R BASIC TOOLS - From APS 2015 and ISPP 2018 Workshops

Neil McRoberts and Paul Esker

Basic introduction to R.

Some online resouces:

http://www.introductoryr.co.uk/R_Resources_for_Beginners.html

http://www.statmethods.net/index.html

http://www.ats.ucla.edu/stat/r/

http://www.r-tutor.com/r-introduction

####**R as a calculator**

R has basics operations to add, subtract, mulitply and divide. Also, R defines certain caluclations, such as pi, based on alpha-numeric nomenclature.

```
7+2

## [1] 9

5/2

## [1] 2.5

6*9

## [1] 54

1000-89

## [1] 911

12/pi

## [1] 3.819719
```

####**Creating Objects** This first step enables us to start creating data vectors, or more simply a list of information of interest.

```
X <- 43
X

## [1] 43

x <- 23
x
```

```
## [1] 23

ls() ### To see that we have both "X" and "x"

## [1] "x" "X"

Y <- c(3,2,6)
Y

## [1] 3 2 6

Y2 <- c(3,X,x)
Y2

## [1]  3 43 23

Y3 <- c(Y,Y2)
Y3

## [1]  3  2  6  3 43 23
```

####**Sequences of numbers**

The following code illustrates different examples of how one can define a list (vector) of information.

```
Set1 <- c(1,2,3,4,5,6,7,8,9)
Set2 <- 1:9
Set3 <- seq(1,10)
Set4 <- seq(1,10,0.5)
### Set4 comes from the following code, more formally defined:
seq(from=1,to=10,by=0.5)
Set4

##  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5
## [15]  8.0  8.5  9.0  9.5 10.0

Set1

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

Set2

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

Set3

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

####**Vectors of data and information**

```
X <- c(4,2,7)
X2 <- seq(3,6)
```

```r
# Combining vectors of information: cbind (column) or rbind (row)
X <- c(1,2,3)
Y <- c(7,8,9)
Z <- cbind(X,Y)
Z

##      X Y
## [1,] 1 7
## [2,] 2 8
## [3,] 3 9

Z2 <- rbind(X,Y)
Z2

##   [,1] [,2] [,3]
## X    1    2    3
## Y    7    8    9

# Understanding how R interprets different types of vectors
example=c(1,2,3,4,5,6,7,8,9,10)
class(example) #numeric

## [1] "numeric"

new.example=c("A","B","C","D")
class(new.example) #character

## [1] "character"
```

####**Matrices and arrays** Matrices and arrays are ways to organize data into a collection of data entries (rows and columns, along with subgroups of information).

```r
# Matrix
Mat1 <- matrix(1,ncol=3,nrow=4) #Matrix of 1's, with 4 rows and 3 columns
Mat1

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

#Let's use X and Y previously defined to make a matrix of data
Z <- c(X,Y)
Z

## [1] 1 2 3 7 8 9

Mat2 <- matrix(Z,ncol=2,byrow=F)
Mat2

##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
## [3,]    3    9

class(Mat2)

## [1] "matrix"
```

####**Creating a matrix using “plant pathological” data**

```
disease=matrix(c(1,1,1,1,2,2,2,2,5,15,35,55,11,30,61,75),ncol=2,nrow=8)
colnames(disease)=c("Trt","Sev")
disease

##      Trt Sev
## [1,]   1   5
## [2,]   1  15
## [3,]   1  35
## [4,]   1  55
## [5,]   2  11
## [6,]   2  30
## [7,]   2  61
## [8,]   2  75

is.matrix(disease)

## [1] TRUE
```

Evaluating a vector - basic methods Basic calculations like: mean() median() var() summary() - Notice here we are provided with a basic 5 (6) number summary.

```
X <- rnorm(20,mean=5,sd=2)
#rnorm() generates a random vector of 20 observations, each from a mean=5
with a standard deviation of 2

length(X)

## [1] 20

mean(X)

## [1] 4.382515

median(X)

## [1] 4.216972

var(X)

## [1] 4.868037

summary(X)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -0.7865  3.5401  4.2170  4.3825  5.7774  8.6711
```

```
#Another example
Rendimiento<-
c(4.2,4.94,4.45,4.36,3.5,4.17,5.4,4.55,5.75,5.15,4.4,3.9,2.82,3.14,3.8,3.74,4
.43,2.92,4.82,3.9,4.5,4.57,5.32,4.35)
mean(Rendimiento)

## [1] 4.295

var(Rendimiento)

## [1] 0.5642

sd(Rendimiento)

## [1] 0.7511325

length(Rendimiento)

## [1] 24

summary(Rendimiento)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   2.820   3.875   4.380   4.295   4.633   5.750
```

####**Matrix calculations**

The same functions can be applied to matrices, but it is important to understand that with some of the functions, for example var(), the calculation is based on the columns and comparisons (covariance as one idea) between them.

```
Mat2 <- matrix(Z,ncol=2,byrow=F)
Mat2

##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
## [3,]    3    9

length(Mat2) # Less straightforward in 2 dimensions

## [1] 6

dim(Mat2) # Rows and columns

## [1] 3 2

mean(Mat2) # Can you see how this was calculated?

## [1] 5

median(Mat2)

## [1] 5
```

```
var(Mat2) # Notice now we are working in 2 dimensions for this calculation -
variances-covariances-correlations

##      [,1] [,2]
## [1,]    1    1
## [2,]    1    1

summary(Mat2) # By column

##        V1            V2     
##  Min.   :1.0   Min.   :7.0  
##  1st Qu.:1.5   1st Qu.:7.5  
##  Median :2.0   Median :8.0  
##  Mean   :2.0   Mean   :8.0  
##  3rd Qu.:2.5   3rd Qu.:8.5  
##  Max.   :3.0   Max.   :9.0  

## Another example (much larger)
Mat3 <- matrix(seq(1,50),ncol=2,byrow=T)
Mat3

##       [,1] [,2]
##  [1,]    1    2
##  [2,]    3    4
##  [3,]    5    6
##  [4,]    7    8
##  [5,]    9   10
##  [6,]   11   12
##  [7,]   13   14
##  [8,]   15   16
##  [9,]   17   18
## [10,]   19   20
## [11,]   21   22
## [12,]   23   24
## [13,]   25   26
## [14,]   27   28
## [15,]   29   30
## [16,]   31   32
## [17,]   33   34
## [18,]   35   36
## [19,]   37   38
## [20,]   39   40
## [21,]   41   42
## [22,]   43   44
## [23,]   45   46
## [24,]   47   48
## [25,]   49   50

head(Mat3) # gives the first 6 rows by default
```

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

```
head(Mat3, n=10)
```

```
##       [,1] [,2]
##  [1,]    1    2
##  [2,]    3    4
##  [3,]    5    6
##  [4,]    7    8
##  [5,]    9   10
##  [6,]   11   12
##  [7,]   13   14
##  [8,]   15   16
##  [9,]   17   18
## [10,]   19   20
```

```
tail(Mat3) # gives the last 6 rows by default
```

```
##       [,1] [,2]
## [20,]   39   40
## [21,]   41   42
## [22,]   43   44
## [23,]   45   46
## [24,]   47   48
## [25,]   49   50
```

####**Working with matrices - operations**

```
Mat2
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
## [3,]    3    9
```

```
5*Mat2
```

```
##      [,1] [,2]
## [1,]    5   35
## [2,]   10   40
## [3,]   15   45
```

```
5+Mat2
```

```
##      [,1] [,2]
## [1,]    6   12
## [2,]    7   13
## [3,]    8   14
```

```
Mat2[,1] <- Mat2[,1] + 100 # Changing just the first column of Mat2
Mat2
```

```
##      [,1] [,2]
## [1,]  101    7
## [2,]  102    8
## [3,]  103    9
```

####**Data frames**

More commonly, we will employ a database created in another program, for example Excel. In this case, we are working with a data frame that has mixed information, such as alpha-numerica, data, etc. Nonetheless, we can handle this information in R like the previously examples. In this part, we will take a matrix and turn this into a data.frame, but after, we will see generically, examples of introducing the data from file.

```
disease
```

```
##      Trt Sev
## [1,]   1   5
## [2,]   1  15
## [3,]   1  35
## [4,]   1  55
## [5,]   2  11
## [6,]   2  30
## [7,]   2  61
## [8,]   2  75
```

```
GreatData <- data.frame(disease)
GreatData
```

```
##   Trt Sev
## 1   1   5
## 2   1  15
## 3   1  35
## 4   1  55
## 5   2  11
## 6   2  30
## 7   2  61
## 8   2  75
```

```
names(GreatData)
```

```
## [1] "Trt" "Sev"
```

```
# Renaming columns
names(GreatData) <-c('Variety', 'Severity')
GreatData
```

```
##   Variety Severity
## 1       1        5
## 2       1       15
## 3       1       35
## 4       1       55
## 5       2       11
## 6       2       30
## 7       2       61
## 8       2       75
```

```
GreatData$Variety
```

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

```
GreatData$Severity
```

```
## [1]  5 15 35 55 11 30 61 75
```

Integrating Functions In R, as well as in many other programming languages, we can combine functions to simply the number of lines of code.

```
dap<-c(7,14,21,28,35,42,49)
dis1<-c(0,5,7,25,55,60,75)
dis2<-c(3,14,33,50,65,75,78)

progress<-data.frame(cbind(dap,dis1,dis2)) #We combined data.frame() and 
cbind()
progress
```

```
##   dap dis1 dis2
## 1   7    0    3
## 2  14    5   14
## 3  21    7   33
## 4  28   25   50
## 5  35   55   65
## 6  42   60   75
## 7  49   75   78
```

```
class(progress)
```

```
## [1] "data.frame"
```

####**Lists**

Understanding how R interprets and formats your data is critical since at times you will need to identify specific components of an output for further analyses.

```
L28 <- list(c(1,2,3),1000,seq(1,2,.1))
L28
```

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

```
##
## [[2]]
## [1] 1000
##
## [[3]]
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

```
L28[[3]] # third component of list
```

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

```
L28[[3]][4] # fourth entry in third component of list
```

```
## [1] 1.3
```

####**Illustration of a list of information**

```
field.work=list(loc="Janesville",year=2010,field="Soybean",trts=c("A","B","C"
),assess=c(7,14,21,28,35,42))

field.work
```

```
## $loc
## [1] "Janesville"
##
## $year
## [1] 2010
##
## $field
## [1] "Soybean"
##
## $trts
## [1] "A" "B" "C"
##
## $assess
## [1]  7 14 21 28 35 42
```

```
names(field.work)
```

```
## [1] "loc"    "year"   "field"  "trts"   "assess"
```

```
field.work$field
```

```
## [1] "Soybean"
```

####**Logical operators**

```
8 < 10 # Try this
```

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

```
8 == 10 # The double equal signs are used for logical statements
```

```
## [1] FALSE

8 != 10 # The exclamation point means ‘not’

## [1] TRUE

X <- 1:10
X < 8

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

X < 8 & X > 3 # The ampersand means ‘and’, both must be true

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

X < 3 | X > 8 # The ‘|’ means ‘or’, either must be true

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

sum(X < 8)/10 * 100

## [1] 70
```

####**Character Data**

```
A1 <- c('Severity', 'Yield')
A1[1]

## [1] "Severity"

A2 <- paste('Disease', 'Severity')
A2

## [1] "Disease Severity"

A3 <- paste('B', 1:10, sep='') # specifies no space between betwen the
characters
A3

##  [1] "B1"  "B2"  "B3"  "B4"  "B5"  "B6"  "B7"  "B8"  "B9"  "B10"

A4 <- paste('B', 1:10, sep='-') # a dash goes between the characters
A4

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

D1 <- 'Mississippi'
substring(D1, 1,4) # takes letters 1 through 4

## [1] "Miss"

C1 <- paste('B', 1:10, sep=' ')
C1

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

```
substring(C1,1,1)

##  [1] "B" "B" "B" "B" "B" "B" "B" "B" "B" "B"

substring(C1,1,2)

##  [1] "B " "B " "B " "B " "B " "B " "B " "B " "B " "B "

substring(C1,1,3)

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

substring(C1,1,4) #Where is the difference with the previous example?

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

Indices for Selecting Subsets

```
# Suppose we have a set of labels for experimental units
D5n <- rep(1:10,3)
D5c <- c(rep('A',10), rep('B',10), rep('C',10))
D5 <- paste(D5c,D5n,sep='')
D5

##  [1] "A1"  "A2"  "A3"  "A4"  "A5"  "A6"  "A7"  "A8"  "A9"  "A10" "B1"
## [12] "B2"  "B3"  "B4"  "B5"  "B6"  "B7"  "B8"  "B9"  "B10" "C1"  "C2"
## [23] "C3"  "C4"  "C5"  "C6"  "C7"  "C8"  "C9"  "C10"

# We can make an index to select only those in treatment A
Aindex <- substring(D5,1,1) == 'A'
# Suppose this is the corresponding list of yields
Yield <- 1:30
# We can apply the logical index to select only those yields corresponding to
treatment A
Yield[Aindex]

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

####**Loops**

Many times we are interested in repeating some calculations. In R, there are many methods to do this, including the use of loops. We will also see some other methods that improve programming performance when we are interested in repeating a specific function many times.

```
K1 <- c(4,2,8,5)
L1 <- c(1,3,4,2)
M1 <- 0*1:4  # This in the object where we will place the answer to our query
M1

## [1] 0 0 0 0
```

```
# This loop finds the maximum of K1 and L1 at each position
for (j in 1:4){
  M1[j] <- max(K1[j],L1[j])
}

M1

## [1] 4 3 8 5
```

####**Apply a function**

While loops work well for some functions and programming, they are rather inefficient for large operations. In R, we can take advantage of the functions: apply(), lapply() and tapply().

Using the help() options, we can see that:

apply() = returns a vector or array or list of values obtained by applying a function to margins of an array or matrix

lapply() = returns a list of the same length of X, where each element is the result of applying a function to the corresponding element of X

tapply() = apply a function to each cell a ragged array, meaning that the function is applied to each, non-empty group of values given by a unique combination of the levels of certain factors

```
#apply() - works on rows or columns

group1<-rnorm(10,5,2)
group2<-rnorm(10,10,5)
group3<-rnorm(10,15,7)

example.apply<-cbind(group1,group2,group3)
apply(example.apply, MARGIN=2, mean)

##    group1    group2    group3 
##  4.062398  9.758647 14.255837

apply(example.apply, MARGIN=2, sd)

##   group1   group2   group3 
## 2.376613 4.004573 8.995733

apply(example.apply, MARGIN=2, function (x) sd(x)/mean(x))

##    group1    group2    group3 
## 0.5850270 0.4103614 0.6310210

#lapply() - works on a list
L28 <- list(c(1,2,3),1000,seq(1,2,.1))
L28
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] 1000
##
## [[3]]
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

```
lapply(L28,mean)
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 1000
##
## [[3]]
## [1] 1.5
```

```
#tapply() - works to summarize information by some defined factor
factor<-rep(c("A","B","C","D","E"), each=2)
tapply(example.apply[,1], factor, mean) #Summarizing the first column, by
factor
```

```
##        A        B        C        D        E
## 1.122121 4.135168 2.565367 6.125899 6.363437
```

```
tapply(example.apply[,1], factor, sd) #Summarizing the first column, by
factor
```

```
##         A         B         C         D         E
## 0.4134547 0.4139875 3.0603428 0.4798451 0.0861750
```

```
tapply(example.apply[,1], factor, function (x) sd(x)/mean(x)) #CV by factor
```

```
##          A          B          C          D          E
## 0.36845806 0.10011382 1.19294558 0.07833057 0.01354221
```