STAT 40001/STAT 50001 Statistical Computing

Lecture 2

Department of Mathematics and Statistics

Outline-R objects

The fundamental data type in R is the vector.

- **Null (empty object): Null**
- **Logical (Boolean): TRUE, FALSE or T, F**
- **Numeric (real number): 1,2,3, pi, 1e-5**
- **Complex(complex number): 2+i, 2i**
- **Character(chain of characters): "ABC", "hi"**

In order to know the mode of an object, say x, in R we can use

> mode(x)

Example:

```
> x=c(1,2,3,4,5,6)
>  is.null(x)
[1] FALSE

>  is.numeric(x)
[1] TRUE

> is.character(x)
[1] FALSE

> y=c("A","B","C")
> is.character(y)
[1] TRUE

> z=c(3+2i,2-3i, 2i,7i)
> is.complex(z)
[1] TRUE
```

```
>  is.logical(x)
 [1] FALSE

>  is.complex(x)
[1] FALSE

> is.numeric(y)
[1] FALSE

> is.numeric(z)
 [1] FALSE
```

Numeric Vectors

- Use collect function or concatenate: **c**
- Construction by **sequence operator**
- Construction by **rep function**
- Construction by **scan function**

Example:

```
> x=c(2, 4, 5, 6.8, 7.2) # Numeric vector with 5 entries
> x
[1] 2.0 4.0 5.0 6.8 7.2
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(1,10, by=3) # We can simply use seq(1,10,3)
[1]  1  4  7 10
> seq(1,50,length=8)
[1]  1  8 15 22 29 36 43 50
> rep(1,6)
[1] 1 1 1 1 1 1
> rep(c(1,2,3,4), each=3)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4
> rep(c(1,2,3,4), 3)
 [1] 1 2 3 4 1 2 3 4 1 2 3 4
```

Example: scan

```
> y
[1] 3 5 7
> y<-scan()
1: 2
2: 5
3: 6
4: 7
5: 8
6: 9
7: 9
8:
Read 7 items
> y
[1] 2 5 6 7 8 9 9
```

Character Vectors

It is possible to create character vectors in the same way as the numeric vectors with functions **c** or **rep**

```
> x<-c("A", "BB", "CCC", "Example")
>  x
[1] "A"       "BB"      "CCC"     "Example"
>  y<-rep("A", 10)
> y
 [1] "A" "A" "A" "A" "A" "A" "A" "A" "A" "A"
> z<-rep('STAT',8)
> z
 [1] "STAT" "STAT" "STAT" "STAT" "STAT" "STAT" "STAT" "STA
 > LETTERS[seq( from = 1, to = 10 )]
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
 > letters[seq( from = 1, to = 10 )]
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

Note that in R " " and ' ' are essentially the same.

More about Character Vectors

```
> LETTERS
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N"
[15] "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
> letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
[15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
> which(letters=="j")
[1] 10
```

For the purposes of printing you might want to suppress the quotes that appear around character strings by default. The function to do this is called noquote:

```
> noquote(letters)
 [1] a b c d e f g h i j k l m n o p q r s t u v w x y z
```

In order to repeat character string use strrep

```
> strrep(c("A", "B", "C"), 1 : 3)
[1] "A"   "BB"  "CCC"
>
```

Logical Vectors

Boolean vectors are usually generated with logical operations: “ > ”, “ >= ”, “ < ”, “ <= ”, “ == ”, “ != ”, “|”, “&” etc.

```
> 1>0
[1] TRUE
> 2<=5
[1] TRUE
> 2>5
[1] FALSE
> 2>=5
[1] FALSE
> 7!=10
[1] TRUE
```

Vector of logical values

```
> log_values <- c(TRUE, FALSE, TRUE, FALSE)
> log_values
[1]  TRUE FALSE  TRUE FALSE
##########Sorting and changing items
> fruits <-c("banana","apple","orange","mango", "lemon")
> numbers <- c(13, 3, 5, 7, 20, 2)
> sort(fruits)  # Sort a string
[1] "apple"  "banana" "lemon"  "mango"  "orange"
> sort(numbers) # Sort numbers
[1]  2  3  5  7 13 20
> fruits <-c("banana","apple","orange","mango", "lemon")
> # Change "banana" to "pear"
> fruits[1] <- "pear"
> # Print fruits
> fruits
[1] "pear"   "apple"  "orange" "mango"  "lemon"
```

cat() function in R

The cat() function in R can be used to concatenate together several objects in R.
Syntex

```
cat(..., file = "", sep = " ", append = FALSE))
```

```
>cat("This is Fall semester")
>cat("This", "is", "Fall", "semester")
>cat("This", "is", "Fall", "semester", sep="-")
>cat("This", "is", "Fall", "semester", sep="\n")
>cat("This","is", "Fall", sep="\n", file="data.csv")
#append results of this concatenation to first file
>cat("how","are","you",sep="\n",file="data.csv", append=T)
```

One can use paste() function which will create a character string

```
> paste("This","is", "Fall", "semester")
[1] "This is Fall semester"
```

Using all() and any()

```
> x <- 1:10
> any(x > 8)
[1] TRUE
> any(x > 88)
[1] FALSE
> all(x > 88)
[1] FALSE
> all(x > 0)
[1] TRUE
```

Selecting Part of a Vector

Selections are made using the selection operator [ ] and a selection vector:> x[indexvector]

```
> x=c(2,4,6,8,4,6,7,8,9,0,12,13,14,15)
> x
 [1]  2  4  6  8  4  6  7  8  9  0 12 13 14 15
> x[5]
[1] 4
> x[-5]
 [1]  2  4  6  8  6  7  8  9  0 12 13 14 15
> v=1:15
> v
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> v[(v<4)|(v>12)] # | means OR
[1]  1  2  3 13 14 15
> v[(v<4)&(v>12)] # & means AND
integer(0)
```

which() and match()

- **which()** Indices of a vector where the condition is TRUE
- **which.max()** Location of the maximum element of a numeric vector
- **which.min()** Location of the minimum element of a numeric vector
- **match()** First position of an element in a vector

```
> x<-c(4,7,2,12,9,0)
> which(x>4)
[1] 2 4 5
> which.max(x)
[1] 4
> x[which.max(x)] # Exact value of the maximum value
[1] 12
> y <- rep(1:5, times=5:1)
> match(1:5,y)
[1]  1  6 10 13 15
```

which() and match()

```
> x<-c(4,7,2,12,9,0)
> y <- rep(1:5, times=5:1)
> x
[1]  4  7  2 12  9  0
> y
 [1] 1 1 1 1 1 2 2 2 2 3 3 3 4 4 5
> which(x>4)
[1] 2 4 5
> which.max(x) # Location of the maximum x value
[1] 4
> which.min(x)# Location of the minimum x value
[1] 6
> x[which.max(x)] # Value of the maximum x value
[1] 12
> match(1:5,y)# It matches the locations
[1]  1  6 10 13 15
```

match function

The match function answers the question "Where do the values in the second vector appear in the first vector?"

```
> first<-c(5,8,3,5,3,6,4,4,2,8,8,8,4,4,6)
> second<-c(8,6,4,2)
> match(first,second)
 [1] NA  1 NA NA NA  2  3  3  4  1  1  1  3  3  2
```

The first thing to note is that match produces a vector of subscripts (index values) and that these are subscripts within the second vector. The length of the vector produced by match is the length of the first vector (15 in this example). If elements of the first vector do not occur anywhere in the second vector, then match produces NA.

Testing Vector Equality

```
> x <- 1:3
> y <- c(1,3,4)
> x = = y
[1] TRUE FALSE FALSE
> identical(x,y)
[1] FALSE
```

Note that ":" produces integers while "c()" produces floating-point numbers.

```
> x=c(1,2,3); y=1:3
[1] 1 2 3
[1] 1 2 3
> identical(x,y)
[1] FALSE
> typeof(x)
[1]  "double" # Double is same as numeric
> typeof(y)
[1] "integer"
```

Peculiarites of floating-point numbers and Integer

```
> 0.45 == 3*0.15
[1] FALSE
> round(0.45 == 3*0.15)
[1] 0
######################
> (0.5 - 0.3) == (0.3 - 0.1)
[1] FALSE
> all.equal((0.5 - 0.3), (0.3 - 0.1))
[1] TRUE
#########################
> is.integer(7)
[1] FALSE
> as.integer(7)
[1] 7
> round(7)==7
[1] TRUE
```

Missing Value

For a number of reasons, certain elements of data may not be collected during and experiment or study. In R, missing values are represented by the symbol **NA** (not available) but impossible values (e.g., dividing by zero) are represented by the symbol **NaN** (not a number) and **Inf** for infinity.

```
> var(5) # Variance of a single observation
[1] NA
> log(-2)
[1] NaN
Warning message:
In log(-2) : NaNs produced
> exp(1e10)
[1] Inf
```

Detecting Missing Value

```
> x<-c(1,2,3,4,5,6,7,8,4,5,7)
> is.na(x)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> y<-c(1,4,NA,6,9,NA,7)
> is.na(y)
[1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
>y<-c(1,4,NA,6,9,NA,7)
> any(is.na(y)) # checking if there is any missing value
[1] TRUE
>which (is.na(y)) # Which one is NA?
[1] 3  6      # Answer: the third one and sixth one
```

Dealing with Missing Value

The user can specify if **NA** should be last or first in a sorted order by indicating TRUE or FALSE for the **na.last** argument.

```
>y<-c(1,4,NA,6,9,NA,7)
> sort(y, na.last = TRUE)
[1]  1  4  6  7  9 NA NA
```

na.omit and **na.exclude** returns the object with observations removed if they contain any missing values

```
> x <- c(88,NA,12,168,13)
[1]  88  NA  12 168  13
> y=x[!is.na(x)]
> y
[1]  88  12 168  13
> mean(x)
[1] NA
> mean(x, na.rm=T)
[1] 70.25
> mean(na.omit(x))
[1] 70.25
```

NA Vs. NULL

Once can use "NULL" if the values really are counted as nonexistent

```
> x <- c(88,NULL,12,168,13) # R has skipped the 2nd observa
> mean(x)
[1] 70.25
> u <- NULL
> length(u)
[1] 0
> v <- NA
> length(v)
[1] 1
```

The sample Function

This function shuffles the contents of a vector into a random sequence while maintaining all the numerical values intact. It is extremely useful for randomization in experimental design, in simulation and in computationally intensive hypothesis testing.

Example: consider a data set: 8, 3, 5, 7, 6, 6, 8, 9, 2, 3, 9, 4, 10, 4, 11

```
> x
 [1]  8  3  5  7  6  6  8  9  2  3  9  4 10  4 11
> sample(x)
 [1]  9  5 11  7  3  4  8  2  9  3  8  4  6  6 10
> sample(x)
 [1]  4  2  5  6  3 10  8  8  9  7 11  6  4  3  9
```

The order of the values is different each time that sample is invoked, but the same numbers are shuffled in every case. This is called sampling without replacement. You can specify the size of the sample you want as an optional second argument:

```
> sample(x, 5)
[1]  2  4 11  4  8
```

The option "replace=T" allows for sampling with replacement

Examples

```
> x=c(8, 3, 5, 7, 6, 6, 8, 9, 2, 3, 9, 4, 10, 4, 11)
> sample(x, replace=T)
 [1]  6  4 11  8  3  2  2  3  8  9  7  7  9  9  9
> sample(x,5, replace=T)
[1] 11  3  3  8  3
```

Factor

Factors offer an alternative way to store character data. For example, a factor with four elements and having the two levels control and treatment can be created using

```
> group <- c("control", "treatment", "control", "treatment"
> group
[1] "control"   "treatment" "control"   "treatment"
> grp <- factor(group)
> grp
[1] control   treatment control   treatment
Levels: control treatment
> as.integer(group)
[1] NA NA NA NA
Warning message:
NAs introduced by coercion
> as.integer(grp)
[1] 1 2 1 2
```

Creating a function

```
> addition<- function(a,b,c) {
  a+b+c
}
> addition(1,2,3)
[1] 6

> DegreeToFahrenheit<- function(F) {
  (5/9)*(F-32)
}

> DegreeToFahrenheit(100)
[1] 37.77778
```

Loops and Repeats

The classic, Fortran-like loop is available in R. The syntax is a little different, but the idea is identical; you request that an index, i, takes on a sequence of values, and that one or more lines of commands are executed as many times as there are different values of i. Here is a loop executed five times with the values of i from 1 to 5: we print the square of each value:

```
> for (i in 1:5) print(i^2)
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

The if() statement

The if()statement allows us to control which statements are executed, and sometimes this is more convenient.

Syntax:

if (condition) commands when TRUE

if(condition)commands when TRUE else commands when FALSE

$$|x| = \begin{cases} x, & \text{if } x \geq 0, \\ -x, & \text{if } x < 0, \end{cases}$$

```
absolute<- function(x) {
  ifelse(x < 0, -x, x)
}
> absolute(-3:3)
[1] 3 2 1 0 1 2 3
################################
> abs(-3:3)
 [1] 3 2 1 0 1 2 3
```

More Example

```
> x=c("a","b","c", "d")
> x
[1] "a" "b" "c" "d"
> for(i in 1:4){
+ print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

```
> x<-matrix(1:6,2,3)
> x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
> for(i in seq_len(nrow(x))){
+ for(j in seq_len(ncol(x))){
+ print(x[i,j])
+ }
```

while Loop

While loops begin by testing a condition. If it is true, then they execute the loop body

```
> x<-0
> while(x<10){
+ print(x)
+ x<-x+1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
```

More Example

```
> f<-function(a, b) {
+ a^2
+ }
> f(5)
[1] 25
```

```
>  f<-function(a, b) {
+ a^2+b^2
+ }
> f(4,5)
[1] 41
```