c("boy", "girl"), each = 4))
[1] "boy" "boy" "boy" "boy" "girl" "girl" "girl" "girl"
> # ifelse() returns a vector when receiving a vector
> ifelse(Malory == "boy", "Good boy.", "Good girl")
[1] "Good boy." "Good boy." "Good boy." "Good boy." "Good girl" "Good
    girl"
[7] "Good girl" "Good girl"
```

`if` 문과 `goto` 문이 결합해서 프로그램의 흐름을 복잡하게 만드는 주범으로 알려져 있다. 다행히 R에는 `goto` 문, 또는 그와 유사한 점핑 동작을 하는 함수가 아예 존재하지 않는다.

`if` 문과 `ifelse()` 함수는 필요한 만큼 중첩해서 호출할 수 있다.

```
if (cond1) {
   expr1
  if (cond2) {expr2} else if (cond3) {expr3} else {expr4}
} else {expr5}
```

cond1 조건이 맞으면 expr1이 실행된 뒤에 다시 cond2가 참이면 expr2가 실행되고 거짓이면 cond3에 따라 expr3가 실행되거나 cond3가 거짓이면 expr4가 실행된다. 처음 조건인 cond1이 거짓이면 expr5가 실행된다.

2.4 반복 구문

`if` 문과 `goto` 문으로 프로그램의 특정 지점으로 갈 수 없지만 R은 구조화된 반복 구문을 만들 수 있다. 사람은 하지 못하는 일; 루프, 즉 반복 구문은 컴퓨터 프로그래밍의 꽃이라고도 말할 수 있다. `for()` 구문과 `while()` 구문과 `repeat()` 구문이 대표적이다. `for()` 구문이 직관적으로 쉽다.

```
for (indexing variable in range) {
   expr1
}
```

색인변수를 정하고(i, j, k, ...를 흔히 쓰지만 상관 없다) 반복할 구간(1–100?)만 정하면 된다. 1부터 100까지 정수의 합을 계산하는 `for()` 구문이다.

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

`while()` 구문에서는 condition이 참인 이상 expr1을 무한히 수행한다. 프로그래밍 초년병들을 무한루프에 빠지게 하는 흔한 상황이 주로 `while()` 구문에서 발생한다. 따라서 탈출할 수 있는 조건이(즉 condition1을 깨는 행위가) expr1 안에서 이루어져야 한다.

```
while (condition) {
```

```
    expr1
}
```

1부터 100까지 정수의 합을 계산하는 `while()` 구문이다. 탈출할 수 있는 조건으로 인덱서 i를 1씩 더해 나갔고 100이 넘는 순간 루프에서 빠져 나온다.

```
> s <- 0
> i <- 1
> while (i <= 100) {
+  s <- s + i
+  i <- i + 1
+ }
> print(s)
[1] 5050
```

`repeat` 구문은 처음 루프로 들어갈 때 점검하는 조건문이 없으므로 탈출조건이 따로 만들어지지 않는다면 무한히 작업을 반복한다. 탈출조건이 맞으면 `break`로 반복문에서 빠져나온다.

```
repeart {
    expr1
   if (escape condition) break
}
```

실제 사용은 다음처럼 한다.

```
> s <- 0
> i <- 0
> repeat {
+  i <- i + 1
+  s <- s + i
+  if (i == 100) break
+ }
> print(s)
[1] 5050
```

이상으로 예시한 세 가지 반복구문들의 각각의 특성이 눈에 보인다. `for()`

구문은 반복할 횟수와 인덱서의 값이 반복문 최초에 미리 정해진다. `while()` 구문은 최초에, 그리고 매 반복의 시작부에서 반복 여부를 결정하기 때문에 가변적인 반복문을 만드는 데 좋지만 탈출조건을 만들어 두지 않으면 무한 루프에 빠진다. `repeat` 구문은 조건 없이 시작해서 반복 도중에도 점검하는 지속 조건이 없다. 그저 일정한 탈출 조건만 주어질 뿐으로서 `if()`와 `goto()`의 결합과 비슷하다. 일반적인 용도라면 `for()` 구문과 `while()` 구문만으로 원하는 바를 성취할 수 있다.

관점을 바꿔 보자. 앞 절에서는 R의 독특한 무기는 벡터 연산이라고 쓴 적이 있다. 1부터 100까지 정수의 합을 구하는 데 굳이 반복 구문이 필요했을까? 정답부터 말하자면 아니다. R은 구성요소의 합을 구하는 `sum()` 함수를 이미 탑재하고 있다. 즉

```
> sum(1:100)
[1] 5050
```

만으로 동일한 합을 얻을 수 있다. 이전의 루프들과 시간 차이를 비교하려면 1부터 100까지의 합을 구하는 식으로는 얻을 수 없다. 차이를 벌리기 힘든 (매우 간단한) 문제이므로 1부터 1억까지의 정수의 합을 구한다고 해야 루프와 벡터 연산 사이에 비로소 차이가 벌어진다. `system.time()`은 주어진 코드가 실행되는 데 소요되는 시간을 측정해 주는 R의 기본 함수이다.

```
> end.value <- 100000000
> system.time ({ # performance of for-loop
+   s <- 0
+   for (i in 1:end.value)
+     s <- s + i
+   print(s)
+ })
[1] 5e+15
   user  system elapsed
  3.655   0.021   3.672
> system.time ({ # performance of while-loop
+   s <- 0
+   i <- 1
+   while (i <= end.value) {
```

```
+    s <- s + i
+    i <- i + 1
+  }
+  print(s)
+ })
[1] 5e+15
   user system elapsed
  8.153  0.036  8.155
> system.time ({ # performance of repeat-loop
+  s <- 0
+  i <- 0
+  repeat {
+    i <- i + 1
+    s <- s + i
+    if (i == end.value) break
+  }
+  print(s)
+ })
[1] 5e+15
   user system elapsed
  8.295  0.023  8.311
> system.time({ # performance of vector sum()
+  sum(1:end.value)
+  print(s)
+ })
[1] 5e+15
   user system elapsed
      0      0      0
```

1부터 1억까지 정수의 합을 구하는 연산을 비교하였더니 맨 마지막에 나온 벡터 연산이 소수점 두 자리에서 측정불가(0.00초)로서 가장 빨랐고 `for()` 구문이 2등, 뒤를 `while()`과 `repeat()`가 뒤따랐다. 벡터 연산을 할 것인가 루프에 넣을 것인가? 또는 질문을 바꿔서 반복 구문이 편한가? 벡터 연산이 편한가? 언제나 맞는 정답은 없다. `sum()` 함수를 지금은 알았지만 조금 전에는 몰랐으며 나중에 기억해 낸다는 보장도 없다. 게다가 무수히 많은

벡터 연산 함수를 다 기억해서 쓸 수도 없는 노릇이다. 주어진 반복 작업을 구현하고자 할 때 반복 구문이 언제나 쉽게 머리에 떠오르는 반면, 모든 함수 이름을 외울 수는 없다. 어떤 질문이건간에 루프 안에 넣어서 답을 얻을 수 있다. 시간이 오래 걸린다고 하지만 우리가 만드는 코드가 달착륙선을 쏘아 올리는 것도 아니며 성운의 전자파 미세변동을 추적하는 것도 아니다. 비효율적인 코드는 대개 알아보기 쉬운 코드라는 장점을 수반한다. R은 엄청나게 빠른 프로그래밍 언어가 아니지만 R이 지향하는 업무의 특성을 고려할 때 충분히 빠르다. 도널드 누스의 격언을 음미하자:

"프로그래머들은 썩 중요하지도 않은 부분의 속도를 높이고자 머리를 짜내고 전전긍긍하면서 어마어마한 시간을 낭비하고 있다. 효율을 높이려는 노력들은 오히려 더 많은 디버깅과 유지 보수 노력을 부르는 역효과를 초래한다[10]."

3 R의 실행 모드

R는 일명 번역어(interpreted language)로 분류된다. 다음처럼 설명하는 것이 번역어의 정확한 정의는 아니지만 번역어의 현상을 이해하는 데에는 도움이 된다: "R는 사용자가 엔터키를 입력할 때마다 단계별로 키 입력을 단위 해석해서 실행하고 다음 명령을 기다린다." 즉, C++ 류의 대표적 언어들의 코드를 컴파일 하는 것처럼 코드 블록을 한꺼번에 실행 코드로 바꾸어 둘 수 없기 때문에 실행속도의 면에서는 한계를 안고 시작하는 셈이지만 번역어의 행 단위 실행 덕분에 코드를 이해가 쉽고 행 단위의 결과를 바로 얻을 수 있어서 디버깅 면에서 유리하다.

3.1 대화 모드(Interactive mode)

사용자는 R의 한 인스턴스(one instance)의 프롬프트에 명령문을 한 줄씩 입력하는 방식으로 R 프로그래밍을 구현할 수 있다.

```
> 1 + 2
```

[10] "Wickham, Hadley. Advanced R. p 349"로부터 중복 인용

```
[1] 3
> t.test(distance ~ Sex, data = nlme::Orthodont)

   Welch Two Sample t-test

data: distance by Sex
t = 4.5333, df = 102.33, p-value = 1.58e-05
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 1.305529 3.336517
sample estimates:
  mean in group Male mean in group Female
            24.96875             22.64773
```

많은 R 사용자는 전혀 프로그래밍을 하지 않고 R을 사용할 수 있는데 행 단위의 대화 모드에서도 R가 가진 거의 모든 기능을 발휘할 수 있기 때문이며, 특히 R를 쓰고 있는 많은 사용자들의 목표인, "통계분석"을 구현하는 데 손색이 없기 때문일 것이다. R는 통계분석을 위해서 개발되었으며, 여전히 R 사용자의 일부는 R의 통계분석 기능만 쓰면서 전혀 프로그래밍을 하지 않는다.

3.2 **배치 모드**(Batch mode)

R에서 실행해야 할 긴 코드를 가지고 있다면 대화 모드보다는 배치 모드로 실행하는 것이 더 편하다. 가령, 어린이들의 치아 성장 자료철과 연령별, 성별 성장을 개체-내, 개체-간에서 분석하는 코드를 "스크립트(script)"로 부르는 코드 파일에 담고 R 인스턴스를 열지 않은 채 실행하는 방식이다.

```
data(Orthodont, package = "nlme")
library(nlme)
model1 <- lme(distance ~ age, random = ~ 1 | Subject, method = "ML",
    data = Orthodont)
model2 <- update(model1, distance ~ age * Sex)
summary(model2)
anova(model1, model2)
```

위 스크립트가 orthodont.R이라는 이름으로 저장되어 있을 때 다음 명령으로 R의 배치 모드를 가동할 수 있다.

```
$ R CMD BATCH orthodont.R
```

달러 표시($)는 시스템 터미널 프롬프트이다. 아무 일도 벌어지지 않은 것처럼 프롬프트가 복귀했다. 결과물은 orthodont.Rout 파일에 들어 있으며 다음의 터미널 명령로 출력할 수 있다.

```
$ cat orthodont.Rout
> data(Orthodont, package = "nlme")
> library(nlme)
> model1 <- lme(distance ~ age, random = ~ 1 | Subject, method = "ML",
    data = Orthodont)
> model2 <- update(model1, distance ~ age * Sex)
> summary(model2)
Linear mixed-effects model fit by maximum likelihood
 Data: Orthodont
      AIC     BIC   logLik
  440.6391 456.7318 -214.3195

Random effects:
 Formula: ~1 | Subject
       (Intercept) Residual
StdDev:   1.740851 1.369159

Fixed effects: distance ~ age + Sex + age:Sex
                Value Std.Error DF t-value p-value
(Intercept) 16.340625 0.9814310 79 16.649795 0.0000
age          0.784375 0.0779963 79 10.056564 0.0000
SexFemale    1.032102 1.5376069 25 0.671239 0.5082
age:SexFemale -0.304830 0.1221968 79 -2.494580 0.0147
 Correlation:
            (Intr) age   SexFml
age         -0.874
SexFemale   -0.638 0.558
age:SexFemale 0.558 -0.638 -0.874
```

```
Standardized Within-Group Residuals:
       Min         Q1        Med         Q3        Max
-3.64686407 -0.46341443 0.01556892 0.52172245 3.73335102

Number of Observations: 108
Number of Groups: 27
> anova(model1, model2)
      Model df    AIC     BIC   logLik  Test L.Ratio p-value
model1    1  4 451.3895 462.1181 -221.6948
model2    2  6 440.6391 456.7318 -214.3195 1 vs 2 14.75048 6e-04
>
> proc.time()
   user  system elapsed
  0.270   0.053   0.312
```

내용은 각자 음미하기로 하되, 대화 모드에서 실행하더라도 결과가 같았음을 아는 것은 어렵지 않다.