Vectors and operators

Yufeng Huang

Associate Professor of Marketing, Simon Business School

July 26-27, 2022

Data types and structures

- From this lecture, we'll start from the basics
  - basic data types
  - operators on data types
  - basic data structures and how to locate "where things are"
- These are like alphabet in a spoken language
  - boring, but very necessary!

Things we want to learn today

- Types of atomic data
- Atomic structures: scalars and vectors
- Finding where things are in a vector
- Operator on numeric data

Scalars are special cases of vectors

Scalars

- Data structure: how do I arrange my data?
- Vectors are the most basic *structures* of data
- Scalars are not a distinct class from vectors
  - scalar is just a vector with length 1
  - but it's useful to talk about it first
- Five basic types of vectors (scalars)
  - logical
  - integer
  - double
  - complex
  - character

Scalars

```r
# we've seen most of these before

# assign a scalar
x <- 1
x

## [1] 1

# operator on scalars
y <- 2
x + y

## [1] 3

# function on scalars
z <- 3
log(z)

## [1] 1.098612

# scalar is a vector of length 1
length(z)

## [1] 1
```

5 basic types of data

```r
# logical
x <- TRUE       # or T

# integer
y <- 2L         # need to assert integer; compare with below

# double (real)
z <- 3

# complex
a <- 4 + 5i

# character
b <- "hi"
```

Numeric (integer and double)

```r
# we're used to working with numeric values in datasets (mostly doubles)
a <- 3 + 4
5 + log(a)
```

```
## [1] 6.94591
```

```r
# we use integers for computational efficiency (less memory required)
#    but for the purpose of this class, think of the two as equivalent
```

Quotes to express a character (string)

```
# double or single quotes (equivalent) to express a character string
"a character string using double quotes"

## [1] "a character string using double quotes"

'a character string using single quotes'

## [1] "a character string using single quotes"

# can insert single quotes in double quotes (vice versa)
'you can insert "double quotes" into a string'

## [1] "you can insert \"double quotes\" into a string"

"you can insert 'single quotes' into a string"

## [1] "you can insert 'single quotes' into a string"

# long strings can be a single scalar
length("bla bla bla bla bla")

## [1] 1
```

Logical (boolean)

```
# logical (or boolean) values are TRUE and FALSE
a <- TRUE
b <- F
b

## [1] FALSE

# they come out of a logical expression
5 > 3

## [1] TRUE

a == "a"        # character "a" is not variable a

## [1] FALSE

# logicals can be combined with & (and) and | (or)
a <- 3
a > 2 | 2 < 1

## [1] TRUE

a > 2 & 2 < 1

## [1] FALSE

# more on logicals later
```

5 basic types of data, conversion[1]

```r
# logical is (sort of) integer 1
T == 1L

## [1] TRUE

# integer and double
2L == 2.0

## [1] TRUE

# complex (with zero in the imaginary part) and double
4 + 0i == 4

## [1] TRUE

# character of course is not numeric
"1.0" == 1.0

## [1] FALSE
```

[1] Which means that most of the time you don't need to distinguish among different types of real numbers; more formal arguments later

Trivia: different data classifications

```r
# type of 40
typeof(40)
```

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

```r
# mode of "Rochester"
mode("Rochester")
```

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

```r
# class of c(1, 2, 3)
class(c(1, 2, 3))
```

```
## [1] "numeric"
```

Trivia: different data classifications

- The difference is only at how you classify real numbers
  - integer or double (double-precision float number) are both considered to be numeric
- Throughout this class we're interested in the first 3 types

example value	type	mode	class
40	double	numeric	numeric
TRUE	logical	logical	logical
"Rochester"	character	character	character
$2 + 3i$	complex	complex	complex
2L	integer	numeric	integer

NaN: “not a number”

```r
# math operation with un-defined output generates NaN
log(-1)

## Warning in log(-1):  NaNs produced

## [1] NaN

# scalar NaN stands for "not a number"
length(NaN)

## [1] 1

# but NaN is a double for programming convenience
typeof(NaN)

## [1] "double"

# can verify whether something is NaN
is.nan(NaN)

## [1] TRUE
```

NA: missing value or "not available"

```r
# it's a scalar
length(NA)
```

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

```r
# can use is.na() to detect NA
is.na(NA)
```

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

```r
# it is logical but it's not true or false
is.logical(NA)
```

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

```r
# it's not comparable to anything, including itself and NaN
NA == NA
```

```
## [1] NA
```

```r
NA == NaN
```

```
## [1] NA
```

```r
# in fact, it is "not there", except it has to
#    be there to maintain the vector structure
#    in other words, it is a placeholder
```

Vectors are the most basic data structure

Vectors

- A vector is a sequence of cells that contain data
  - specifically: same type of data
- These are the most basic data structures in R
- Can be of any length (including zero)
- Have length but no dimensions (unlike Matlab)

To create a vector

```r
# function c() creates a vector
x <- c(1, 2, 3, 4, 5)

# a vector of characters
y <- c("one", "two", "three", "four", "five")

# a vector of logicals
z <- c(T, F, T, F, F)   # elements separated by comma
```

To create a vector

```
# another way to define a vector
a <- 1:10       # 1 to 10
a

##  [1]  1  2  3  4  5  6  7  8  9 10

# (a point specific to R) is this vector 1, 1.1, 1.2, 1.3, ...?
b <- 1:0.1:2
```

To create a vector

```r
# another way to define a vector
a <- 1:10       # 1 to 10
a

##  [1]  1  2  3  4  5  6  7  8  9 10

# (a point specific to R) is this vector 1, 1.1, 1.2, 1.3, ...?
b <- 1:0.1:2
```

```r
# answer is NO
b

## [1] 1 2

# CORRECT approach
b <- seq(1, 2, length.out = 11)
b

##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

seq() function: deterministic sequence with patterns

```r
# the following are equivalent
1:5
seq(5)
seq(1, 5)
seq(from = 1, to = 5, by = 1)
```

```r
# but seq() can do more advanced things
seq(from = 1, to = 2, by = 0.25)
```

```
## [1] 1.00 1.25 1.50 1.75 2.00
```

```r
# equivalent
seq(from = 1, to = 2, length.out = 5)
```

```
## [1] 1.00 1.25 1.50 1.75 2.00
```

```r
# note: please pay attention to the consistency between arguments
```

To replicate vectors

```r
# the following are equivalent
c(1, 1, 1, 1, 1)
seq(from = 1, to = 1, length.out = 5)
rep(1, times = 5)
```

```r
# replicate different times
rep(1:3, times = c(3, 2, 1))
```

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

```r
# replicate till fixed vector length
rep(c(2, 4, 6), length.out = 5)
```

```
## [1] 2 4 6 2 4
```

```r
# replicate each element before another
rep(c(2, 4, 6), each = 2)
```

```
## [1] 2 2 4 4 6 6
```

Scalar functions that operate element-by-element

```r
# many math functions are built in (many in {base})
sqrt(9)

## [1] 3

# they work element-wise on vectors
log(c(1, 2, 3))

## [1] 0.0000000 0.6931472 1.0986123

c(log(1), log(2), log(3))

## [1] 0.0000000 0.6931472 1.0986123

# more examples
cos(seq(from = 1, to = 5))

## [1]  0.5403023 -0.4161468 -0.9899925 -0.6536436  0.2836622

# element-wise (or "vectorized") operation is a very important concept
#    more on this point in the second half of the course
```

Vector (or array) functions that map into a single scalar

```
a <- c(1, 3, 7, 8)

mean(a)

## [1] 4.75

var(a)  # variance

## [1] 10.91667

sum(a)

## [1] 19

length(a)

## [1] 4

cor(a, -a)      # correlation coefficient

## [1] -1

# and many other examples...
```

Sub-setting: Locating elements in a vector

Locating elements

- What happens if I need parts of a vector?
- What if I know which part I need?
  - e.g. first 3 elements
- What if I need all parts that satisfy a condition?
  - e.g. elements larger than 2

Recall: length of a vector

```r
# all vectors have a length
a <- c("one", "two", "three")
length(a)

# what's the output?
```

Elements in a vector with known locations

```
# recall
a <- c("one", "two", "three")

# refer to their location to retrieve elements
a[1]    # VERY IMPORTANT: use brackets for subsetting

## [1] "one"

# or multiple elements
a[2:3]

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

Elements in a vector with known locations

```
# recall
a <- c("one", "two", "three")

# can subset non-adjacent elements
a[c(1, 3)]

## [1] "one"   "three"

# can re-order them
a[3:2]

## [1] "three" "two"

# rev() is the reverse function
rev(a)

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

# the following is equivalent to a[3:2]
rev(a)[1:2]

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

```
# what is the output?
rev(a[1:2])
```

Names of elements in a vector

```r
# can assign names of elements in a vector
b <- c(first = 1, second = 2, third = 3)
b
```

```
##  first second  third
##      1      2      3
```

```r
# then we can use names to retrieve it
b["first"]
```

```
## first
##     1
```

```r
# comment: often we don't specific assign names to a vector
#    but similar notations can be used for more complex data structures
```

Use logicals to subset a vector

```
# recall
a <- c("one", "two", "three")

# can use logicals to find elements
a[c(TRUE, FALSE, TRUE)]

## [1] "one"   "three"

# equivalent
a[c(1, 3)]

## [1] "one"   "three"

# which() finds the position where a logical vector is T
which(c(T, F, T))

## [1] 1 3
```

If don't know which element but know a condition

```r
# assign a numeric vector to a
a <- c(1, 2, 3)

# elements of vector a larger than 1.5
a[a > 1.5]

## [1] 2 3

# simultaneously satisfying two conditions ("&" for and)
a[a > 0.5 & a < 2.5]

## [1] 1 2

# satisfy at least one of the two conditions ("|" for or)
a[a > 0.5 | a < 2.5]

## [1] 1 2 3

# satisfy the reverse of a condition ("!" for not)
a[a != 2]

## [1] 1 3
```

Let's understand what happened

```r
# the logical operator returns logicals
x <- a > 1.5
x
```

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

```r
# finding where the logical is T
y <- which(x)
y
```

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

```r
# equivalent
a[x]
```

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

```r
a[c(2, 3)]
```

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

Interlude: 3 minutes on data frame

- R has many other ways to structure data
  - we'll cover them in the next week
  - but we'll take a look at the most common one – data frame
- Data frame is like a table or spreadsheet
  - has rows (observations)
  - and columns (variables)
- Loosely speaking
  - can view a data frame as a matrix
  - and a column as a vector

3 minutes on data frame

```r
# construct a data frame
df <- data.frame(id = c(1, 2, 3),
                 age = c(30, 29, 28))

# data frame looks like a table
df

##   id age
## 1  1  30
## 2  2  29
## 3  3  28

# columns are vectors (taken by $)
df$id

## [1] 1 2 3

df$age

## [1] 30 29 28
```

Why is subsetting so important? An example

- Load country-level savings data from the 1960s

```
# load savings data and convert variables into vectors
#    for now just copy and paste, will explain early next week

sr <- LifeCycleSavings$sr        # savings rate
gr <- LifeCycleSavings$ddpi      # growth rate of disposible income
dpi <- LifeCycleSavings$dpi      # disposible income
ctry <- row.names(LifeCycleSavings)     # list of countries

# source: {datasets}, LifeCycleSavings (1960-1970
#    Saving, GDP growth and disposible income data)
#    original: Sterling, Arnie (1977) Unpublished BS Thesis from MIT.
```

- What is the correlation between savings rate and growth of disposible income?

```
# correlation coefficient between the two
cor(gr, sr)

## [1] 0.3047872
```

- Conclude that saving and growth are related

- Does this relationship depend on whether the people are rich or poor?
  - why do we think that disposible income might play a role?
  - poor – savings as buffer stock against bad weather or wars
  - rich – savings invested either in capital (stock market) or education

```r
# for the poor countries
#    e.g. many African, Asian, South American countries in the 60s
cor(gr[dpi < 500], sr[dpi < 500])

## [1] 0.1461259

# for the moderately-rich countries
#    e.g. Spain, Italy, Brazil, etc.
cor(gr[dpi > 500 & dpi <= 1500], sr[dpi > 500 & dpi <= 1500])

## [1] 0.5931484

# for the rich countries
#    e.g. Switzerland, UK, US, etc.
cor(gr[dpi > 1500], sr[dpi > 1500])

## [1] 0.6478972
```

- Implementation: take correlation for subsets of the vectors
    - confirm the intuition => to the extent that the correlation between savings and growth is important, looking at this correlation by country group is crucial!
    - should really subset the data instead of vectors (wait until we cover data.frame)

Taking stock

- Vectors are the most basic atomic structures

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

- Elements in a vector are located based on its position or based on logicals

```
a[2]     # what is this?
a[c(F, T, F)]   # equivalent
```

- Vector subseting is the most important point today (and the rest of the class will be to strengthen our understanding on vectors)

A "tricky" example

- I don't recommend coding in this way, but the following example helps us understand subsetting notation versus function calls
- Define vector c

```
c <- c("one", "two", "three")

c

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

- Why are the following statements different?

```
# Your turn: What are the console output?
c[2]    # subset the second element in vector c
c(2)    # call function c() with argument 2
```

More about vector sub-setting: adding, dropping, and sorting elements

Can add elements to a vector

```r
# define vector a
a <- c(4, 5)

# add scalar 6
a[3] <- 6
a
```

```
## [1] 4 5 6
```

```r
# equivalent
a <- c(4, 5)
a <- c(a, 6)
a
```

```
## [1] 4 5 6
```

Can add non-adjacent elements

```
# define vector a
a <- c(4, 5)

# add scalar 7 to position 4
a[4] <- 7
a

## [1]  4  5 NA  7

# equivalent (don't have to specify 'after =')
a <- c(4, 5)
a <- c(a, c(NA, 7))      # recall what NA means
a

## [1]  4  5 NA  7
```

Can squeeze things into a vector

```r
# define vector a
a <- c(4, 5)

# add 4.5 between 4 and 5
a <- append(a, 4.5, after = 1)  # cannot use c() easily
a
```

```
## [1] 4.0 4.5 5.0
```

Subtract elements using “-”

```
# define vector a
a <- c(4, 5, 6)

# finding all elements of a *except for* index 2
a[-2]

## [1] 4 6

# equivalent
a[c(1, 3)]

## [1] 4 6
```

Can't have both positive and negative indices at the same time

```
# define vector a
a <- c(4, 5, 6)

# get rid of first element and retain the 3rd element
a[c(-1, 3)]

## Error in a[c(-1, 3)]:  only 0's may be mixed with negative subscripts

# what to do with element 2?
```

To sort elements in a vector

```r
num <- c(9, 4, 5, 1, 4, 1, 4, 7)

# sort elements
sort(num)

## [1] 1 1 4 4 4 5 7 9

sort(num, decreasing = TRUE)

## [1] 9 7 5 4 4 4 1 1

# Q: what is num now?
```

To sort elements in a vector

```r
num <- c(9, 4, 5, 1, 4, 1, 4, 7)        # same num

# reverse the order of elements
rev(num)
```

```
## [1] 7 4 1 4 1 5 4 9
```

```r
# position of sorted elements
#    e.g. the two 1's are in position 4 and 6
order(num)
```

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

```r
# what does the above mean?
num[order(num)]
```

```
## [1] 1 1 4 4 4 5 7 9
```

To detect duplicates in a vector[2]

```r
num <- c(9, 4, 5, 1, 4, 1, 4, 7)

# unique elements
unique(num)

## [1] 9 4 5 1 7

# duplicated elements
#    marks as duplicated only if element appears beyond the first time
duplicated(num)

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

# what are we doing now?
num[-which(duplicated(num))]

## [1] 9 4 5 1 7
```

[2] duplicated() finds duplicates, in the order from first to last element

Your turn

```
# without typing, tell me the results

a <- c(1, 1, 2, 2, 3)

a == sort(a, decreasing = F)

duplicated(a)

unique(a)

rep(unique(a), length.out = length(a))
```

Operators on numeric vectors

Operators on numeric vectors

- Arithmetic operators : $+$, $/$, ...
- Logical operators: $>$, $<=$, ...
- How does R handle vector operations?

Arithmetic operators

```
# simple stuff first
1 + 1

## [1] 2

2 * 1

## [1] 2

5 / 2

## [1] 2.5

7 ^ 2

## [1] 49

29 %/% 7        # integer division

## [1] 4

29 %% 7         # modulus (remainder after the integer division)

## [1] 1
```

Logical operators

```r
# comparison produce logical value
5 > 1

## [1] TRUE

# works on vectors and same rules apply
c(2, 1) <= c(2, 3)

## [1] TRUE TRUE

# double equal is comparison
5 == 4

## [1] FALSE

# single equal is assignment
5 = 4   # 5 is not a legal variable name

## Error in 5 = 4:  invalid (do_set) left-hand side to assignment
```

Logical operators between characters

```r
# can compare across characters
"abc" < 'def'
```

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

```r
# comparison is done in a dictionary-like way
"ab" < "abc"
```

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

```r
"abc" < "abd"
```

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

```r
"aed" < "afa"
```

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

Logical operators on logicals

```
# & for AND, | for OR
TRUE | FALSE

## [1] TRUE

2 > 1 & 3 > 2

## [1] TRUE

# ! for NOT
!TRUE

## [1] FALSE

# xor for exclusive OR (either but not both)
xor(2 > 1, 2 <= 1)

## [1] TRUE

xor(2 > 1, 3 > 2)

## [1] FALSE
```

How do operators work on vectors?

```
# between vectors of same length
c(2, 5, 32) - c(1, 2, 1)

## [1]  1  3 31

# between vector and scalar
c(1, 2, 1) + 2

## [1] 3 4 3

# what's the logic behind this?
c(3, 2, 4) * c(2, 3)

   ## Warning in c(3, 2, 4) * c(2, 3):  longer object length is not a
                    multiple of shorter object length

## [1] 6 6 8

# where does the number 8 come from?
```

Recycling rule

The recycling rule: the shorter vector is **replicated** enough times so that the result has the length of the longer vector

```
# the following two statements are equivalent
c(1, 2, 1) + 2

## [1] 3 4 3

c(1, 2, 1) + rep(2, length.out = length(c(1, 2, 1)))

## [1] 3 4 3
```

Recycling rule (con'd)

- Internally, R works as if there is always a rep() function before the operator

```
# the following two statements are equivalent
c(3, 2, 4) * c(2, 3)

## Warning in c(3, 2, 4) * c(2, 3):  longer object length is not a
multiple of shorter object length

## [1] 6 6 8

c(3, 2, 4) * rep(c(2, 3), length.out = length(c(3, 2, 4)))

## [1] 6 6 8
```

- But to prevent mistakes, I suggest you always allign elements before any operation (exception: scalar-vector operation)

Recycling rule works on comparison operators

```r
# comparison produce logical value
5 > c(1, 1, 1, 1)

## [1] TRUE TRUE TRUE TRUE

# another example
c(2, 1) <= c(2, 3, 2)

## Warning in c(2, 1) <= c(2, 3, 2):  longer object length is not a
multiple of shorter object length

## [1] TRUE TRUE TRUE
```

Your turn

```r
# what are the output?
sqrt(c(1, 4, 9, 16)) + c(1, 2)


sqrt(c(1, 4, 9, 16)) <= c(1, 2)


(sqrt(c(1, 4, 9, 16)) < c(1, 2)) <
        (sqrt(c(1, 4, 9, 16)) > c(1, 2))
```

Conversion between data types

Conversion

- One can convert between different data types
- Can explicitly coerce a vector into another type

```
# example
a <- c(1, 2, 3)

as.character(a)

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

- Or implicitly coerce in a given context

Implicit coercion

```r
# what if I do mix stuff up?
z <- c(T, "two", 3)

# question: what is z?
```

Implicit coersion

```r
# what if I do mix stuff up?
z <- c(T, "two", 3)

# question: what is z?
```

```r
z

## [1] "TRUE" "two"  "3"

# z is coerced to characters
```

Hierarchy in coercion

```r
# logical and integer
typeof(c(T, 2L))
```

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

```r
# logical, integer and double
typeof(c(T, 2L, 3))
```

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

```r
# logical, integer, double and complex
typeof(c(T, 2L, 3, 4 + 5i))
```

```
## [1] "complex"
```

```r
# logical, integer, double, complex and character
typeof(c(T, 2L, 3, 4 + 5i, "six"))
```

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

Explicit coersion

```
# can explicitly coerce vectors
a <- c(T, F, T, F)
as.numeric(a)

## [1] 1 0 1 0

# can always move to the "more general" type
x <- c(1, 2, 3)
y <- as.character(x)
y

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

Explicit coercion (con'd)

```
# coming back sometimes works
z <- as.numeric(y)
z

## [1] 1 2 3

# but this does not always work
b <- c("one", "two", "three")
as.numeric(b)

                ## Warning:  NAs introduced by coercion

## [1] NA NA NA
```

Coersion rule works when comparising across data types

```r
# what's 2L?
2L == 2
```

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

```r
# what's TRUE under coersion?
2.05 > TRUE
```

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

```r
# but can't compare across two complex
3 + 5i >= 2.5 + 2i
```

```
## Error in 3 + (0+5i) >= 2.5 + (0+2i):  invalid comparison with complex
                                    values
```

Coersion works when adding elements

```r
# define vector a
a <- c(1, 2)

# add a character to a
b <- "something > 1 but < 2"
a <- append(a, b, after = 1)

# note the type change
a

## [1] "1"                     "something > 1 but < 2" "2"
```

Taking stock

- Vectors are the most basic structure
  - types of (atomic) data; explicit and implicit coersion
- Vector sub-setting
  - by index

```
a <- c(1, 4, 7)
a[-1]    # what does the -1 index mean?
a        # what is a?
```

  - by logical

```
b <- c(1, 2, 3)
a[b > 2]         # what is this?
```

- Vector operations

```
c(1, 4, 5, 7) * c(2, 1) # what is this?
c(1, 4, 5, 7) > 1       # what is this?
```