Introduction to R

This assignment is an opportunity to try the R statistical package and to start to learn some of its behaviors and options.

Text like this will be general comments.

`Text like this will be my commands to R, the R prompt is a "greater than" sign (>).`

1. Assignment and basics

Assignment to an object name may be done using 1) an equals sign =, 2) a "left arrow" <- (less than, hyphen), or 3) a "right arrow" -> (hyphen, greater than).

You can type the name of any object to look at that object.

```
> n <- 15
> n
[1] 15
> a = 12
> a
[1] 12
> 24 -> z
> z
[1] 24
```

Variables must start with a letter, but may also contain numbers and periods. R is case sensitive.

```
> N <- 26.42
> N
[1] 26.42
> n
[1] 15
```

To see a list of your objects, use ls( ). The ( ) is required, even though there are no arguments.

```
> ls()
[1] "a" "n" "N" "z"
```

Use rm to delete objects you no longer need.

```
> rm(n)
> ls()
[1] "a" "N" "z"
```

You may see online help about a function using the help command or a question mark.

```
> ?ls

> help(rm)
```

Several commands are available to help find a command whose name you don't know. Note that anything after a pound sign (#) is a comment and will not have any effect on R.

```
> help.search("help")   # "help" in name or summary; note quotes!

> help.start()     # also remember the R Commands web page (link on
                   # class page)
```

Other data types are available. You do not need to declare these; they will be assigned automatically.

```
> name <- "Mike"  # Character data
> name
[1] "Mike"

> q1 <- TRUE            # Logical data
> q1
[1] TRUE

> q2 <- F
> q2
[1] FALSE
```

1. Simple calculation

R may be used for simple calculation, using the standard arithmetic symbols +, -, *, /, as well as parentheses and ^ (exponentiation).

```
> a <- 12+14
> a
[1] 26
> 3*5
[1] 15
> (20-4)/2
[1] 8
> 7^2
[1] 49
```

Standard mathematical functions are available.

```
> exp(2)
[1] 7.389056
> log(10)   # Natural log
[1] 2.302585
> log10(10) # Base 10
[1] 1
> log2(64)  # Base 2
[1] 6
> pi
[1] 3.141593
> cos(pi)
[1] -1
> sqrt(100)
[1] 10
```

2. Vectors

Vectors may be created using the c command, separating your elements with commas.

```
> a <- c(1, 7, 32, 16)
> a
[1]  1  7 32 16
```

Sequences of integers may be created using a colon (:).

```
> b <- 1:10
> b
 [1]  1  2  3  4  5  6  7  8  9 10

> c <- 20:15
> c
[1] 20 19 18 17 16 15
```

Other regular vectors may be created using the seq (sequence) and rep (repeat) commands.

```
> d <- seq(1, 5, by=0.5)
> d
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0

> e <- seq(0, 10, length=5)
> e
[1]  0.0  2.5  5.0  7.5 10.0

> f <- rep(0, 5)
> f
[1] 0 0 0 0 0

> g <- rep(1:3, 4)
> g
 [1] 1 2 3 1 2 3 1 2 3 1 2 3

> h <- rep(4:6, 1:3)
> h
[1] 4 5 5 6 6 6
```

Random vectors can be created with a set of functions that start with r, such as rnorm (normal) or runif (uniform).

```
> x <- rnorm(5)   # Standard normal random variables
> x
[1] -1.4086632  0.3085322  0.3081487  0.2317044 -0.6424644

> y <- rnorm(7, 10, 3)  # Normal r.v.s with mu(µ) = 10, sigma = 3
> y
[1] 10.407509 13.000935  8.438786  8.892890 12.022136  9.817101  9.330355

> z <- runif(10)  # Uniform(0, 1) random variables
> z
 [1] 0.925665659 0.786650785 0.417698083 0.619715904 0.768478685 0.676038428
 [7] 0.050055548 0.727041628 0.008758944 0.956625536
```

If a vector is passed to an arithmetic calculation, it will be computed element-by-element.

```
> c(1, 2, 3) + c(4, 5, 6)
[1] 5 7 9
```

If the vectors involved are of different lengths, the shorter one will be repeated until it is the same length as the longer.

```
> c(1, 2, 3, 4) + c(10, 20)
[1] 11 22 13 24

> c(1, 2, 3) + c(10, 20)
[1] 11 22 13
Warning message:
longer object length
        is not a multiple of shorter object length in: c(1, 2, 3) + c(10, 20)
```

Basic mathematical functions will apply element-by-element.

```
> sqrt(c(100, 225, 400))
[1] 10 15 20
```

To select subsets of a vector, use square brackets ([ ]).

```
> d
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
> d[3]
[1] 2
> d[5:7]
[1] 3.0 3.5 4.0
```

A logical vector in the brackets will return the TRUE elements.

```
> d > 2.8
[1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> d[d > 2.8]
[1] 3.0 3.5 4.0 4.5 5.0
```

The number of elements in a vector can be found with the length command.

```
> length(d)
[1] 9
> length(d[d > 2.8])
[1] 5
```

3. Simple statistics

There are a variety of mathematical and statistical summaries which can be computed from a vector.

```
> 1:4
[1] 1 2 3 4

> sum(1:4)
[1] 10

> prod(1:4)        # product
[1] 24

> max(1:10)
[1] 10
> min(1:10)
[1] 1
> range(1:10)
[1]  1 10

> X <- rnorm(10)
> X
 [1]  0.2993040 -1.1337012 -0.9095197 -0.7406619 -1.1783715  0.7052832
 [7]  0.4288495 -0.8321391  1.1202479 -0.9507774

> mean(X)
[1] -0.3191486

> sort(X)
 [1] -1.1783715 -1.1337012 -0.9507774 -0.9095197 -0.8321391 -0.7406619
 [7]  0.2993040  0.4288495  0.7052832  1.1202479

> median(X)
[1] -0.7864005

> var(X)
[1] 0.739266

> sd(X)
[1] 0.8598058
```

4. Matrices

Matrices can be created with the matrix command, specifying all elements (column-by-column) as well as the number of rows and number of columns.

```
> A <- matrix(1:12, nr=3, nc=4)
> A
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

You may also specify the rows (or columns) as vectors, and then combine them into a matrix using the **rbind (cbind)** command.

```
> a <- c(1,2,3)
> a
[1] 1 2 3
> b <- c(10, 20, 30)
> b
[1] 10 20 30
> c <- c(100, 200, 300)
> c
[1] 100 200 300
> d <- c(1000, 2000, 3000)
> d
[1] 1000 2000 3000

> B <- rbind(a, b, c, d)
> B
  [,1] [,2] [,3]
a    1    2    3
b   10   20   30
c  100  200  300
d 1000 2000 3000

> C <- cbind(a, b, c, d)
> C
     a  b   c    d
[1,] 1 10 100 1000
[2,] 2 20 200 2000
[3,] 3 30 300 3000
```

To select a subset of a matrix, use the square brackets and specify **rows before the comma**, and columns after.

```
> C[1:2,]
     a  b   c    d
[1,] 1 10 100 1000
[2,] 2 20 200 2000

> C[,c(1,3)]
     a   c
[1,] 1 100
[2,] 2 200
[3,] 3 300

> C[1:2,c(1,3)]
     a   c
[1,] 1 100
[2,] 2 200
```

Matrix multiplication is performed with the operator %*%. **Remember that order matters**!

```
> B%*%C
      a      b       c       d
a    14    140    1400 1.4e+04
```

```
b   140   1400   14000 1.4e+05
c  1400  14000  140000 1.4e+06
d 14000 140000 1400000 1.4e+07

> C%*%B
        [,1]    [,2]    [,3]
[1,] 1010101 2020202 3030303
[2,] 2020202 4040404 6060606
[3,] 3030303 6060606 9090909
```

You may apply a summary function to the rows or columns of a matrix using the apply function.

```
> C
     a  b   c    d
[1,] 1 10 100 1000
[2,] 2 20 200 2000
[3,] 3 30 300 3000

> sum(C)
[1] 6666

> apply(C, 1, sum)       # sums of rows
[1] 1111 2222 3333

> apply(C, 2, sum)       # sums of columns
   a    b    c    d
   6   60  600 6000
```

4.1 Mixed modes and data frames

All elements of a matrix must be the same mode (numeric, character, logical, etc.). If you try to put different modes in a matrix, all elements will **be coerced to the most general – usually character.**

```
> Name <- c("Bob", "Bill", "Betty")
> Test1 <- c(80, 95, 92)
> Test2 <- c(40, 87, 90)

> grades <- cbind(Name, Test1, Test2)
> grades
     Name    Test1 Test2
[1,] "Bob"   "80"  "40"
[2,] "Bill"  "95"  "87"
[3,] "Betty" "92"  "90"
```

The solution is another complex object called a **data frame**. **The data frame views rows as cases and columns as variables.** All elements in a column must be the same mode, but different columns may be different modes.

```
> grades.df <- data.frame(Name, Test1, Test2)
> grades.df
   Name Test1 Test2
```

```
1   Bob    80    40
2  Bill    95    87
3 Betty    92    90
```

Summary functions applied to a data frame will be applied to each column.

```
> mean(grades.df)
    Name
      NA
Warning message:
argument is not numeric or logical: returning NA in: mean.default(X[[1]], ...)

> mean(grades.df[,2])
   Test1
89.00000
```

Note: as similar as matrices and data frames appear, R considers them to be quite different. Many functions will work on one or the other, but not both. You can convert from one to the other using **as.matrix or as.data.frame**.

```
> C.df <- data.frame(a,b,c,d)
> C.df
  a  b   c    d
1 1 10 100 1000
2 2 20 200 2000
3 3 30 300 3000

> C.df%*%B
Error in C.df %*% B : requires numeric matrix/vector arguments
> as.matrix(C.df)%*%B
     [,1]    [,2]    [,3]
1 1010101 2020202 3030303
2 2020202 4040404 6060606
3 3030303 6060606 9090909

> C
     a  b   c    d
[1,] 1 10 100 1000
[2,] 2 20 200 2000
[3,] 3 30 300 3000

> mean(C)
[1] 555.5
```

5. Data Import – Text Files

Data files should most easily be set up as text files with rows as cases and columns as variables. Save them to a text file and use read.table to read them into R as data frames.

```
> iris<-read.table("C:\\...\\Iris.txt",header=T)
> iris
    Species SepalLength SepalWidth PetalLength PetalWidth
```

```
1          1          5.1          3.5          1.4          0.2
2          1          4.9          3.0          1.4          0.2
3          1          4.7          3.2          1.3          0.2
4          1          4.6          3.1          1.5          0.2
5          1          5.0          3.6          1.4          0.2
6          1          5.4          3.9          1.7          0.4
:          :           :            :            :            :
:          :           :            :            :            :
:          :           :            :            :            :
149        3          6.2          3.4          5.4          2.3
150        3          5.9          3.0          5.1          1.8
```

Other possible file format includes:

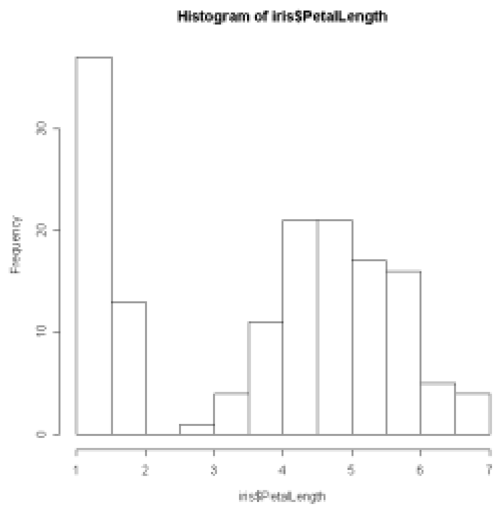
```
>iris<-read.csv("C:\\Teaching_Advising\\sta519---Multivariate Statistics\\08
Spring\\Data\\IRIS.csv")
```

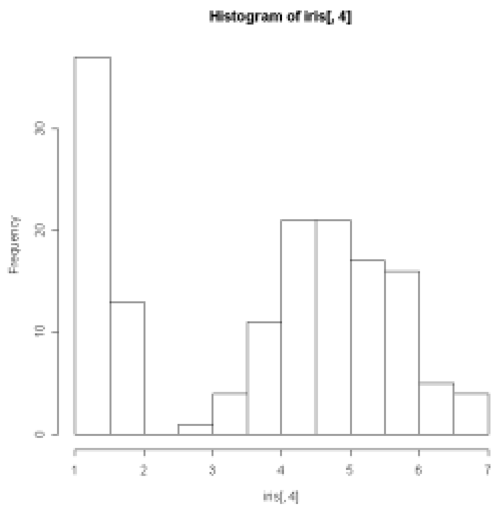
6. Graphics

R has functions to automatically plot many standard statistical graphics. Histograms and boxplots may be generated with **hist and boxplot**, respectively.

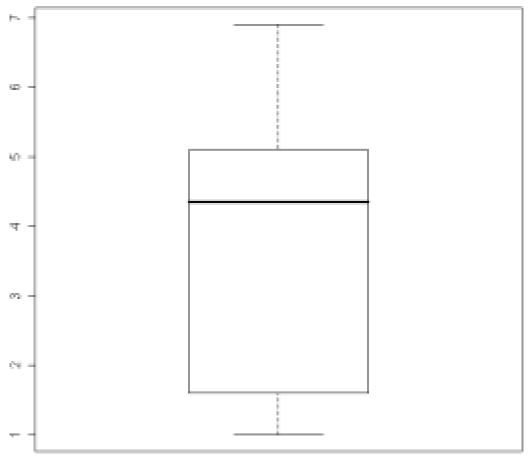
Once you have a graphic you're happy with, you can copy the entire thing. Make sure that the graphics window is the active (selected) window, and select "Copy to clipboard as bitmap" from the file menu. You can then paste your figure into Word and resize to taste.

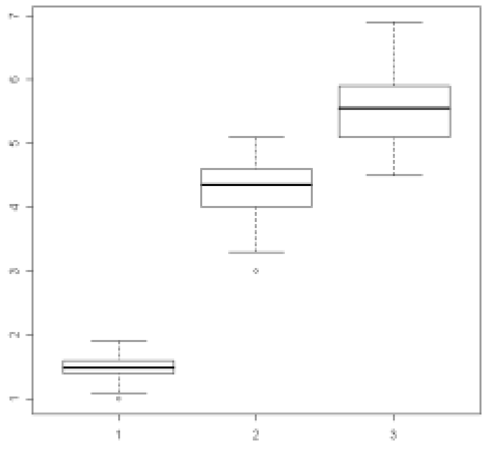
```
> hist(iris$Petal.Length)
> hist(iris[,4])# alternative specification
```





```
> boxplot(iris$Petal.Length)
> boxplot(Petal.Length~Species, data=iris)      # Formula description,
                                               # side-by-side boxplots
```

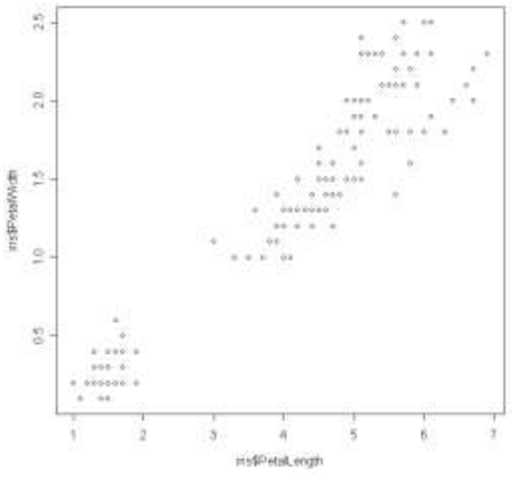


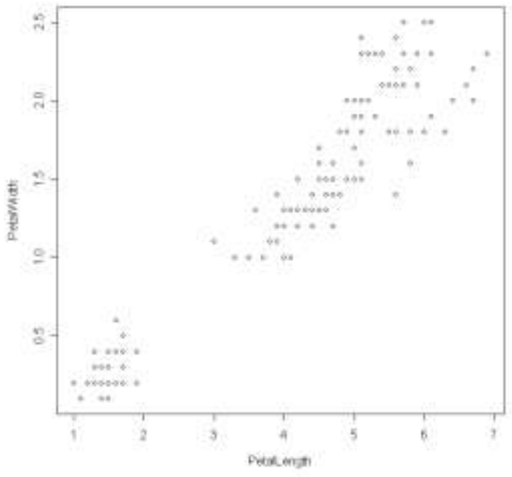


7. Scatterplots and simple linear regression

Scatterplots may be produced by using the plot command. For separate vectors, the form is plot(x, y). For columns in a dataframe, the form is plot(yvar ~ xvar, data=dataframe).

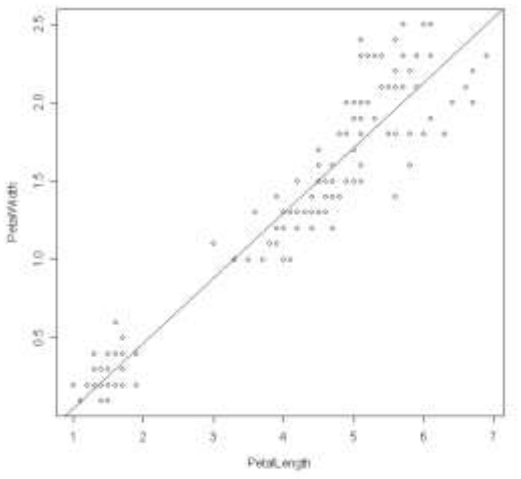
```
> plot(iris$Petal.Length, iris$Petal.Width)
> plot(Petal.Width~Petal.Length, data=iris)
```





Linear regression is done with the lm command, with a form similar to the second version of the scatterplot command.

```
> PetalReg <- lm(Petal.Width~Petal.Length, data=iris)

Call:
lm(formula = PetalWidth ~ PetalLength, data = iris)

Residuals:
     Min       1Q   Median       3Q      Max
-0.56515 -0.12358 -0.01898  0.13288  0.64272

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.363076   0.039762  -9.131  4.7e-16 ***
PetalLength  0.415755   0.009582  43.387  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2065 on 148 degrees of freedom
Multiple R-Squared: 0.9271,     Adjusted R-squared: 0.9266
F-statistic:  1882 on 1 and 148 DF,  p-value: < 2.2e-16

> abline(PetalReg)      # add the regression line to the plot
```



Check out some misc R commands like:

```
> ls.str(); ls.str; ?ls.str
> ls()
> getwd()     #get working directory
> setwd()
> dir()
> search()
> searchpaths()
> attach()
> detach()
> history()
> save.history()
> save.image()
> demo()
> demo(package = .packages(all.available = TRUE))
> help.search()
> help.search("linear models")
> help.search(keyword = "hplot")
> help(library=car)
> library()
> demo()
> demo(package = .packages(all.available = TRUE))
```

In addition to the standard packages listed in the search,

```
> search()
".GlobalEnv"         "package:stats"      "package:graphics"
"package:grDevices" "package:utils"      "package:datasets"
"package:methods"    "Autoloads"          "package:base"
```

R- Conditional Statements

R - If...Else Statement

An **if** statement can be followed by an optional **else** statement which executes when the boolean expression is false.

Syntax

The basic syntax for creating an **if...else** statement in R is −

```
if(boolean_expression) {
   // statement(s) will execute if the boolean expression is true.
} else {
   // statement(s) will execute if the boolean expression is false.
}
```

If the Boolean expression evaluates to be **true**, then the **if block**of code will be executed, otherwise **else block** of code will be executed.

Example

```
x <- c("what","is","truth")


if("Truth" %in% x) {

   print("Truth is found")

} else {

   print("Truth is not found")

}
```

When the above code is compiled and executed, it produces the following result −

```
[1] "Truth is not found"
```

Here "Truth" and "truth" are two different strings.

The if...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using **if**, **else if**, **else** statements there are few points to keep in mind.

- An **if** can have zero or one **else** and it must come after any **else if**'s.
- An **if** can have zero to many **else if's** and they must come before the else.
- Once an **else if** succeeds, none of the remaining **else if**'s or **else**'s will be tested.

Syntax

The basic syntax for creating an **if...else if...else** statement in R is −

```
if(boolean_expression 1) {
   // Executes when the boolean expression 1 is true.
} else if( boolean_expression 2) {
   // Executes when the boolean expression 2 is true.
} else if( boolean_expression 3) {
   // Executes when the boolean expression 3 is true.
} else {
   // executes when none of the above condition is true.
}
```

Example

```
x <- c("what","is","truth")


if("Truth" %in% x) {
   print("Truth is found the first time")
} else if ("truth" %in% x) {
   print("truth is found the second time")
} else {
   print("No truth found")
}
```

When the above code is compiled and executed, it produces the following result −

```
[1] "truth is found the second time"
```

for Loops

For loops are pretty much the only looping construct that you will need in R. While you may occasionally find a need for other types of loops, in my experience doing data analysis, I’ve found very few situations where a for loop wasn’t sufficient.

In R, for loops take an interator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```
> for(i in 1:10)
{  print(i)  }

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

```
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

The following loop have the same behavior.

```
> x <- c("a", "b", "c", "d")
> for(i in 1:4)
{
  ## Print out each element of 'x'
 print(x[i])  }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

Nested for loops

for loops can be nested inside of each other.

```
x <- matrix(1:6, 2, 3)
for(i in seq_len(nrow(x)))
{ for(j in seq_len(ncol(x)))
   { print(x[i, j]) } }
```

while Loops

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth, until the condition is false, after which the loop exits.

```
> count <- 0
 > while(count < 10) {
  print(count)
 count <- count + 1
 }

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