

Matrices - Intro

R also supports **matrices**, which are objects that typically refer to a numeric array of rows and columns.

Matrices are ideal for storing information on gene expression and metabolomic data as well as many other types of scientific information.

Arrays, (matrices with dimensions greater than 2), can easily handle multi-dimensional research types.

There are two common ways to create matrices in R:

Matrices - Creating

There are two common ways to create matrices in R:

1) The “dim” command turns the vector into a matrix

```
myvec = c(1:12)
```

```
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))
```

```
Error in dim(myvec = c(5, 4)) :
```

```
supplied argument name 'myvec' does not match 'x'
```

Matrices - Creating

There are three common ways to create matrices in R:

2) Using the matrix command

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

```
mymat
```

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

You can specify explicitly the nrow and ncol arguments. Note also that you can 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
```

Matrices - Naming Rows and Columns

It is useful to name the rows and columns of a matrix.

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

```
X
      [,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 and 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
```

The R Book – Michael J. Crawley

Matrices - Naming Rows and Columns

And we can do 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

The R Book – Michael J. Crawley

Matrices - Naming Rows and Columns

You aren't restricted to naming things with a pattern (though it is usually preferable).

```
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

The R Book - Michael J. Crawley

Matrices - Indexing by Name

Names provide a convenient way to index into a matrix

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 #1
```

aspirin	paracetamol	nurofen	hedex	placebo
1	4	1	2	1

```
# Get's the nurofen column for Trial.1
```

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

The R Book - Michael J. Crawley

Matrices - Indexing By Name

```
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[, 'nurofen']                # Get all Trials for nurofen
Trial.1 Trial.2 Trial.3 Trial.4
      1      1      4      1

X[, 'nurofen', drop=FALSE]    # Preserves matrix structure if desired
      nurofen
Trial.1      1
Trial.2      1
Trial.3      4
Trial.4      1
```

The R Book - Michael J. Crawley

Matrices - Numeric Indexing

It is more common to use numeric indexing.

```
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]      # First row, First Column
[1] 1
```

```
X[2,2]      # Second row, Second Column
[1] 4
```

```
X[3,3]      # Third row, Third column
[1] 1
```

```
diag(X)      # Ah, there is a function that gets the diagonal.
[1] 1 4 1     # Always check to see if there is already a function
              # to do what you want
```

Matrices - Indexing

You need to know how to extract information from a matrix. This can be confusing at first but becomes much easier with practice:

X

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

X[1:2,1] # Gets First and second rows and the first column
[1] 1 2

X[1:2,2] # Gets First and second rows and the second column
[1] 3 4

X[1:2,] # Gets First and second rows and ALL columns
 [,1] [,2] [,3]
[1,] 1 3 1
[2,] 2 4 3

Matrices - Indexing Like a Vector

Keep in mind that a matrix is basically a vector with dimensions so you can index into it as if it were a vector. This might not be so intuitive at first:

```
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)  # Returns which elements are greater or equal to 2
```

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

Matrices - Indexing Like a Vector

Keep in mind that a matrix is basically a vector with dimensions so you can index into it as if it were a vector. This might not be so intuitive at first:

X

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

X %% 2 == 0 # Returns a logical matrix

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

> X[X %% 2 == 0] # Finds the actual element values
[1] 2 4 0

Matrices - Indexing Like a Vector

Keep in mind that a matrix is basically a vector with dimensions so you can index into it as if it were a vector. This might not be so intuitive at first:

X

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

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

X

	[,1]	[,2]	[,3]
[1,]	1	3	1
[2,]	99	99	3
[3,]	1	99	1

Matrices - Indexing Like a Vector

There are two functions called `row` and `col` that return the numeric row and column, respectively of the matrix. Kinda weird but useful. An example is in order.

X

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

`row(X)`

	[,1]	[,2]	[,3]
[1,]	1	1	1
[2,]	2	2	2
[3,]	3	3	3

The values correspond to the actual row number

`col(X)`

	[,1]	[,2]	[,3]
[1,]	1	2	3
[2,]	1	2	3
[3,]	1	2	3

The values correspond to the actual col number

Matrices - Indexing Like a Vector

There are two functions called `row` and `col` that return the numeric row and column, respectively of the matrix. Kinda weird but useful. An example is in order.

```
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 4 1
```

```
X[row(X) == col(X)] = 0 # Put zeroes in the diagonal
```

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

Matrices - Adding Rows and Columns

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

Matrices - Adding Rows and Columns

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

Matrices - Doing Calculations

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

```
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
```

```
var(X[3,]) # Variance of the 3rd row  
[1] 0.3333333
```

Matrices - Doing Calculations

Let's look at some examples involving calculations on matrices. But there are some general functions to help with this kind of thing:

```
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
```

Maybe columns represent protein expression and you are trying to determine if there are differences between the mean expression levels.

The R Book – Michael J. Crawley

Matrices - Doing Calculations

But there are some general functions to help with this kind of thing:

```
rowMeans(X)
  Trial.1  Trial.2  Trial.3
1.666667 3.000000 0.666667

colMeans(X)
  aspirin paracetamol  nurofen
1.333333  2.333333  1.666667

colMeans(X)[3]
nurofen
1.666667
```

These are fast and can work on very large matrices. Though be careful if you have missing values in your data.

The R Book – Michael J. Crawley

Matrices - Linear Algebra

R supports common linear algebra operations also.

```
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

$$\begin{bmatrix} 1 & 2 \\ 3 & 8 \\ 2 & 9 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 2 \\ 2 & 8 & 9 \end{bmatrix}$$

<http://bendixcarstensen.com/APC/linalg-notes-BxC.pdf>

Matrices - Linear Algebra

A

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

B = matrix(c(5,8,4,2),2,2)

A %% B

	[,1]	[,2]
[1,]	21	8
[2,]	79	28
[3,]	82	26

$$\begin{aligned} \begin{bmatrix} 1 & 2 \\ 3 & 8 \\ 2 & 9 \end{bmatrix} \begin{bmatrix} 5 & 4 \\ 8 & 2 \end{bmatrix} &= \left[\begin{bmatrix} 1 & 2 \\ 3 & 8 \\ 2 & 9 \end{bmatrix} \begin{bmatrix} 5 \\ 8 \end{bmatrix} : \begin{bmatrix} 1 & 2 \\ 3 & 8 \\ 2 & 9 \end{bmatrix} \begin{bmatrix} 4 \\ 2 \end{bmatrix} \right] \\ &= \begin{bmatrix} 1 \cdot 5 + 2 \cdot 8 & 1 \cdot 4 + 2 \cdot 2 \\ 3 \cdot 5 + 8 \cdot 8 & 3 \cdot 4 + 8 \cdot 2 \\ 2 \cdot 5 + 9 \cdot 8 & 2 \cdot 4 + 9 \cdot 2 \end{bmatrix} = \begin{bmatrix} 21 & 8 \\ 79 & 28 \\ 82 & 26 \end{bmatrix} \end{aligned}$$

<http://bendixcarstensen.com/APC/linalg-notes-BxC.pdf>

Matrices - Linear Algebra

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

```
A = matrix(1:4,2,2)
```

A

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

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

B

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

```
A %% B      # We get the identity matrix
```

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

<http://bendixcarstensen.com/APC/linalg-notes-BxC.pdf>

Matrices - Linear Algebra

Suppose you have the following system of equations. This can be represented as:

$$\begin{array}{rcl} x_1 + 3x_2 & = & 7 \\ 2x_1 + 4x_2 & = & 10 \end{array}$$

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 7 \\ 10 \end{bmatrix} \text{ i.e. } Ax = b$$

A

$$\begin{array}{cc} & [, 1] & [, 2] \\ [1,] & 1 & 3 \\ [2,] & 2 & 4 \end{array}$$

b = c(7,10)

x = solve(A) %*% b

x

$$\begin{array}{cc} & [, 1] \\ [1,] & 1 \\ [2,] & 2 \end{array}$$

Since $A^{-1}A = I$ and since $Ix = x$ we have

$$x = A^{-1}b = \begin{bmatrix} -2 & 1.5 \\ 1 & -0.5 \end{bmatrix} \begin{bmatrix} 7 \\ 10 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

<http://bendixcarstensen.com/APC/linalg-notes-BxC.pdf>

Supplemental - Matrices - Linear Algebra

B

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

```
diag(B)          # Fetches the diagonal
[1] 5 2
```

```
diag(c(1,2,3))    # Creates a matrix with 1,2,3 on the diagonal
      [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

```
diag(1,4)         # Creates a 4 x 4 Identity matrix
      [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

<http://bendixcarstensen.com/APC/linalg-notes-BxC.pdf>

Supplemental - Matrices - eigen values

Eigen values and vectors show up a lot in statistics - like with Principal Components Analysis.

```
my.wines = read.csv("http://www.bimcore.emory.edu/wine.csv", header=T)

my.scaled.wines = scale(my.wines)    # Scale the data

my.cov = cov(my.scaled.wines)        # Get the covariance matrix

my.eigen = eigen(my.cov)              # Now find the eigen values/vectors

options(digits=3)

my.eigen                                # Check out the Eigen values and vectors
$values
[1]  4.76e+00  1.81e+00  3.53e-01  7.44e-02  3.73e-16 -2.61e-16 -2.99e-16

$vectors
      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
[1,] -0.3965  0.1149  0.80247  0.0519 -1.46e-01  0.00e+00 -4.02e-01
[2,] -0.4454 -0.1090 -0.28106 -0.2745  4.84e-01 -5.18e-01 -3.64e-01
[3,] -0.2646 -0.5854 -0.09607  0.7603  5.41e-16  3.75e-16 -1.16e-15
[4,]  0.4160 -0.3111  0.00734 -0.0939  3.24e-01  4.88e-01 -6.15e-01
[5,] -0.0485 -0.7245  0.21611 -0.5474 -2.16e-01 -3.23e-02  2.80e-01
[6,] -0.4385  0.0555 -0.46576 -0.1687 -5.67e-01  3.86e-01 -2.97e-01
[7,] -0.4547  0.0865  0.06430 -0.0835  5.20e-01  5.85e-01  4.01e-01

$loadings = my.eigen$vectors
```

Supplemental - Matrices - eigen values

Eigen values and vectors show up a lot in statistics - like with Principal Components Analysis.

The loadings are the principal components

```
loadings = my.eigen$vector
```

The scores are the product of the matrix multiplication between the scaled.wines and the loadings. This takes the original wine data and re-expresses it in terms of the “principal components”.

```
scores = my.scaled.wines %*% loadings
```

Supplemental - Matrices - Cluster Analysis

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.

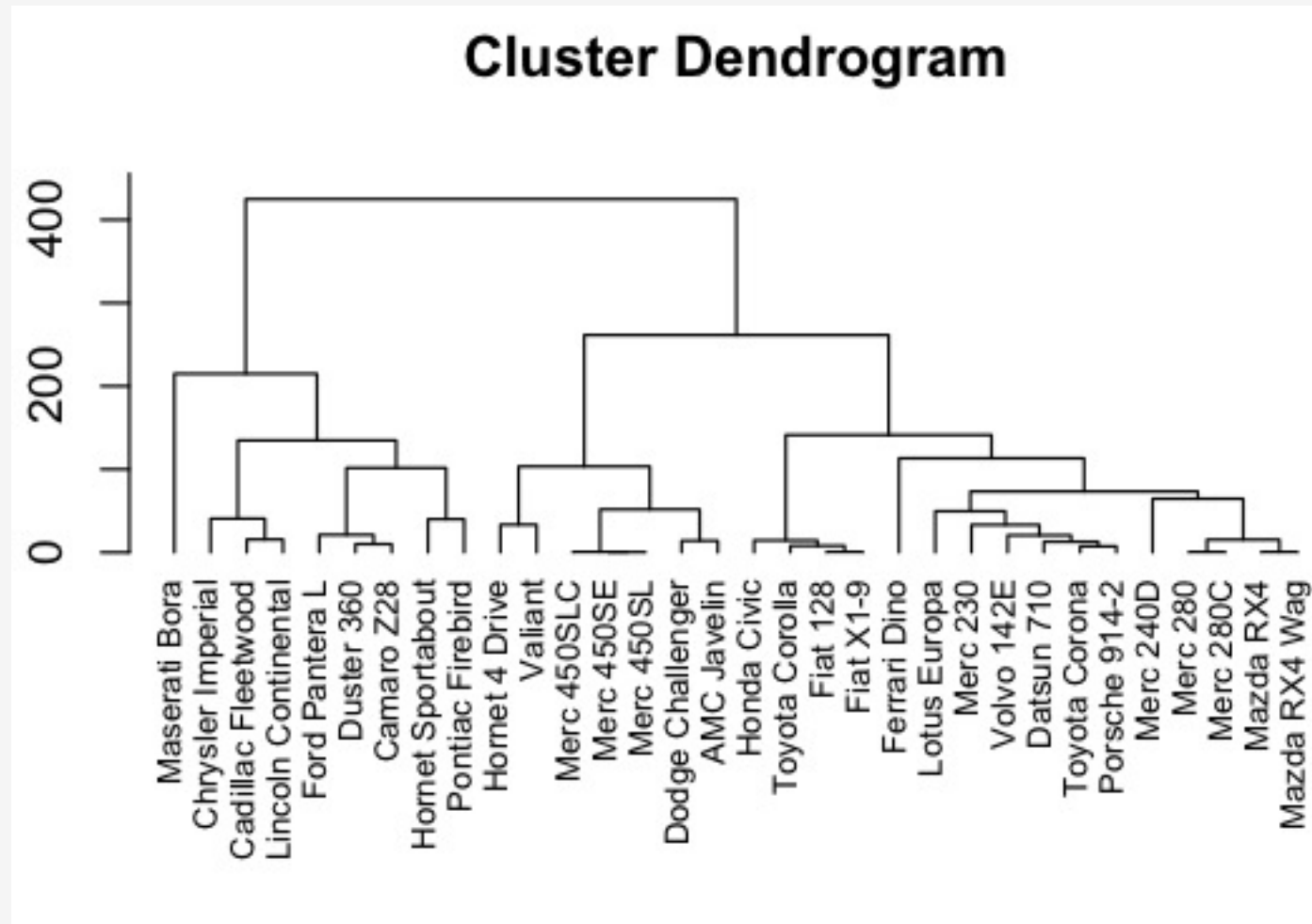
```
mtcars
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.62	16.5	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.88	17.0	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.32	18.6	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.21	19.4	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.44	17.0	0	0	3	2
..											
..											

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

```
hc <- hclust(dist(mtcars[,2:11])) # The first column is a label.  
plot(hc, hang = -1, cex=0.7)
```

Supplemental - Matrices - Cluster Analysis



Supplemental - Matrices - Alternative Ways to

We can create matrices using the replicate command. This approach is useful if you are trying to capture the results of repeated sampling activity like when bootstrapping. In the simplest case here is an example. This generates a 4 column matrix with 5 rows. Each time we generate a new column we are effectively getting a new sample of data from a normal distribution.

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

	[,1]	[,2]	[,3]	[,4]
[1,]	-1.181720384	0.2717525	-1.4716542	2.26654104
[2,]	0.268970133	0.3423814	0.9553185	0.07749788
[3,]	0.007413904	-1.2102476	0.2273662	-0.46742459
[4,]	1.726961040	0.9977138	2.0491924	0.77174367
[5,]	0.950821481	-1.8599874	-0.8587209	0.95906263

```
some.population = rnorm(1000)
```

```
replicate(4,sample(some.population, 5, replace=TRUE))
```

	[,1]	[,2]	[,3]	[,4]
[1,]	-0.4680211	0.27727612	-0.5346220	0.94161600
[2,]	0.3138391	0.36105532	0.1108916	0.35186402
[3,]	-1.8416441	-0.05812402	1.3535505	0.05288187
[4,]	-0.9483933	-0.24572418	1.6950778	1.30636068
[5,]	1.0369327	-0.66983941	0.3055545	1.57318148

Supplemental - Matrices - Alternative Ways to

So if we have done a repeated sample from a population we could then process each column to see if a hypothesized mean fell into a confidence interval. So in this case we kind of know what the true mean is but let's pretend we don't.

```
set.seed(177)
```

```
some.pop = rnorm(1000)
```

```
mean(some.pop)  
[1] -0.01982588
```

```
my.boot = replicate(4, sample(some.pop, 5, replace=T))
```

```
      [,1]      [,2]      [,3]      [,4]  
[1,] -0.06820582 -0.1160524  0.3222247  1.1403365  
[2,] -0.96508173  0.4916324 -0.9598382 -1.9968074  
[3,]  0.26232581 -0.7655528  0.9046675 -0.3021725  
[4,] -0.02585224 -1.1985142 -1.0101444 -0.3309738  
[5,]  0.92775589 -1.1999768 -1.2637646  0.5954983
```

Supplemental - Matrices - Some Functions

<code>A * B</code>	Element-wise multiplication
<code>A %**% B</code>	Matrix multiplication
<code>A %o% B</code>	Outer product. AB'
<code>crossprod(A,B)</code> <code>crossprod(A)</code>	$A'B$ and $A'A$ respectively.
<code>t(A)</code>	Transpose
<code>diag(x)</code>	Creates diagonal matrix with elements of x in the principal diagonal
<code>diag(A)</code>	Returns a vector containing the elements of the principal diagonal
<code>diag(k)</code>	If k is a scalar, this creates a $k \times k$ identity matrix. Go figure.
<code>solve(A, b)</code>	Returns vector x in the equation $b = Ax$ (i.e., $A^{-1}b$)
<code>solve(A)</code>	Inverse of A where A is a square matrix.
<code>ginv(A)</code>	Moore-Penrose Generalized Inverse of A . <code>ginv(A)</code> requires loading the MASS package.

Supplemental - Matrices - Some Functions

<code>y<-eigen(A)</code>	<code>y\$val</code> are the eigenvalues of <code>A</code> <code>y\$vec</code> are the eigenvectors of <code>A</code>
<code>y<-svd(A)</code>	Single value decomposition of <code>A</code> . <code>y\$d</code> = vector containing the singular values of <code>A</code> <code>y\$u</code> = matrix with columns contain the left singular vectors of <code>A</code> <code>y\$v</code> = matrix with columns contain the right singular vectors of <code>A</code>
<code>R <- chol(A)</code>	Choleski factorization of <code>A</code> . Returns the upper triangular factor, such that $R'R = A$.
<code>y <- qr(A)</code>	QR decomposition of <code>A</code> . <code>y\$qr</code> has an upper triangle that contains the decomposition and a lower triangle that contains information on the Q decomposition. <code>y\$rank</code> is the rank of <code>A</code> . <code>y\$qlraux</code> a vector which contains additional information on Q. <code>y\$pivot</code> contains information on the pivoting strategy used.
<code>cbind(A,B,...)</code>	Combine matrices(vectors) horizontally. Returns a matrix.
<code>rbind(A,B,...)</code>	Combine matrices(vectors) vertically. Returns a matrix.
<code>rowMeans(A)</code>	Returns vector of row means.
<code>rowSums(A)</code>	Returns vector of row sums.
<code>colMeans(A)</code>	Returns vector of column means.
<code>colSums(A)</code>	Returns vector of column means.

Factors - Intro

R supports factors, which are a special data type for, among other things, managing categories of data.

“One of the most important uses of factors is in statistical modeling; since categorical variables enter into statistical models differently than continuous variables, storing data as factors insures that the modeling functions will treat such data correctly”.

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.

Factors - Intro

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(1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1)
```

```
table(transvec)    # Count 'em up. Which are Auto and Manual ?
```

```
transvec
```

```
 0  1
```

```
19 13
```

```
mytransfac = factor(transvec, levels = c(0,1), labels = c("Auto","Man") )
```

```
table(mytransfac)
```

```
mytransfac
```

```
Auto  Man
```

```
 19   13
```

```
levels(mytransfac)
```

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

```
mytransfac
```

```
[1] Man  Man  Man  Auto Auto Auto Auto Auto Auto Auto Auto Auto Auto Auto Auto Auto Auto
```

```
[16] Auto Auto Man  Man  Man  Auto Auto Auto Auto Auto Auto Man  Man  Man  Man  Man
```

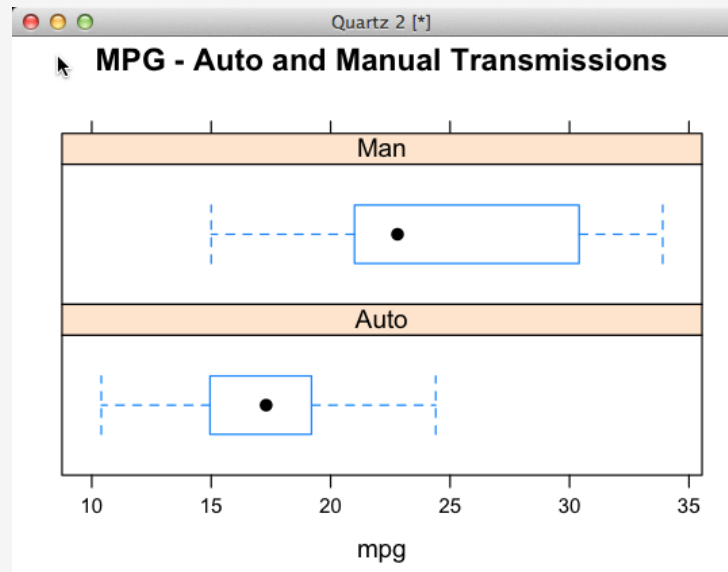
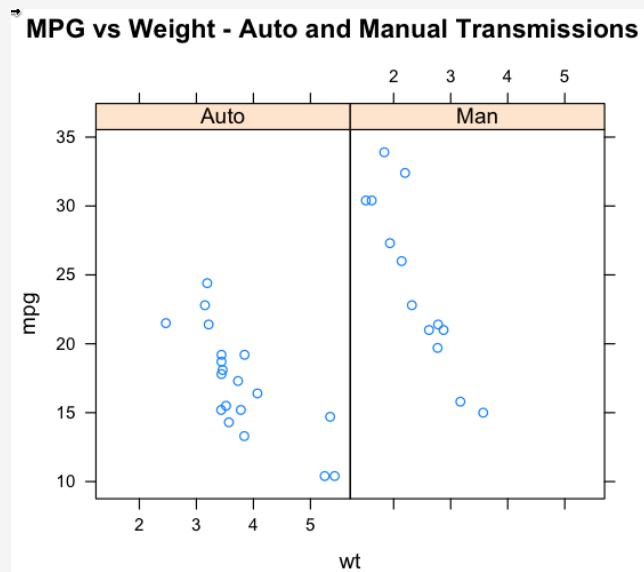
```
[31] Man  Man
```

```
Levels: Auto Man
```

Factors - Intro

R knows how to handle factors when doing plots. Here we 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))
```



Factors - 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,  
          10.4,14.7,32.4,30.4,33.9,21.5,15.5,15.2,13.3,19.2,27.3,26,30.4,15.8,19.7,15,21.4)  
  
tapply( continuous_value_to_summarize, factor_or_grouping_variable, function_for_summary)  
  
tapply(mympg,mytransfac,mean)  
      Auto      Man  
17.14737 24.39231
```

Factors - cut

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 1 and 10 into 4 intervals. It looks kind of messy:

```
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]      (5,7.5]      (5,7.5]
[7.5,10]      (7.5,10]
[11] (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
```

Factors - cut

Well that was cool but people like to read labels:

```
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
```

Factors - cut

But you can 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
```


Factors - cut

Another example. Let's say we have some exam scores. Let's summarize them according to the typical US 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)

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]

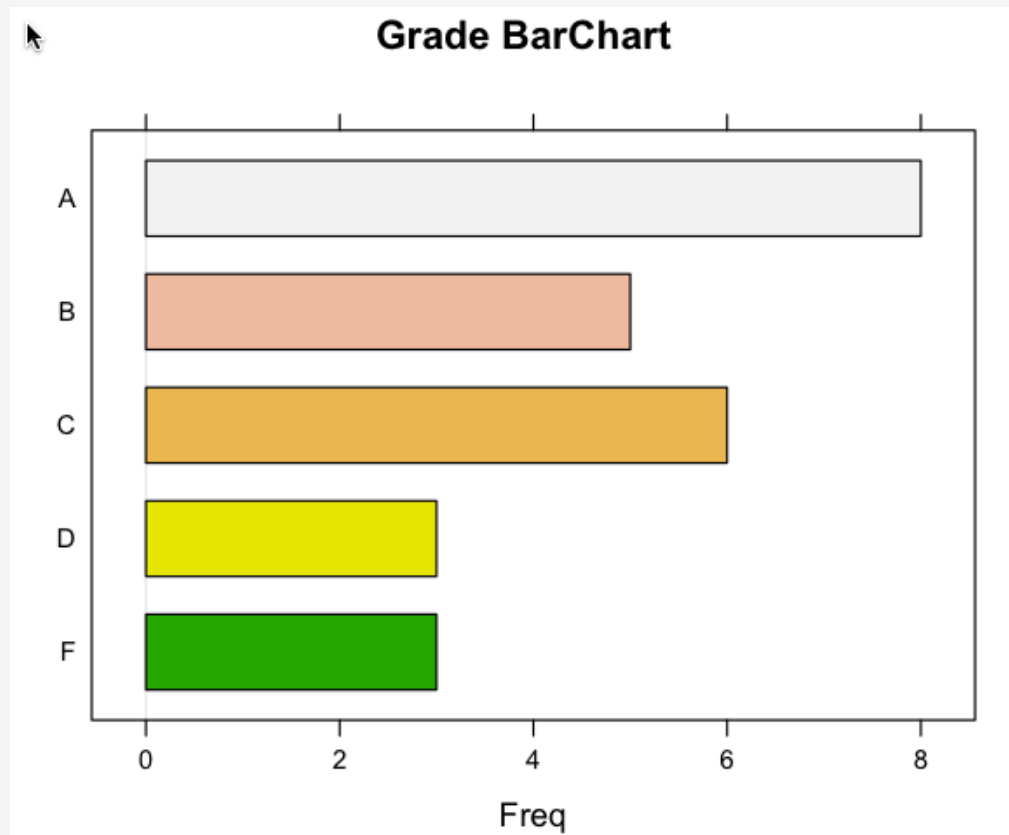
cut(exam.score,breaks=c(50,60,70,80,90,100),labels=c("F","D","C","B","A"))
[1] D B C A A F C A C C A C B C F A D F D A A B B A B
Levels: F D C B A

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

F D C B A
3 3 6 5 8

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

Factors - cut



Factors - cut

We have a small problem in that the intervals don't exactly match the grading scheme. In this scheme someone getting a grade of 90 will get a B although we intend for them to get an A. This is where you should be paying attention to the (and] characters. To make the interval exclude the "right side" of the interval we specify the "right=F" argument.

```
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)
```

Factors - cut

So if you don't think that the cut command doesn't do something interesting then here is how you would have had to the last example with the exams:

```
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) & (exam.score[ii] < 90)) {bcount = bcount +1} else
          if ((exam.score[ii] >= 90) & (exam.score[ii] <= 100)) {acount = acount +1}
}
cat("acount bcount ccount dcount fcount")
cat(acount,bcount,ccount,dcount,fcount)
acount bcount ccount dcount fcount
8 5 7 3 2
```

Factors - Ordered

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

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

Warning message:

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

```
levels(my.fact.mons)  
[1] "Apr" "Feb" "Jan" "Jun" "Mar" "May"
```

<http://www.stat.berkeley.edu/classes/s133/factors>

Factors - Ordered

```
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[2]
[1] FALSE
```

```
table(my.fact.mons)
my.fact.mons
Jan Feb Mar Apr May Jun
  2   2   2   2   2   2
```

```
levels(my.fact.mons) # This is what we want !
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun"
```

<http://www.stat.berkeley.edu/classes/s133/factors>

Supplemental Factors - AOV example

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.

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

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
factor(gear)	2	483.24	241.622	10.901	0.0002948 ***
Residuals	29	642.80	22.166		

```
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
> print(model.tables(aov.ex1,"means"))
Tables of means
Grand mean

20.09062

gear
      3      4      5
16.11 24.53 21.38
rep 15.00 12.00  5.00

par(mfrow=c(2,2))
plot(aov.ex1)
```

Supplemental Factors - AOV example

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.

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

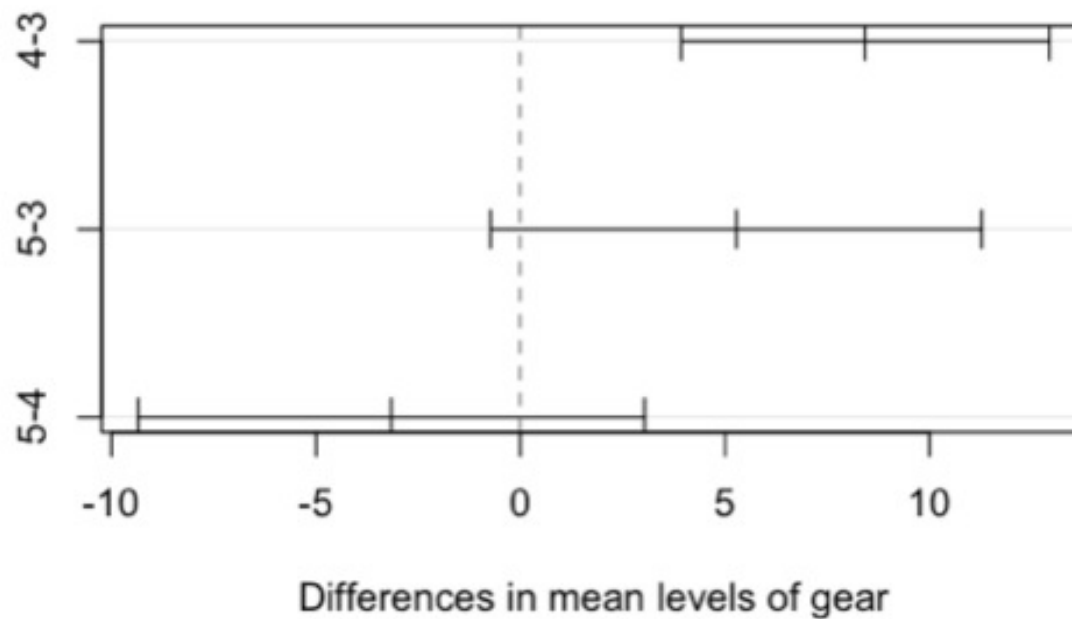
Differences between Gears are significant at 5% level if the confidence interval around the estimation of the difference does not contain zero

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


Supplemental Factors - AOV example

Using R - Factors - Tukey-MultiComp Plot

95% family-wise confidence level



Lists - Intro

- * Lists address the situation where we need to store information of different types in a single structure.
- * Remember that vectors and matrices restrict us to only one data type at a time.
- * Many functions in R return lists.

Lists - Creating

```
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
```

Lists - Indexing

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

family1$agesofkids      # If the list elements have names then use "$" to access the element
[1] 8 11 14

family1$agesofkids[1:2]
[1] 8 11

sapply(family1,class)
      husband      wife numofchildren  agesofkids
"character" "character"    "numeric"    "numeric"

sapply(family1,length)
      husband      wife numofchildren  agesofkids
           1           1           1           3
```

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
```

Lists - Indexing

If the list elements have no names then you have to use numeric indexing. But try to create lists with names as its easier to work with.

```
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
```

Lists - Indexing

You can do "unlist" on any list to turn it into a vector. Since the list has mixed data types all of the elements of the vector will be converted to a single data type. In this case character

```
unlist(family1)
      husband      wife numofchildren  agesofkids1  agesofkids2
      "Fred"      "Wilma"           "3"           "8"           "11"
      agesofkids3
      "14"
```

```
as.numeric(unlist(family1))
[1] NA NA  3  8 11 14
```

Normally we don't create lists as a "standalone" object 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 lots of information.
- 2) As a precursor to creating a data frame which is a hybrid between a list and a matrix. We'll investigate this momentarily.

Lists - Functions

R has lots of statistical functions that return lists of information. In fact this is the norm.

```
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
37.285	-5.344

But there is a lot more information

```
typeof(mylm)
```

```
[1] "list"
```

Lists - Functions

R has lots of statistical functions that return lists of information. In fact this is the norm.

```
str(mylm,give.attr=F) # Lots of stuff here
```

List of 12

```
$ coefficients : Named num [1:2] 37.29 -5.34
$ residuals    : Named num [1:32] -2.28 -0.92 -2.09 1.3 -0.2 ...
$ effects      : Named num [1:32] -113.65 -29.116 -1.661 1.631 0.111 ...
$ rank         : int 2
$ fitted.values: Named num [1:32] 23.3 21.9 24.9 20.1 18.9 ...
$ 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 ...
..$ qraux     : num [1:2] 1.18 1.05
..$ pivot     : int [1:2] 1 2
..$ tol       : num 1e-07
..$ rank      : int 2
$ df.residual  : int 30
$ xlevels      : Named list()
$ call         : language lm(formula = mpg ~ wt, data = mtcars)
$ terms        :Classes 'terms', 'formula' length 3 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 ...
```


Lists - Functions

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

mylm$effects
(Intercept)      wt
-113.6497374 -29.1157217 -1.6613339  1.6313943  0.1111305 -0.3840041
-3.6072442  4.5003125  2.6905817  0.6111305 -0.7888695  1.1143917
 0.2316793 -1.6061571  1.3014525  2.2137818  6.0995633  7.3094734
 2.2421594  6.8956792 -2.2010595 -2.6694078 -3.4150859 -3.1915608
 2.7346556  0.8200064  0.5948771  1.7073457 -4.2045529 -2.4018616
-2.9072442 -0.6494289

# Some use the $ notation to extract desired information they want straight from the function call

lm(mpg ~ wt, data = mtcars)$coefficients
(Intercept)      wt
 37.285126    -5.344472
```

Lists - Functions

When we create our own functions we can package things up into a list and return things.

```
my.summary <- function(x) {  
  return.list = list()      # Declare the 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] 3.02765  
  
$var  
[1] 9.166667  
  
names(my.summary(1:10))  
[1] "mean" "sd"  "var"  
  
my.summary(1:10)$var      # Here we exploit the $ notation to get only what we want  
[1] 9.166667
```

Lists - Functions

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] "This" "is"    "a"     "test"
```

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

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

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

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

```
unlist(mys)
```

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

Lists - Twitter

```
delta.tweets = searchTwitter('@delta', n = 100) # Uses the add-on twitterR package

class(delta.tweets)
[1] "list"

delta.tweets
[[1]]
[1] "sotsoy: Apparently if you use your frequent flier miles on @delta they stick you at the back of the plane on every flight next to the bathroom"

[[2]]
[1] "ImTooNonFiction: My @Delta flight has been delayed for the last 2 hrs. We've been on plane at gate for 2+ hours and no mention of a voucher or compensation"

[[3]]
[1] "ShaneNHara: @Delta and @DeltaAssist, thank you for a swift boarding process here at SEA en route to LAX. Taking care of your loyal flyers = appreciated."

[[4]]
[1] "NaiiOLLG: RT @TheRealNickMara: ThankYou @Delta for a great flight!! #Work!!!"

[[5]]
[1] "forbeslancaster: @bsideblog @Delta just saw a commercial highlighting delta awesome service. Totes NOT true"

..
```

Lists - Twitter

```
supply(delta.tweets,function(x) x$text()) # Pulls out the text of the tweet
```

```
[1] "Apparently if you use your frequent flier miles on @delta they stick you at the back of the  
plane on every flight next to the bathroom"
```

```
[2] "My @Delta flight has been delayed for the last 2 hrs. We've been on plane at gate for 2+ hours  
and no mention of a voucher or compensation"
```

```
[3] "@Delta and @DeltaAssist, thank you for a swift boarding process here at SEA en route to LAX.  
Taking care of your loyal flyers = appreciated."
```

```
[4] "RT @TheRealNickMara: ThankYou @Delta for a great flight!! #Work!!!"
```

```
[5] "@bsideblog @Delta just saw a commercial highlighting delta awesome service. Totes NOT true"
```

```
..  
..  
..
```

```
other results omitted due to obscenities...
```

Lists - sapply

Lastly, While we could use `sapply` to apply some statistical function across all elements of a list it might not make sense since you have different data types:

```
sapply(family1, mean)
      husband      wife numofchildren      agesofkids
           NA         NA              3             11
```

Warning messages:

```
1: In mean.default(X[[1L]], ...) :
  argument is not numeric or logical: returning NA
2: In mean.default(X[[2L]], ...) :
  argument is not numeric or logical: returning NA
```

We could write our own function to ignore non-numeric data:

```
sapply(family1, function(x) { if (is.numeric(x)) print(mean(x)) })
[1] 3
[1] 11
$husband
NULL
$wife
NULL

$numofchildren
[1] 3

$agesofkids
[1] 11
```