0508 R 시작 + 2 장 데이터타입

1. R 설치
대상폴더 신경쓰기!

2. www.java.com/kr - jdk 설치
(몇몇 패키지 자바와 연동)
대상폴더 변경

3. R studio 설치

※설치시 위치가 C:₩Program Files(x86) 이면 나중에 연동 문제 생김!
대상폴더 항상 확인하고 C:₩Program Files 로 바꿔주기

패키지 설치하기
install.packages("KoNLP")

한국어 자연어 사전 패키지 (자바 기반 패키지)

설치시 Other Mirrors -> Korea(Seoul) 선택

패키지 로딩하기
library(KoNLP)

오류로 로딩이 안 되면!

내컴퓨터-속성-고급시스템설정-환경변수-시스템변수 중 Path 편집
Home 키 누르고 ; 앞부분만
C:₩Program Files₩Java₩jre1.8.0_**171_;** 로 추가
버전에 따라 다름! 설치된 위치 찾아서 따라하기

0509 R 책 ~66p

작업디렉토리 지정 및 확인

setwd("C:₩₩r_temp")
getwd() # 현재 설정된 작업 디렉터리를 확인
(₩이 R 에서 특수문자이기 때문에 두 번 써야 함)

또는
setwd("C:/r_temp")
getwd()
->나중에 작업디렉토리에 있는 파일명만 가져올 수 있음! 위치 지정 안 하고

+
R 프로그램에서 속성 확인 -> 시작위치가 작업디렉토리

2 장 데이터타입

01 변수

#모든 변수 확인

objects() / 숨김변수도 확인 objects(all.names=T)

#변수 제거

rm() / 모든 변수 제거 rm(list = ls())

변수 이름 규칙

- 첫 글자는 알파벳 또는 .으로 시작
- 알파벳, 숫자, _(언더스코어), .(마침표)로 구성
- .으로 시작한다면 숫자가 올 수 없음 + 숨김변수 설정

올바른 변수명	올바르지 않은 변수명
a b a1 a2 .x	2a .2 a-b

변숫값 할당

- <-, <<-, = 사용 가능
- <- 주로 사용
- =는 명령의 최상위 수준에서만 사용할 수 있어 표현력에 한계가 있음

변수 사용 예제

```
var <- "aaa"          # 문자 - " " 사용
var2 <- 111
var3 <- Sys.Date()        # R : 대소문자 구분함 ! in 패키지, 함수
var4 <- c("a", "b", "c")
```

02 함수 호출 시 인자 지정

ex. 가상의 함수 foo

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

- 인자의 위치를 기억하면 순서대로 쓰기만 해도 가능

foo(3, 4)

foo(3, 4, 1)

foo(3, 4, 1, 2)

- 인자 이름을 사용하면 원래 프로토타입의 인자 순서와 달라도 같은 함수가 실행된다.

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

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

foo(3, 4, d=5)

03 스칼라

- 스칼라 : 단일 차원의 값

1) 숫자형과 주요 산술연산자

산술연산자	의미	예시
%%	**나머지 구하기**	5**%%**4 -> 1
^, **	**승수 구하기**	3^2 -> 9, 3**3 -> 27

> 100000 # 0 이 5 개부터는 e 로 표시됨

1**e**+05

> 1000000

1**e**+06

> 1**e2** # 1 * **10^2**

100

> 3**e3** # 3 * **10^3**

300

> 3**e-1** # -1 : 소수점 1 자리까지 표시하라

0.3

> 3**e-2** # -2 : 소수점 2 자리까지 표시하라

0.03

강제로 숫자형으로 변환하기(cf. 오라클 to_number)

> as.numeric('1') + as.numeric('2')

-----숫자형 변환시---------
1st. class 로 데이터 타입 확인하기
2nd. as.numeric()으로 변환시키기
----

2) NA & NULL 형
NA : 잘못된 값이 들어올 경우(Not Applicable, Not Available) / 결측값
NULL : 값이 없음, 변수가 초기화되지 않았을 때 사용함 / 자동으로 무시됨, 출력시 생략됨

- is.na() : NA 값이 저장되어 있는지 확인
-is.null : NULL 값이 저장되어 있는지 확인

※유의! NA, NULL 의 계산※

```
> cat(1, NA, 2)
1 NA 2

> cat(1, NULL, 2)
1 2

> sum(1, NA, 2)
[1] NA

> sum(1, NULL, 2)
[1] 3
```

3) 문자형
- 따옴표 ("", '')로 묶어주기

4) 날짜형

날짜 표시

- Sys.Date() # 날짜만 표시
- Sys.time() # 날짜 + 시간
- date() # 미국식 날짜, 시간 표시

문자를 날짜로 강제변환하기 **as.Date('문자로 표시한 날짜', format=' ')**

ex.

as.Date('2018-05-09')

as.Date('20110101','%Y%m%d')

as.Date('2011 년 01 월 01 일','%Y 년 %m 월 %d 일')

as.Date(**as.character**(20110101),'%Y%m%d')

- in R only 문자 -> 날짜 (cf. oracle 숫자 -> 날짜도 가능)
- 날짜 기본 포맷 yyyy-mm-dd

형식	의미		
%d	일자를 숫자로 인식	%H	시
%m	월을 숫자로 인식	%M	분
%b	월을 영어 약어로 인식	%S	초
%D	월을 전체 이름으로 인식		
%y	년도를 숫자 두 자리로 인식		
%Y	년도를 숫자 네 자리로 인식		

---

as.Date('20110101') (X) as.Date(20110101,'%Y%m%d') (X)

일, 월, 년 추가하기

```
date1 <- seq(from=as.Date('2014-01-01'),
             to=as.Date('2014-01-31'),by=1) # 1 일씩 추가 (기본 단위 : 일)
date1 <- seq(from=as.Date('2014-01-01'),
             to=as.Date('2014-05-31'),by='month') #한 달씩 추가
date1 <- seq(from=as.Date('2014-01-01'),
             to=as.Date('2020-05-31'),by='year') # 1 년씩 추가
```

날짜 연산하기

```
> '2014-11-30' - '2014-11-01'   # 문자임 -> 오류
> class("2014-11-30")

[1] "character"
> as.Date("2014-11-30") - as.Date("2014-11-01")
> as.Date("2014-11-01") + 5
```

- 날짜 형태로 바꿨기 때문에 연산 가능 / 기본단위 : 일

lubridate 패키지로 날짜와 시간 제어하기 / /국제표준시간 for 월별, 일별 등 통계 사용

install.packages("lubridate")

library(lubridate) # default 에 설정할 수도 있음

date <- now() # 현재 날짜와 시간 넣기 (요기서 date 는 변수임! 함수는 반드시 괄호가 있는 채로 쓰기)

date

year(date) # 년도만 출력하기

month(date,label=T) # 월을 출력하되 영문이름으로 출력하기

month(date,label=F) # 월을 출력하되 숫자로 출력하기

day(date) # 일 출력하기

wday(date,label=T) # 요일을 출력하되 영문이름으로 출력하기

wday(date,label=F) # 요일을 출력하되 가중치 숫자로 출력하기 . 일요일부터 1 로 시작함.

※ 영문일 때만 !

month(date) <- 2 # 2 월로 설정하기

date - **days(2)** # 2 일 전 날짜 출력하기

date+**years(1)** # 1 년 추가하기

date+**months(1**) **# 1 개월 추가하기** cf. orcale / months_between

date+**hours(1)** # 1 시간 추가하기

date+**minutes(1)** # 1 분 추가하기

date+**seconds(1)** #1 초 추가하기

date <- hm("22:30") ; date # 시간 분 지정하기 / 엔터 안 쓰고 연달아서 수행할 때 ; 사용

date <- hms("22:30:15") ; date # 시간 분 초 지정하기

5) 진릿값

- TRUE, FALSE : 예약어 (변수명으로 사용 불가)
- &(AND), |(OR), !(NOT) 연산자 사용

```
> TRUE & TRUE
[1] TRUE
> TRUE & FALSE
[1] FALSE
> TRUE | TRUE
[1] TRUE
> TRUE | FALSE
[1] TRUE
> !TRUE
[1] FALSE
> !FALSE
[1] TRUE
```

+ 숫자 ~ TRUE / FALSE

```
> 3&0      # 3 * 0
[1] FALSE
> 3&2      # 3 * 2
[1] TRUE
> 3|0
[1] TRUE
> !0
[1] TRUE
> !3
[1] FALSE
```

in 숫자

- 0 : FALSE 값으로 인식
- 0 외의 숫자 : TRUE 값
- if 문에서 0 유의

6) 팩터

- 팩터 : 범주형 데이터(자료)를 표현하기 위한 데이터 타입
- 범주형 데이터 : 명목형 데이터 (크기비교 X)
  순서형 데이터 (크기비교 O) / cf. 수치형 데이터 - 값 측정 가능

팩터 관련 함수

factor(x, levels, ordered) - 팩터값을 생성한다.

x : 표현하고자 하는 값(주로 문자열 벡터
levels : 값의 레벨 - 범위
ordered : TRUE : 순서형, FALSE(기본값) : 명목형

- **nlevels()** : 팩터에서 레벨의 개수를 반환한다.
- **levels()** : 팩터에서 레벨의 목록을 반환한다.
- **is.factor()** : 주어진 값이 팩터인지 판단한다.
- **ordered** : 순서형 팩터를 생성한다.
- **is.ordered** : 순서형 팩터인지를 판단한다.

```
> sex <- factor("m", c("m", "f"))
> sex
[1] m
Levels: m f          # levels 같이 표시됨 - 순서 X -> 범주형

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

> levels(sex)[1]       # cf. python 색인 시작 : 0
[1] "m"
> levels(sex)[2]
[1] "f"

> levels(sex) <- c("male", "female") # 팩터변수에서 레벨 값을 직접 수정
> sex                                                  또는 변수명 동일하게 factor 로 덮어쓰기
[1] male
Levels: male female

> factor(c('m', 'm', 'f'), c('m','f'))
[1] m m f
Levels: m f
> factor(c('m','m','f'))  #levels 인자 생략시 자동으로 레벨의 목록 파악
[1] m m f
Levels: f m

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

04 벡터

- 벡터 : 배열의 개념 (like column)
- 한 가지 스칼라 데이터 타입의 데이터를 저장할 수 있음
- 벡터 각 셀에 이름 부여 가능, 이름 역시 색인 가능
- 하나의 벡터에 다른 형 데이터가 함께 저장 -> 표현력이 높은 데이터 타입으로 자동 형 변환
- 벡터는 중첩할 수 없다 (cf. 리스트 : 중첩 가능)

벡터 생성

- c( ) : 벡터생성
- names( ) : 객체 이름 반환
- names <- : 객체에 이름 저장

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

※ 괄호로 코드를 묶으면 괄호 안의 문장을 수행 + 결과 값을 화면에 출력함

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

```
# 벡터의 각 셀에 names<-() 함수를 사용해 이름을 부여
> x <- c(1, 3, 4)
> names(x) <- c("kim", "seo", "park")
> x
 kim  seo park
   1    3    4
```

벡터 데이터 접근

- 색인, 이름을 사용하여 인덱스

문법	의미	예시
X[n]	벡터 x 의 n 번째 요소 N 은 숫자 또는 셀(행/열)의 이름을 뜻하는 문자열	X[3] X["a"]
X[-n]	벡터 x 에서 n 번째 요소를 제외한 나머지.	X[-3]
X[idx_vector] 슬라이스색인	x 로부터 idx_vector 에 지정된 요소를 얻어옴. Idx_vector : 색인을 표현하는 숫자 벡터 또는 셀의 이름을 표현하는 문자열 벡터	X[c(1,2)] X[c(4,1)]
X[start:end]	벡터 x 의 start 부터 end 까지의 값을 반환함. Start 와 end 의 위치 값을 모두 포함함	X[1:3]

cf. python

- -색인 없음

- 1 : 3 / 2nd ~3rd 값 (시작색인값:0, end 이전까지만 표시)

숫자, 날짜 연속적 값 대입 가능 / 문자는 불가능

```
seq1 <- 1:5
date1 <- seq(from=as.Date('2014-01-01'),
             to=as.Date('2014-01-31'),by=1)    #연속적인 일 추가하기, 기본단위 일
```

cf. in ORACLE, to_date()

벡터의 길이

- length() : 객체의 길이를 반환한다. / 객체, 팩터, 배열, 리스트를 지정
- **NROW()** : 인자가 벡터인 경우 벡터를 **n 행 1 열의 행렬로 취급**해 행의 수를 반환함 / 벡터, 배열, 또는 데이터프레임

cf. nrow() : 행렬과 데이터프레임의 행의 수를 리턴

-> NROW()가 가장 무난하게 어디든 쓸 수 있다. (대문자!)

주의

```
vec1 <- 1:6
vec1[9] <- 9        # 벡터의 크기보다 큰 벡터를 수정할 때 NULL 이 아닌 NA 가 채워짐
```

1 2 3 4 5 6 NA NA 9

append() - 추가하기

```
append(vec1,10,after=3) # 3rd 위치 다음에 10 을 넣으라는 의미
```

[1] 1 2 3 10 4 5 6 NA NA 9

```
append(vec1,c(10,11),after=3) # 3rd 위치 다음에 c(10,11)을 넣으라는 의미
```

[1] 1 2 3 10 11 4 5 6 NA NA 9

반복문을 통해 벡터에 새로운 값을 계속 추가할 때 사용하는 함수

(맨 마지막 위치에 계속 추가 - length() / NROW())

벡터의 연산

두 벡터의 길이가 다를 경우 순환 원리가 적용됨

배열의 개수가 맞지 않을 경우 산술 연산은 개수가 작은 쪽 배열 값이 차례대로 반복되어 연산

벡터 연산 함수

- identical() : 객체가 동일한지 판단한다.
- union() : 합집합을 구한다 / 중복된 값은 제거된 상태로 출력
- intersect() : 교집합을 구한다.
- setdiff() : 차집합을 구한다. / 순서 중요
- setequal() : x 와 y 가 같은 집합인지 판단한다.

※ identical() : 요소 값 + 요소 개수 일치

setequal() : 요소 값 일치

벡터 연산자

- value **%in%** x : 벡터 x 에 value 가 저장되어있는지 판단함 **(not 패턴!)**
- x + n : 벡터 x 의 모든 요소에 n 을 더한 벡터를 구함 (*,/,-,== 연산 가능)

```
> identical(c(1,2,3),c(1,2,3))
[1] TRUE
> identical(c(1,2,3),c(1,2,100))
[1] FALSE
> setequal(c("a","b","c"),c("a","d"))
[1] FALSE
> setequal(c("a","b","c"),c("a","b","c","c"))
[1] TRUE
```

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

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

※ if 문 등의 조건문에서는 단 하나의 참 또는 거짓 값을 사용하기 때문에 identical()을 사용함

==, !=를 사용하지 않음!

연속된 숫자로 구성된 벡터

연속된 숫자로 구성된 벡터 관련 함수

seq(from, to, by)

- 시퀀스를 생성한다.

#시작값, 끝값, 증가치

 # 숫자, 날짜만 사용 가능 / 문자는 사용 불가능!

ex. seq2 <- "a" : "f" **(X)**

seq_along(x)

- seq_along : 주어진 객체의 길이만큼 시퀀스를 생성한다

반환값은 x 의 길이가 N 일 때, 1 부터 N 까지의 숫자를 저장한 벡터

시퀀스 생성 문법

from : end from 부터 end 까지의 숫자를 저장한 벡터를 반환함

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

반복된 값을 저장한 벡터

rep(x, times, each)

- rep : 주어진 값을 반복한다

x : 반복할 값이 저장된 벡터
times : 전체 벡터의 반복 횟수 (집단)
each : 개별 값의 반복 횟수 (원소)

tip : 데이터 프레임에서 색인~rep()를 사용하여 그룹핑함

```
> 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 2 2 2 2
```

0510

05 리스트

- 데이터를 중간에 삽입하는 데 유리한 구조
- **(key, value)** 형태의 데이터를 담는 연관 배열
- 벡터와 달리 값이 서로 다른 데이터 타입을 담을 수 있음 + 중첩가능
- cf. 파이썬 - 딕셔너리

(1) 리스트 생성 + 데이터 접근

list(key1 = value1, key2 = value2, ...)

- 반환값은 key1 에 value1, key2 에 value2 등을 저장한 리스트다.

x[n] : 리스트 x 에서 n 번째 데이터의 서브리스트 / key 값
x[[n]] : 리스트 x 에서 nth 키에 번째 저장된 값(value) / value 값
x$key : 리스트 x 에서 키 값 key 에 해당하는 값 / value 값

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

$height
[1] 1 3 5

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

> x[[1]]
[1] "foo"
> x[[2]]
[1] 1 3 5
> x$name
[1] "foo"

> x$height[2] # height 의 두 번째 값만 빼고 싶을 때 - 이 표현 더 권장
[1] 3
> x[[2]][2]
[1] 3
------------------------------------------------------------
y <- list (a=list(val=c(1,2,3)), b=list(val=c(1,2,3,4))) # 리스트 중첩
$`a`
$`a`$`val`
[1] 1 2 3

> y[1]
$`a`
$`a`$`val`
[1] 1 2 3

> y$b$val[3] # b 키의 val 키의 세 번째 값만 빼고 싶을 때 - 이 표현 더 권장
[1] 3
> y[[2]][[1]][3]
[1] 3
```

(2) List 에 새로운 요소 추가/ 삭제하기

추가하기

x$birth <- '1975-10-23' ##기존에 중복되지 않는 이름으로 새로운 키값 추가 // 맨 끝에 추가됨

cf.이름이 없는 list 형태에 새로운 요소 추가

```
> list2<-list(c(1,2,3),c("a","b","c"))
> list2
[[1]]
[1] 1 2 3

[[2]]
[1] "a" "b" "c"

> list2[[3]] <- c(4,5,6)
> list2
[[1]]
[1] 1 2 3

[[2]]
[1] "a" "b" "c"

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

cf. 존재하는 key 에 추가로 value 붙이기

```
> list1<-list(col1=c(1,2,3),col2=c("a","b","c"))
> list1
$`col1`
[1] 1 2 3

$col2
[1] "a" "b" "c"

> list1$col1[4] <- 3
> list1
$`col1`
[1] 1 2 3 3

$col2
[1] "a" "b" "c"
```

/ 아니면 append 함수 사용

cf. list1$col1 <- 3 **(X)** #기존 키 전체에 값을 덮어쓰는 것

키를 삭제하기

```
> x$birth <- NULL
```

NA 랑 다름! 출력됨 / NULL 은 출력되지 않음

cf. 특정 값을 삭제? 불가능..!

06 행렬

- 행, 열의 수가 지정된 2 차원 구조
- 동일한 데이터 타입만 허용 - 한 가지 유형의 스칼라만 저장할 수 있음

(1) 행렬 생성

matrix(data, nrow, ncol, byrow=FALSE, dimnames=NULL)

nrow / ncol 중 하나만 !

byrow= TRUE 가 되면 행 우선으로 데이터를 채운다. default 는 FALSE 값, 열 우선

dimnames = NULL : 행+열에 이름 붙이기. list() 사용

관련 함수

dimnames (x) 객체의 각 차원에 대한 이름을 가져온다

dimnames(x) <- 객체의 차원에 이름을 설정한다 - 반드시 **list()** 형태로! / list 는 행-열 순서로

rownames(x) 행렬의 행 이름을 가져온다

rownames(x) <- 행렬의 행 이름을 설정한다

colnames(x) 행렬의 열 이름을 가져온다

colnames(x) <- 행렬의 열 이름을 설정한다

```
> a<-1:9
> matrix(a, nrow=3)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> matrix(a, nrow=3, byrow=TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

```
> x<-matrix(1:9,ncol=3)
> 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
```

```
> y<-matrix(1:9,nrow=3)
> rownames(y)
NULL                # 행 이름이 없으므로 NULL 로 반환됨
```

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

(2) 행렬 데이터 접근

A[ridx, cidx]

행렬 A 의 ridx 행, cidx 열에 저장된 값. 각각은 벡터로 사용할 수도 있고 생략할 수도 있다.
행/열 이름이 있는 경우 이름을 사용할 수도 있다.

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

> x[3,1]    #3 행 1 열
[1] 3

> x[-3, ]    # 특정 행/열 제외 색인 가능
   c1 c2 c3
r1  1  4  7
r2  2  5  8

> x[1:2, ]  #벡터 형태 색인 가능
   c1 c2 c3
r1  1  4  7
r2  2  5  8

> x[c(1,3),2:3]
   c2 c3
r1  4  7
r3  6  9

> x["r1",]  #행, 열에 이름을 부여했다면 이름 사용으로도 가능하다.
c1 c2 c3
 1  4  7
```

(3) 행렬 추가

rbind() / cbind() 사용하기

벡터 추가

```
> v1 <- c(1,2,3,4)
> v2 <- c(5,6,7,8)
> rbind(v1,v2)
   [,1] [,2] [,3] [,4]
v1    1    2    3    4
v2    5    6    7    8
> cbind(v1,v2)
     v1 v2
[1,]  1  5
[2,]  2  6
[3,]  3  7
[4,]  4  8
```

- **matrix 에 추가**

```
> mat4 <- matrix(1:9, nrow=3)
> mat4 <- rbind(mat4,c(1,2,3))
> mat4
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
[4,]    1    2    3
```

주의

+ 여러개의 벡터들도 동일한 벡터타입이어야 함
+ 길이(크기) 가 같아야 함

(4) 행렬 연산

연산자

A + x	행렬 A 의 모든 값에 스칼라 x 를 더한다. (-, *, / 도 사용 가능)
A + B	행렬 A 와 행렬 B 의 합을 구한다. 행렬 간 차는 - 연산자를 사용한다.
A **%*%** B	행렬 A 와 행렬 B 의 곱을 구한다 (행렬곱 inner product)

cf. 파이썬은 벡터연산 불가능

행렬 연산 함수

t(x)	행렬 또는 데이터 프레임의 전치행렬을 구한다
solve(a,b)	수식 A **%*%** x = B 에서 x 를 구한다. ---**역행렬 구할 때 사용(b 생략)**
nrow/ncol(x)	배열의 행/열의 수를 구한다.
dim(x)	배열의 차원 수를 구한다 - 벡터형식
dim(x) <-	객체의 차원 수를 지정한다 (행렬 모양 변경)

```
> x <- matrix(1:9, nrow=3)
> x
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> x %*% x
     [,1] [,2] [,3]
[1,]   30   66  102
[2,]   36   81  126
[3,]   42   96  150
```

```
> x<-matrix(1:4,nrow=2)
> x
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> t(x)                          #x 의 전치행렬
     [,1] [,2]
[1,]    1    2
[2,]    3    4
> solve(x)              #x 의 역행렬
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
> x %*% solve(x)        #역행렬 확인
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

```
> y<-matrix(1:6,nrow=2)
> dim(y)                #차원 출력
[1] 2 3

> y
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> dim(y) <- c(3,2)      #차원 변경
> y
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

07 배열

- 다차원 데이터
- 동일한 데이터 타입으로 구성

(1) 배열 생성

array(data, dim, dimnames=NULL)

data : 데이터를 저장한 벡터

dim : 배열의 차원. 이 값을 지정하지 않으면 1 차원 배열이 생성된다. c(행,열,층)

cf. 파이썬 -> 층,행,열

dimnames : 차원의 이름

```
> 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

> (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
```

(2) 배열 데이터 접근

```
> x[1, 1, 1]
[1] 1

> x[1, 2, 3]
[1] 11

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

> dim(x)   #행렬 차원이 늘어난 형태로 이해
[1] 2 2 3
```

08 데이터 프레임

- 표 형태로 정리한 모습
- 각 열은 서로 다른 데이터 타입을 가질 수 있다.
- 문자열 사용할 때 stringsAsFactors=FALSE 쓰는 편 습관 들이는게 편함!

(1) 데이터 프레임 생성

data.frame(data, stringsAsFactors=TRUE)

- **data** : value 또는 tag = value 형태로 표현된 데이터 값.
  여러 데이터 나열시 데이터 개수 동일해야함
- **stringsAsFactors** : TRUE(기본) ~ **문자형** 데이터를 순서형 데이터(factor)로 처리. **(for 수치화)**
  FALSE ~ 단순히 문자열로, 명목형 데이터

str( )

임의의 R 객체의 내부 구조를 보인다. column 의 개수, 데이터타입, factor 여부 확인 가능.

cf. in oracle, DESC 테이블명 비슷

---

cf. list 와 key - value 값 형태로 가진다는 건 비슷. 하지만 데이터의 정렬 형태가 다름

list : 쭉쭉 나열 / data.frame - 칼럼형태로 정리됨

+ 똑같이 **rownames(), colnames()** 사용 가능**. names()**를 사용해도 colnames()와 같은 결과

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

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


> d$w <- c("A","B","C","D","E") # 새로 추가할 때에는 factor 변수화 안 됨!
> d
  x  y z w
1 1  2 M A
2 2  4 F B
...
> str(d)
'data.frame':           5 obs. of  4 variables:
 $ x: int  1 2 3 4 5
 $ y: num  2 4 6 8 10
 $ z: Factor w/ 2 levels "F","M": 2 1 2 1 1
 $ w: chr  "A" "B" "C" "D" ...
```

(2) 데이터 프레임 접근

d$colname

d[m, n, drop=TRUE]

- 데이터 프레임 m 행 n 열에 저장된 데이터.

★차원축소★

- **drop = TRUE** 차원축소 됨(matrix 랑 비슷) - 기본값

  drop = FALSE 사용시 원래 데이터프레임 형식 그대로, 차원축소 되지 않고 나타남!

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

> d[,c("x")]            #차원축소 된 상태!
[1] 1 2 3 4 5
> d[,c("x"),drop=FALSE]          #차원축소 안 됨!
  x
1 1
2 2
3 3
4 4
5 5

> b<-data.frame(a=1:3,b=4:6,c=7:9)

#주어진 값이 벡터에 존재하는지 판별하는 %in% 연산자와
데이터 프레임의 컬럼 이름을 반환하는 names()를 이용하면 특정 컬럼 선택 작업을 쉽게 할 수 있다.

> b[,names(b) %in% c("b","c")]     # 잘 안 씀
  b c
1 4 7
2 5 8
3 6 9
> b[, !names(b) %in% c("a")] #!~-색인  형태와 같군
  b c
1 4 7
2 5 8
3 6 9
```

(3) 유틸리티 함수

head(x, n=6L)

tail(x, n=6L)

n - 반환할 결과 값의 크기

view(x,title)

title - 뷰어 윈도우의 제목

```
> d <- data.frame(x=1:10000)
> head(d)
  x
1 1
2 2
3 3
4 4
5 5
6 6
> tail(d)
        x
9995   9995
9996   9996
9997   9997
9998   9998
9999   9999
10000 10000
```

09 타입 판별

데이터 타입 판단 함수

class(x) 객체 x 의 클래스 (데이터 타입)

str(x) 객체 x 의 내부 구조

is.factor(x) 주어진 객체 x 가 팩터인가

is.numeric(x) 주어진 객체 x 가 숫자를 저장한 벡터인가

is.character(x) 주어진 객체 x 가 문자열을 저장한 벡터인가

is.matrix(x) 주어진 객체 x 가 행렬인가

is.array(x) 주어진 객체 x 가 배열인가

is.data.frame(x) 주어진 객체 x 가 데이터프레임인가

10 타입 변환

as.factor(x) 주어진 객체 x 를 팩터로 변환
as.numeric(x) 주어진 객체 x 를 숫자를 저장한 벡터로 변환
as.character(x) 주어진 객체 x 를 문자열을 저장한 벡터로 변환
as.matrix(x) 주어진 객체 x 를 행렬로 변환
as.array(x) 주어진 객체 x 를 배열로 변환
as.data.frame(x) 주어진 객체 x 를 데이터 프레임으로 변환

```
> 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"
```

```
> 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

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

```
> 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
```

as.factor 와 factor 의 차이점

as.factor -> 알파벳 순서로 자동 레벨 순서 지정

factor -> 레벨 자의적으로 순서 설정 가능