BIOS 545 Lecture 3

Factors, Matrices, Lists

Steve Pittard, wsp@emory.edu

Let's talk about Factors.

- R supports factors which are a special data type for managing categories of data
- Identifying categorical variables is usually straightforward. These are the variables by which you might want to summarize some continuous data
- Categorical variables usually take on a definite number of values
- Many times R functions will convert character labels to factors by default but not always
- Storing data as factors insures that R modeling functions will treat such data correctly

Let's say we have some automobile data that tells us if a car has an automatic transmission (0) or a manual transmission (1). We store this into a vector called transvec

```
transvec <- c(1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1)
table(transvec) #How many are Auto and Manual ? Count 'em up. transvec
```

```
## transvec
##  0  1
## 19 13
```

```
# Make a vector of factors
mytransfac <- factor(transvec,
                     levels = c(0,1),
                     labels = c("Auto","Man") )

# Get the levels of the factor vector
levels(mytransfac)
```

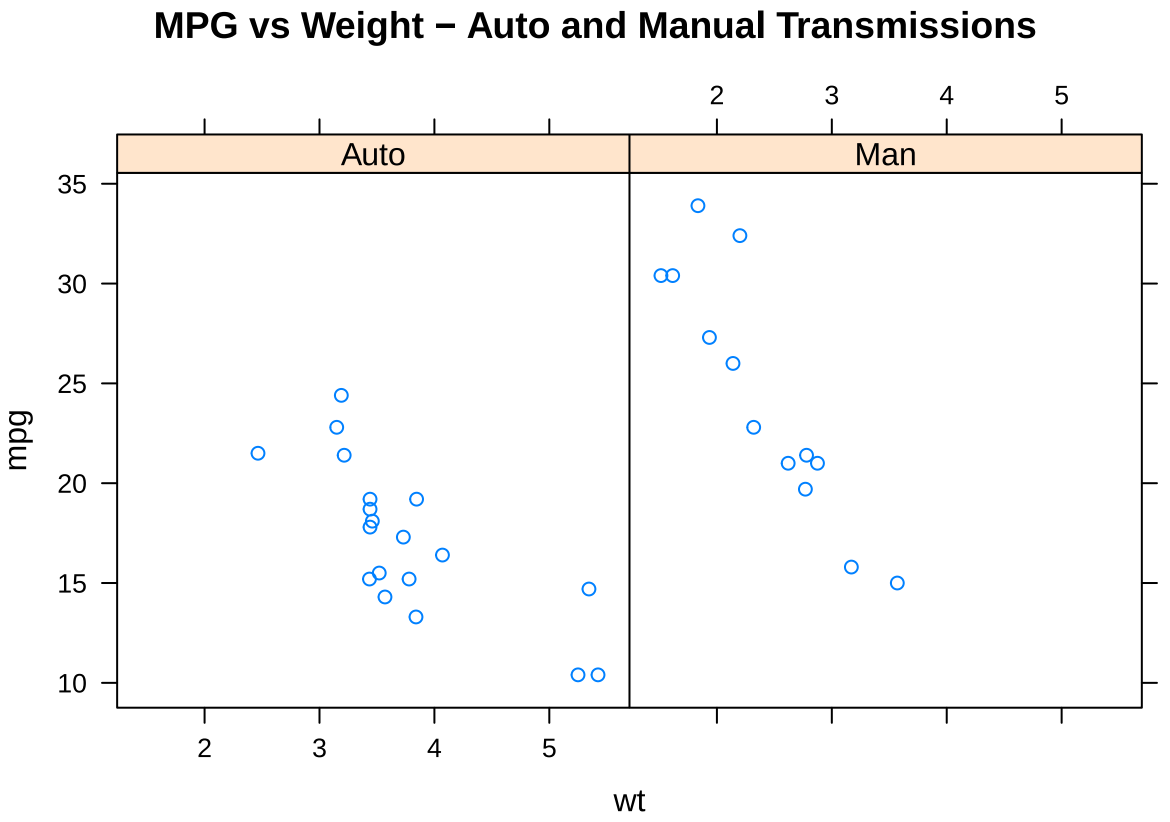
```
## [1] "Auto" "Man"
```

```
mytransfac
```

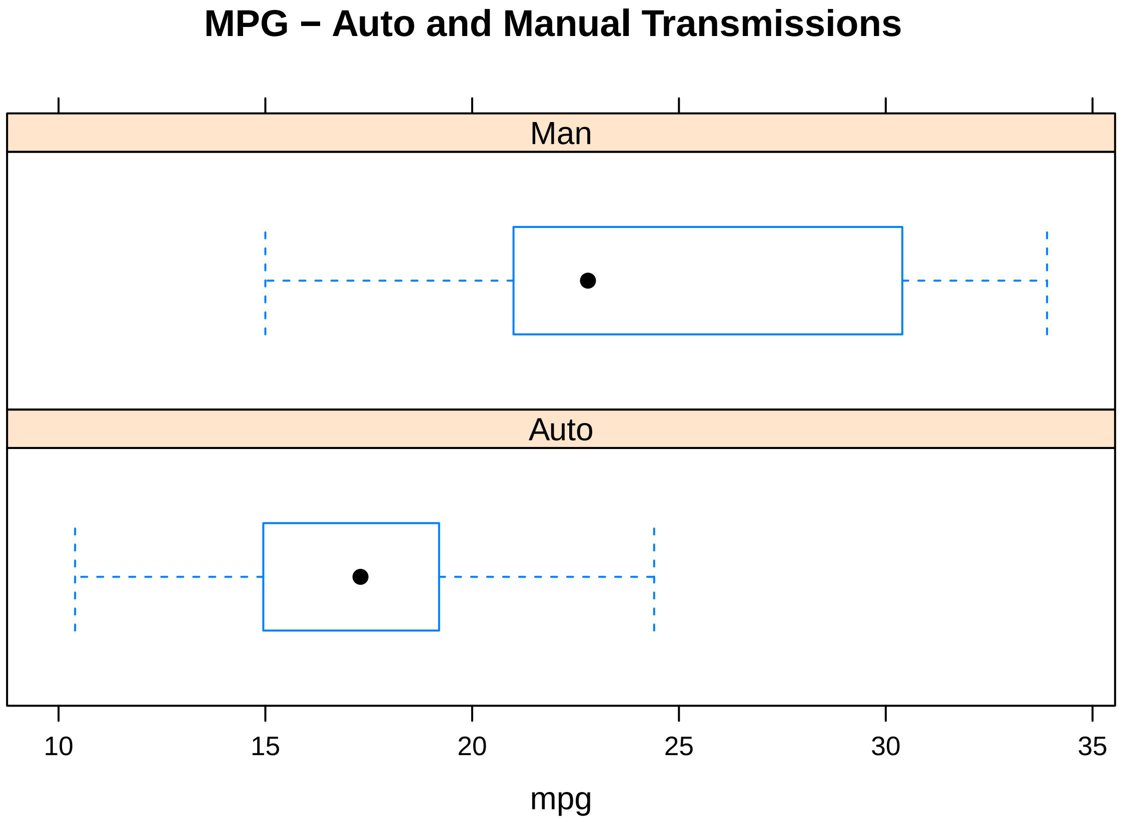
```
##  [1] Man  Man  Man  Auto Auto Auto Auto Auto Auto Auto Auto Auto Auto Auto Auto
## [16] Auto Auto Man  Man  Man  Auto Auto Auto Auto Auto Man  Man  Man  Man  Man
## [31] Man  Man
## Levels: Auto Man
```

R knows how to handle factors when doing plots. Here were get an X/Y plot and a Box Plot with very little work since R knows that mytransfac is a factor

```
library(lattice)
xyplot(mpg~wt | mytransfac,
       mtcars,
       main="MPG vs Weight - Auto and Manual Transmissions")
```



```
bwplot(~mpg|mytransfac, mtcars,
main="MPG - Auto and Manual Transmissions",layout=c(1,2))
```



Aggregation Preview

With our knowledge of factors and vectors we can do some basic aggregation using the tapply command. We have a factor vector called mytransfac. Let's summarize some MPG data that corresponds to the automobiles used in the mytransfac vector. So for each car we have its MPG figure and whether it has an automatic or manual transmission.

```
mympg <- c(21,21,22.8,21.4,18.7,18.1,14.3,24.4,22.8,19.2,17.8, 16.4,17.3,15.2,10.4,10.4,14.7,32.4,30.4,3

# tapply( continuous_value_to_summarize, factor_or_grouping variable, function_for_summary)
tapply(mympg, mytransfac, mean)
```

```
##     Auto      Man
## 17.14737 24.39231
```

The cut function

It is sometimes useful to take a continuous variable and chop it up into intervals or categories for purposes of summary or grouping. R has a function to do this called "cut"to accomplish this.

Let's work through some examples to understand what is going on. Let's cut up the numbers between 0 and 10 into 4 distinct intervals

```
cut(0:10,breaks=4)
```

```
##  [1] (-0.01,2.5] (-0.01,2.5] (-0.01,2.5] (2.5,5]     (2.5,5]     (2.5,5]
##  [7] (5,7.5]     (5,7.5]     (7.5,10]    (7.5,10]    (7.5,10]
## Levels: (-0.01,2.5] (2.5,5] (5,7.5] (7.5,10]
```

```
table(cut(0:10,breaks=4))
```

```
##
## (-0.01,2.5]     (2.5,5]     (5,7.5]    (7.5,10]
##           3           3           2           3
```

Well that was cool but since we are creating 4 intervals we should probably name them

```
(my.cut <- cut(0:10,breaks=4,labels=c("Q1","Q2","Q3","Q4")))
```

```
##  [1] Q1 Q1 Q1 Q2 Q2 Q2 Q3 Q3 Q4 Q4 Q4
## Levels: Q1 Q2 Q3 Q4
```

```
table(my.cut)
```

```
## my.cut
## Q1 Q2 Q3 Q4
##  3  3  2  3
```

But you can to take finer-grained control over how the intervals are made.

```
quantile(0:10)
```

```
##   0%  25%  50%  75% 100%
##  0.0  2.5  5.0  7.5 10.0
```

```
table(cut(0:10,breaks=quantile(0:10),include.lowest=TRUE))
```

```
##
##  [0,2.5]  (2.5,5]  (5,7.5] (7.5,10]
##        3        3        2        3
```

Let's summarize some exam scores according to a typical grading system

- F: < 60,
- D: 60-70:
- C: 70-80:
- B: 80-90
- A: 90-100

```
set.seed(123)
exam.score <- runif(25,50,100) # Simulate some scores
cut(exam.score,breaks=c(0,60,70,80,90,100))
```

```
##  [1] (60,70]  (80,90]  (70,80]  (90,100] (90,100] (0,60]   (70,80]  (90,100]
##  [9] (70,80]  (70,80]  (90,100] (70,80]  (80,90]  (70,80]  (0,60]   (90,100]
## [17] (60,70]  (0,60]   (60,70]  (90,100] (90,100] (80,90]  (80,90]  (90,100]
## [25] (80,90]
## Levels: (0,60] (60,70] (70,80] (80,90] (90,100]
```

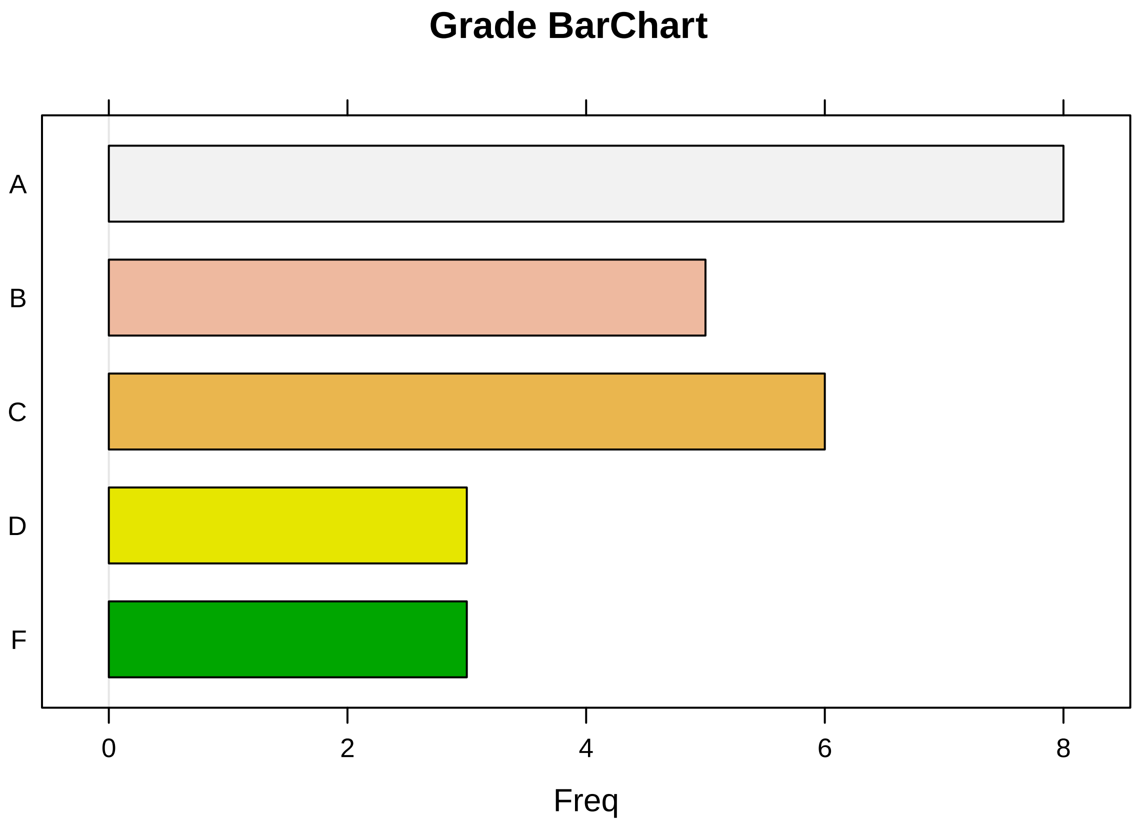
```
cut(exam.score,breaks=c(50,60,70,80,90,100),labels=c("F","D","C","B","A"))
```

```
##  [1] D B C A A F C A C C A C B C F A D F D A A B B A B
## Levels: F D C B A
```

```
my.table <- table(cut(exam.score,
                      breaks=c(0,60,70,80,90,100),
                      labels=c("F","D","C","B","A")))
my.table
```

```
##
## F D C B A
## 3 3 6 5 8
```

```
barchart(my.table,main="Grade BarChart",col=terrain.colors(5))
```



- But the intervals don't exactly match the grading scheme
- A score of 90 will get a B when it should get an A
- Note the [and) characters show us if the interval boundary includes a score
- There is an argument that can help right)

```
set.seed(123)
cut(exam.score,breaks=c(50,60,70,80,90,100))
```

```
##  [1] (60,70]  (80,90]  (70,80]  (90,100] (90,100] (50,60]  (70,80]  (90,100]
##  [9] (70,80]  (70,80]  (90,100] (70,80]  (80,90]  (70,80]  (50,60]  (90,100]
## [17] (60,70]  (50,60]  (60,70]  (90,100] (90,100] (80,90]  (80,90]  (90,100]
## [25] (80,90]
## Levels: (50,60] (60,70] (70,80] (80,90] (90,100]
```

```
cut(exam.score,breaks=c(50,60,70,80,90,100),right=F)
```

```
##  [1] [60,70)  [80,90)  [70,80)  [90,100) [90,100) [50,60)  [70,80)  [90,100)
##  [9] [70,80)  [70,80)  [90,100) [70,80)  [80,90)  [70,80)  [50,60)  [90,100)
## [17] [60,70)  [50,60)  [60,70)  [90,100) [90,100) [80,90)  [80,90)  [90,100)
## [25] [80,90)
## Levels: [50,60) [60,70) [70,80) [80,90) [90,100)
```

So if you don't think that the cut command is cool then here is how you would have had to solve the problem - ughhh !

```
set.seed(123)
exam.score <- runif(25,50,100)
acount <- 0
bcount <- 0
ccount <- 0
```

```
dcount <- 0
fcount <- 0
exam.score <- runif(25,50,100)
for (ii in 1:length(exam.score)) {
if (exam.score[ii] < 60) {fcount = fcount + 1} else
if ((exam.score[ii] >= 60) & (exam.score[ii] < 70)) {dcount = dcount + 1} else
if ((exam.score[ii] >= 70) & (exam.score[ii] < 80)) {ccount = ccount +1} else if ((exam.score[ii] >= 80
if ((exam.score[ii] >= 90) & (exam.score[ii] <= 100)) {acount = acount +1} }

cat("acount bcount ccount dcount fcount")
```

```
## acount bcount ccount dcount fcount
```

```
cat(acount,bcount,ccount,dcount,fcount)
```

```
## 3 4 6 7 5
```

Sometimes we want our factors to be ordered. For example, we intuitively know that January comes before February and so on. Can we get R to create ordered factors ?

```
mons <- c("Jan","Feb","Mar","Apr","May","Jun","Jan", "Feb","May","Jun", "Apr","Mar")
my.fact.mons <- factor(mons)

my.fact.mons[1] < my.fact.mons[2]
```

```
## Warning in Ops.factor(my.fact.mons[1], my.fact.mons[2]): '<' not meaningful for
## factors
```

```
## [1] NA
```

Sometimes we want our factors to be ordered. For example, we intuitively know that January comes before February and so on. Can we get R to create ordered factors ?

```
my.fact.mons <- factor(mons,
                       labels=c("Jan","Feb","Mar","Apr","May","Jun"),
                       ordered=TRUE)
my.fact.mons
```

```
##  [1] Mar Feb May Jan Jun Apr Mar Feb Jun Apr Jan May
## Levels: Jan < Feb < Mar < Apr < May < Jun
```

```
my.fact.mons[1] < my.fact.mons
```

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

```
table(my.fact.mons)
```

```
## my.fact.mons
## Jan Feb Mar Apr May Jun
##   2   2   2   2   2   2
```

```
levels(my.fact.mons)
```

```
## [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun"
```

Let's do an AOV on the mtcars data set variables MPG and number of gears the latter of which takes on the values 3,4,5. So it is well suited to be a factor.

```
# Turn gear into a factor aov.ex1 = aov(mpg ~ gear,mtcars)
mtcars$gear <- factor(mtcars$gear)
```

```
aov.ex1 = aov(mpg ~ gear,mtcars)
summary(aov.ex1)
```

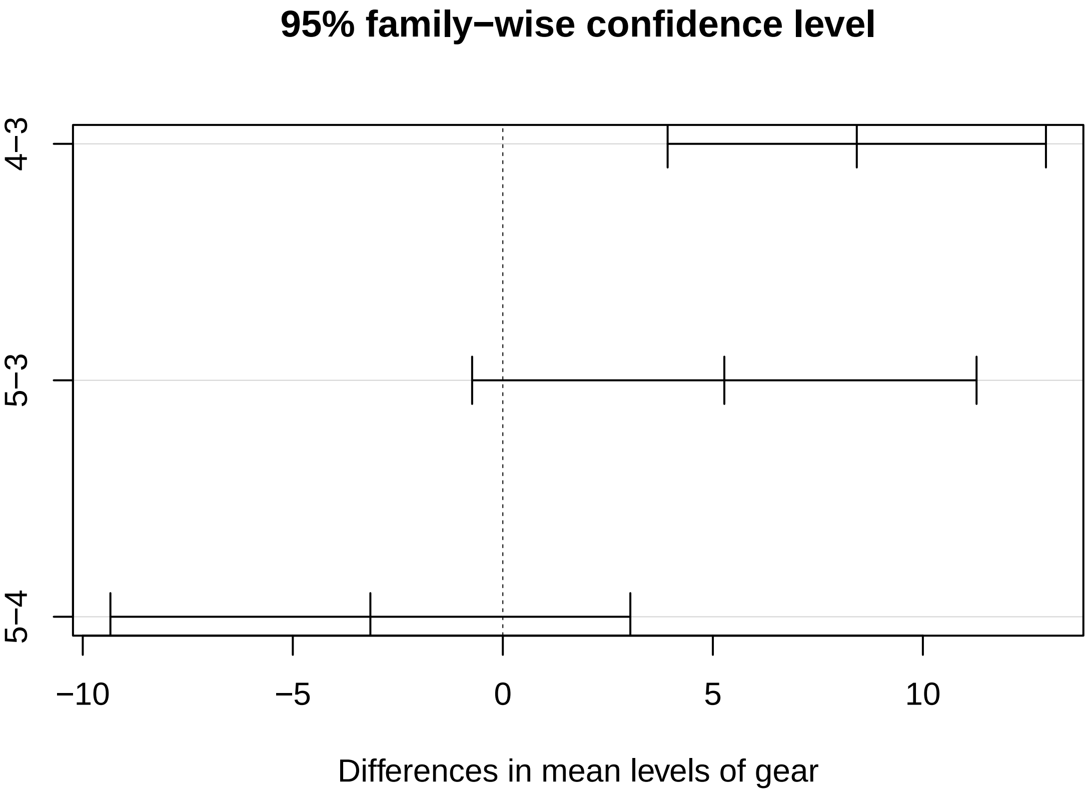
```
##             Df Sum Sq Mean Sq F value   Pr(>F)    
## gear         2  483.2  241.62    10.9 0.000295 ***
## Residuals   29  642.8   22.17                     
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Let's do an AOV on the mtcars data set variables MPG and number of gears the latter of which takes on the values 3,4,5. So it is well suited to be a factor.

```
# Tukey Multiple Comparisons my.tukey
(my.tukey <- TukeyHSD(aov.ex1,"gear"))
```

```
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
## Fit: aov(formula = mpg ~ gear, data = mtcars)
##
## $gear
##          diff        lwr       upr     p adj
## 4-3  8.426667  3.9234704 12.929863 0.0002088
## 5-3  5.273333 -0.7309284 11.277595 0.0937176
## 5-4 -3.153333 -9.3423846  3.035718 0.4295874
```

```
plot(my.tukey)
```



Differences between Gears are significant at the five percent level if the confidence interval around the estimation of the difference does not contain zero.

Matrices

- Matrices are important mathematical structures for which R has excellent support
- Matrices are ideal for storing information on scientific data
- Think of a matrix as being a vector with dimnensions
- There are two common ways to create a matrix

1) Using the dim() function. You can think of the following vector as being a matrix with one row and twelve columns.

```
myvec <- c(1:12)
```

To create, for example, a 3x4 matrix use the dim() function to adjust the dimensions of the vector

```
dim(myvec) <- c(3,4)
myvec
```

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

Note that columns are “filled” before rows. Note also that the requested dimension must make sense with the available number of elements

```
dim(myvec) <- c(5,4)
```

Using the matrix() function

```
mymat <- matrix( c(7, 4, 2, 4, 7, 2), nrow=3, ncol=2)
```

You can use the nrow and ncol arguments to explicity specify the desired number of rows and columns. You can also request that the rows get filled first as opposed to the columns:

```
mymat <- matrix( c(7, 4, 2, 4, 7, 2), nrow=3, ncol=2, byrow=TRUE)
mymat
```

```
##      [,1] [,2]
## [1,]    7    4
## [2,]    2    4
## [3,]    7    2
```

It is useful to name the rows and columns of matrices

```
set.seed(123)
(X <- matrix(rpois(20,1.5),nrow=4))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    1    2    1
## [2,]    2    0    1    2    0
## [3,]    1    1    4    0    1
## [4,]    3    3    1    3    4
```

Let’s say that these refer to four trials. We want to label the rows ”Trial.1”, ”Trial.2”, etc.

```
rownames(X) <- paste("Trial",1:nrow(X),sep=".")
X
```

```
##         [,1] [,2] [,3] [,4] [,5]
## Trial.1    1    4    1    2    1
## Trial.2    2    0    1    2    0
## Trial.3    1    1    4    0    1
## Trial.4    3    3    1    3    4
```

We can something similar with the columns

```
colnames(X) <- paste("P",1:ncol(X),sep=".")
X
```

```
##         P.1 P.2 P.3 P.4 P.5
## Trial.1   1   4   1   2   1
## Trial.2   2   0   1   2   0
## Trial.3   1   1   4   0   1
## Trial.4   3   3   1   3   4
```

We aren't restricted to naming things according to a pattern

```
drug.names <- c("aspirin","paracetamol","nurofen","hedex","placebo")
colnames(X) <- drug.names
X
```

```
##         aspirin paracetamol nurofen hedex placebo
## Trial.1       1           4       1     2       1
## Trial.2       2           0       1     2       0
## Trial.3       1           1       4     0       1
## Trial.4       3           3       1     3       4
```

Names provide an intutive way to index into a matrix structure

```
X
```

```
##         aspirin paracetamol nurofen hedex placebo
## Trial.1       1           4       1     2       1
## Trial.2       2           0       1     2       0
## Trial.3       1           1       4     0       1
## Trial.4       3           3       1     3       4
```

```
X['Trial.1',] # Gets all columns for Trial
```

```
##     aspirin paracetamol     nurofen       hedex     placebo
##           1           4           1           2           1
```

```
X['Trial.1','nurofen']
```

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

It is more common to use numeric indexing especially if you are accessing parts of a matrix from a program

```
set.seed(123)
X <- matrix(rpois(9,1.5),nrow=3)
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    1
## [2,]    2    4    3
## [3,]    1    0    1
```

```
X[1,1]
```

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

```
X[2,2]
```

```
## [1] 4
```

```
X[3,3]
```

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

```
diag(X)
```

```
## [1] 1 4 1
```

Extracting information from a matrix is an important skill. Practice makes perfect

```
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    1
## [2,]    2    4    3
## [3,]    1    0    1
```

```
X[1:2,1]
```

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

```
X[1:2,2]
```

```
## [1] 3 4
```

```
X[1:2,]
```

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

Extracting information from a matrix is an important skill. Practice makes perfect

```
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    1
## [2,]    2    4    3
## [3,]    1    0    1
```

```
X[,c(1,3)]
```

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

```
X[,-2]
```

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

Remember that a matrix is just a vector with dimensions so you could index into a matrix using single bracket notation.

```
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    1
## [2,]    2    4    3
## [3,]    1    0    1
```

```
X[1:4]
```

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

```
X >= 2
```

```
##       [,1]  [,2]  [,3]
## [1,] FALSE  TRUE FALSE
## [2,]  TRUE  TRUE  TRUE
## [3,] FALSE FALSE FALSE
```

```
X[X >= 2]    # Returns which values are greater or equal to 2
```

```
## [1] 2 3 4 3
```

```
which(X >= 2)
```

```
## [1] 2 4 5 8
```

Remember that a matrix is just a vector with dimensions so you could index into a matrix using single bracket notation.

```
X %% 2 == 0
```

```
##       [,1]  [,2]  [,3]
## [1,] FALSE FALSE FALSE
## [2,]  TRUE  TRUE FALSE
## [3,] FALSE  TRUE FALSE
```

```
X[X %%2==0]
```

```
## [1] 2 4 0
```

Remember that a matrix is just a vector with dimensions so you could index into a matrix using single bracket notation.

```
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    1
## [2,]    2    4    3
## [3,]    1    0    1
```

```
X[X%% 2==0]<-99
```

There are two functions called row and col that return the numeric row and column, respectively of the matrix.

```
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    1
## [2,]   99   99    3
## [3,]    1   99    1
```

```
row(X)
```

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

```
col(X)
```

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

There are two functions called row and col that return the numeric row and column, respectively of the matrix.

```
row(X) == col(X)
```

```
##       [,1]  [,2]  [,3]
## [1,]  TRUE FALSE FALSE
## [2,] FALSE  TRUE FALSE
## [3,] FALSE FALSE  TRUE
```

```
X[row(X) == col(X)]
```

```
## [1]  1 99  1
```

```
X[row(X) == col(X)] <- 0
```

Matrices - rbind/cbind

Sometimes we need to add rows and columns to a matrix. There are two commands to do this: rbind and cbind.

```
set.seed(123)
X <- matrix(rpois(9,1.5),nrow=3)
colnames(X) <- c("aspirin","paracetamol","nurofen")
rownames(X) <- paste("Trial",1:3,sep=".")
rbind(X,Trial.4=c(4,7,5))
```

```
##         aspirin paracetamol nurofen
## Trial.1       1           3       1
## Trial.2       2           4       3
## Trial.3       1           0       1
## Trial.4       4           7       5
```

Binding columns works pretty much the same way:

```
X
```

```
##         aspirin paracetamol nurofen
## Trial.1       1           3       1
## Trial.2       2           4       3
## Trial.3       1           0       1
```

```
rowSums(X)
```

```
## Trial.1 Trial.2 Trial.3
##       5       9       2
```

```
cbind(X,rowsums = rowSums(X))
```

```
##         aspirin paracetamol nurofen rowsums
## Trial.1       1           3       1       5
## Trial.2       2           4       3       9
## Trial.3       1           0       1       2
```

Let's look at some examples involving calculations on matricies:

```
set.seed(123)
X <- matrix(rpois(9,1.5),nrow=3)
colnames(X) <- c("aspirin","paracetamol","nurofen")
rownames(X) <- paste("Trial",1:3,sep=".")
X
```

```
##         aspirin paracetamol nurofen
## Trial.1       1           3       1
## Trial.2       2           4       3
## Trial.3       1           0       1
```

```
mean(X[,3]) # Mean of the 3rd column [1] 1.666667
```

```
## [1] 1.666667
```

```
var(X[3,]) #
```

```
## [1] 0.3333333
```

There are functions optimized specifcally for use with matrices. They are very fast and work well on large matrices

```
X
```

```
##         aspirin paracetamol nurofen
## Trial.1       1           3       1
## Trial.2       2           4       3
## Trial.3       1           0       1
```

```
rowSums(X)
```

```
## Trial.1 Trial.2 Trial.3
##       5       9       2
```

```
colSums(X)
```

```
##     aspirin paracetamol     nurofen
##           4           7           5
```

There are functions optimized specifcally for use with matrices. They are very fast and work well on large matrices. However be careful if the matrix has missing data in wich case you will need to include the na.rm=TRUE arguement.

```
rowMeans(X)
```

```
##   Trial.1   Trial.2   Trial.3
## 1.6666667 3.0000000 0.6666667
```

```
colMeans(X,na.rm=TRUE)
```

```
##     aspirin paracetamol     nurofen
##    1.333333    2.333333    1.666667
```

```
colMeans(X)[3]
```

```
##  nurofen
## 1.666667
```

There are functions optimized specifcally for use with matrices. They are very fast and work well on large matrices. However be careful if the matrix has missing data in wich case you will need to include the na.rm=TRUE arguement.

```
X[1,2] <- NA
X
```

```
##         aspirin paracetamol nurofen
## Trial.1       1          NA       1
## Trial.2       2           4       3
## Trial.3       1           0       1
```

```
colMeans(X)
```

```
##     aspirin paracetamol     nurofen
##    1.333333          NA    1.666667
```

```
colMeans(X, na.rm=TRUE)
```

```
##     aspirin paracetamol     nurofen
##    1.333333    2.000000    1.666667
```

There is a function called apply which simplifies looping over the rows or columns of a matrix. The“apply family”of functions is important in R.

```
X
```

```
##         aspirin paracetamol nurofen
## Trial.1       1          NA       1
## Trial.2       2           4       3
## Trial.3       1           0       1
```

```
apply(X,1,range)
```

```
##      Trial.1 Trial.2 Trial.3
## [1,]      NA       2       0
## [2,]      NA       4       1
```

```
apply(X,2,range)
```

```
##      aspirin paracetamol nurofen
## [1,]       1          NA       1
## [2,]       2          NA       3
```

R supports common linear algebra operations

```
A = matrix(c(1,3,2,2,8,9),3,2)
A
```

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

```
t(A)
```

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

```
A
```

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

```
diag(A)
```

```
## [1] 1 8
```

```
diag(c(1,2,3))
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    2    0
## [3,]    0    0    3
```

```
diag(1,4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

The inverse of a n x n matrix A is the matrix B (which is also n x n) that when multiplied by A gives the identity matrix

```
(A <- matrix(1:4,2,2))
```

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

```
(B = solve(A))
```

```
##      [,1] [,2]
## [1,]   -2  1.5
## [2,]    1 -0.5
```

```
A %*% B
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Eigen values/vectors show up a lot in math - like with Principal Components

```
url <- "https://raw.githubusercontent.com/steviep42/youtube/master/YOUTUBE.DIR/wines.csv"
my.wines <- read.csv(url, header=T)
my.scaled.wines <- scale(my.wines[,-1]) # Scale the data
my.cov <- cor(my.wines[,-1])
my.eigen <- eigen(my.cov)
```

```
options(digits=3)
my.eigen
```

```
## eigen() decomposition
## $values
## [1]  4.76e+00  1.81e+00  3.53e-01  7.44e-02  4.02e-16  1.08e-17 -6.54e-16
##
## $vectors
##         [,1]    [,2]     [,3]    [,4]      [,5]      [,6]      [,7]
## [1,] -0.3965 -0.1149  0.80247  0.0519 -2.23e-01  0.00e+00  3.65e-01
## [2,] -0.4454  0.1090 -0.28106 -0.2745  5.71e-01  1.22e-02  5.56e-01
## [3,] -0.2646  0.5854 -0.09607  0.7603 -2.19e-15  1.80e-17  1.18e-16
## [4,]  0.4160  0.3111  0.00734 -0.0939 -2.04e-01  7.08e-01  4.23e-01
## [5,] -0.0485  0.7245  0.21611 -0.5474 -7.02e-02 -2.27e-01 -2.65e-01
## [6,] -0.4385 -0.0555 -0.46576 -0.1687 -7.41e-01 -9.10e-03  1.01e-01
## [7,] -0.4547 -0.0865  0.06430 -0.0835  1.71e-01  6.69e-01 -5.46e-01
```

Eigen values/vectors show up a lot in math - like with Principal Components. The loadings dervied in the previous slide are the principal componenents

```
(loadings <- my.eigen$vectors)
```

```
##         [,1]    [,2]     [,3]    [,4]      [,5]      [,6]      [,7]
## [1,] -0.3965 -0.1149  0.80247  0.0519 -2.23e-01  0.00e+00  3.65e-01
## [2,] -0.4454  0.1090 -0.28106 -0.2745  5.71e-01  1.22e-02  5.56e-01
## [3,] -0.2646  0.5854 -0.09607  0.7603 -2.19e-15  1.80e-17  1.18e-16
## [4,]  0.4160  0.3111  0.00734 -0.0939 -2.04e-01  7.08e-01  4.23e-01
## [5,] -0.0485  0.7245  0.21611 -0.5474 -7.02e-02 -2.27e-01 -2.65e-01
## [6,] -0.4385 -0.0555 -0.46576 -0.1687 -7.41e-01 -9.10e-03  1.01e-01
## [7,] -0.4547 -0.0865  0.06430 -0.0835  1.71e-01  6.69e-01 -5.46e-01
```

```
(scores <- my.scaled.wines %*% loadings)
```

```
##        [,1]  [,2]    [,3]    [,4]      [,5]      [,6]      [,7]
## [1,] -2.330  1.10  0.6713  0.0943  8.05e-16  0.00e+00 -1.11e-16
## [2,] -2.084 -1.22 -0.6563  0.1337 -2.08e-15  0.00e+00  9.99e-16
## [3,]  0.167 -0.37  0.0608 -0.4809  1.39e-15 -1.14e-17 -7.46e-17
## [4,]  1.784  1.71 -0.5492  0.0646 -8.60e-16  2.22e-16  1.11e-16
## [5,]  2.463 -1.21  0.4735  0.1882  7.49e-16 -3.33e-16 -6.66e-16
```

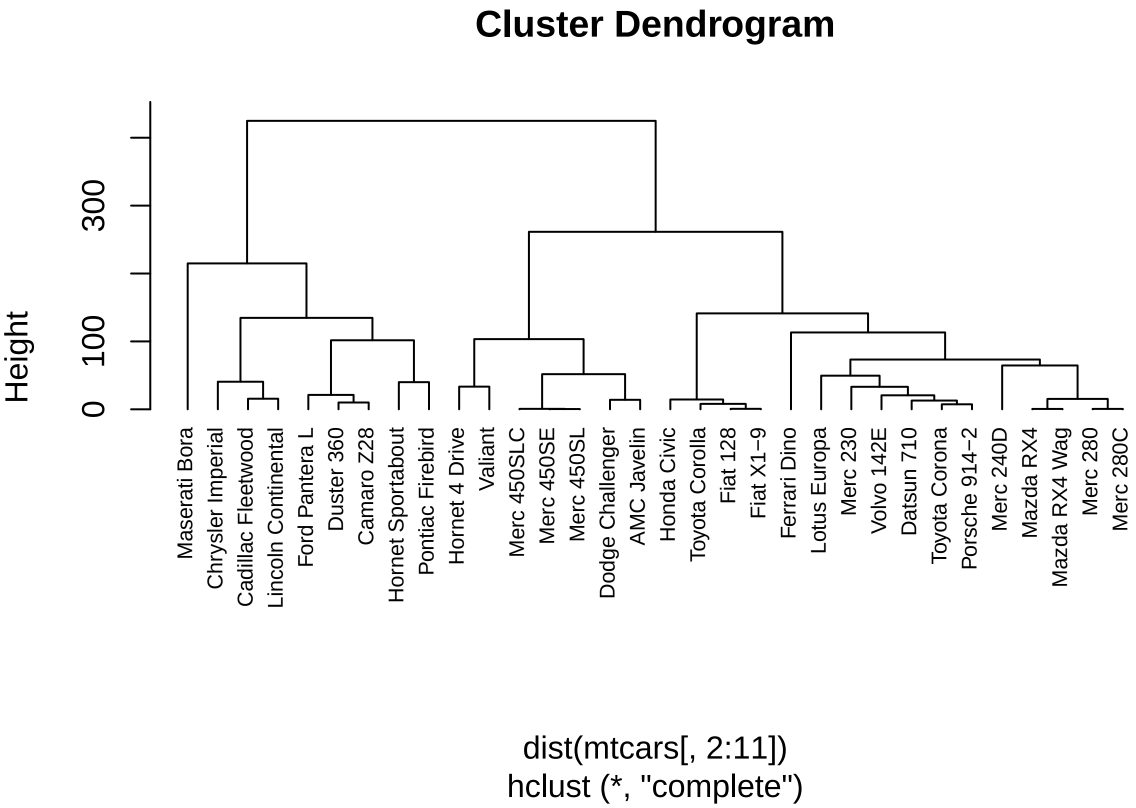
Matrices are also used a lot in cluster analysis. Let's look at a matrix of 32 cars and attempt to cluster them according to their various attributes such as MPG, Number of Cylinders, Gears, Weight, etc. This data set (mtcars) is internal to R so you can refer to it easily.

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat   wt qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.62 16.5  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.88 17.0  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.32 18.6  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.21 19.4  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.44 17.0  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.46 20.2  1  0    3    1
```

We first compute a distance between the rows and then cluster them.

```
hc <- hclust(dist(mtcars[,2:11]))
plot(hc, hang = -1,cex=0.7)
```



We can create matrices using the replicate command Useful when doing repeated sampling activity like bootstrapping

```
replicate(4,rnorm(5))
```

```
##         [,1]  [,2]    [,3]    [,4]
## [1,] -0.109 1.690 -1.0489 -0.7335
## [2,] -0.117 0.504  1.2948 -0.2159
## [3,]  0.183 2.528  0.8255 -0.3349
## [4,]  1.281 0.549 -0.0557 -1.0857
## [5,] -1.727 0.238 -0.7844 -0.0854
```

```
some.population <- rnorm(1000)
replicate(4,sample(some.population, 5, replace=TRUE))
```

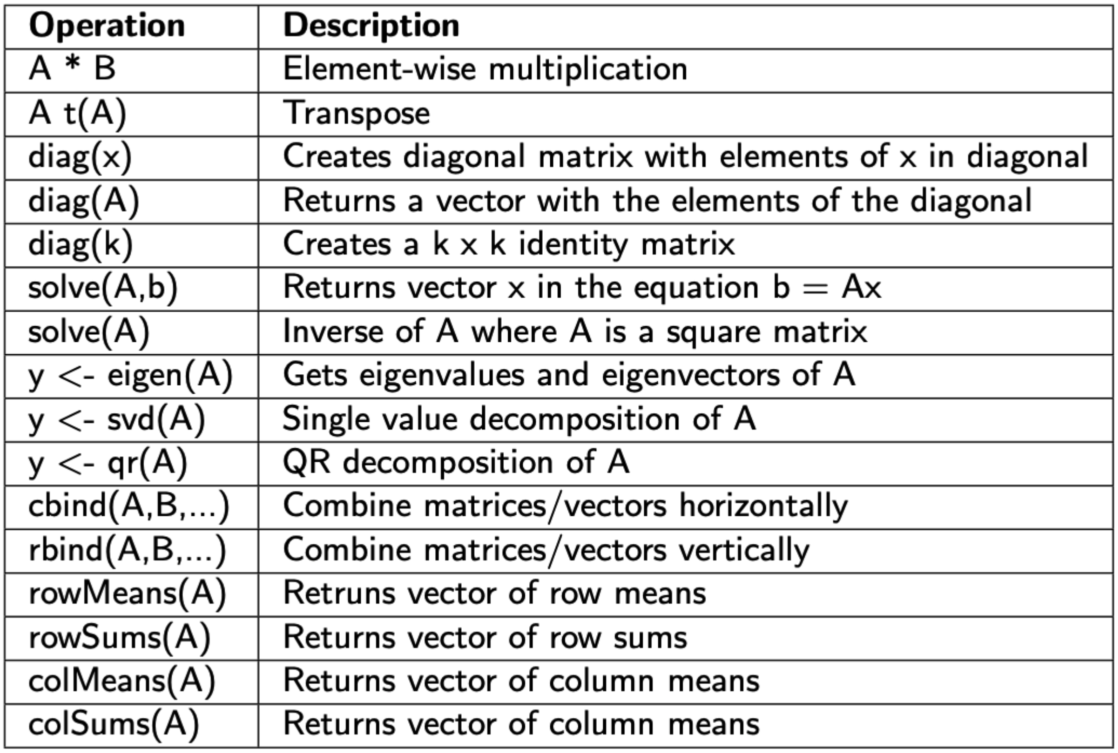
```
##         [,1]     [,2]    [,3]    [,4]
## [1,] -0.884 -0.84250 -0.1524 -0.0460
## [2,]  1.201  0.96038  0.0687  0.8483
## [3,]  0.322  0.83415  0.1498 -1.6614
## [4,]  0.440 -0.00514  1.3806 -0.0249
## [5,]  0.834 -0.79715  0.6822 -0.3125
```

Here are some Matrix functions:

Lists

Lists provide a way to store information of different types within a single data structure

Remember that vectors and matrices restrict us to only one data type at a time. That is we cannot mix, for example, characters and numbers within a vector or matrix.



Operation	Description
A * B	Element-wise multiplication
A t(A)	Transpose
diag(x)	Creates diagonal matrix with elements of x in diagonal
diag(A)	Returns a vector with the elements of the diagonal
diag(k)	Creates a k x k identity matrix
solve(A,b)	Returns vector x in the equation b = Ax
solve(A)	Inverse of A where A is a square matrix
y <- eigen(A)	Gets eigenvalues and eigenvectors of A
y <- svd(A)	Single value decomposition of A
y <- qr(A)	QR decomposition of A
cbind(A,B,...)	Combine matrices/vectors horizontally
rbind(A,B,...)	Combine matrices/vectors vertically
rowMeans(A)	Retruns vector of row means
rowSums(A)	Returns vector of row sums
colMeans(A)	Returns vector of column means
colSums(A)	Returns vector of column means

Figure 1: Some useful matrix functions

Many functions in R return information stored in lists

Consider the following example wherein we store information about a family. Not all this information is of the same type

```
family1 <- list(husband="Fred",
                wife="Wilma",
                numofchildren=3,
                agesofkids=c(8,11,14))

 length(family1)  # Has 4 elements
```

```
## [1] 4
```

```
 family1
```

```
## $husband
## [1] "Fred"
##
## $wife
## [1] "Wilma"
##
## $numofchildren
## [1] 3
##
## $agesofkids
## [1]  8 11 14
```

```
 str(family1)
```

```
## List of 4
##  $ husband      : chr "Fred"
##  $ wife         : chr "Wilma"
##  $ numofchildren: num 3
##  $ agesofkids   : num [1:3] 8 11 14
```

If possible, always create named elements. It is easier for humans to index into a named list

```
family1 <- list(husband="Fred",
                wife="Wilma",
                numofchildren=3,
                agesofkids=c(8,11,14))
 # If the list elements have names then use "$" to access the element
family1$agesofkids
```

```
## [1]  8 11 14
```

If the list elements have no names then you have to use numeric indexing

```
(family2 <- list("Barney","Betty",2,c(4,6)))
```

```
## [[1]]
## [1] "Barney"
##
## [[2]]
## [1] "Betty"
##
## [[3]]
## [1] 2
```

```
##
## [[4]]
## [1] 4 6
```

```
family2 <- list("Barney","Betty",2,c(4,6))

family2[4] # Accesses the 4th index and associated element
```

```
## [[1]]
## [1] 4 6
```

```
family2[[4]] # Accesses the 4th element value only - more direct
```

```
## [1] 4 6
```

```
family2[3:4] # Get 3rd and 4th indices and associate values
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 4 6
```

As newcomers to R we usually doesn't create lists except in two major cases:

1) We are writing a function that does some interesting stuff and we want to return to the user a structure that has information of varying types
2) As a precursor to creating a a data frame, which represents a hybrid object with characteristics of a list and a matrix

As an example of the first case, R has lots of statistical functions that return lists of information.

```
data(mtcars) # Load mtcars into the environment
mylm <- lm(mpg ~ wt, data = mtcars)
print(mylm)
```

```
##
## Call:
## lm(formula = mpg ~ wt, data = mtcars)
##
## Coefficients:
## (Intercept)           wt
##       37.29        -5.34
```

But there is a lot more information

```
str(mylm,attr=FALSE)
```

```
## List of 12
##  $ coefficients : Named num [1:2] 37.29 -5.34
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "wt"
##  $ residuals    : Named num [1:32] -2.28 -0.92 -2.09 1.3 -0.2 ...
##   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive" ...
##  $ effects      : Named num [1:32] -113.65 -29.116 -1.661 1.631 0.111 ...
##   ..- attr(*, "names")= chr [1:32] "(Intercept)" "wt" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:32] 23.3 21.9 24.9 20.1 18.9 ...
##   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive" ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:32, 1:2] -5.657 0.177 0.177 0.177 0.177 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive" ...
##   .. .. ..$ : chr [1:2] "(Intercept)" "wt"
##   .. ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.18 1.05
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 30
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = mpg ~ wt, data = mtcars)
##  $ terms        :Classes 'terms', 'formula'  language mpg ~ wt
##   .. ..- attr(*, "variables")= language list(mpg, wt)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "mpg" "wt"
##   .. .. .. ..$ : chr "wt"
##   .. ..- attr(*, "term.labels")= chr "wt"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(mpg, wt)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "mpg" "wt"
##  $ model        :'data.frame':   32 obs. of  2 variables:
##   ..$ mpg: num [1:32] 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##   ..$ wt : num [1:32] 2.62 2.88 2.32 3.21 3.44 ...
##   ..- attr(*, "terms")=Classes 'terms', 'formula'  language mpg ~ wt
##   .. .. ..- attr(*, "variables")= language list(mpg, wt)
##   .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. ..$ : chr [1:2] "mpg" "wt"
##   .. .. .. .. ..$ : chr "wt"
##   .. .. ..- attr(*, "term.labels")= chr "wt"
##   .. .. ..- attr(*, "order")= int 1
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 1
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. .. ..- attr(*, "predvars")= language list(mpg, wt)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. .. ..- attr(*, "names")= chr [1:2] "mpg" "wt"
##  - attr(*, "class")= chr "lm"
```

```
names(mylm)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

```
mylm$effects
```

```
## (Intercept)          wt                                                  
##    -113.650     -29.116      -1.661       1.631       0.111      -0.384 
##                                                                         
##      -3.607       4.500       2.691       0.611      -0.789       1.114 
##                                                                         
##       0.232      -1.606       1.301       2.214       6.100       7.309 
##                                                                         
##       2.242       6.896      -2.201      -2.669      -3.415      -3.192 
##                                                                         
##       2.735       0.820       0.595       1.707      -4.205      -2.402 
##                                                                         
##      -2.907      -0.649
```

```
lm(mpg ~ wt, data = mtcars)$coefficients
```

```
## (Intercept)          wt 
##       37.29       -5.34
```

Some other basic R functions will return a list - such as some of the character functions:

```
mystring <- "This is a test"
mys <- strsplit(mystring, " ")
str(mys)
```

```
## List of 1
##  $ : chr [1:4] "This" "is" "a" "test"
```

```
mys[[1]][1]
```

```
## [1] "This"
```

```
mys[[1]][1:2]
```

```
## [1] "This" "is"
```

```
unlist(mys)
```

```
## [1] "This" "is"   "a"    "test"
```

When we create our own functions we can return a list

```
my.summary <- function(x) {
  return.list <- list()
  return.list$mean <- mean(x)
  return.list$sd <- sd(x)
  return.list$var <- var(x)
  return(return.list)
}
```

```
my.summary(1:10)
```

```
## $mean
## [1] 5.5
## 
## $sd
## [1] 3.03
##
```

```
## $var
## [1] 9.17
```

Remember the sapply command ? We use it to apply a function over each element of a list or a vector. This helps us avoid having to write a “for-loop” every time we want to process a list or a vector.

```
family1 <- list(husband="Fred",
                wife="Wilma",
                numofchildren=3,
                agesofkids=c(8,11,14))

sapply(family1,class)
```

```
##       husband          wife numofchildren    agesofkids
##   "character"   "character"     "numeric"     "numeric"
```

```
sapply(family1,length)
```

```
##       husband          wife numofchildren    agesofkids
##             1             1             1             3
```

sapply tries to return a “simplified” version of the output (either a vector or matrix) hence the ”s”in the ”sapply”. If you don’t use something like sapply then the example on the previous slide would look this:

```
family1 <- list(husband="Fred",
                wife="Wilma",
                numofchildren=3,
                agesofkids=c(8,11,14))

 for (ii in 1:length(family1)) {
    cat(names(family1)[ii]," : ",class(family1[[ii]]),"\n")
}
```

```
## husband  :  character
## wife  :  character
## numofchildren  :  numeric
## agesofkids  :  numeric
```

Similar to sapply, the lapply function let’s you“apply”some function over each element of a list or vector. It will return a list version of the output hence the“l”in the“lapply”. So deciding between sapply and lapply simply is a question of format. What do you want back ? A vector or list ? Most of the time I use sapply.

```
# lapply( vector_or_list, function_to_apply_to_each element)
family1 <- list(husband="Fred",
                wife="Wilma",
                numofchildren=3,
                agesofkids=c(8,11,14))

lapply(family1,class)
```

```
## $husband
## [1] "character"
##
## $wife
## [1] "character"
##
## $numofchildren
## [1] "numeric"
```

```
##
## $agesofkids
## [1] "numeric"
```

check out the following:

```
lapply(family1,mean)
```

```
## Warning in mean.default(X[[i]], ...): argument is not numeric or logical:
## returning NA

## Warning in mean.default(X[[i]], ...): argument is not numeric or logical:
## returning NA

## $husband
## [1] NA
##
## $wife
## [1] NA
##
## $numofchildren
## [1] 3
##
## $agesofkids
## [1] 11
```

We can write our own processing function that checks to see if the list element is valid input for the mean function.

```
my.func <- function(x) { if(class(x)=="numeric") {
    return(mean(x))
  }
}
lapply(family1, my.func)
```

```
## $husband
## NULL
##
## $wife
## NULL
##
## $numofchildren
## [1] 3
##
## $agesofkids
## [1] 11
```