WRITING FUNCTIONS IN R

Manpreet S. Katari

LEARNING OBJECTIVES

After completing this module, you will be able to:

- Write your own functions in R
- Create loops and if-else statements to control the flow of your code
- Use the apply function to perform operations in parallel
- Create plots
- Manipulate and search for string patterns

Why write your own function?

- Provide your code to someone else.
- Can include complicated steps of an algorithm.
- Can include simply steps of a workflow such as running all the steps for normalizing data.
- You can use the apply function to make your code more efficient.

Architecture of a function

- A function is divided into 3 parts:
  - *Name* – the name of the function that has to be used when calling the function
  - *Arguments* – the required and optional objects and settings that the software can/will use
  - *Expression* – the code that does the work
    - It is a good idea to include a return command in the expression block

```
name<-function(argument1, argument2, …) {
                    Expression
                    return(myresults)
                    }
```

A function example

- To create a function you must run the code where it is defined. If you make any changes, you must re-enter the entire code
- To call the function simply use its name and provide the arguments

```
squareme < - function(x){
                                result = x**2
                    return(result)
                }

squareme(3)
squareme( "hello"  )
squareme(c(3,4,5,6))
```

Control statements

- If-else is a control statement that is used to evaluate a condition
  - If the condition is TRUE a given statement will be executed and if it is FALSE then a different statement will be executed.
- If-else can be nested
- If-else evaluates a vector of length 1

```
a<-20
if(a>5) {
   print(a)
} else {
   print("a is less than 5")
}
[1] 20

a<-2
if(a>5) {
   print(a)
} else {
   print("a is less than 5")
}
[1] "a is less than 5"
```

Create a function with nested if -else

```
f<-function(a) {
        if(a<5) {
                print("a is < 5")
        } else {
                if(a<10) {
                        print("a is < 10 but > 5")
                } else {
                        print ("a is >=  10")
                }
        }
}
>f(2)
[1] "a is < 5"
>f(7)
[1] "a is < 10 but > 5"
>f(20)
[1] "a is >=  10"
```

ifelse

- Ifelse can be used to evaluate a vector of longer length
  - First argument is the condition, Second is statements to execute if TRUE, and third is statement to execute if FALSE

```
a<-c(1:8)
ifelse(a <= 5, a, "greater than 5")
[1] "1" "2" "3" "4" "5" "greater than 5"
[7] "greater than 5" "greater than 5"
```

Loop statements (for and while)

- There are cases where you need to loop through each element of a data object.
  - Loops through the code that is provide within curly brackets.
- *for loops* need to know the number of iterations

```
for (i in 1:length(a)) { }
```

- *while loops* evaluate a condition and loops until it is no longer true

```
while (a<4) { }
```

Loop statements (repeat)

- Repeat loops
  - No condition required. It iterates until break is called which ca n be put in a control statement.

```
>repeat
{
…
 if(a>4) break
}
```

Calculate the mean of each row

```
sampleData<-matrix(sample(20:160, 20,
replace=T)/10, ncol=4, nrow=5)

#your sampleData will be different
because it is generated randomly

sampleData
     [,1] [,2] [,3] [,4]
[1,]  7.5  7.1 10.0 10.4
[2,]  8.1 15.3 11.6  4.5
[3,]  7.7  8.1  6.2  6.0
[4,]  9.6  6.6 11.0 14.7
[5,]  7.7  4.5  3.1  5.8
```

Calculate the mean of each row

```
myrowmeans=numeric()

#####################################
for (i in 1:nrow(sampleData) {
      myrowmeans[i] =
mean(sampleData[i,])
}

#####################################
i=1
while (i <= nrow(sampleData) {
      myrowmeans[i] =
mean(sampleData[i,])
      i=i+1
```

Apply function family

- The functions apply(), tapply() and lapply() allows you to perform a specified function across array objects.
  - apply() – first provide the array then whether to apply the function by row (1), column(2) or both (c(1,2), finally the function.
  - tapply() – similar to apply but pass a factor vector instead of row or column
  - lapply() – simply provide a list and the function to apply to each vector in the list. Result is a list.

Calculate the mean using apply()

```
sampleData
     [,1] [,2] [,3] [,4]
[1,]  7.5  7.1 10.0 10.4
[2,]  8.1 15.3 11.6  4.5
[3,]  7.7  8.1  6.2  6.0
[4,]  9.6  6.6 11.0 14.7
[5,]  7.7  4.5  3.1  5.8

apply(sampleData, 1, mean)
[1]  8.750  9.875  7.000 10.475
5.275
```

Convert matrix to dataframe

```
sampleData.df<-as.data.frame(sampleData)

colnames(sampleData.df)<-c("ctr11","ctrl2",
                           "trt1", "trt2")

rownames(sampleData.df)<-c("gene1","gene2",
                "gene3", "gene4", "gene5")

sampleData.df
      ctrl1 ctrl2  trt1  trt2
gene1   7.5   7.1  10.0  10.4
gene2   8.1  15.3  11.6   4.5
gene3   7.7   8.1   6.2   6.0
gene4   9.6   6.6  11.0  14.7
gene5   7.7   4.5   3.1   5.8
```

Calculate mean based on group

```
expgroups = factor(c("ctrl", "ctrl",
                     "trt", "trt"))

tapply(as.numeric(sampleData.df[1,]),
                  expgroups, mean)
 ctrl    trt
  7.3   10.2

apply(sampleData.df, 1, tapply,
          expgroups, mean)

 gene1 gene2 gene3 gene4 gene5
ctrl   7.3 11.70   7.9  8.10  6.10
trt   10.2  8.05   6.1 12.85  4.45
```

PLOTTING IN R

Packages

- The package we will be using is a part of the base package called “graphics”
- A newer package called “grid” is much more involved.
- Packages will often come with their own plotting functions but generally the arguments are the same

Choosing a graphical output device

- In the R GUI, as a default, plots will be printed in a new window.
- On a windows machine it is actually a device called “windows”. On a mac it is called “quartz”.
- Different devices include: PDF, PostScript, bitmap, jpeg, png, and LaTeX.
    - These devices will not print to the GUI, instead they will save a file in the working directory.

```
>#open a new window
>windows() #on a pc
>quartz() #on a mac

>#current device
>dev.cur()

>#start pdf device
>pdf()

>#list of devices
>dev.list()

>#select a device
>dev.set(3)

>#end a device or close a file
>dev.off()
```

Generating graphics with Plot

- The output of the function depends on the object it is passed.
- ldeaths is an object of class “ts” (time series) that is provided in R. It is the number of deaths in UK from 1974 -1979 due to lung disease. Since it is a numerical vector the plot function plots a line.

```
plot(ldeaths)
```

Graphical Arguments

- ann – allow annotation [boolean]
- bty – plot border type
- cex – scaling of point size and text
- cex.axis – scaling of axis label text size
- cex.lab – scaling of axis title text size
- cex.main – scaling of plot title text size
- col – default color plotting
- family – font family(arial, courier,etc)
- font – Font selection (bold, italic, etc)

More Useful Graphical Arguments

- lab – number of tick marks on each axis
- las – orientation of axes labels
- lty – line type (dashed, solid, dotted,etc)
- lwd – line width
- mfrow – number of plots to be drawn in a window
- pch – style of points (circle, cross, start, etc)
- tck – draw gridlines of tick marks
- xaxt/yaxt - Plot x or y tick marks and labels

Multiple plots in the same device

- The mfrow argument in the par() function allows you to define the number of rows and columns you want in the device.

- Every time the plot it will fill the device. If plot is called more then the number of plots then a new window is opened with the same mfrow options.

Type of plots

- “p” – points
- “l” – lines
- “h” – histogram-like vertical lines
- “s” – draw lines as steps
- “n” – draw only axes.

```
par(mfrow = c(2,2))

plot(ldeaths, type="p")
plot(ldeaths, type="l")
plot(ldeaths, type="h")
plot(ldeaths, type="s")
```

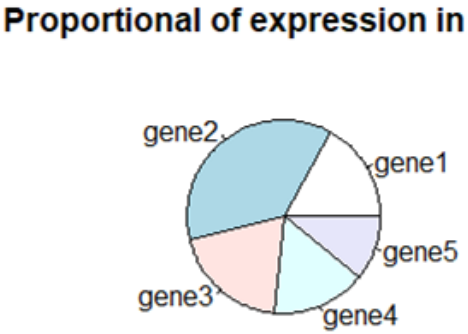
Adding to a plot using functions

- axis() – add an axis
- title() – add a title and subtitle
- legend() – add a legend
- text() – add text such as point labels
- points() – add new points to a plot
- lines() – draw lines specified by points provided as argument
- polygon() – draw polygon given the coordinates
- abline() – draw a diagonal, horizontal, or

Some Common Plots

- Pie charts are useful in visualizing proportion of a given category

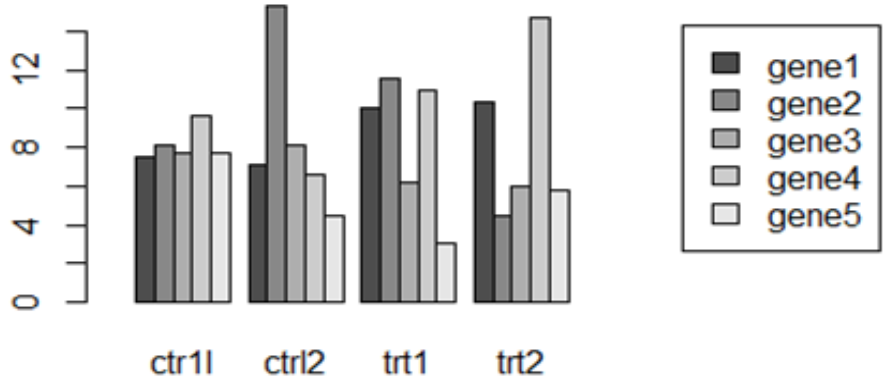
```
pie(sampleData.df[,2],
labels=rownames(sampleData.df),
main="Proportional of expression in Exp 2")
```



Some Common Plots

- Bar plots are the most common plots and are useful for comparing data values.

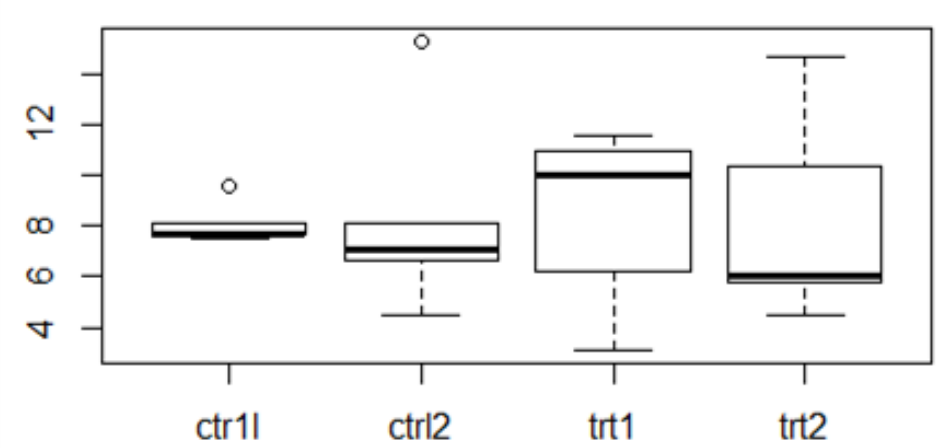
```
barplot(as.matrix(sampleData.df), beside=T,
legend.text = rownames(sampleData.df),
xlim=c(0,40) )
```



Some Common Plots

- Boxplots allow you to compare the median and quartiles of the data sets.

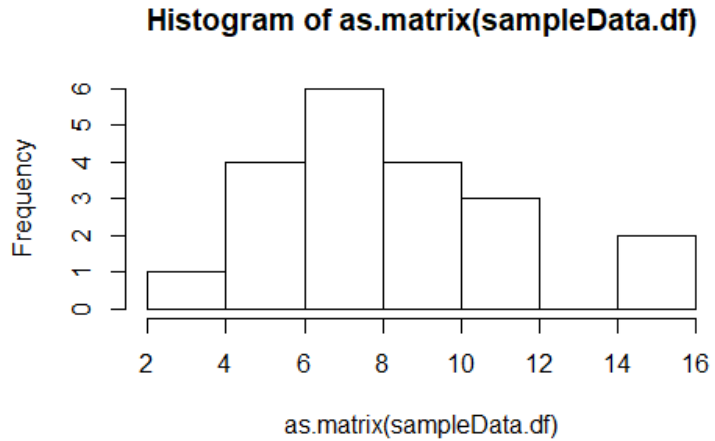
```
boxplot(sampleData.df)
```



Some Common Plots

- Histograms help visualize the distribution of the data values.

```
hist(as.matrix(sampleData.df))
```



Some Cool R commands

```
sampleData = matrix(sample(10:100, 10000, replace=T),
                    nrow=5000, ncol=10)
head(sampleData)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]   78   82   17   85   53   39   60   92   62    75
[2,]   36   95   95   65   95   90   68   71   98    31
[3,]  100   25   31   32   15   46   62   53   33    21
[4,]   54   75   92   82   59   15   16   81   68    27
[5,]   32   37   93   59   84   52  100   88   35    48
[6,]   69   71   57   24   28  100   53   71   48    62

tail(sampleData)
        [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[4995,]   91   14   71   99   95   28   51   65   72    82
[4996,]   14   27   28   62   53   46   63   81   75    45
[4997,]   30   67   88   75   89   20   89   30   30    32
[4998,]   64   71   82   17   28   24   18   78   43    82
[4999,]   75   12   91   82   14   16   31   42   87    74
[5000,]   73   14   16   75   27   89   93   46   98    96
```

String Matching

```
fruits = c("strawberry", "banana", "orange", "apple",
"blueberry", "cranberry", "rasberry", "banana")
fruits == "banana" #exact match
[1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
which(fruits == "banana") #use which to get position
[1] 2 8
lunch = c("apple", "banana")
fruits %in% lunch
[1] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE
match(lunch, fruits) #notice that only the first match is
returned
[1] 4 2
fruits == "bana" # how do we search for pattern ?
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
lunch.ab = c("app","bana")
pmatch(lunch.ab, fruits) #notice that bana is not unique so it
won't work
[1]  4 NA
grep("bana", fruits) # grep works but one pattern at a time
[1] 2 8
grep("bana", fruits, value=T)
[1] "banana" "banana"
```