Solving Quantitative Problems with R

Data and Introductory Programming

Session 2

R Data Structures and Objects

September 25, 2018

R Language Fundamentals

R Datatypes

R Objects

- Vectors
- Matrices
- Lists
- Data Frames
- Factors

Conversion

Overview

R Language Fundamentals

R Datatypes

R Objects
- Vectors
- Matrices
- Lists
- Data Frames
- Factors

Conversion

Thinking R: What R is about

The basic element in R is a **vector** (not a scalar!). A vector is set of values that are all of the same type.

```
2
```

```
[1] 2
```

```
matrix(2:9,2,4)
```

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

```
"a"
```

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

```
letters[1:3]
```

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

Overview

R Language Fundamentals

R Datatypes

R Objects
- Vectors
- Matrices
- Lists
- Data Frames
- Factors

Conversion

Datatypes

The existing datatypes are

- integer: 1, 2, 3, ..
- numeric: 2.12, π, 4.0
- character: “joe”, “ strawberry fields”
- complex: 4+3i
- logical: TRUE or FALSE (abbreviated as T or F)

Integers are a subclass of numeric, you don’t need to convert explicitly before using them together.

For each of these datatypes there is a set of operations. Since the basic element is a vector you also need to think of this operations as operations on vectors (or list of elements).

Operation on Numerical Vectors

A non-exhaustive list of operations on vectors is given below.

	Description
`x+y`	Addition (elementwise) of x and y
`x-y`	Substraction (elementwise) of x and y
`x*y`	Multiplication (elementwise) of x and y
`x/y`	Division (elementwise) of x and y
`x<y`	Comparison (elementwise) (may also be $>$, $\geq$,$\leq$,==, !=)
`sum(x)`	Sum of all elements in x
`mean(x)`	Mean of all elements in x
`prod(x)`	Product of all elements of x
`diff(x)`	Differences of x, i.e., $x_2 - x_1$, $x_3 - x_2$,..

Where `==` is the operator for equal and `!=` for unequal. However, some caution is needed when doing comparisons on non-integer values (cf. Session 1)

```
sqrt(2)^2 == 2

[1] FALSE

sqrt(2)^2 -2

[1] 4.440892e-16
```

Operations on character vectors

- Concatenation: To paste together two character objects / strings: `paste(x,y)`
  - `paste(x, y, sep = sepstring)` adds a `sepstring` between `x` and `y`, by default `sep = " "`.
  - `paste(x, collapse = collapsestring)` collapses the vector of characters `x` into a single character with elements separated by `collapsestring`

Examples

```
paste("A","B")

[1] "A B"

x <- c("a","b","c")

y <- c("A","B")

paste(x,y, sep = "--")

[1] "a--A" "b--B" "c--A"

paste(y,x, sep ="")

[1] "Aa" "Bb" "Ac"

paste(x)

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

paste(x,collapse="<->")

[1] "a<->b<->c"
```

Operation on Logical Vectors

A non-exhaustive list of operations on logical vectors is given below.

	Description
`x & y`	Logical and (elementwise) of x and y
`x | y`	Logical or (elementwise) of x and y
`!x`	Logical negation (elementwise) of x
`all(x)`	TRUE if all elements of x are TRUE
`any(x)`	TRUE if any element of x is TRUE

You may also use all the operations defined for numerical vectors on logical vectors: `TRUE` is considered as 1 and `FALSE` as 0.

Boolean Algebra

Table for logical AND

&	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

Table for logical OR

$\vert$	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

Table for logical negation

x	!x
TRUE	FALSE
FALSE	TRUE

Like this we may for instance conclude that if A and B are two logical statements that

$$!A\&!B \Leftrightarrow !(A|B)$$

Overview

- R Language Fundamentals
- R Datatypes
- R Objects
  - Vectors
  - Matrices
  - Lists
  - Data Frames
  - Factors
- Conversion

Overview

- R Language Fundamentals
- R Datatypes
- R Objects
  - Vectors
  - Matrices
  - Lists
  - Data Frames
  - Factors
- Conversion

Creating Vectors – Logicals

To create vectors the usual way is to invoke the concatenation operator `c`.
Logical vectors

```
x <- c(TRUE, FALSE, TRUE, FALSE)

x

[1]  TRUE FALSE  TRUE FALSE

y <- c(T,F,T,F)

y

[1]  TRUE FALSE  TRUE FALSE

str(y)

 logi [1:4] TRUE FALSE TRUE FALSE

summary(y)

   Mode   FALSE    TRUE
logical       2       2
```

Creating Vectors – Logicals

A helpful way is also to use the repeat operator `rep`:

```
x <- rep(c(TRUE, FALSE), times = 4)
x
```

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

```
y <- rep(c(TRUE, FALSE), each = 4)
y
```

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

Creating Vectors – Numerical

Just as with logical vectors you can create numerical vectors with `c` to concatenate several values

```
x1 <- c(-10, 0, 1, 12, 12.1, pi, 3)

x1

[1] -10.000000   0.000000   1.000000  12.000000  12.100000
    3.141593   3.000000
```

Alternatively, you can create ranges:

```
x2 <- 1:10

x2

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

x3 <- -10:10

x3

 [1] -10  -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1   2   3   4
      5   6   7   8   9  10
```

Creating Vectors – Numerical Sequence Operator

For more control over the behavior of ranges you may use the sequence operator `seq`:

```
x4 <- seq(-10, 10, by = 2)

x4

 [1] -10  -8  -6  -4  -2   0   2   4   6   8  10
```

or alternatively by specifying the number of elements in the range

```
x5 <- seq(-10, 10, length.out=7)

x5

[1] -10.000000  -6.666667  -3.333333   0.000000   3.333333
    6.666667  10.000000
```

Creating Vectors – Character Vectors & Complex Vectors

Just as with logical and numerical vectors you may get vectors of characters by using `c` as follow

```
names <- c("John", "Joe", "Paul")

names

[1] "John" "Joe"  "Paul"

# or

names <- rep(c("John", "Joe"), each = 3)

names

[1] "John" "John" "John" "Joe"  "Joe"  "Joe"
```

Although less frequently used R can handle complex numbers

```
mycmplx <- c(0 + 1i, 2 + 2i, 4, 0 +1i)

mycmplx

[1] 0+1i 2+2i 4+0i 0+1i

mycmplx^2

[1] -1+0i  0+8i 16+0i -1+0i
```

If you mix elements of different modes you'll obtain a character vector that contains only elements of the "covering mode":

```
mix1 <- c("Paul", TRUE, F, 12, 2.83)

mix1

[1] "Paul"  "TRUE"  "FALSE" "12"    "2.83"

str(mix1)

 chr [1:5] "Paul" "TRUE" "FALSE" "12" "2.83"
```

However, note what happens if you only mix numerical values with logical values

```
mix2 <- c(TRUE, TRUE, 2, -27, 2.3, FALSE, 0, 1, 17, pi)

mix2

 [1]   1.000000   1.000000   2.000000 -27.000000   2.300000
     0.000000   0.000000   1.000000  17.000000   3.141593

str(mix2)

 num [1:10] 1 1 2 -27 2.3 ...
```

Operations on (numerical) Vectors

R carries out operations on vectors **element wise** (unlike Matlab).

```
x <- c(1, 2, 3, 4, 5, 6)

y <- rep(c(2, 3), times = 3)

x

[1] 1 2 3 4 5 6

y

[1] 2 3 2 3 2 3

x * y  # multiply

[1]  2  6  6 12 10 18

x + y  # add

[1] 3 5 5 7 7 9

x^y  # exponentiate

[1]   1   8   9  64  25 216

x %/% y  # integer division

[1] 0 0 1 1 2 2

x %% y  # modulo aka remainder

[1] 1 2 1 1 1 0
```

Operations on Vectors – Recycling

And important concept in this context is the so called **"recycling"** behavior. Whenever a shorter vector in used in an operation with a longer vector the elements of the shorter vector are recycled until its length matches the length of the longer vector.

```
x <- rep(1,10)

x

 [1] 1 1 1 1 1 1 1 1 1 1

y <- 1:4

y

[1] 1 2 3 4

x * y


Warning in x * y: longer object length is not a multiple of shorter
     object length

 [1] 1 2 3 4 1 2 3 4 1 2

x + y


Warning in x + y: longer object length is not a multiple of shorter
     object length

 [1] 2 3 4 5 2 3 4 5 2 3
```

Accessing Elements of a vector

Elements of a vector can either be accessed by an integer vector or by a logical vector.

```
x <- c(1,12,3,12)
x[c(1,2)]
[1]  1 12
x[2:3]
[1] 12  3
x[c(TRUE,TRUE, FALSE,FALSE)]
[1]  1 12
x[c(TRUE,FALSE)]##Recycling!
[1] 1 3
```

Suppose now you want all elements that are smaller than 4:

```
relevantindices <- x < 4
relevantindices
[1]  TRUE FALSE  TRUE FALSE
x[relevantindices]
[1] 1 3
```

Vector Operations

Let x and y be two vectors

Command	Effect
`union(x, y)`	$x \cup y$
`intersect(x, y)`	$x \cap y$
`setdiff(x,y)`	$x \setminus y$
`x %in% y`	A vector of length of x consisting of booleans indicating whether each element is part of y
`unique(x)`	Unique values of x
`sort(x)`	A sorted version of x (see `?sort` for options, the sorting obviously depends on the type of x)

```
x <- c(1, 2, 5, 18)

y <- c(2, 3, 6, 8, 9, 10, 11, 12, 18)

setdiff(x, y)

[1]  1  5

setdiff(y, x)

[1]  3  6  8  9 10 11 12

x %in% y

[1] FALSE  TRUE FALSE  TRUE
```

Constructing vectors from vectors

Quite often it is also helpful to remove elements from a vector. This can be achieved by using negative indices, i.e.

```
x <- c(1, 2, 5, 18)
x[-c(1,3)]
[1]  2 18
```

Helpful in this context is the `which` function which returns the indices of values that are true in a logical vector. An alternative way to construct `setdiff(x,y)` would thus be

```
y <- c(2, 3, 6, 8, 9, 10, 11, 12, 18)
xisiny <- x %in% y
xisiny
[1] FALSE  TRUE FALSE  TRUE
indices <- which(xisiny)
indices
[1] 2 4
x[-indices]
[1] 1 5
```

Overview

- R Language Fundamentals
- R Datatypes
- R Objects
  - Vectors
  - Matrices
  - Lists
  - Data Frames
  - Factors
- Conversion

Matrices

Matrices in R are essentially 2-dimensional vectors which may consist of a single type only. Suppose you want to have

$$A = \begin{pmatrix} 2 & 9 & 3 \\ 1 & 3 & 7 \end{pmatrix}$$

which you could get in R as

```
A <- matrix(c(2,1,9,3,3,7),nrow=2,ncol=3)

A

     [,1] [,2] [,3]
[1,]    2    9    3
[2,]    1    3    7

str(A)

 num [1:2, 1:3] 2 1 9 3 3 7
```

Matrices

Alternatively, you could also have a matrix of characters, i.e.

$$B = \begin{pmatrix} \text{John} & \text{Joe} & \text{Jim} \\ \text{Jane} & \text{Julia} & \text{Jackie} \end{pmatrix}$$

which would translate to the following R code

```
B <- matrix(c("John","Jane","Joe","Julia","Jim","Jackie"),nrow=2,
    ncol=3)

B

     [,1]   [,2]    [,3]
[1,] "John" "Joe"   "Jim"
[2,] "Jane" "Julia" "Jackie"
```

Your Turn: Explore what happens if you only specify `nrow` or `ncol`, also check out the `byrow` option

Accessing Elements of Matrices

As in Matlab you may access elements of a matrix by (1-based) indices or again by logicals. Suppose A is again given as

$$A = \begin{pmatrix} 2 & 9 & 3 \\ 1 & 3 & 7 \end{pmatrix}$$

```
A <- matrix(c(2,1,9,3,3,7),nrow=2,ncol=3)

A[,1]

[1] 2 1

A[2,]

[1] 1 3 7

A[c(1,2),1]

[1] 2 1

A[c(1,3),1]

Error in A[c(1, 3), 1]: subscript out of bounds

A[c(TRUE,FALSE),1]

[1] 2
```

where the first element denotes the the row and the second the column. You can again use negative indices and ranges.

Matrix Operations

Let A and B appropriately sized matrices, x and y appropriately sized vectors and n an integer.

Command	Effect
`A%*%B`	$A \cdot B$ (matrix multiplication)
`A%*%x`	$A \cdot x$ (matrix times vector)
`t(A)`	$A' = A^T$)
`solve(A)`	A^{-1}
`solve(A, x)`	$A^{-1} \cdot x$[1]
`crossprod(x, y)`	$x \cdot y = \langle x, y \rangle$
`A%o%B`	Kronecker product of A and B
`diag(A)`	Returns the diagonal elements of A
`diag(x)`	Constructs a quadratic matrix with x on the diagonal
`diag(n)`	Gives a n-dimensional identity matrix
`eigen(A)`	Gives the eigensystem of A
`det(A)`	Gives the determinant of A
`chol(A)`	The Cholesky decomposition of A
`cbind(A, x)`	Add (attach) column x to A
`rbind(A, x)`	Add (attach) row x to A

[1] Numerically more stable than `solve(A)%*%x`

Overview

R Language Fundamentals

R Datatypes

R Objects

- Vectors
- Matrices
- Lists
- Data Frames
- Factors

Conversion

Lists

A list is loosely speaking a more general vector (similar to e.g., the Java Vector object) A list can contain objects of different types and can be easily accessed by names. Suppose you have two employees, Jane and Joe, that have the following characteristics

	Employee 1	Employee2
Name	Jane	Joe
Gender	Female	Male
Salary	62'000	60'000
Payment Group	A	G

In R you could cook this up as follows

```
emp1 <- list(name="Jane",male=FALSE,salary=62000,paymentgroup="A")
emp2 <- list(name="Joe",male=TRUE,salary=60000,paymentgroup="G")
```

Lists

If you then display an employee you will find the actual list structure

```
emp1

$name
[1] "Jane"

$male
[1] FALSE

$salary
[1] 62000

$paymentgroup
[1] "A"
```

Accessing Elements of Lists

Elements of list can be accessed differently, depending on your needs.

Command	Effect
`list[i]`	Gets sublist a position i
`list[i:j]`	Gets sublist from position i to j
`list[[i]]`	Gets the object at position i
`list$name`	Gets the object at 'name'
`names(list)`	Names of the elements of `list`

```
a <- list(1,2,3)

names(a)

NULL

names(a) <- c("elem1","elem2","elem3")

a[1]

$elem1
[1] 1

names(a)

[1] "elem1" "elem2" "elem3"
```

Accessing Elements of Lists

To understand what has been introduced on the previous slides consider the following examples:

```
emp1[1]

$name
[1] "Jane"

emp1[1:2]

$name
[1] "Jane"

$male
[1] FALSE

emp1[[1]]

[1] "Jane"

emp1[[1:2]]

Error in emp1[[1:2]]: subscript out of bounds

emp1$salary

[1] 62000
```

Accessing Elements of Lists

Your Turn: Enclose all of the 5 statements of above in `str` command. See what the differences are, i.e.

```
str(emp1[1])

str(emp1[1:2])

str(emp1[[1]])

str(emp1[[1:2]])

str(emp1$salary)
```

Overview

- R Language Fundamentals
- R Datatypes
- R Objects
  - Vectors
  - Matrices
  - Lists
  - Data Frames
  - Factors
- Conversion

Data Frames

Data frames is what you will probably encounter most frequently when using R for applied work. A data frame is (typically) a two-dimensional structure whose columns may contain different information types, i.e.

ID	Name	Height	Male
12	Jane	170	FALSE
18	Joe	178	TRUE
99	Al	193	TRUE
⋮	⋮	⋮	⋮

Typically, you do not construct data frame manually, but they are the result of some process (reading data, collecting data, etc.).
Dataframes are strictly speaking simply lists composed of vectors (columns) of different datatypes with some added functionality.

Dealing with Data Frames

Command	Effect
`df[i,]`	The *i*th row of `df`
`df[,j]`	The *j*th column of `df`
`df$colname`	The column with name ‘colname’
`df[, "colname"]`	The column with name ‘colname’
`with(df,colname)`	The column with name ‘colname’
`dim(df)`	Dimension of a `df`
`nrow(df)`	Dimension of a `df`
`ncol(df)`	Dimension of a `df`
`head(df)`	The first 6 rows
`tail(df)`	The last 6 rows
`head(df, x)`	The first `x` rows
`tail(df, x)`	The last `x` rows
`names(df)`	The column names of a data frame.
`summary(df)`	Summary of a `df`

As with vectors and matrices you can also use ranges `i:j` or negative indices.

Dealing with Data Frames

Suppose you have a data frame named `persons` that contains the name, the height, the gender and the id of several people.

```
head(persons)
```

```
   id   name height  male
1  12   Jane    170 FALSE
2  18    Joe    178  TRUE
3  99     Al    193  TRUE
4 123 Martha    172 FALSE
5   7  Peter    182  TRUE
6  74  Julie    167 FALSE
```

```
summary(persons)
```

```
       id             name               height         male        
 Min.   :  7.0   Length:10          Min.   :166.0   Mode :logical  
 1st Qu.: 32.0   Class :character   1st Qu.:170.5   FALSE:6        
 Median : 82.5   Mode  :character   Median :173.5   TRUE :4        
 Mean   : 71.2                      Mean   :175.2                  
 3rd Qu.:102.0                      3rd Qu.:177.8                  
 Max.   :123.0                      Max.   :193.0                  
```

Dealing with Data Frames

Display only people that are no taller than 170:

```
persons[persons$height <= 170,]
```

```
   id   name height  male
1  12   Jane    170 FALSE
6  74  Julie    167 FALSE
7 103 Jackie    166 FALSE
```

Display only females

```
persons[persons$male == FALSE,]
```

```
    id   name height  male
1   12   Jane    170 FALSE
4  123 Martha    172 FALSE
6   74  Julie    167 FALSE
7  103 Jackie    166 FALSE
9   83  Cathy    174 FALSE
10 111 Maggie    173 FALSE
```

Dealing with Data Frames – Subsets

Display only people that are no taller than 170:

```
subset(persons,height<=170)
```

```
   id   name height  male
1  12   Jane    170 FALSE
6  74  Julie    167 FALSE
7 103 Jackie    166 FALSE
```

Display only females

```
subset(persons,male==FALSE)
```

```
    id   name height  male
1   12   Jane    170 FALSE
4  123 Martha    172 FALSE
6   74  Julie    167 FALSE
7  103 Jackie    166 FALSE
9   83  Cathy    174 FALSE
10 111 Maggie    173 FALSE
```

Dealing with Data Frames – Subsets

The `subset` command is often handier since it allows for easier notation. `subset` returns the rows of the data frame that match the criteria given. Moreover, criteria can be combined with logical operators:

```
## all records that are male AND shorter than 180
subdataframe <- subset(persons,male==TRUE & height <= 180)
subdataframe$name
[1] "Joe" "Bob"
## all records that are either female OR greater than 180
subdataframe <- subset(persons,male==FALSE | height >= 185)
subdataframe$name
[1] "Jane"   "Al"     "Martha" "Julie"  "Jackie" "Cathy"  "Maggie"
```

Dealing with Data Frames – Subsets

Combining Logical Statements can also be done. Suppose you want to get all records that are

(female **AND** greater than 170)
OR
(male **AND** greater than 185)

```
subdataframe <- subset(persons,(male == F & height>=170) | (male==T
     & height>=185))

nrow(subdataframe)

[1] 5

subdataframe$name

[1] "Jane"   "Al"      "Martha" "Cathy"  "Maggie"
```

Dealing with Data Frames – Modifying Data

Suppose that in the persons data frame the height of all men has been measured on scale which always gives 2cm too much. You want to correct this in the data frame.

```
## create a copy
pmod <- persons
## subtract 2 cm
pmod[pmod$male==F,"height"] <- pmod[pmod$male==F,"height"] - 2
```

or alternatively

```
## create a copy
pmod <- persons
## subtract 2 cm
pmod$height[pmod$male==F] <- pmod$height[pmod$male==F]-2
```

Dealing with Data Frames – Modifying Data Frames

Suppose you have the weights of these persons in a vector called `mweights` (measured weights). To add it to the data frame you may proceed as follows

```
extpersons <- persons

extpersons$weight <- mweights

head(extpersons ,2)

  id name height  male weight
1 12 Jane    170 FALSE     62
2 18  Joe    178  TRUE     85

## or

extpersons <- persons

extpersons[,"weight"]<- mweights

## or

extpersons <- cbind(persons ,mweights)

head(extpersons ,2)

  id name height  male mweights
1 12 Jane    170 FALSE       62
2 18  Joe    178  TRUE       85
```

Overview

R Language Fundamentals

R Datatypes

R Objects
- Vectors
- Matrices
- Lists
- Data Frames
- Factors

Conversion

Factors

The reason for having factors is induced by the following taxonomy of attributes. An attribute can be of any of the three types

Scale	Example
Nominal	gender, colors, etc.
Ordinal	grades, day of week, etc.
Interval	Age, time, etc.

Interval scales are mapped to R by `numeric`, however, to account for either a nominal or ordinal scale of a variable R has the `factor` object.

Factors vs. vectors of characters

```
x <- c("blue", "green", "blue", "blue", "red")

x

[1] "blue"  "green" "blue"  "blue"  "red"

str(x)

 chr [1:5] "blue" "green" "blue" "blue" "red"

summary(x)

   Length     Class      Mode 
        5 character character 

xf <- factor(c("blue", "green", "blue", "blue", "red"))

xf

[1] blue  green blue  blue  red  
Levels: blue green red

str(xf)

 Factor w/ 3 levels "blue","green",..: 1 2 1 1 3

summary(xf)

 blue green   red 
    3     1     1 
```

Ordered Factors

Consider for instance weekdays as example for an ordinal scale.

```
wdays <- c("Sat", "Mon", "Tue", "Wed", "Thu", "Sat", "Sun", "Mon",
    "Tue", "Wed")

fwdays <- factor(wdays)

levels(fwdays)

[1] "Mon" "Sat" "Sun" "Thu" "Tue" "Wed"
```

However, for weekdays it makes sense to maintain their natural order. This can be done as follows

```
ofwdays <- factor(wdays, levels=c("Mon","Tue","Wed","Thu","Fri","
    Sat","Sun"))

ofwdays

 [1] Sat Mon Tue Wed Thu Sat Sun Mon Tue Wed
Levels: Mon Tue Wed Thu Fri Sat Sun

levels(ofwdays)

[1] "Mon" "Tue" "Wed" "Thu" "Fri" "Sat" "Sun"
```

Overview

R Language Fundamentals

R Datatypes

R Objects
- Vectors
- Matrices
- Lists
- Data Frames
- Factors

Conversion

Going round – Converting objects

Since each object in R has its own type it sometimes required to convert from one to another. A non-exhaustive list is given below

`Command`	Effect
`as.character`	converts to a character objects (from e.g, numerical or factor)
`as.numeric`	converts to numerical object
`as.matrix`	converts an object (data frame) to a matrix (if possible, watch out if the original object consists of different types)
`as.vector`	converts to a vector (from e.g., a matrix)
`as.data.frame`	converts an object (typically a matrix) to a data frame
`as.Date`	converts an object (typically a character object) to a Date object (not yet introduced)

Converting Objects

Some hints:

- Converting a factor to a numerical value: Make sure to first convert it to a character vector and only then to a numerical vector, otherwise you will just get the factor levels.
- To see all possible conversions: `methods("as")`