```
> #################### LOADING DATA #########################
> # Read in the data
> forbes <- read.table("forbes")
> # and now look at it
> forbes
      V1     V2
1     Tb   logP
2  194.5 131.79
3  194.3 131.79
4  197.9 135.02
5  198.4 135.55
6  199.4 136.46
7  199.9 136.83
8  200.9 137.82
9  201.1 138.00
10 201.4 138.06
11 201.3 138.05
12 203.6 140.04
13 204.6 142.44
14 209.5 145.47
15 208.6 144.34
16 210.7 146.30
17 211.9 147.54
18 212.2 147.80
> # Try reading in the data while recognizing the headers
> forbes <- read.table("forbes",header=T)
> # and now look at it.
> forbes
      Tb   logP
1  194.5 131.79
2  194.3 131.79
3  197.9 135.02
.... (these dots mean that more output was given, I'm just saving space)
> #
> # Note: a slick way to load data posted online is
> # data <- read.table("http://webaddressofdata", header = T)
> #
> # We can now reference the colums of 'forbes' using the headers.
> forbes$Tb
 [1] 194.5 194.3 197.9 198.4 199.4 199.9 200.9 201.1 201.4 201.3 203.6 204.6 209.5
[14] 208.6 210.7 211.9 212.2
> forbes$logP
 [1] 131.79 131.79 135.02 135.55 136.46 136.83 137.82 138.00 138.06 138.05 140.04
[12] 142.44 145.47 144.34 146.30 147.54 147.80
> # If we 'attach' the object forbes, we won't have to use the $
> attach(forbes)
> # Now reference the columns by name
> Tb
 [1] 194.5 194.3 197.9 198.4 199.4 199.9 200.9 201.1 201.4 201.3 203.6 204.6 209.5
[14] 208.6 210.7 211.9 212.2
> ################## MANIPULATING DATA ##################
> # First you can learn more about the data using the summary command
> summary(forbes)
       Tb             logP
 Min.   :194.3   Min.   :131.8
 1st Qu.:199.4   1st Qu.:136.5
 Median :201.3   Median :138.1
 Mean   :203.0   Mean   :139.6
 3rd Qu.:208.6   3rd Qu.:144.3
 Max.   :212.2   Max.   :147.8
> and the attribute command
> attributes(forbes)
$names
```

```
[1] "Tb"   "logP"

$class
[1] "data.frame"

$row.names
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14" "15"
[16] "16" "17"
> # We can access just the second column of 'forbes'.
> forbes[,2]
 [1] 131.79 131.79 135.02 135.55 136.46 136.83 137.82 138.00 138.06 138.05 140.04
[12] 142.44 145.47 144.34 146.30 147.54 147.80
> # or just the third row
> forbes[3,]
     Tb   logP
3 197.9 135.02
> # or just the second element of the column 'Tb'
> Tb[[2]]
[1] 194.3
> # Note: there are no scalars in R, hence the value returned
> # above is a vector of dim [1]. Also, the above manipulations
> # were just to show the use of the ',' operator.
> # Since 'forbes' is a data.frame, we normally access its
> # columns using the column names, Tb, logP, as shown previously.
> #
> # We can make a subset of the forbes data
> forbes.highT <- data.frame(forbes[ Tb >= 200,])
> # The comma after 200 brings the corresponding values of logP
> # into the new data.frame
> forbes.highT
      Tb   logP
7  200.9 137.82
8  201.1 138.00
9  201.4 138.06
....
16 211.9 147.54
17 212.2 147.80
> # Convert the temperature to Kelvin
> Tb_Kelv <- (5/9) * (Tb - 32 ) + 273
> # To add the temperature in Kelvin to the forbes data
> # first we make the vector Tb_Kelv into a data.frame
> Tb_Kelv <- data.frame(Tb_Kelv)
> # Now use cbind to combine the two data frames
> forbes.expanded <- cbind(forbes,Tb_Kelv)
> forbes.expanded
      Tb   logP  Tb_Kelv
1  194.5 131.79 363.2778
2  194.3 131.79 363.1667
3  197.9 135.02 365.1667
4  198.4 135.55 365.4444
5  199.4 136.46 366.0000
....
> # Rename the last column to "Tk"
> attr(forbes.expanded,"names") <-c("Tb","logP","Tk")
> forbes.expanded
      Tb   logP       Tk
1  194.5 131.79 363.2778
2  194.3 131.79 363.1667
3  197.9 135.02 365.1667
....
> # We are starting to see how R treats vectors.
> # To make a 'list'use the concatenate command c().
> # Above we made a list of names.
```

```
> # Below we make a list of numbers (i.e. a vector).
> v <- c(1,1,2,2,3,0)
> v
[1] 1 1 2 2 3 0
> v[[3]]
[1] 2
> # R does vector math.
> # Here's an example of element by element multiplication
> T_sq = Tb * Tb
> T_sq
 [1] 37830.25 37752.49 39164.41 39362.56 39760.36 39960.01 40360.81 40441.21
 [9] 40561.96 40521.69 41452.96 41861.16 43890.25 43513.96 44394.49 44901.61
[17] 45028.84
> # and here's an example of vector multiplication
> T_norm = sqrt( t(Tb) %*% Tb )
> T_norm
         [,1]
[1,] 837.1135
> # t() is the transpose, though R is smart enough that
> T_norm = sqrt( Tb %*% Tb )
> # works just as well
> #
> # Arrays are made out of vectors of data, to which we assign
> # dimensions.  For example, make a vector of numbers, 1 through 27.
> index <- seq(1:27)
> # Then divide this vector 'index' up into a 3x3x3 array.
> dim(index) <- c(3,3,3)
> index
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18

, , 3

     [,1] [,2] [,3]
[1,]   19   22   25
[2,]   20   23   26
[3,]   21   24   27

> # We see that arrays are stored column by column, similar to FORTRAN
> # Finally, you can save the data you were working with and load it
> # later
save("index", file="savetest",ascii=FALSE)
rm(index)
load(file="savetest")
index()
, , 1

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

```
> ####################################################################
> #    E X A M P L E :  S I M P L E   L I N E A R  R E G R E S S I O N
> ####################################################################
> # Now suppose we would like to fit a linear model to this data.
> # We do this using the function 'lm'.
> # To regress Y on X, use lm(Y ~ X). So in our case we have
> fit <- lm(logP ~ Tb)
> fit

Call:
lm(formula = logP ~ Tb)

Coefficients:
(Intercept)           Tb
   -42.1309       0.8955
> #
> # So this gave us a two parameter fit, intercept and slope.
> ########################## RMS ERROR ###############################
> # What else do we want to know about our linear model? How about the
> # residual mean square (sigma_hat^2)
> # The formula for sigma_hat^2 is
> # sigma_hat^2 = RSS / DOF
> # We could use our model to calulate the residuals, then square them,
> # then sum them.... but we don't actually need to do all that work ourselves.
> # Here's an easier way (but still not the easiest). From attributes(fit) or
> # help(lm) we learn that the residuals are already calculated for us.
> fit$resid
           1            2            3            4            5            6
-0.246590305 -0.067497800 -0.061162889  0.021105848  0.035643323 -0.042087939
....
> # or equivalently
> resid(fit)
           1            2            3            4            5            6
-0.246590305 -0.067497800 -0.061162889  0.021105848  0.035643323 -0.042087939
....
> # and the DOF are stored as df.resid
> # so we can calculate the residual mean square
> sum( (fit$resid)^2 ) / fit$df.resid
[1] 0.1435546
> #
> # While it's informative to know how to access the individual model
> # properties such as resid, df.resid, etc., there is an even easier way to calculate
> # the residual mean square. Use the 'summary' command.
> #
> summary(fit)

Call:
lm(formula = logP ~ Tb)

Residuals:
     Min       1Q   Median       3Q      Max
-0.32261 -0.14530 -0.06750  0.02111  1.35924

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -42.13087    3.33895  -12.62 2.17e-09 ***
Tb            0.89546    0.01645   54.45  < 2e-16 ***
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1

Residual standard error: 0.3789 on 15 degrees of freedom
Multiple R-Squared: 0.995,  Adjusted R-squared: 0.9946
F-statistic:  2965 on 1 and 15 DF,  p-value: < 2.2e-16
```

```
> # From the summary we see that the residual standard error, sigma_hat, which
> # is also often called the standard error of regression, is 0.3789.  Square
> # this to get the residual mean square.
> 0.3789^2
[1] 0.1435652
> ######################## VARIANCE #############################################
> # Want to know the variance of the parameters? Summary lists the Std. Error, and
> # variance is just the square of the standard error.
> # Heres a fancy way to square these parameters.
> beta_hat <- data.frame(summary(fit)$coef)
> # Using the 'data.frame' command allows us to access elements by name
> beta_hat$Std..Error
[1] 3.33895220 0.01644562
> beta_hat.var <- beta$Std..Error^2
> beta_hat.var
[1] 1.114860e+01 2.704585e-04
> #
> # Now consider using analysis of variance to test the null hypothesis
> # that the intercept should be at the origin.  Creat a new model, and
> # force it to go through the origin.
> fit.org <- lm( logP ~ 0 + Tb )
> anova(fit.org, fit)
Analysis of Variance Table

Model 1: logP ~ 0 + Tb
Model 2: logP ~ Tb
  Res.Df     RSS Df Sum of Sq      F    Pr(>F)
1     16 25.0092
2     15  2.1533  1   22.8559 159.21 2.170e-09 ***
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1
> # The probability that this is true is 2.17e-09, so we reject the
> # null hypothesis.
> #
> ######################## CONFIDENCE INTERVALS ######################
> # What if we want to know the 95% confidence interval for the model
> # intercept. We just saw how to get the standard error, so now all we
> # need is t-test value.
> t <- qt( 1- 0.025, fit$df.residual )
> t
[1] 2.131450
> # To calculate the interval bounds (lamda), first change the loaded object
> detach(forbes)
> attach(beta_hat)
> # Now caluclate the bounds as two elements of a vector c(lower,upper).
> lamda_intercept <- c(Estimate[1] - t * Std..Error[1], Estimate[1] + t * Std..Error[1])
> lamda_intercept
[1] -49.24768 -35.01406
> ############################# PLOTTING #########################
> # It's always a good idea to plot the data
> attach(forbes)
> plot(Tb,logP)
> #
> # Usually it's good to look at residuals vs fitted values.
> # While we know how to access the residuauls (fit$resid), the plot
> # function recognizes 'lm' object, and will give us this, and other
> # plots, automatically.
> plot(fit)
Hit <Return> to see next plot:
Hit <Return> to see next plot:
Hit <Return> to see next plot:
Hit <Return> to see next plot:
> # To see how the fit matches with the data, use abline.
```

```
> # Note: abline adds a line to a prexisting plot, so you must have
> # already done plot(Tb, logP)
> plot(Tb,logP)
> abline(fit)
> # Add confidence intervals to the plot.
> # Use the function predict, in "prediction" mode. By default
> # it will calculate values for all of the X data in the model
> # you give it ( see help(predict.lm) ).
> limits <- data.frame(predict(fit,interval="prediction"))
> limits
        fit      lwr      upr
1  132.0366 131.1544 132.9188
2  131.8575 130.9729 132.7421
3  135.0812 134.2315 135.9308
....
16 147.6176 146.7294 148.5058
17 147.8863 146.9943 148.7782
> # Pick out the endpoints, and make them the Y vector of the X,Y
> # coordinates to feed to the function 'lines'..  lty=2 gives
> # us a dotted line.
> lines(c(Tb[1],Tb[17]),c(limits$lwr[1],limits$lwr[17]), lty=2)
> lines(c(Tb[1],Tb[17]),c(limits$upr[1],limits$upr[17]), lty=2)
# Add a title
> title(main = "Data, fit and ± 95% confidence band")
> ###############################################################
> # (dkh, 10/03/04)
```