2b_data_struct

Code

data objects

data are stored as objects.

1. vector

1.0 Intro

1. one dim array: a systematic arrangement of similar objects
2. vector is the simplest. a scalor is a vector with len=1
3. One vector single data type

1.1 create vector

1. basic

```
v = c(0, 1, 2)
```

2. declare with name

```
c(mean=1,stdev=2,median=3,min=4,max=5)
```

note: can use double quote can don't use

3. other functions
   1. `seq()`:
      1. syntax: `seq(from = 1, to = 1, by = ((to - from)/(length.out - 1))`
      2. note:
         i start and end are both included
         can also specify by parameter `length.out` : what is the length of the output
   2. `:`
      Similar to the basics of `seq`

      ```
      1:5 gets 1 2 3 4 5
      5:1 gets 5 4 3 2 1
      ```

   3. create by the mode type:

      ```
      > numeric(12)
       [1] 0 0 0 0 0 0 0 0 0 0 0 0
      > character(10)
       [1] "" "" "" "" "" "" "" "" "" ""
      > logical(10)
       [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
      ```

1.2 accessing elements

```
x[1]
x[1:3]
x[c(1, 3, 5)]
x[x>3]    # all elements that > 3 in x

x[-1]   # exclude first
x[c(-1, -2)]
x[c(-1, -2, 3)]   # results in error
x[-6]   # even if length of x is 5, no error
```

1.25 modify elements

Idea: just how to access, assign the expression with a new value then is OK. Also work for other data objects.

```
# change the value of an exising element
> v <- 1:3
> v[1] <- 0
> v
[1] 0 2 3

# add a new element previously not existed:
> x <- 1
> (x[3] <- 3)
[1]  1 NA

# quick change
v[v>0] <- v[v>0] * 2
v <- factor(v); levels(v) <- c("a", "b")
```

1.3 calculation

see section 2a

1.4 misc operations

1. get name: `names()` Also work for other objects. Can also change in this way following the modification principal
2. get length: `length()`
   Also work for other objects.
3. combination of vectors
   use `c()` . Can combine diff datatype, but will convert to the same:
   1. numeric + logical: logical F -> 0, T -> 1 1. character + numerical/logical: numerical/logical -> char
4. explicit conversion of data type (also in section 2a)

   `as.numeric(v)` : eg characters that cannot be converted will be `NA`
5. unique()
   return unique elements as a list

   ```
   > unique(c("C", "D", "H", "D", "H"))
   [1] "C" "D" "H"
   ```

6. split()
   split the vector to several vectors and return as a list

   ```
   > g = c('a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c')
   > x = c(1, 2, 3, 4, 5,  6, 7, 8, 9)
   > split(x, g)
   $a
   [1] 1 4 7

   $b
   [1] 2 5 8

   $c
   [1] 3 6 9
   ```

7. repeat

   ```
   # repeat a number: rep(x, times)
   rep(3, 12)

   # repeat a vector:
   rep(seq(1,3), 2)          # 1, 2, 3, 1, 2, 3
   rep(c(1,2,3), c(1,2,3)) # 1, 2, 2, 3, 3, 3; note: if num not match, error
   rep(c(1,2,3), each=2)    # 1, 1, 2, 2, 3, 3
   ```

8. `tapply`
   1. Intro: Apply a function to subsets of a vector or array, split by factors
   2. Syntax `tapply(data, INDEX, FUN, ...)`

      data: The vector or array you want to summarize.
      INDEX: A list of factors or grouping variables used to split the data.
      FUN: The function you want to apply to each subset.
   3. Eg.

      ```
      scores <- c(85, 90, 78, 92, 88, 76, 84, 92, 95, 89)
      grades <- c("A", "A", "B", "A", "B", "C", "B", "A", "A", "B")

      # Using tapply() to calculate the average score for each grade
      tapply(scores, grades, mean)

          A     B     C
      90.80 84.75 76.00
      ```

9. `outer(X, Y, FUN = "*", ...)`

   The outer product of the arrays X and Y is the array A with dimension c(dim(X), dim(Y)) where element `A[i, j] = FUN(X[i], Y[j], ...)` . `FUN` can also be `+`, ...

   ```
   > outer(c(1, 2, 3), c(0.1, 0.2), "+")
         [,1] [,2]
   [1,]  1.1  1.2
   [2,]  2.1  2.2
   [3,]  3.1  3.2
   ```

10. `table()`
    1. desc: return a contingency table (which is of class table) of the counts at each combination of factor levels.
    2. eg
       1. basic use as counter

          ```
          grades <- c("A", "A", "B", "A", "B", "C", "B", "A", "A", "B")
          names <- c('Alice', 'Alice', 'Alice', 'Alice', 'Bob', 'Bob', 'Bob', 'Bo
          b', 'Bob')
          > table(g)
          g
          a b c
          4 3 2
          ```

       2. 2-way table

          ```
          > names = c('Alice', 'Alice', 'Alice', 'Alice', 'Bob', 'Bob', 'Bob', 'Bo
          b', 'Bob')
          > table(names, grades)
                grades
          names   a b c
            Alice 2 1 1
            Bob   2 2 1
          ```

2. matrix (& multi dimensional arrays)

2.0. intro

n by m means n rows m cols

2.1. creation

1. Syntax
   `matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,dimnames = NULL)`
2. eg

   ```
   # col-wise fill in by default
   > m<-matrix(1:12,nrow=3,ncol=4)
   > m
        [,1] [,2] [,3] [,4]
   [1,]    1    4    7   10
   [2,]    2    5    8   11
   [3,]    3    6    9   12
   ```

3. Notes

1. if can't match

```
> matrix(1:9, nrow=3) # no error
> matrix(1:10, nrow=3)    # no error but warning
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8    1
[3,]    3    6    9    2
警告信息:
In matrix(1:10, nrow = 3) :
  data length [10] is not a sub-multiple or multiple of the number of rows [3]
```

2.2. inspection

```
> dim(m)
[1] 3 4
> nrow(m)
[1] 3
> ncol(m)
[1] 4
```

2.3. access

```
m[2,3]    # element, slicing, -1 ... same with vector
m[2,]     # select row (result is vector)
m[,3]     # select col (result is vector)

m[1:2, 3:4]   # (result is a matrix)
```

2.35 change

follow the change of vector how to access how to change

2.4. misc op

1. combine:

```
m = cbind(mab, mac)   # bind diff cols. If mab is axb, mac is axc, then m is ax(b+c)
m = rbind(mac, mbc)   # similarly
```

   1. Note
      1. can also be used in dataframe, see dataframe section
2. transpose: `t()`
3. `prop.table(rs, margin=1)` : along axis=margin (1 for row, 2 for col), return the proportion (percentage)
4. `col()` , `row()`
   Returns a matrix of integers indicating their row number in a matrix-like object, or a factor indicating the row labels.

```
> row(matrix(0, 2, 3))
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
```

5. `apply` :
   1. description: apply a function to margins of an array or matrix, return a vector or array or list of values.
   2. syntax

```
apply(X, MARGIN, FUN, ..., simplify = TRUE)
```

      1. MARGIN: a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, 3 indicates the 3rd axis for 3D matrix … c(1, 2) indicates rows and columns.
   3. eg

```
# apply for a single dim
> y<-matrix(runif(500,0,20), ncol=5)
> m<-apply(y,2,mean)    # find mean along the 5 cols separately
> m
[1] 10.528686  9.941802  9.340321  9.888169 10.840206


# for mul-dim array, along multiple dimesions
# how to interpret: the result array `ap`'s (i, j) element: the mean of array `z`'s element in both i for dim1 and j for dim3
> (z=array(rep(seq(20,by=-2,length=12),2), dim=c(3,4,2)))
, , 1

     [,1] [,2] [,3] [,4]
[1,]   20   14    8    2
[2,]   18   12    6    0
[3,]   16   10    4   -2

, , 2

     [,1] [,2] [,3] [,4]
[1,]   20   14    8    2
[2,]   18   12    6    0
[3,]   16   10    4   -2

> (ap=apply(z, c(1,3), mean))
     [,1] [,2]
[1,]   11   11
[2,]    9    9
[3,]    7    7
```

6. col/row stats
   1. desc: Form row and column sums and means for numeric arrays (or data frames)
   2. syntax

```
colSums (x, na.rm = FALSE, dims = 1)
rowSums (x, na.rm = FALSE, dims = 1)
colMeans(x, na.rm = FALSE, dims = 1)
rowMeans(x, na.rm = FALSE, dims = 1)
```

2.5. side track: mul-dim array

1. create
   `array(data = NA, dim = length(data), dimnames = NULL)`
   mind the dim! first 2 dim is the array, then is the third dim
   also mind for filling

```
> array(data=1:24, dim=c(4, 3, 2))
, , 1

     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

3. **list**

3.0. intro

most general obj. compared to vectors and matrices: can hold diff type of obj

3.1. creation

1. normal creation

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

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

[[3]]
[1] TRUE
```

2. create with name

```
w<-list("ab"=1:3, "y"=c("a","b"), "z"=T)

w<-list(ab=1:3, y=c("a","b"), z=T)    # OK
w<-list(ab=1:3, y=c("a","b"), 1=T)    # error
w<-list(ab=1:3, y=c("a","b"), "1"=T)  # OK
```

3.2. name

```
assign name:
> names(w) = c("x", "y", "z")
$x
[1] 1 2 3

$y
[1] "a" "b"

$z
[1] TRUE
```

3.3. access

1. single square bracket `[]` : return value is still a list. Can both access index or name.
   1. eg

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

[[2]]
[1] TRUE
```

   2. Notes
      1. Access name: still write in bracket. But cannot partially write

```
names(w) = c("ab" ,"y", "z")

> w["x"]
$x
[1] 1 2 3

# using `[]`, cannot (in dataframe, is error)
> w["a"]
$<NA>
NULL
```

2. double square bracket `[[]]` : return is the element of the list. Can also be accessed with `$` if named.
   1. Note:
      1. In this way, can only return 1 element, can't do slicing.
         eg. This is actually `w[[1]][2]` but strangly no error

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

2. names are access with $; index are accessed with [[]]

    1. basic eg

```
w[[2]]  # OK
w$x     # OK
w$"x"   # OK

w[["x"]]    # error
w$1         # error
```

    2. special case: partially write out the name

```
names(w) = c("ab" ,"y", "z")

# using `$`, can partially
> w$a
[1] 1 2 3
```

3.3a change

follow the change of how to access how to change

```
# change the value of an existing element
> (w<-list("ab"=1:3, "y"=c("a","b"), "z"=T))
$ab
[1] 1 2 3
$y
[1] "a" "b"
$z
[1] TRUE
> w$y <- c(1, 2)
> w
$ab
[1] 1 2 3
$y
[1] 1 2
$z
[1] TRUE

# add an element that previously not existed
> w<-list("ab"=1:3, "y"=c("a","b"), "z"=T)
> w$new = c(T, T, F)
> w
$ab
[1] 1 2 3
$y
[1] "a" "b"
$z
[1] TRUE
$new
[1]  TRUE  TRUE FALSE
```

strange case

```
> (w<-list("ab"=1:3, "y"=c("a","b"), "z"=T))
$ab
[1] 1 2 3

$y
[1] "a" "b"

$z
[1] TRUE

> w[2]
$y
[1] "a" "b"

> w[2] <- c(1, 2)
警告信息:
In w[2] <- c(1, 2) : 被替换的项目不是替换值长度的倍数
> w
$ab
[1] 1 2 3

$y
[1] 1

$z
[1] TRUE
```

3.4 misc op

1. `unlist()` : change the list obj to a long vector, using similar data type casting rule

4. dataframe

4.1. intro

data frame is a special kind of list where each member (col, but not row) are vectors of equal length

```
> women   # built-in dataframe in R
   height weight
1      58    115
2      59    117
3      60    120
4      61    123
5      62    122
```

Each row is called an **observation** and each column is called a **variable**

4.2 create dataframe

1. basic

```
member <- data.frame(
  name = c("Tom", "May"),
  age = c(22, 20)
)
```

2. Convert from matrix

```
> m <- matrix(1:6, nrow=2)
> data.frame(m)
  X1 X2 X3
1  1  3  5
2  2  4  6
```

4.3 access

1. Access with column first: as it is a “list”

    1. Similar to list, `[[]]` or `$` : return as a vector

```
> women$weight
 [1] 115 117 120 123 126 129 132 135 139 142 146 150 154 159 164
> women[[2]]
 [1] 115 117 120 123 126 129 132 135 139 142 146 150 154 159 161
```

2. Access like an array

    Just do `[1, c(2, 3)]` ...

```
> women[c(1, 2),]   # still return a dataframe
  height weight
1      58    115
2      59    117

> women[c(1, 2),1]  # return a vector
[1] 58 59
```

3. `with()` : directly access columns w/o using $:

```
with(women, weight/height)
# gives out a vector: [1] 1.982759 1.983051 2.000000 2.016393 ...
```

4.4 change

1. modify existing: just how to access, assign new to the expression
2. add new col:

    1. `df$new_row <- c('new', 'row', 'info')`
    2. use `cbind`

```
> (rand6 <- sample(6))
[1] 5 6 4 1 2 3
> cbind(head(women), rand6)
  height weight rand6
1     58    115     5
2     59    117     6
3     60    120     4
4     61    123     1
5     62    126     2
6     63    129     3
```

4.5 inspect

Note: some methods can also be used in list, matrix, …

1. `names()` ~
2. `head()` and `tail()`
    1. syntax

    ```
    head(x, n = 6L, ...)
    tail(x, n = 6L, keepnums = FALSE, addrownums, ...)
    ```

    1. `n` number of lines to display. By default is 6

4.7 misc methods

1. `summary(df)` : provides the mean max …
2. `by()` : similar to `tapply()` , for dataframe
    1. Syntax:
       `by(data, INDICES, FUN)`
    2. eg

```
> data <- data.frame(
    Class = c("A", "A", "B", "B", "C", "C"),
    Student = c("John", "Alice", "Bob", "Eve", "Charlie", "David"),
    Score = c(85, 90, 78, 88, 92, 95),
    Score2 = c(80, 92, 74, 82, 93, 96)
  )

> by(data$Score, data$Class, mean)
data$Class: A
[1] 87.5
------------------------------------
data$Class: B
[1] 83
------------------------------------
data$Class: C
[1] 93.5

> by(data[, c(3, 4)], data$Class, colMeans)   # multiple col
data$Class: A
 Score Score2
  87.5   86.0
------------------------------------
data$Class: B
 Score Score2
    83     78
------------------------------------
data$Class: C
 Score Score2
  93.5   94.5
```

3. `aggregate()` : apply a function to dataframe separated by something
    1. syntax Note: also have syntax for other objects, here only list for dataframe

    ```
    aggregate(x, data, FUN, ..., subset, na.action = na.omit)
    ```

    x is the name of the columns to apply function / be separated by; separated by ~ data is the dataframe number FUN is the function to apply to. eg `mean` .

    2. eg

```
# one ~ one
aggregate(weight ~ feed, data = chickwts, mean)

# one ~ many: just use + don't ask why
aggregate(breaks ~ wool + tension, data = warpbreaks, mean)

# many ~ one: just use cbind don't ask why
aggregate(cbind(Ozone, Temp) ~ Month, data = airquality, mean)

# many ~ many
aggregate(cbind(ncases, ncontrols) ~ alcgp + tobgp, data = esoph, sum)

## full eg of last
> head(esoph)
  agegp     alcgp    tobgp ncases ncontrols
1 25-34 0-39g/day 0-9g/day      0        40
2 25-34 0-39g/day    10-19      0        10
3 25-34 0-39g/day    20-29      0         6
4 25-34 0-39g/day      30+      0         5
5 25-34     40-79 0-9g/day      0        27
6 25-34     40-79    10-19      0         7

> aggregate(weight ~ feed, data = chickwts, mean)
       feed   weight
1    casein 323.5833
2 horsebean 160.2000
3   linseed 218.7500
4  meatmeal 276.9091
5   soybean 246.4286
6 sunflower 328.9167

> aggregate(cbind(ncases, ncontrols) ~ alcgp + tobgp, data = esoph, sum)
       alcgp    tobgp ncases ncontrols
1  0-39g/day 0-9g/day      9       252
2      40-79 0-9g/day     34       145
3     80-119 0-9g/day     19        42
4       120+ 0-9g/day     16         8
5  0-39g/day    10-19     10        74
6      40-79    10-19     17        68
7     80-119    10-19     19        30
8       120+    10-19     12         6
9  0-39g/day    20-29      5        37
10     40-79    20-29     15        47
11    80-119    20-29      6        10
12      120+    20-29      7         5
13 0-39g/day      30+      5        23
14     40-79      30+      9        20
15    80-119      30+      7         5
16      120+      30+     10         3
```

5. **factor**

Intro

Is effective when there are a lot of same elements: categorize the data, represent & store it on multiple levels.
A level is the representation of a kind of element. Name of levels are string. Levels coded as integers (so effective to store data with repeated element).

Notes

1. Storage issues Levels are internally coded as intergers, and the levels are stored as strings (even if original is numeric or logical).
    1. eg1: logical expressions should be all expressed in character from (since is stored as strings) (but not exactly the case?)

    ```
    > nums <- factor(c(3, 2, 2, 2, 3))
    > nums == "3"   # expected
    [1]  TRUE FALSE FALSE FALSE  TRUE
    > nums == 3     # ? don't know why but don't do this
    [1]  TRUE FALSE FALSE FALSE  TRUE
    ```

    2. eg2: `mode(factor(c("a","b")))` gets "numeric";
    but `class(factor(c("a","b")))` gets "factor" not "numeric".
    3. eg3: illustration of how is it coded in integer

    ```
    # see the code (int representation) of levels
    > as.integer(grp)
    [1] 1 2 1 2

    # inspect levels
    > levels(grp)
    [1] "control" "treatment"

    > levels(grp)[as.integer(grp)]
    [1] "control" "treatment" "control" "treatment"
    ```

create

```
# create from list
> grp <- factor(c("control", "treatment", "control", "treatment"))
> grp
[1] control   treatment control   treatment
Levels: control treatment
```

operations

```
# relabel
> levels(grp) <- c("1","2")
> grp
[1] 1 2 1 2
Levels: 1 2

# compare / find wanted element
# Note: need to compare as string
> grp == "1"
[1]  TRUE FALSE  TRUE FALSE

# common use: rename a repeating section
participants <- data.frame(
    destination = c("Austria","Korea","Japan","Austria","Sweden"),
    age = c(18,20,22,NA,20)
)

participants$destination <- factor(participants$destination)
levels(participants$destination) <- c("Australia","Japan","Korea","Sweden")
```