Data structure

Yufeng Huang

Associate Professor of Marketing, Simon Business School

August 1, 2022

Review: vector and vector operations

- What are the five data types?
  - in particular what are the differences between numeric, character and logical?
- How do vector operations work
  - 1) between two vectors with different length?
  - 2) between two vectors of different data type?
- How to read help files?
- Indentation, space and comment when writing clean code

Vectors are the most basic atomic structures

Vectors as **atomic** structures

```
# define a
a <- 1:10

# element is a vector (scalar in this case)
a[2]
```

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

```
# full subset of a is itself
identical(a[1:10], a)   # returns TRUE if two objects are EXACTLY the same
```

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

```
#    note: different from a == a[1:10]

# some other subsets of a; also a vector
a[c(1, 3, 8)]
```

```
## [1] 1 3 8
```

A vector has length but no dimension

```
# length
length(a)

## [1] 10

# a is a vector and has no dimension
dim(a)

## NULL

# in short, NULL means "nothing"
```

All elements of a vector must be the same type

```
x <- c(1, 2)

# recall coersion
x[3] <- "beautiful day"

x
```

```
## [1] "1"              "2"              "beautiful day"
```

```
# can get type of the vector but not each element
typeof(x)
```

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

Vectors

- Vectors are **atomic** structures
  - elements of a vector are vectors
- All elements in a vector must be of the **same type**
  - i.e. they have to be **all** logicals, integers, double, complex or character
  - if you mix types, R will impose coersion rules
- Vectors are the most basic atomic structures that has **no dimensions**

Arrays are atomic structures with dimensions

Converting a vector to an array

```
# define vector x
x <- c("one", "two", "three", "four", "five", "six", "seven", "eight")

# can convert x into an array y
y <- array(x, dim = c(2, 2, 2))

# maintains length of x
length(y)       # has length
```

```
## [1] 8
```

```
# dimension of y
dim(y)  # and dimensions
```

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

```
# console's display of a 2x2x2 array
y

## , , 1
##
##      [,1]  [,2]
## [1,] "one" "three"
## [2,] "two" "four"
##
## , , 2
##
##      [,1]   [,2]
## [1,] "five" "seven"
## [2,] "six"  "eight"

# Wait: how are elements arranged in an array?
```

Order: by dimension

- When you look up a dictionary for "cab", you:
  - find first element "c" (e.g. after "boy")
  - given "c", find second element "a" (e.g. before "cow"), and so on...
- Similarly, to order elements in R
  - first fix all other dimensions of y at 1, arrange x into **first** dimension in order
  - so x[1] goes to y[1, 1, 1], x[2] goes to y[2, 1, 1], etc.
  - when first dimension of y is full, turn to y[1, 2, 1], y[2, 2, 1], and so on...

Can convert into array by simply adding dimensions

```
# simply add dimensions will make x an array
dim(x) <- c(8, 1)
x
```

```
##       [,1]
## [1,] "one"
## [2,] "two"
## [3,] "three"
## [4,] "four"
## [5,] "five"
## [6,] "six"
## [7,] "seven"
## [8,] "eight"
```

```
# more generally, "dim" is an attribute (now make x a row)
attr(x, "dim") <- c(1, 8)
x
```

```
##       [,1]  [,2]  [,3]    [,4]   [,5]   [,6]  [,7]    [,8]
## [1,] "one" "two" "three" "four" "five" "six" "seven" "eight"
```

```
# same as dim(x) <- c(1, 8)
```

Can convert array into vector by simply taking away dimensions

```
# can convert back
# assigning something as NULL will eliminate it
dim(x) <- NULL
x
```

```
## [1] "one"   "two"   "three" "four"  "five"  "six"   "seven" "eight"
```

Retrieving elements of the array

```
# single element
y[1, 1, 1]
```

```
## [1] "one"
```

```
# one (the second) dimension
y[1, , 1]
```

```
## [1] "one"   "three"
```

```
# two dimensions
y[1, , ]
```

```
##      [,1]    [,2]
## [1,] "one"   "five"
## [2,] "three" "seven"
```

```
# multiple elements (happened to be identical)
y[1, c(1, 2), 1:2]
```

```
##      [,1]    [,2]
## [1,] "one"   "five"
## [2,] "three" "seven"
```

```
# subsets of an array are arrays
dim(y[1, , ])
```

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

Note 1: can still subset the array by index (which is different from subsetting by dimension)

```
# subset by index
y[3]    # cell 3 (same as subsetting a vector)
```

```
## [1] "three"
```

```
# subset by dimension
y[1, 2, 1]      # first row, second column, first page
```

```
## [1] "three"
```

Note 2: subsetting array by index of each dimension

```
# three arguments, some of which are vectors
y[1, c(1, 2), 1:2]      # first row, column 1 and 2, page 1 to 2

##      [,1]    [,2]
## [1,] "one"   "five"
## [2,] "three" "seven"

# compared to
y[1, 1, 2, 1:2]

      ## Error in y[1, 1, 2, 1:2]:  incorrect number of dimensions
```

Note 3: “trivial” dimensions are collapsed

```
# collapse first and second dimensions
y[1, 2, 1:2]

## [1] "three" "seven"

# collapse first dimension
y[1, c(2, 1), 1:2]

##      [,1]    [,2]
## [1,] "three" "seven"
## [2,] "one"   "five"
```

Note 4: when subsetting by dimensions, empty argument represents “all”

```
# second (column) argument left empty,
#    represents "all elements" in that dim
y[1, , c(2, 1)]
```

```
##      [,1]    [,2]
## [1,] "five"  "one"
## [2,] "seven" "three"
```

Matrix as a two-dimensional array

```
z <- matrix(1:4, nrow = 2, ncol = 2)
z
```

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

```
# certain operation ONLY apply to matrices
```

```
t(z) # e.g. transpose
```

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

```
z %*% z # e.g. matrix multiply
```

```
##      [,1] [,2]
## [1,]    7   15
## [2,]   10   22
```

Let's look at a few variations

```
# recall y
y <- array(1:8, dim = c(2, 2, 2))

# subset y by dimension
y[1, 1, 1]

## [1] 1

y[c(1, 2), 1, 1]

## [1] 1 2

y[1, c(1, 2), 1]

## [1] 1 3

y[1, 1, c(1, 2)]

## [1] 1 5

# flip order
y[1, 1, c(2, 1)]

## [1] 5 1

# subset repeated elements
y[1, 1, c(1, 2, 1, 2, 1)]

## [1] 1 5 1 5 1
```

Example: looking up elements in a matrix

```r
# assign a
a <- array(1:100, dim = c(10, 10))

# what's the element in cell (location) [6, 7] of a?
```

Example: looking up elements in a matrix

```r
# assign a
a <- array(1:100, dim = c(10, 10))

# what's the element in cell (location) [6, 7] of a?
```

```r
# the count goes column wise
head(a, n = 3)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1   11   21   31   41   51   61   71   81    91
## [2,]    2   12   22   32   42   52   62   72   82    92
## [3,]    3   13   23   33   43   53   63   73   83    93
```

```r
# so first 6 columns are cycled through completely
# so (7 - 1)*10

# last column goes to the 6th row
# so 6

# add together and find location
a[(7 - 1)*10 + 6]
```

```
## [1] 66
```

```r
a[6, 7]
```

```
## [1] 66
```

Another example

```r
# assign a
b <- array(1:625, dim = c(5, 5, 5, 5))

# what's the element in cell (location) [2, 3, 4, 5] of a?
```

Another example

```r
# assign a
b <- array(1:625, dim = c(5, 5, 5, 5))

# what's the element in cell (location) [2, 3, 4, 5] of a?
```

```r
# last dim of the cell location is 5, so cycle through dimensions
#    1, 2, 3 FOUR TIMES:
# (5 - 1)*5*5*5

# fourth dim is 4, so cycle through dimensions 1 and 2 THREE TIMES:
# (4 - 1)*5*5

# third dim is 3, so cycle through dimension 1 TWICE:
# (3 - 1)*5

# first dim is 2, so take 2
# 2

# add together and find location
b[(5 - 1)*5*5*5 + (4 - 1)*5*5 + (3 - 1)*5 + 2]
```

```
## [1] 587
```

```r
b[2, 3, 4, 5]
```

```
## [1] 587
```

Functions and operators on arrays

Math functions on arrays

```
# define a matrix
a <- array(1:9, c(3, 3))

a
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
# most math functions work element-wise on arrays
log(a)
```

```
##           [,1]     [,2]     [,3]
## [1,] 0.0000000 1.386294 1.945910
## [2,] 0.6931472 1.609438 2.079442
## [3,] 1.0986123 1.791759 2.197225
```

Operators on arrays

```
# element-wise operation between two arrays
b <- array(1, c(3, 3))
a - b
```

```
##      [,1] [,2] [,3]
## [1,]    0    3    6
## [2,]    1    4    7
## [3,]    2    5    8
```

```
# recycling rule works on arrays
a + 3
```

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

Your turn

```
# recall
a <- array(1:9, c(3, 3))

# recycling works with array and vector
a + c(1, 2, 3)

# quiz: what's the output?
#    hint: recycling rule + how things are ordered in an array
```

So far...

- What are the differences between arrays and vectors?
  - Vectors and arrays are atomic in what sense?
  - How are elements ordered when I form a vector into an array?
  - How do math functions operator on an array
- So far: we have focused entirely on atomic data structures

Beyond atomic structures: factors, lists and data frames

Why extend beyond atomic structures?

- Vectors and arrays are atomic structures
- Meaning: they can store one type of data (recall: coersion)
- In real life, many datasets contain multiple types of data
- Example: contact information for 4 people

name	age	female	entry
Anita	26	T	2014
Linda	27	T	2015
Harikesh	28	F	2015
Yufeng	29	F	2016

Factors: concept

- If raw data is like this
  - name: Anita, Anita, Anita, Linda, Linda, Linda, Linda, Linda, Linda
- It is more efficient to store these data in the following format
  - name: 1, 1, 1, 2, 2, 2, 2, 2, 2
  - levels: 1: Anita, 2: Linda

Factors are numeric but they **can** represent information in character vectors

```
# convert characters into factors
a <- c("one", "two", "three")
f_a <- factor(a)

# what is f_a?
f_a
```

```
## [1] one   two   three
## Levels: one three two
```

```
# contains a vector of integer and a vector of characters

# and is recognized as an integer vector
typeof(f_a)
```

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

```
# can retrieve the unique values that the integers represent (i.e., levels)
levels(f_a)
```

```
## [1] "one"   "three" "two"
```

What is the advantage of using a factor?

```r
# a vector of months
months_in_a_year <- c("January", "February", "March",
        "April", "May", "June", "July", "August",
        "September", "October", "November", "December")

# and we have 100 years of data
months <- rep(months_in_a_year, 100)

# factorize
f1_months <- factor(months)

# show head of the factor
head(f1_months, n = 4)
```

```
## [1] January  February March    April
## 12 Levels: April August December February January July June March ... Septem
```

Advantage of using factors (advantage 1)

```
# size of the original vector
object.size(months)

## 10352 bytes

# size of factorized vector
object.size(f1_months)

## 6064 bytes
```

Ordered factor: **poor** example

```
# recall that months is rep(months_in_a_year, 100)

# cannot compare unordered factors
f1_months[1] > f1_months[2]

     ## Warning in Ops.factor(f1_months[1], f1_months[2]):  '>' not
                          meaningful for factors

## [1] NA

# create an ordered factor
f2_months <- factor(months, ordered = TRUE)

# note that the order is alphabetical
head(f2_months, n = 4)

## [1] January  February March    April
## 12 Levels: April < August < December < February < January < July < ... < Sep
```

Ordered factor: **good** example (advantage 2)

```
# specify the order yourself
f3_months <- factor(months, ordered = TRUE,
        levels = months_in_a_year)

# note: shortcut because we defined order of levels before

# this time months are ordered
head(f3_months, n = 4)
```

```
## [1] January  February March    April
## 12 Levels: January < February < March < April < May < June < ... < December
```

"Implicit" translation between factor and the underlying value (advantage 3)

```
a <- c("one", "two", "three")
f_a <- factor(a)

# what happens if we compare factor and characters?
f_a == "two"
```

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

```
# thus, oftentimes we can just treat factors as equal to the
#    underlying value they represent
```

So far...

- Factors as a more efficient way to handle characters (in a given column/vector)
- So far we do not have a good way to deal with the contact information because it's a mixture of character and numeric data

Lists

Lists are the most general data structure

```
# list of names
names <- list("Anita", "Linda", "Harikesh", "Yufeng")
names
```

```
## [[1]]
## [1] "Anita"
##
## [[2]]
## [1] "Linda"
##
## [[3]]
## [1] "Harikesh"
##
## [[4]]
## [1] "Yufeng"
```

```
# it's not a vector but rather a list of scalars
#    also note the double bracket [[]]

names[[1]]      # first element in a list
```

```
## [1] "Anita"
```

Double bracket ([[]]) to extract an element, single bracket ([]) to extract a list that *contains* the element

```
# first element in a list (!!without preserving list structure)
names[[1]]

## [1] "Anita"

# a list that !!contains the first element
names[1]

## [[1]]
## [1] "Anita"
```

Can create a list of vectors

```
# list of vectors
clist <- list(
        c("Anita", "Linda", "Harikesh", "Yufeng"),
        26:29,
        c(T, T, F, F),
        c(2014, 2015, 2015, 2016)
)

# your turn: what's the output?
clist[[3]][2]
```

Can create a list of vectors

```
# list of vectors
clist <- list(
        c("Anita", "Linda", "Harikesh", "Yufeng"),
        26:29,
        c(T, T, F, F),
        c(2014, 2015, 2015, 2016)
)

# your turn: what's the output?
clist[[3]][2]
```

```
# [[]] gives the element, which is a vector
# [] gives the element in the vector
clist[[3]][2]
```

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

Can convert back to vectors but not always

```
#
x <- list(1, 2, 3)
x_vec <- unlist(x)
x_vec
```

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

```
y <- list(1, c(1, 2, 3), array(1:8, c(2,2,2)))
y_vec <- unlist(y)
y_vec   # forced into vector
```

```
##  [1] 1 1 2 3 1 2 3 4 5 6 7 8
```

```
z <- list(1, "a", 3 + 4i)
z_vec <- unlist(z)
z_vec   # elements coerced
```

```
## [1] "1"    "a"    "3+4i"
```

Cannot use math functions on a list (non-atomic)

```
# recall: math function on vector
a <- c(1, 2, 3)
log(a)

# now a list
b <- list(1, 2, 3)
log(b)
```

```
## Error in log(b):  non-numeric argument to mathematical function
```

Can name elements in a list

```
# list of vectors
clist <- list(
        name = c("Anita", "Linda", "Harikesh", "Yufeng"),
        age = 26:29,
        female = c(T, T, F, F),
        entry = c(2014, 2015, 2015, 2016)
)

# easier to understand
clist["female"][2]
```

Can name elements in a list

```
# list of vectors
clist <- list(
        name = c("Anita", "Linda", "Harikesh", "Yufeng"),
        age = 26:29,
        female = c(T, T, F, F),
        entry = c(2014, 2015, 2015, 2016)
)

# easier to understand
clist["female"][2]
```

```
clist["female"][2]      # Wait, what?

## $<NA>
## NULL
```

```r
# use double bracket to refer to elements in a list
clist[[1]][2]
```

```
## [1] "Linda"
```

```r
# refer to element names, also with double bracket
clist[["female"]][2]
```

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

```r
# or replace [[]] with "$" (only works with element names)
clist$female[2]
```

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

Special object: NULL

```
# NULL is an object with length 0
length(NULL)

## [1] 0

# NULL is its own kind
typeof(NULL)

## [1] "NULL"

# represents "things that do not exist" (contrast with NA)
v <- c(1, 2, 3)
dim(v)  # dim() not defined for a vector

## NULL
```

Special object: NULL (con'd)

```
# NULL can be used to eliminate list elements
names[[1]] <- NULL

names
```

```
## [[1]]
## [1] "Linda"
##
## [[2]]
## [1] "Harikesh"
##
## [[3]]
## [1] "Yufeng"
```

Lists are very general: can even create a list of different structures

```
# list of all kinds of structures
clist <- list(
        list("Anita", "Linda", "Harikesh", "Yufeng"), # list
        array(26:29, dim = c(2, 2)), # an array (2x2)
        "T, T, F, F",              # a scalar (character)
        c(2014, 2015, 2015, 2016)       # a vector
)

# but this is the opposite of being practical
```

So far...

- Lists are the most general structures
- Use double brackets [[]] to find *elements* in a list
  - as opposed to using single brackets [] to find the *subset* of a list (or subset/element of a vector)
- Can create very general list structures if needed

Data frame: an introduction

The problem with lists is that they are too general

```
# e.g. the one we just defined

clist[[1]]
```

```
## [[1]]
## [1] "Anita"
##
## [[2]]
## [1] "Linda"
##
## [[3]]
## [1] "Harikesh"
##
## [[4]]
## [1] "Yufeng"
```

```
clist[[2]]
```

```
##      [,1] [,2]
## [1,]   26   28
## [2,]   27   29
```

```
clist[[3]]
```

```
## [1] "T, T, F, F"
```

```
clist[[4]]
```

```
## [1] 2014 2015 2015 2016
```

```
# I don't want to work with something like this
```

Data frame are specific lists with more structures

```
cdata <- data.frame(
        name = c("Anita", "Linda", "Harikesh", "Yufeng"),
        age = 26:29,
        female = c(T, T, F, F),
        entry = c(2014, 2015, 2015, 2016)
)

# guess what it looks like?
```

Data frame are specific lists with more structures

```
cdata <- data.frame(
        name = c("Anita", "Linda", "Harikesh", "Yufeng"),
        age = 26:29,
        female = c(T, T, F, F),
        entry = c(2014, 2015, 2015, 2016)
)

# guess what it looks like?
```

```
##       name age female entry
## 1    Anita  26   TRUE  2014
## 2    Linda  27   TRUE  2015
## 3 Harikesh  28  FALSE  2015
## 4   Yufeng  29  FALSE  2016
```

Data frame

- Primary ways to handle tabular data in R
  - tabular: data in the form of tables
  - like spreadsheets in Excel
- Displayed like a table
- Stored as a list

So list notations work on a data frame

```
cdata$female[2] # a scalar

## [1] TRUE

cdata[["name"]] # a vector

## [1] "Anita"    "Linda"    "Harikesh" "Yufeng"
```

But data frame are also tables

```
cdata[c("name", "female")]      # list notation, now maintain the data frame
```

```
##       name female
## 1    Anita   TRUE
## 2    Linda   TRUE
## 3 Harikesh  FALSE
## 4   Yufeng  FALSE
```

```
# note the difference between [[]] and []
```

```
cdata[c(1, 3)]
# same as above
```

```
cdata[, c(1, 3)]        # same as above, but now note the array-like notation
```

Note that we can take sub-sets of a list the same way

```
clist[c(1, 3)]  # same info but different shape
```

```
## [[1]]
## [[1]][[1]]
## [1] "Anita"
##
## [[1]][[2]]
## [1] "Linda"
##
## [[1]][[3]]
## [1] "Harikesh"
##
## [[1]][[4]]
## [1] "Yufeng"
##
##
## [[2]]
## [1] "T, T, F, F"
```

But compare with the following: we can subset data frames in rows

```
cdata[1, ]

##    name age female entry
## 1 Anita  26   TRUE  2014

# we can cut a data frame row-wise
```

But compare with the following: we can subset data frames in rows

```
cdata[1, ]
```

```
##    name age female entry
## 1 Anita  26   TRUE  2014
```

```
# we can cut a data frame row-wise
```

```
clist[1, ]
```

```
## Error in clist[1, ]:  incorrect number of dimensions
```

```
# can't do it with a list because it's not rectangular
```

We can also subset by conditions

```
# select females
cdata[cdata$female==T, ]

##    name age female entry
## 1 Anita  26   TRUE  2014
## 2 Linda  27   TRUE  2015
```

We can also subset by conditions

```
# select females
cdata[cdata$female==T, ]

##    name age female entry
## 1 Anita  26   TRUE  2014
## 2 Linda  27   TRUE  2015
```

```
# select old males
cdata[cdata[[3]] == F & cdata[["age"]] > 27, ]

##       name age female entry
## 3 Harikesh  28  FALSE  2015
## 4   Yufeng  29  FALSE  2016

# note the equivalence of different ways to select variables
# but in practice, recommend sticking to one convention
```

Summary

- Arrays
  - what's the difference between a vector and an array?
- Lists
  - why do we say lists are not atomic data?
  - how do we subset a list? what's the difference between '[[]]' and '[]'
- Data frames
  - how are data frame shaped?
  - are data frames matrices or lists?