Vectors and iteration

박찬영

2024-09-01

이번엔 R 베이스를 보자

벡터

벡터는 원시벡터와 리스트로 구분된다. 원시벡터는 한 자료형에 대한 벡터이고, 리스트는 자료형에 구애받지 않는다.

c("Aasd","agd") 는 문자형 원시벡터 c(FALSE, TRUE) 는 논리형 원시벡터 c(1,3,4) 는 수치형 원시벡터이다. 수치형 원시벡터는 다시 double과 integer로 구분된다.

벡터가 비어있으면 NULL이다.

후에 알아보겠지만 R은 벡터가 아닌 데이터가 없다. 즉 스칼라도 1원소 벡터로 되는것이다. 예를 들어 charactor라는 자료형은 문자형 벡터라는 자료형이다.

```
letters #문자형  원시벡터
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

```
typeof(letters) #charactor라고  뜸  이는  문자형벡터라는  뜻
```

```
## [1] "character"
```

```
typeof("a") #하나짜리도  벡터임  사실
```

```
## [1] "character"
```

```
typeof(c("asdf",12,TRUE)) #원시벡터로  만들면  한번에  여러  자료형을  못가진다
```

```
## [1] "character"
```

```
a=c("asdf",12,TRUE)
#a[2]+1 #에러뜸  12를  문자열로  취급하기  때문
a[2]
```

```
## [1] "12"
```

```r
a[3] #얘도 문자열 취급
```

```
## [1] "TRUE"
```

```r
#벡터 자료형에 대한 함수로는 is.*()이 있다


is.numeric(c(1,23)) #TRUE
```

```
## [1] TRUE
```

```r
c(a=1,b=2,c=5) #벡터의 각 요소에 이름을 붙일 수 있음
```

```
## a b c
## 1 2 5
```

```r
names(c(a=1,b=2,v=5)) #이름을 반환
```

```
## [1] "a" "b" "v"
```

```r
set_names(c(1,1,3),c("a","b","v")) #이름을 붙이는 함수
```

```
## a b v
## 1 1 3
```

```r
#리스트를 알아보자
typeof(list("Asdf",1,TRUE)) #리스트는 이렇게 생성, 여러 자료형을 한번에 가진다
```

```
## [1] "list"
```

```r
#리스트와 벡터 둘다 자신의 형식을 부분집합으로 갖는다
#즉 벡터의 한 요소는 벡터이고
#리스트의 한 요소는 리스트이다
#벡터의 요소가 벡터인건 자명하다 R은 모두 벡터이기 때문


b <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))
b #리스트이다
```

```
## $a
## [1] 1 2 3
##
## $b
## [1] "a string"
##
## $c
## [1] 3.141593
##
```

```
## $d
## $d[[1]]
## [1] -1
##
## $d[[2]]
## [1] -5
```

```
b[1] #이것도 리스트로써의 객체이다, 벡터를 요소로 갖는 1원소 리스트이다
```

```
## $a
## [1] 1 2 3
```

```
b[2] #이것도 리스트 문자열을 요소로 갖는 1원소 리스트
```

```
## $b
## [1] "a string"
```

```
#그럼 b의 첫번째 요소인 1:3의 리스트가아닌 벡터자체에 접근하고 싶을경우
b[[1]] #이렇게 하면된다
```

```
## [1] 1 2 3
```

```
b[[1]][2] #이러면 첫번째 요소에 직접 들어가서 2번째 인덱싱
```

```
## [1] 2
```

```
#원시벡터의 복제
c(1,2) + c(1)
```

```
## [1] 2 3
```

```
#길이가 짧은 두 벡터의 연산은 길이를 긴쪽에 맞춰서 작은쪽이 복제된다
#연산은 각 원소끼리 대응시켜서 연산한다

c(1,2,3,4) > 2
```

```
## [1] FALSE FALSE  TRUE  TRUE
```

```
#2라는 것도 사실 1원소 벡터라서 4개로 복제되어
c(1,2,3,4) > c(2,2,2,2) #와 같다
```

```
## [1] FALSE FALSE  TRUE  TRUE
```

```
#결과는 당연히 벡터

#속성

#벡터에 메타 데이터를 속성을 이용해 넣을 수 있다
```

```r
x=1:10

attr(x,"greeting") #속성을 생성
```

```
## NULL
```

```r
attr(x,"greeting") = "Hi!" #속성값을 지정해줘야한다
x #Hi!
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
## attr(,"greeting")
## [1] "Hi!"
```

```r
attr(x,"farewell") = "Bye!"
x #Hi! Bye!
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
## attr(,"greeting")
## [1] "Hi!"
## attr(,"farewell")
## [1] "Bye!"
```

```r
attributes(x) #속성을 보는법
```

```
## $greeting
## [1] "Hi!"
##
## $farewell
## [1] "Bye!"
```

```r
#확장벡터
#팩터형, 데이트형, 데이트타임형, 티블형은 일반적인 벡터에 객체지향적 요소를 담은 객체이다
#확장된 벡터로 일반 원시벡터와 다르게 작동한다

x=factor(c("a","b","c"),levels=letters) #팩터
typeof(x) #팩터형은 정수형 기반이다
```

```
## [1] "integer"
```

```r
class(x) #팩터라는 것은 정수형 벡터기반 객체이다
```

```
## [1] "factor"
```

```r
x=as_date("1971/01/01") #데이트형 생성
typeof(x) #더블형
```

```
## [1] "double"
```

```
class(x) #데이트 객체
```

```
## [1] "Date"
```

```
unclass(x) #객체를 풀면 실제값은 1970-01-01로 부터 지난 일 수를 가짐
```

```
## [1] 365
```

```
attributes(x) #데이트라는 속성을 가짐
```

```
## $class
## [1] "Date"
```

```
x=ymd_hm("1970/01/01 01:00")
x
```

```
## [1] "1970-01-01 01:00:00 UTC"
```

```
typeof(x) #더블형
```

```
## [1] "double"
```

```
class(x) #POSIXct 객체
```

```
## [1] "POSIXct" "POSIXt"
```

```
unclass(x) #오케이?
```

```
## [1] 3600
## attr(,"tzone")
## [1] "UTC"
```

```
attributes(x) #객체속성과 tzone에 대한 속성이 있다
```

```
## $class
## [1] "POSIXct" "POSIXt"
##
## $tzone
## [1] "UTC"
```

```
x=tibble(x=1:5,y=5:1)
x
```

```
## # A tibble: 5 x 2
##       x     y
##   <int> <int>
## 1     1     5
```

```
## 2     2     4
## 3     3     3
## 4     4     2
## 5     5     1
```

```
typeof(x) #얘는 확장 리스트이다
```

```
## [1] "list"
```

```
class(x) #클래스는 3개 data.frame이 있음에 주목, data.frame 클래스를 그대로 상속받음
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
attributes(x) #클래스 열이름 행이름의 속성을 가짐
```

```
## $class
## [1] "tbl_df"     "tbl"        "data.frame"
##
## $row.names
## [1] 1 2 3 4 5
##
## $names
## [1] "x" "y"
```

반복문

코딩을 잘하려면 제어문을 잘써야한다. 반복문과 조건문이지

```
df = tibble(
    a=rnorm(10),
    b=rnorm(10),
    c=rnorm(10),
    d=rnorm(10)
)
#각열의 중앙값을 계산해보자
#median(df) 같은게 안된다

for(i in 1:length(df)){
    print(median(df[[i]]))
}
```

```
## [1] -0.03358715
## [1] -0.2257747
## [1] 0.2180721
## [1] -0.5076686
```

```r
#for문 사용 i가 1:length(df)의 요소를 순회한다
for(i in seq_along(df)){
    print(median(df[[i]]))
}
```

```
## [1] -0.03358715
## [1] -0.2257747
## [1] 0.2180721
## [1] -0.5076686
```

```r
#이러면 안전 length(df)가 0인 케이스를 피해준다
#저장해보자

output=c()
for(i in seq_along(df)){
    output[i]=median(df[[i]])
}
output #이렇게 할 수 있다
```

```
## [1] -0.03358715 -0.22577474  0.21807210 -0.50766863
```

```r
output=vector("double",ncol(df))
for(i in seq_along(df)){
    output[i]=median(df[[i]])
}
output #좀 더 메모리를 효율적으로 쓰는 법
```

```
## [1] -0.03358715 -0.22577474  0.21807210 -0.50766863
```

```r
x=c(1,3,5,7)
y=1:10
for(i in x){
    print(y[i])
} #i는 x를 순회한다 y의 1 3 5 7번째 요소만 낸다
```

```
## [1] 1
## [1] 3
## [1] 5
## [1] 7
```

```r
#결과값의 길이를 모르면 어떡하죠?
#일단 반복수만큼의 리스트를 만들어 저장하고
#리스트를 해제합니다
```

```r
means=c(0,1,2)
out <- vector("list", length(means))
for (i in seq_along(means)) {
  n <- sample(100, 1)
  out[[i]] <- rnorm(n, means[[i]])
}
str(out)
```

```
## List of 3
##  $ : num [1:12] -0.443 -1.254 0.779 -1.458 -0.841 ...
##  $ : num [1:44] 1.674 2.525 1.347 0.211 -0.394 ...
##  $ : num [1:31] 3.52 1.73 2.6 2.23 3.15 ...
```

```r
unlist(out) #이러면 원하는 결과 받는다
```

```
##  [1] -0.44294121 -1.25380713  0.77889891 -1.45790584 -0.84072739  0.07651466
##  [7]  0.30074439  1.14133207 -0.98800826  0.28666703 -0.89073466 -0.84081883
## [13]  1.67399224  2.52487302  1.34650446  0.21131829 -0.39351397  0.30862288
## [19]  1.82083934  0.54137943 -0.70840694  0.45931627  2.40676679  0.52299102
## [25] -1.95219126  1.18682312  2.03180631  2.12065178  1.31323558  2.60973808
## [31]  0.20156030  1.88084528  0.92400517  0.30159427  1.06973619  2.21777571
## [37]  2.13561774  2.28151251  1.95622588  0.23276629  0.89491945  0.91967555
## [43]  0.54893696  2.43823383  1.97658608  1.53068658 -0.20880198  1.94563606
## [49]  1.48256944  1.57937589  0.55790533  0.93104961  2.71161084  1.71278610
## [55]  1.45953488 -0.51094982  3.51799034  1.72779571  2.60152632  2.23408923
## [61]  3.15386779  2.65953476  1.72546309  2.38483502  0.92451699  2.24469173
## [67]  2.17971106  2.36259460  1.73394550  2.28209342  2.74867677  2.69622279
## [73]  1.38410077  1.71155574  1.61389868  1.74250782  2.62038555  1.06239397
## [79]  2.22197560  2.83438333  2.29559840  2.20372283  2.70204708  0.42654861
## [85]  2.03677738  2.10003911  1.81114274
```

```r
#얼마나 반복해야하나요?
#while()문을 씁시다

flip=function() sample(c("T","H"),1)

head_flip=function() {
flips=0
nhead=0
while(nhead<3) {
    a=flip()
```

```
    if(a=="H")
        nhead=nhead+1
    else
        nhead=0
    flips=flips+1
}

return(flips) #오호
}
head_flip()
```

```
## [1] 21
```

```
# + \alpha 표본평균을 구해볼까..
output=vector("double",100)

for(i in 1:100) {
    output[i]=head_flip()
}
output
```

```
##   [1] 25 13 19 17  3  6 13  4 33  3 15 15 21  3  7  4  3  5 18 10 17  4  6  3  8
##  [26] 13  6 18 18  7  3  6 11  7  6  3 15  3  5  3  8  6 15  7 20 48 11  6  3  5
##  [51] 23 11  4 17  7 14 10 24 12 17  3 10  3  5  4  8 34 12  4 42 14 10 16 13  5
##  [76] 23 26  3 11  9 13 10 17  8 23 11 13  9 13  3 27 45  9  3 19  7 11  5 19  6
```

```
mean(output) #그렇다네요
```

```
## [1] 11.93
```

```
#함수형 프로그래밍
col_summary <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
    out[i] <- fun(df[[i]])
  }
  out
}
#이는 함수에 인수로 함수를 전달하는 방법으로
#이렇게하면 mean median sd에대해 각각의 함수를 일일히 만들 필요가 없다

col_summary(df,median)
```

```
## [1] -0.03358715 -0.22577474  0.21807210 -0.50766863
```

```
col_summary(df,mean)
```

```
## [1]  0.191717553  0.123722845 -0.005452945 -0.567111408
```

```
col_summary(df,sd)
```

```
## [1] 0.8513636 0.8109885 0.7649798 0.6228917
```

```
#함수원형을  넣어줘야한다
```

purrr 라이브러리

purrr 라이브러리는 강력하고 for문을 대체한다

```
#각  데이터에  같은  함수를  적용하는  경우가  꽤나  많다
#map함수는  벡터를  순회하며  작업을  해준다


map(df,mean) #각  열에  평균함수  적용
```

```
## $a
## [1] 0.1917176
##
## $b
## [1] 0.1237228
##
## $c
## [1] -0.005452945
##
## $d
## [1] -0.5671114
```

```
map(df, median)
```

```
## $a
## [1] -0.03358715
##
## $b
## [1] -0.2257747
##
## $c
## [1] 0.2180721
##
## $d
```

```
## [1] -0.5076686
```

```
df %>% map(sd) #파이프 이용도 가능~
```

```
## $a
## [1] 0.8513636
##
## $b
## [1] 0.8109885
##
## $c
## [1] 0.7649798
##
## $d
## [1] 0.6228917
```

```
#근데 결과는 리스트이다
class(map(df,mean))
```

```
## [1] "list"
```

```
map_dbl(df, mean) #이러면 벡터로 나옴
```

```
##           a           b           c           d
##  0.19171755  0.12372284 -0.00545294 -0.56711140
```

```
x= list(list(1,2,3),list(4,5,6), list(8,9,10))

map(x, 2) #각 리스트에서 두번째 요소만 출력
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 5
##
## [[3]]
## [1] 9
```

```
#map에 숫자만 넣으면 인덱싱의 개념

mtcars %>%
    split(.$cyl) %>%
    map(function(df) lm(mpg ~ wt, data=df))
```

```
## $`4`
##
## Call:
## lm(formula = mpg ~ wt, data = df)
##
## Coefficients:
## (Intercept)           wt
##      39.571       -5.647
##
##
## $`6`
##
## Call:
## lm(formula = mpg ~ wt, data = df)
##
## Coefficients:
## (Intercept)           wt
##       28.41        -2.78
##
##
## $`8`
##
## Call:
## lm(formula = mpg ~ wt, data = df)
##
## Coefficients:
## (Intercept)           wt
##      23.868       -2.192
```

```
#cyl별로 쪼개고 lm을 적용하는 함수
#익명함수의 적용이 귀찮다 함수를 정의해줘야함 새롭게

models = mtcars %>%
    split(.$cyl) %>%
    map(~lm(mpg ~ wt, data=.)) #단축 공식 .의 의미는 현재 리스트 요소

models %>%
    map(summary) %>%
    map_dbl(~.x$r.squared) #r squared 출력
```

```
##         4         6         8
## 0.5086326 0.4645102 0.4229655
```

```
models %>%
    map(summary) %>%
    map_dbl("r.squared") #단축방법
```

```
##         4         6         8
## 0.5086326 0.4645102 0.4229655
```

```
#safely 함수를 알아보자
safe_log = safely(log)

safe_log(10) #결과와 오류가 뜬다
```

```
## $result
## [1] 2.302585
##
## $error
## NULL
```

```
safe_log("a") #에러내용을 알 수 있음
```

```
## $result
## NULL
##
## $error
## <simpleError in .Primitive("log")(x, base): non-numeric argument to mathematical function>
```

```
x = list(1,10,"a")
y= x %>%
    map(safe_log)
y #각 리스트 요소마다 안전하게 log를 씌워준다
```

```
## [[1]]
## [[1]]$result
## [1] 0
##
## [[1]]$error
## NULL
##
##
## [[2]]
```

```
## [[2]]$result
## [1] 2.302585
##
## [[2]]$error
## NULL
##
##
## [[3]]
## [[3]]$result
## NULL
##
## [[3]]$error
## <simpleError in .Primitive("log")(x, base): non-numeric argument to mathematical function>
```

```
y= y %>% transpose() #이러면 결과 리스트와 오류 리스트로 분류된다

#오류를 검출하는 법
is_ok = y$error %>% map_lgl(is_null)

x[!is_ok] #이러면 오류있는 데이터를 찾을 수 있음
```

```
## [[1]]
## [1] "a"
```

```
y$result[is_ok] %>% unlist() #오류 없는 항 출력
```

```
## [1] 0.000000 2.302585
```

```
x %>% map(possibly(log, NA_real_)) #이거는 오류나는 항을 NA로 대체해준다
```

```
## [[1]]
## [1] 0
##
## [[2]]
## [1] 2.302585
##
## [[3]]
## [1] NA
```

```
#map2는 이중인수를 받는다

mu=list(5,10,-3)
sigma=list(1,5,10)
```

```r
map2(mu, sigma, rnorm, n=5) # 이러면 mu sigma를 대응해 순회하며 rnorm함수를 적용한다 (n=5)
```

```
## [[1]]
## [1] 3.860576 4.416771 5.628931 5.004963 4.572105
##
## [[2]]
## [1]  3.834387 11.558803 16.827943  4.282821  5.869843
##
## [[3]]
## [1] -21.289650  -6.924278  -8.708244  10.950331  -0.278760
```

```r
n=list(1,3,5)
#pmap은 다중인수를 받는다

arg=list(n,mu,sigma) #인수들의 리스트를 만듬
pmap(arg, rnorm) #이러면 원하는 결과생성
```

```
## [[1]]
## [1] 5.864555
##
## [[2]]
## [1]  5.459573  7.990765 14.017443
##
## [[3]]
## [1] -9.238895 11.747765 10.357201  1.537877  2.043711
```

```r
#데이터 프레임을 쓰면 편하다

params <- tribble(
  ~mean, ~sd, ~n,
    5,     1,  1,
   10,     5,  3,
   -3,    10,  5
)

params %>% pmap(rnorm) #굿굿
```

```
## [[1]]
## [1] 4.604245
##
## [[2]]
```

```
## [1]  7.986348 22.983032  9.815639
##
## [[3]]
## [1]   9.282173  -8.995469  -3.194228  -3.558402 -13.424037
```

```
#함수도 순회시키고 싶으면?


f <- c("runif", "rnorm", "rpois")
param <- list(
  list(min = -1, max = 1),
  list(sd = 5),
  list(lambda = 10)
)

invoke_map(f, param, n = 5) %>% str() #param의 첫번째 요소에 f의 첫번째 요소를 적용하는 형식
```

```
## Warning: `invoke_map()` was deprecated in purrr 1.0.0.
## i Please use map() + exec() instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

```
## List of 3
##  $ : num [1:5] -0.758 0.276 -0.586 -0.202 -0.474
##  $ : num [1:5] 5.36 1.59 7.92 1.04 1.37
##  $ : int [1:5] 10 14 13 10 7
```

```
#역시 데이터프레임

sim <- tribble(
  ~f,      ~params,
  "runif", list(min = -1, max = 1),
  "rnorm", list(sd = 5),
  "rpois", list(lambda = 10)
)

rs=sim %>%
    mutate(sim=invoke_map(f,params,n=10))
```

```r
#여러 데이터에 같은 함수를 중복적용하는 방법을 알아보자

dfs <- list(
  age = tibble(name = "John", age = 30),
  sex = tibble(name = c("John", "Mary"), sex = c("M", "F")),
  trt = tibble(name = "Mary", treatment = "A")
)

dfs %>% reduce(full_join) #세 데이터프레임을 full_join 함수를 한번에 적용해준다
```

```
## Joining with `by = join_by(name)`
## Joining with `by = join_by(name)`
```

```
## # A tibble: 2 x 4
##   name    age sex   treatment
##   <chr> <dbl> <chr> <chr>
## 1 John     30 M     <NA>
## 2 Mary     NA F     A
```

```r
#accumulate는 비슷한데 누적한다

x=1:10

reduce(x, `+`) #모든 요소에 덧셈 적용
```

```
## [1] 55
```

```r
accumulate(x,`+`) #누적 벡터
```

```
##  [1]  1  3  6 10 15 21 28 36 45 55
```