Introduction to R
Selected Handouts for 36-401

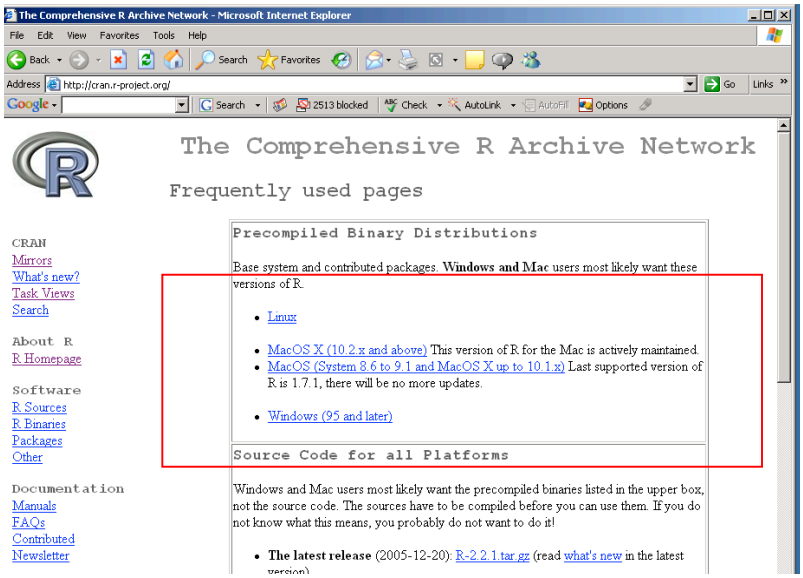
Rebecca Nugent
Department of Statistics
Carnegie Mellon University

Extensive material (more topics, handouts, examples, solutions)
can be found at
http://www.stat.cmu.edu/~rnugent/teaching/introR

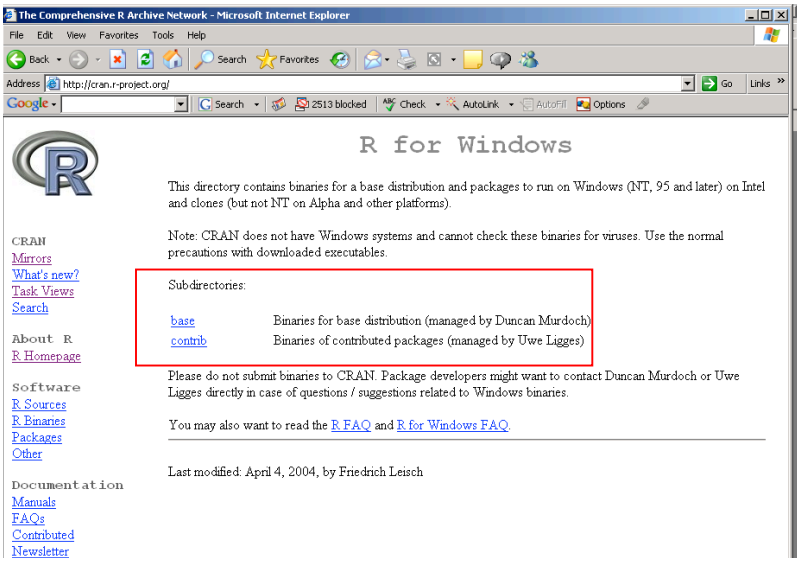
Table of Contents

Installing/Updating R

1) Go to http://cran.r-project.org. Most people will want to click on Windows (95 and later). You may need administrative privileges to install or update (depends on the operating system).

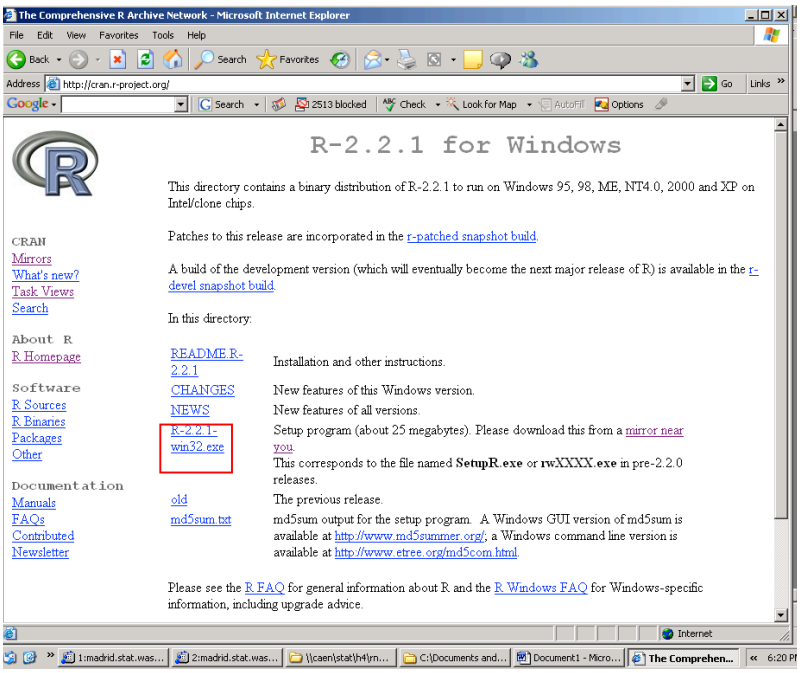


2) Then click on base.

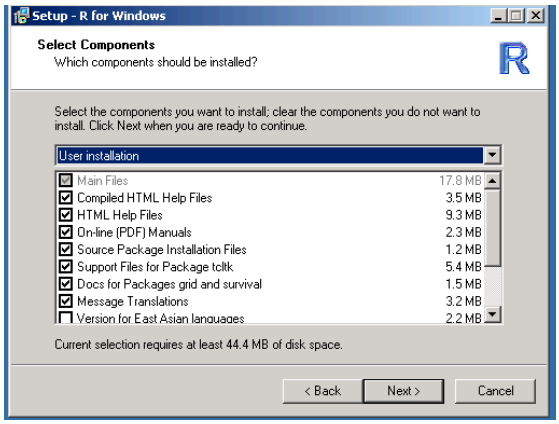


(adapted from Patty Glynn, UW, 12/07/02)

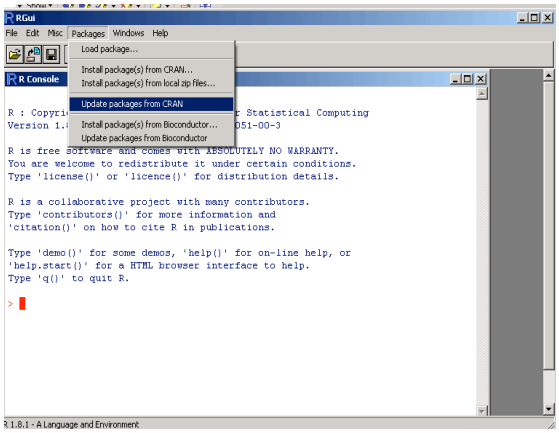
3) The base link will bring you to the latest version (Jan 6th, R-2.2.1).



Click on the link, download, and install (R-2.2.1-win32.exe). Installation may require administrative privileges. If you have the room on your hard disk, you probably want all of the documentation installed.



Update in the Packages menu by choosing Update packages from CRAN. You need administrative privileges and to be on-line.



Storing your R Work
(in Windows)

1) Create directory for storing your R work.
2) Start R either from the Start menu or by clicking on shortcut icon.
3) Go to File menu; choose "Change dir…."
4) Change the directory to your created directory from #1. It might be easier to click Browse and search for your directory instead of typing the pathname.
5) Do your R work.

Now there are two ways to save your workspace:

Either) Before quitting, go to File menu; choose "Save your Workspace". It will be a .RData file. This method has the advantage of not automatically loading the workspace every time you start R. Then quit R by typing q().

Or) Quit R by typing q(). It will ask you to save your workspace (yes/no/cancel). If yes, your workspace will be stored in your created directory.

Regardless, if you want your workspace back, you can go to your R work directory and double click on the .Rdata file. R will be started, and your workspace will be loaded. Or you can start R from the Start menu and then go to the File menu and choose Load Workspace.

(on a Mac)

Under the Workspace menu, there is an option "Save Workspace File". A box will come up to help you decide where to save the file (don't forget to name it something useful). The workspace can be re-loaded the next time you use R with the "Load Workspace File" in the same menu.

Helpful Commands:
To see what objects are in your directory:
> ls()

To remove an object "obj" from your directory:
> rm(obj)

To remove all objects from your directory:
> rm(list=ls())

To see where you're currently working on your computer:
> getwd() (get working directory)

(adapted from Cori Mar, UW, spring 2005)

Storing your R Commands

The previous page describes how to store the objects you've created in R.
It does not save the commands that you have written.

As you work with R, you'll quickly discover that it will often take you several tries to write the command to give you exactly what you want.
(Everyone does this – even experienced users.)

In order to save yourself incredible frustration, you want to save your commands.

You can use any script/text application (Emacs, Notepad, Word, etc).
Save your commands file as a .R file. (.txt may work as well)
You can either copy and paste your commands into the R command line, or you can "source" your file into R which will then run every command you have in the file.

```
> source("commandsfile.R")
```

After sourcing, any objects created by the commands will now be in your workspace. Note that if you do this, you don't need to save your workspace. You can just quickly source your file and get back to exactly the same point in your work.

On a PC:

Some versions may have a script window that automatically opens up that you may save as a .R file. Others may require you to open up a window from another application. Regardless, if you're having difficulty sourcing, copying and pasting all of the commands works just as well (and is often easier when you just want to re-run a few commands).

On a Mac:

Under the File menu, choose New Document. A script file window will pop up. Click on it to select it and then Save As "yourfilename.R". You can then either copy/paste, source using the source command, or click on the second icon (an R with a .R file on top of it) to choose a file to source from your list of files.

Commenting out text:

You should always comment on your code as you type it. Comments will be invaluable when you go back to look at your code after leaving it for some time. The # sign before any line of text will not be read by R as a command.

```
###testing the mean function
mean(x)
```

R will only execute the second line.

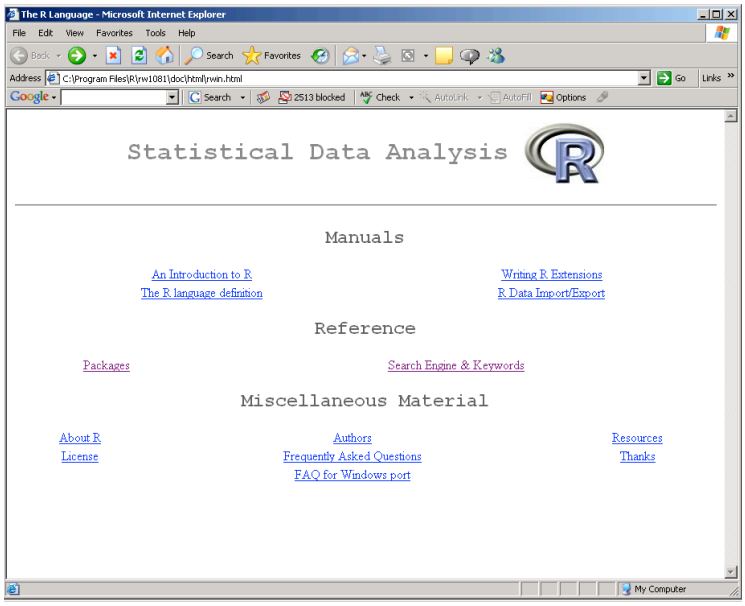
Working with the R Help Pages

HTML Help:

On a PC (Windows):
Go to the Help Menu.
Select HTML Help.

On a Mac:
Go to the Help Menu.
Select R Help.

An R Help browser window will appear.



On a PC, the R Help Main pages will come up in the browser.

On a Mac, there will be a search box in the upper right corner. The left corner has two search options: exact search, fuzzy search. Beneath that there are two buttons: R Help Main Pages and R For Mac OS FAQ. The FAQ are mostly setup/installation questions. The R Help Main pages are the default of the browser.

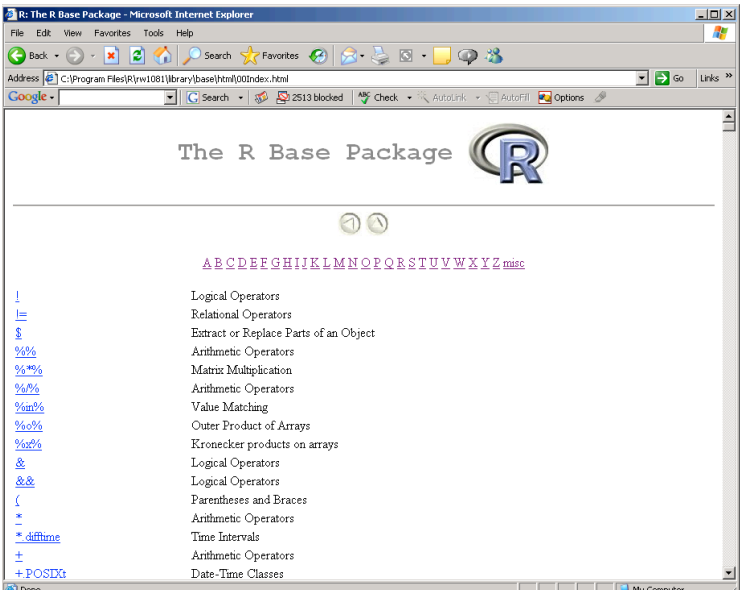
Within the R Help pages:

The most common places to search for help are: Packages and Search Engine/Keywords.

Click on Packages: if you know which function you are looking for and its package

A list of packages will come up. Base and MASS are the two most commonly used packages. Clicking on them will bring up a list of functions.

List of base functions:

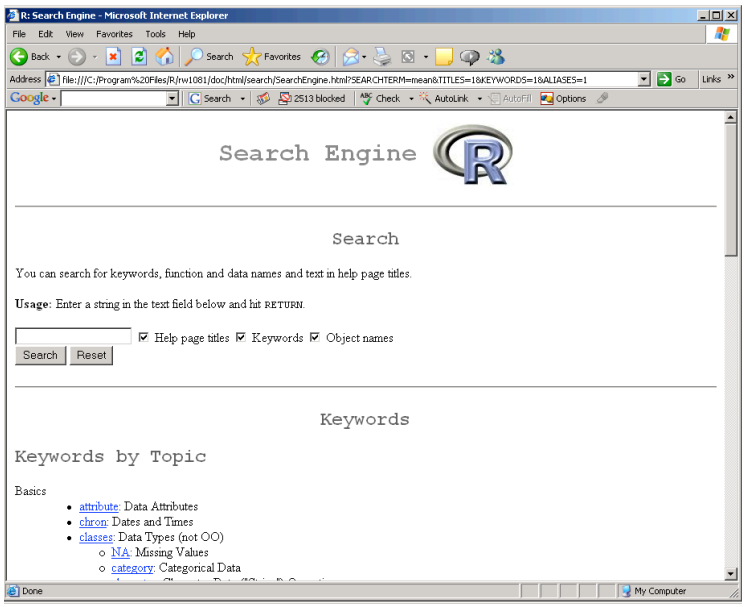


You can either scroll to the function you want or click on the function's first letter for a shorter list of functions to search.

Once you click on a function, its help documentation pages will open up.

Click on Search Engine & Keywords: if you're not sure which function you need

(Note: For search to work, you need Java installed and both Java and JavaScript enabled in your browser. (See R Installation and Administration Help))



If you just want to look by keyword, scroll down to Keywords by Topic. Each keyword has a short description next to it. For example, if you want to look at the functions used for spatial statistics, scroll down and select **spatial**.

You've selected a function and now you have the help documentation pages open. Okay, so what is all this?

Help Documentation Pages:

The header at the top of the page has the function name in the upper left corner, the package in the center, and the words "R Documentation" in the upper right.
The title of the documentation is the subject you would type in an R search engine.

There are up to 10 major sections: Description, Usage, Arguments, Details, Value, Note, Author(s), References, See Also, and Examples.

Description: Describes the purpose of the function.

Usage: the command/line of code that should be typed. There will be a list of arguments (if the function has any). If the argument has been set equal to an option, it is your default option. If you do not change the option, the function will run with the default settings. Note: if you type in the arguments without setting them equal to the argument names (i.e. mean(data) vs. mean(x=data)), the arguments must be typed in the correct order. If you assign them to each argument, the order is not important.

Arguments: Many functions have several arguments that can be chosen. When typing in the arguments, you are telling the function to run with these particular options.
Each argument is described and then a default setting is given. If no default setting is given, you must provide that argument.

Details: Further description of the function. Often this section is present for more complicated functions.

Value: A description of what the function returns (results/answers). Some help pages list the results that you will get depending on your argument settings. Often other functions that may be applied to the results to get more information will be suggested (ex. summary). If more than one result is returned, there will be a list of results, each item with a short description.

Note: Just extra information if there's something kind of tricky.

Author(s): Who came up with the function and/or who designed it.

References: Suggested reference material if you want to learn more.

See Also: Related functions that perform similar tasks

Examples: A very useful section. If you understood nothing in the other sections or are someone who learns by doing, this section is mega-helpful. The example lines of code are lines you can type in and then see what happens. The examples are also commented (text starting with #) to describe what the different examples are showing. I would recommend typing them in rather than cutting and pasting so you get a sense of how to put together the line of code.

Let's look at some help documentation pages:

```
mean                  package:base                  R Documentation

Arithmetic Mean

Description:

     Generic function for the (trimmed) arithmetic mean.

Usage:

     mean(x, ...)

     ## Default S3 method:
     mean(x, trim = 0, na.rm = FALSE, ...)

Arguments:

       x: An R object.  Currently there are methods for numeric data
          frames, numeric vectors and dates.  A complex vector is
          allowed for 'trim = 0', only.

    trim: the fraction (0 to 0.5) of observations to be trimmed from
          each end of 'x' before the mean is computed.

   na.rm: a logical value indicating whether 'NA' values should be
          stripped before the computation proceeds.

     ...: further arguments passed to or from other methods.

Value:

     For a data frame, a named vector with the appropriate method being
     applied column by column.

     If 'trim' is zero (the default), the arithmetic mean of the values
     in 'x' is computed.

     If 'trim' is non-zero, a symmetrically trimmed mean is computed
     with a fraction of 'trim' observations deleted from each end
     before the mean is computed.

References:

     Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) _The New S
     Language_. Wadsworth & Brooks/Cole.

See Also:

     'weighted.mean', 'mean.POSIXct'

Examples:

     x <- c(0:10, 50)
     xm <- mean(x)
     c(xm, mean(x, trim = 0.10))

     data(USArrests)
     mean(USArrests, trim = 0.2)
```

```
lm                    package:base                    R Documentation

Fitting Linear Models

Description:

     'lm' is used to fit linear models. It can be used to carry out
     regression, single stratum analysis of variance and analysis of
     covariance (although 'aov' may provide a more convenient interface
     for these).

Usage:

     lm(formula, data, subset, weights, na.action,
        method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
        singular.ok = TRUE, contrasts = NULL, offset = NULL, ...)

Arguments:

 formula: a symbolic description of the model to be fit. The details of
          model specification are given below.

    data: an optional data frame containing the variables in the model.
           By default the variables are taken from
          'environment(formula)', typically the environment from which
          'lm' is called.

 subset: an optional vector specifying a subset of observations to be
          used in the fitting process.

 weights: an optional vector of weights to be used in the fitting
          process. If specified, weighted least squares is used with
          weights 'weights' (that is, minimizing 'sum(w*e^2)');
          otherwise ordinary least squares is used.

na.action: a function which indicates what should happen when the data
          contain 'NA's.  The default is set by the 'na.action' setting
          of 'options', and is 'na.fail' if that is unset. The
          "factory-fresh" default is 'na.omit'.

  method: the method to be used; for fitting, currently only
          'method="qr"' is supported; 'method="model.frame"' returns
          the model frame (the same as with 'model = TRUE', see below).

model, x, y, qr: logicals.  If 'TRUE' the corresponding components of
          the fit (the model frame, the model matrix, the response, the
          QR decomposition) are returned.

singular.ok: logical. If 'FALSE' (the default in S but not in R) a
          singular fit is an error.

contrasts: an optional list. See the 'contrasts.arg' of
          'model.matrix.default'.

  offset: this can be used to specify an _a priori_ known component to
          be included in the linear predictor during fitting.  An
          'offset' term can be included in the formula instead or as
          well, and if both are specified their sum is used.

     ...: additional arguments to be passed to the low level regression
          fitting functions (see below).
```

```
Details:

     Models for 'lm' are specified symbolically.  A typical model has
     the form 'response ~ terms' where 'response' is the (numeric)
     response vector and 'terms' is a series of terms which specifies a
     linear predictor for 'response'.  A terms specification of the
     form 'first + second' indicates all the terms in 'first' together
     with all the terms in 'second' with duplicates removed.  A
     specification of the form 'first:second' indicates the set of
     terms obtained by taking the interactions of all terms in 'first'
     with all terms in 'second'.  The specification 'first*second'
     indicates the _cross_ of 'first' and 'second'.  This is the same
     as 'first + second + first:second'.  If 'response'  is a matrix a
     linear model is fitted to each column of the matrix. See
     'model.matrix' for some further details.

     'lm' calls the lower level functions 'lm.fit', etc, see below, for
     the actual numerical computations.  For programming only, you may
     consider doing likewise.

Value:

     'lm' returns an object of 'class' '"lm"' or for multiple responses
     of class 'c("mlm", "lm")'.

     The functions 'summary' and 'anova' are used to obtain and print a
     summary and analysis of variance table of the results.  The
     generic accessor functions 'coefficients', 'effects',
     'fitted.values' and 'residuals' extract various useful features of
     the value returned by 'lm'.

     An object of class '"lm"' is a list containing at least the
     following components:

coefficients: a named vector of coefficients

residuals: the residuals, that is response minus fitted values.

fitted.values: the fitted mean values.

    rank: the numeric rank of the fitted linear model.

 weights: (only for weighted fits) the specified weights.

df.residual: the residual degrees of freedom.

   call: the matched call.

   terms: the 'terms' object used.

contrasts: (only where relevant) the contrasts used.

 xlevels: (only where relevant) a record of the levels of the factors
          used in fitting.

       y: if requested, the response used.

       x: if requested, the model matrix used.

   model: if requested (the default), the model frame used.


     In addition, non-null fits will have components 'assign',
```

```
     'effects' and (unless not requested) 'qr' relating to the linear
     fit, for use by extractor functions such as 'summary' and
     'effects'.

Note:

     Offsets specified by 'offset' will not be included in predictions
     by 'predict.lm', whereas those specified by an offset term in the
     formula will be.

Author(s):

     The design was inspired by the S function of the same name
     described in Chambers (1992).  The implementation of model formula
     by Ross Ihaka was based on Wilkinson & Rogers (1973).

References:

     Chambers, J. M. (1992) _Linear models._ Chapter 4 of _Statistical
     Models in S_ eds J. M. Chambers and T. J. Hastie, Wadsworth &
     Brooks/Cole.

     Wilkinson, G. N. and Rogers, C. E. (1973) Symbolic descriptions of
     factorial models for analysis of variance. _Applied Statistics_,
     *22*, 392-9.

See Also:

     'summary.lm' for summaries and 'anova.lm' for the ANOVA table;
     'aov' for a different interface.

     The generic functions 'coef', 'effects', 'residuals', 'fitted',
     'vcov'.

     'predict.lm' (via 'predict') for prediction, including confidence
     and prediction intervals.

     'lm.influence' for regression diagnostics, and 'glm' for
     *generalized* linear models.

     The underlying low level functions, 'lm.fit' for plain, and
     'lm.wfit' for weighted regression fitting.

Examples:

     ## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
     ## Page 9: Plant Weight Data.
     ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
     trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
     group <- gl(2,10,20, labels=c("Ctl","Trt"))
     weight <- c(ctl, trt)
     anova(lm.D9 <- lm(weight ~ group))
     summary(lm.D90 <- lm(weight ~ group - 1))# omitting intercept
     summary(resid(lm.D9) - resid(lm.D90)) #- residuals almost identical

     opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
     plot(lm.D9, las = 1)      # Residuals, Fitted, ...
     par(opar)

     ## model frame :
     stopifnot(identical(lm(weight ~ group, method = "model.frame"),
                         model.frame(lm.D9)))
```

Help from the command line:

1) **help.start( )**

   If you type help.start() at the command line, the HTML help pages will open up in another window (need to have a browser interface).

2) **help()**

   When you know the name of the function you need to use or just want to get more information about a function, type help(function). The help documentation pages for that function will appear. If you're interested in a function (fxn) in a not-often-used package (pkg), you can type help(fxn, package = pkg) to get the help pages. (But what if you don't know the package? See #3.)

3) **help.search(" ")**

   When you just have a topic in mind, type help.search("topic"). This one can be tough as the R documentation sometimes has common functions titled with more complicated titles. You will get a list of functions, their packages, and short descriptions. Selecting one of those will pull up the help documentation pages.

Some help tips:

Choose fuzzy search when you're given a choice. This search will return partial matches as well as exact matches.

Also when looking at the list of packages and their descriptions, you can do a Ctrl+F (or Find on this Page) for your particular topic of interest and it will search for those words in the package descriptions.

Google can be your friend. Just search for what you're looking for with an R in front of it. Such as: R chi-square, R regression, etc.

Also, remember that Splus and R are very very similar. Searching for an Splus function will likely give the correct R function. (Google: Splus regression).

Getting Started and Using Help

Start session via Start menu.

Change Directory to C:\Temp (or the directory of your choice).

Now you have up a console window. At the top you see the copyright information.

In red at the bottom is a cursor >. This is the command line. You type in your command or lines of code and then hit enter/return. R evaluates your code and returns the answer on the next line.

R as a calculator:

```
> 4+3
[1] 7
> 2-1+12-2*5
[1] 3
> 2^4
[1] 16
```

We assign values to objects or variables with the assignment operator <- (or can use =).

```
> x<-47
> y=3
> x+y
[1] 50
```

The objects x and y are currently in the workspace.
They will remain there unless we delete them or if we choose not to save the workspace.

To list the current objects:

```
> ls()
[1] "x" "y"
```

Let's remove one:

```
> rm(x)
> ls()
[1] "y"
```

To quit R:

```
> q()
```

If we say yes when asked about saving the workspace, we can get it back by going into our working directory. We can then start R and load our workspace by double-clicking on the .Rdata file. Note that above the command line it says "previously saved workspace restored".
Now when we list the objects, we should get back what we had at the end of the last session.

```
> ls()
[1] "y"
```

Getting Help:

There are several ways to get help. In Windows, you can use the Html Help option in the Help menu or type `help.start()` at the command line. A browser opens up to the help pages within the R program. Clicking on Packages gives a list of all the available packages. Clicking on a specific package shows you the functions that are available in the package. (base is the automatically loaded package).

For example, clicking on "mean" gives me the documentation for the function that finds the average of a group of numbers.

If I want to see the help documentation for another package, I first must load the package. For example, to simulate from a multivariate normal distribution (`mvnorm`), I need the MASS package.

```
> library(MASS)
```

Now I can get a list of functions in MASS by:

```
> help(package=MASS)
```

You can also get help from the command line on a specific topic.

```
> help.search("regression")
```

This command will open up an R Information window with several regression functions and short descriptions. It also includes each function's package. Then you can type (for example):

```
> help(lm)
```

for more information on a specific function. Then another window opens up with very detailed documentation on the function: what it does (usage), what information you need to give it (arguments), what information you can get back (value), and other related functions.

One of the more helpful sections of this documentation is the example section at the bottom. You can often learn much more from looking at examples of how this function was used than trying to read pages of how to use it.

Typing:

```
> options(chmhelp=TRUE)
> help(lm)
```

will bring up a help box with documentation with Contents/Index/Search Tabs.

Basic Data Management

An object in R is any variable you define or a result of a function: "an object of data"

A variable is a word or letter that you define and assign to some value or value.
The assignment operator in R is: `<-` or `=`. What you type on the right is assigned to what you type on the left. So it reads "left is assigned as right".

For example,

`y <- 4` (we have assigned the value 4 to the variable y)
`x <- 6` (we have assigned the value 6 to the variable x)

`x <- y` (we have assigned the value of the variable y to the variable x, i.e. x = 4)

Variables can start with a letter, digits, or periods. Pretty much anything. Just not a number by itself. Some examples: `result, 2b, .answer` (not 2)

R is case-sensitive (i.e. `Result` is a different variable than `result`)
R is insensitive to white space though.
(i.e. `x <- 2` and `x<-` `2` are the same).

VECTORS:

A vector is a list of numbers. We can create this list using several different ways.

The `c( )` command links a list of numbers together.

```
> x<-c(2,4,5,7)
> x
[1] 2 4 5 7
```

Sequences of numbers can be created with either a colon or the `seq( )` function.

```
> y<-1:5
> y
[1] 1 2 3 4 5
> z<-27:34
> z
[1] 27 28 29 30 31 32 33 34
```

There are several ways to use the `seq( )` function. The most common are:

`seq(from=, to=, by=)` Starts at from, ends at to, steps by by (pos or neg)
`seq(from=, to=, length=)` Starts at from, ends at to, steps defined by length

If no by/length argument is given, sequence will start at from, end at to, and step by 1.

```
> seq(1,5)
[1] 1 2 3 4 5
> seq(1,10,by=2)
[1] 1 3 5 7 9
> seq(45,41,by=-1)
[1] 45 44 43 42 41
> seq(2,6,length=6)
[1] 2.0 2.8 3.6 4.4 5.2 6.0
```

The `rep( )` function will create a vector of repeated values of a given length:
`rep(x, times)`

```
> rep(1,4)
[1] 1 1 1 1
> rep(3:5,2)
[1] 3 4 5 3 4 5
```

These functions can be combined.

```
> rep(c(1,3,5),4)
 [1] 1 3 5 1 3 5 1 3 5 1 3 5
> rep(seq(2,6,by=2),3)
[1] 2 4 6 2 4 6 2 4 6
```

Indexing Vectors: We use brackets [ ] to pick specific elements in the vector.

```
> x<-c(1,3,5,7,9)
> x
[1] 1 3 5 7 9
> x[2]
[1] 3
> x[2:3]
[1] 3 5
> x[c(1,4)]
[1] 1 7
```

We use the `length( )` command to find out how long our vector is

```
> length(x)
[1] 5
```

Sorting and Ordering Vectors:

The `sort( )` function returns a list of ordered numbers.
The `order( )` function returns the order of the numbers, i.e. which position each number should be in if you were to list the numbers in order. It is a type of indexing.

```
> test.vec<-c(3,6,1,5,7,2)
> test.vec
[1] 3 6 1 5 7 2

> sort(test.vec)
[1] 1 2 3 5 6 7
> order(test.vec)
[1] 3 6 1 4 2 5
> test.vec[order(test.vec)]
[1] 1 2 3 5 6 7
```

Sampling from a Vector:

The `sample( )` function can be used to select a random sample from a list of numbers with or without replacement. The default is without replacement. If you do not replace the elements you've sampled, you only can select a sample of size 1 to the length of the vector. If you replace the elements, you can sample any size.

```
> sample(test.vec)
[1] 7 3 5 6 2 1
> sample(test.vec,3)
[1] 1 3 2
> sample(test.vec,replace=T,12)
 [1] 1 2 5 6 5 2 5 2 5 2 5 6
```

Vector Operations:

If two vectors are the same length, they can be added/subtracted element by element.

```
> x<-c(2,3)
> y<-c(4,5)
> x+y
[1] 6 8
```

Similarly for multiplication/division.

```
> x*y
[1]  8 15
> x/y
[1] 0.5 0.6
```

MATRIX:

A matrix stores 2-dimensional data. A matrix has rows and columns.
Each element is indexed by its row and column position [row, col].

Matrices can be created by combining vectors (must be of same length).

`rbind()` treats each vector like a row and stacks the vectors on top of each other.

```
> x<-c(6,5,4,3,2)
> y<-c(8,7,5,3,1)
> m1<-rbind(x,y)
> m1
  [,1] [,2] [,3] [,4] [,5]
x    6    5    4    3    2
y    8    7    5    3    1
```

This matrix has 2 rows and 5 columns. We can index it by using the brackets [ ] with a comma between the two dimensions. Leaving an index blank means you want the whole row or column.

```
> m1[2,2]    #the element in the 2nd row, 2nd column
[1] 7
> m1[,4]     #the 4th column
x y
3 3
```

`cbind( )` treats each vector like a column and lines the vectors up next to each other.

```
> m2<-cbind(x,y)
> m2
     x y
[1,] 6 8
[2,] 5 7
[3,] 4 5
[4,] 3 3
[5,] 2 1
```

This matrix has 5 rows and 2 columns.

```
> m2[3,2]     #the element in the 3rd row, 2nd column
[1] 5
> m2[5,]      #the 5th row
x y
2 1
```

We can also create a matrix from a list of numbers and the number of rows and columns.

```
> matrix(c(1,0,1,0,0,1,0,1,0),3,3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    0    1
[3,]    1    1    0
```

Notice that this filled in the matrix column by column. If you want to fill the matrix by rows, use the argument `byrow=TRUE`.

Also, if you need a matrix of just one number:

```
> m3<-matrix(0,4,4)
> m3
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
[4,]    0    0    0    0
```

We can assign values in the matrix.
In particular, the `diag()` function selects the diagonal elements of the matrix.

```
> m3[3,2]<-4
> m3[2,4]<-6
> diag(m3)<-2
> m3
     [,1] [,2] [,3] [,4]
[1,]    2    0    0    0
[2,]    0    2    0    6
[3,]    0    4    2    0
[4,]    0    0    0    2
```

Matrix Operations:

If two matrices are the same size, they are added/subtracted element by element.

```
> m.a<-matrix(c(1,2,1,2),2,2)
> m.b<-matrix(c(0.2,0.3,0.1,0.4),2,2)
> m.a+m.b
     [,1] [,2]
[1,]  1.2  1.1
[2,]  2.3  2.4
```

Matrix Multiplication:

The multiplication operator for two matrices is : `%*%`
(Recall `*` means element by element multiplication)

```
> m.a%*%m.b
     [,1] [,2]
[1,]  0.5  0.5
[2,]  1.0  1.0
> m.a*m.b
     [,1] [,2]
[1,]  0.2  0.1
[2,]  0.6  0.8
```

Transpose of a Matrix: `t( )`

```
> t(m.a*m.b)
     [,1] [,2]
[1,]  0.2  0.6
[2,]  0.1  0.8
```

Inverse of a Matrix: `solve( )` NOT `()^-1`

```
> solve(m.b)
     [,1] [,2]
[1,]    8   -2
[2,]   -6    4
```

We can return the dimensions of a matrix with `dim( ).`

```
> dim(m.b)
[1] 2 2
```

Sometimes a matrix of data is called a data.frame, a matrix where the columns have been given names. See `is.data.frame()` and `as.data.frame().`

```
> x1<-c(1,0,1)
> x2<-c(2,3,1)
> y<-c(5,4,2)
> df1<-(cbind(x1,x2,y))
> df1
     x1 x2 y
[1,]  1  2 5
[2,]  0  3 4
[3,]  1  1 2
```

ARRAYS: We can continue building storage objects for higher-dimensional data. Each dimension is another indexing level.

A vector is a one-dimensional array.
A matrix is a two-dimensional array.

A three-dimensional array can be built with the `array( )` function.

```
> m1<-matrix(1,2,4)
> m2<-matrix(2,2,4)
> m3<-matrix(3,2,4)

> array1<-array(0,c(2,4,3))
> array1[,,1]<-m1
> array1[,,2]<-m2
> array1[,,3]<-m3
> array1
, , 1

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

, , 2

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

, , 3

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

The `dim( )` function works on larger-dimensional arrays as well.

```
> dim(array1)
[1] 2 4 3
```

Asking Questions About Your Data Objects:

```
> x<-c(4,3,4,6,7,10,13)
> x
[1]  4  3  4  6  7 10 13
```

Whether or not your data is equal to, greater than, less than a specific value:

```
> x==4
[1]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE
> x<=6
[1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
> x>8
[1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
```

Where a specific value is located:

```
> which(x==4)
[1] 1 3
> which(x>10)
[1] 7
> x[which(x<=6)]
[1] 4 3 4 6
```

The and operator: `&`

```
> which(x<6 & x>=10)
numeric(0)
```

The or operator: `|`

```
> which(x<6 | x>=10)
[1] 1 2 3 6 7
```

How many values are equal to, greater than, less than a specific value:

```
> sum(x<=6)
[1] 4
> sum(x==4)
[1] 2
> sum(x>7)
[1] 2
> sum(x>6)
[1] 3
```

What kind of data you have:

Can have numeric and character (words) data.

In general, you can ask many true/false questions by `is.----()`

```
> x
[1]  4  3  4  6  7 10 13
> y<-c("Red", "Green", "Blue")
> y
[1] "Red"   "Green" "Blue"

> is.vector(x)
[1] TRUE
> is.character(y)
[1] TRUE
> is.numeric(x)
[1] TRUE
> is.numeric(y)
[1] FALSE
> is.matrix(x)
[1] FALSE
> is.array(x)
[1] FALSE
```

In particular, using `is.na()` helps you find missing data.

```
> is.na(x)
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> sum(is.na(x))
[1] 0
```

Reading in Data:

We can read in data from a text file or a .dat file or an Excel (sometimes save as .csv) file using the `read.table()` command.

If your data is in the same directory as your R session/.RData file, you can just type the name of the file.

```
>read.table("classexample.dat")
```

If your data is in another directory, you will need to type the whole pathname.
(Note the front slashes).

```
>read.table("//caen/stat/h4/rnugent/classexample.dat")
```

If your data has names for each of the columns on the top row, set `header=TRUE`
If you do not assign `read.table` to a variable, it will read the table right to the commandline.

Another option is the `scan()` function. It is more complicated but more flexible. `read.table()` is more user-friendly.

If you have .csv data, `read.csv()` is similar to `read.table()`

Writing out Data:

`write.table`(the object you're writing out, where you're writing it).

If you leave the destination blank, it will write it to the command line window.

```
> write.table(m1, "m1.dat")
> write.table(m1, "//caen/stat/h4/rnugent/m1.dat")
```

If/Else Statements

We've seen how we can use a TRUE/FALSE vector to identify certain conditions.
For example:
`which(age<35)` identifies who is younger than 35.
`which(gender=="male"| religion=="protestant")` identifies people who are males OR who are Protestant.

But what if you have a specific action you would like to take depending on someone's age or religion or……?

If statement basic structure:

```
if( statement that gives TRUE or FALSE) {

      The action you want to take

}
```

Note that the statement inside the parentheses returns ONE true or false, not a vector of trues or falses.

```
if( age < 35){
      young<-TRUE
}
```

You can combine the if with an else. If the statement is true, do this action. Otherwise, do this other action.

```
if( statement that gives TRUE or FALSE) {
      The action you want to take if TRUE
}

else {
      The action you want to take if FALSE
}
```

Here the if/else belong to each other. An else cannot stand alone.

```
if(age < 35){
      young<-TRUE
}
else young <- FALSE
```

If there are several different possible actions, can put together a string of if/else statements:

```
if( statement that gives TRUE or FALSE) {
     The action you want to take if TRUE
}

else if(another TRUE/FALSE statement){
     The action you want to take if TRUE
}
else if(another TRUE/FALSE statement){
     The action you want to take if TRUE
}
....
else {
     The action you want to take if none of the other
statements were TRUE
}
```

When looking at the code, think of reading it just like the English language. If this happens, I want to do X. But if this other thing happens, I want to do Y. And if neither of them happens, I'll do Z. No matter what, one of {X, Y, Z} will happen.

Example: Coding religion in the 1992 General Social Survey.
People identify religions by several different names. Might want to simplify:

```
if( religion ==" Fundamentalist Protestant" | religion=="Moderate
Protestant" | religion=="Liberal Protestant"){
      rel.cat<-"Protestant"
}
else if (religion == "Catholic" | religion=="Roman Catholic"){
      rel.cat<-"Catholic"
}
else if (religion=="Orthodox Jewish" | religion=="Jewish"){
      rel.cat<-"Jewish"
}
else rel.cat<-"Other"
```

For Loops/While

For loops:

Often we need to repeat an action several times – sometimes over subjects in a dataset.

```
> for(i in 1:n){
+     the action to be repeated
+ }
```

`for` indicates that we're going to loop from a start index to an end index.
`i` is the index we're looping over
`1` is our start index
`n` is the end index
`{` opens the loop; `}` closes the loop.

```
> index<-NULL
```
(Initiates the variable; assigns NULL value)

```
> for(i in 1:4){
+  index<-c(index,i)
+ }
> index
[1] 1 2 3 4
```

Looping over a dataset:

```
> data<-cbind(rnorm(10,0,1),rnorm(10,3,1),rnorm(10,6,1),rnorm(10,9,1))
```

A different way of initiating the variables

```
> mean.vec<-sd.vec<-rep(0,4)
```

Looping over each column:

```
> for(i in 1:4){
+ mean.vec[i]<-mean(data[,i])
+ sd.vec[i]<-sd(data[,i])
+ }
> mean.vec
[1] 0.8071171 2.9904124 5.8173064 9.1275691
> sd.vec
[1] 1.0747551 0.9355267 0.6220746 0.7322954
```

You don't have to loop over a sequence. It can be any vector of numbers.

```
> loop.vector<-c(3,5,7,12,20)
> for(i in loop.vector){
+ cat("i=",i,"\n")
+ }
i= 3
i= 5
i= 7
i= 12
i= 20
```

The `cat( )` function prints a list in order. `"\n"` indicates a new line.

Can loop over a selected sample of rows in your dataset:

```
> sample.vec<-sample(seq(1,40),10)
> sample.vec
 [1] 31 28 18  1  8 20 25 11 30 29

> mean.vec<-rep(0,10)
> for(i in 1:length(sample.vec)){
+ mean.vec[i]<-mean(data[sample.vec[i],])
+ }
```

While Loops:

If you're not sure how many loops you need, you can use a while loop.

```
while( true/false statement) {
      Repeated action
}
```

R checks the true/false statement; if true, it does another loop. If false, the loop stops.

```
> i<-1
> while(i <= 6){
+ cat("i=",i,"\n")
+ i<-i+1
+ }
i= 1
i= 2
i= 3
i= 4
i= 5
i= 6
```

We need an initializing statement to start the while loop: `(i<-1)`. If we did not have the `i<-i+1` statement, `i` would always be 1, and the loop would be infinite.

```
> x<-rnorm(1,0,1)
> while(x<0){
+ cat(x)
+ x<-rnorm(1,0,1)
+ }
-0.572205-0.09479107
> x
[1] 0.5720156

> x<-rnorm(1,0,1)
> while(x<0.5){
+ cat(x)
+ x<-rnorm(1,0,1)
+ }
-1.043909-0.4560672-0.1760368-0.8520931-0.5805316
> x
[1] 1.865334
```

Recoding Variables

Often we want to create another variable based on recoding a variable we already have. For example, recoding a continuous variable as a categorical variable.

We can do this with a for loop.

```
> age.cat<-rep(0,n)
> for(i in 1:n){
+ if(age[i]>10 & age[i]<=20) age.cat[i]<-1
+ if(age[i]>20 & age[i]<=30) age.cat[i]<-2
+ if(age[i]>30 & age[i]<=40) age.cat[i]<-3
+ etc
+ }
```

```
> gen.race<-NULL
> for(i in 1:n){
+ if(gender[i]=="m"&(race[i]==1|race[i]==2)) gen.race<-c(gen.race,1)
+ if(gender[i]=="f"&(race[i]==1|race[i]==2)) gen.race<-c(gen.race,2)
+ if(gender[i]=="m"&(race[i]==3|race[i]==4)) gen.race<-c(gen.race,3)
+ if(gender[i]=="f"&(race[i]==3|race[i]==4)) gen.race<-c(gen.race,4)
+ }
```

We can also do this with a conditional statement.
(Need to initialize the space ahead of time – can't do `age.cat<-NULL`.)

```
> age.cat<-rep(0,n)
> age.cat[age>10 & age<=20]<-1
> age.cat[age>20 & age<=30]<-2
> age.cat[age>30 & age<=40]<-3
etc
```

```
> race.cat<-rep(0,n)
> race.cat[race==1|race==2]<-1
> race.cat[race==3|race==4]<-2
etc
```

Plotting/Graphics in R

One of the best things about R is its graphics capability. You can produce publication-quality graphics that can be copied/pasted into a document or saved as .ps files to put into latex files.

Opening a Graphics Window:

When you use any plot commands, a separate graphics window will open up. Any new plots you make will show up in the same window, so if you would like to page back and forth between plots, click to select the graphics window and select the Recording option in the History menu. (On a Mac, select new Quartz Device Window from the Window menu before making your next plot.)

To Put the Graphics in a Paper/Report:
On a PC, right click on the graphics window and choose either copy or save as .ps depending what you want. Then you can paste or put the .ps file in a latex file or doc.

On a Mac, click on the Active Window and either select copy or save from the R menus. Or, if this goes awry, you can create postscript files instead.

```
postscript("nameoffile.ps", other arguments as needed)
```
then your graphics commands
```
dev.off()
```

The created ps file will be in your working directory. Can also use with `pdf()`.

Multiple Frames:

The default is to have one plot per window.
If you would like more, you can use the following command:

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

Within the `c( , )`, choose how many rows and columns of pictures you would like.

If you have four plots, `par(mfrow=c(2,2))` – 2 rows and 2 columns

If you have six plots, `par(mfrow=c(2,3))` or `par(mfrow=c(3,2))`.

If you have fewer plots than spaces on the window, R will just leave the others blank. However, if you want to go back to one plot per window, you need to reset to `par(mfrow=c(1,1))`.

Note: setting your frames to an equal number of rows and columns: `c(1,1)`, `c(2,2)`, `c(3,3)`, etc. will keep the plots in the nice square format you're used to seeing (width=height). Frames like `c(2,3)` will be narrower and tall; frames like `c(3,2)` will be wider and short.

The basic Plot Command:

```
plot(x,y)
```

`x` and `y` are the coordinates of the points in the plot

If you pass in a matrix with two columns, R will treat the first column as `x` and the second column as `y`.

```
x<-rnorm(30,0,1)
y<-dnorm(x,0,1)
```

There are many types of plots available: `plot (x, y, type=" ")`

Most common:
"p" for points (default)
```
plot(x,y,type="p")
```

"l" for lines
(note: this will connect the first point to the second, the second to the third, etc. So if you're plotting random data, the lines will criss-cross over the plot. You'll need to order the (x,y) pairs by x if you want the line to make sense.

```
plot(x,y,type="l")
plot(sort(x), y[order(x)], type="l")
```

"b" for both
```
plot(sort(x),y[order(x)],type="b")
```

"n" for no plotting
```
plot(x,y,type="n")
```

Often we use this option to get a plot with the appropriate size, location, etc and then we add points, lines, etc to the plot.

Labels:

If you do not specify a label, the x and y axes will be labeled by what you typed in the plot command.

The options `xlab=" "` and `ylab=" "` will let you change those labels.

```
plot(sort(x), y[order(x)],type="b", xlab="x",
ylab="Density")
```

We can add a title in two ways.

a) option `main=" "`

```
plot(sort(x),y[order(x)],type="b",xlab="x",ylab="Density"
,main="Standard Normal")
```

b) with a title command after the plot command

```
plot(sort(x), y[order(x)],type="b", xlab="x",
ylab="Density")
title("Standard Normal")
```

A sub-title can be added with the option `sub=" "`.
This title appears at the bottom of the graph.

```
plot(sort(x),y[order(x)],type="b",xlab="x",ylab="Density",
main="Standard Normal", sub="N(0,1)")
```

Limits:

The default is for R to plot a big enough space to contain all the (x,y) pairs.
We can change this with the options `xlim=c( , )` and `ylim=c( , )`.

```
plot(sort(x),y[order(x)], type="b", xlim=c(-5,5),
ylim=c(0,1.5))

plot(sort(x),y[order(x)], type="b", xlim=c(-2,2),
ylim=c(0,0.2))
```

These are the basics of the plot. But we can do ever so much more…….

```
help(par)
```

This will get you a list of the graphical parameters that you can set.

Some common ones:

`cex` = changes the character size of the of the points
(2 – double the size; 0.5 - half the size)

```
plot(sort(x), y[order(x)], type="b", cex=2)
```

`col` = changes the color of the points, lines, etc.
Can specify by a number, a name, or RGB components
`colors( )` to get the list of colors by name
The numbers for colors can vary with the version of R but usually
1 = black, 2 – red, 3 – green or blue, etc.

```
plot(x,y,col=1)
plot(x,y,col=2)
plot(x,y,col=3)
```

`lty` = line type – specified by an integer
(0 – blank, 1 – solid, 2 – dashed, etc)

```
plot(sort(x), y[order(x)], type="l", lty=2)
```

`lwd` = line width (the higher the integer, the wider the line)

```
plot(sort(x), y[order(x)], type="l", lty=2, lwd=3)
```

`pch` = specifies different plotting symbols; can be an integer or a text character in quotes

```
plot(x,y,pch=1)
plot(x,y,pch=2)
plot(x,y,pch=3)
plot(x,y,pch=4)
plot(x,y,pch=16)
```

Can add points or lines to the current plot:

```
plot(sort(x),y[order(x)], pch=16,type="b")
x2<-rnorm(30,0,3); y2<-dnorm(x2,0,3)
points(x2,y2,pch=16,col=2)
lines(sort(x2),y2[order(x2)],col=2)
```

Bar Plots:

```
help(barplot)
```

We can give it a vector of heights for each of the bars.

```
barplot(c(1,2,3,4,5))
```

We can also add color, shading, titles, etc.

```
barplot(c(1,2,3,4,5), col=c(1,2,3,4,5), names=c('Group 1", "Group2", "Group3", "Group4", "Group5")
```

Histograms:

```
help(hist)
```

We pass in a vector of data; R will construct the histogram for you.

```
z<-rnorm(100,0,1)
hist(z)
hist(z,col=3)
```

We can change the cutpoints for the bins.

```
hist(z, breaks=seq(-3,3,by=1))
hist(z, breaks=20)
hist(z, breaks=20, col=seq(1,20))
```

The same options are available like: `xlim, ylim, main`, etc.

If we want to change the histogram to show probability instead of frequency, we choose `probability = TRUE`.

```
hist(z, breaks=20, col=seq(1,20),probability = TRUE)
```

We can put two histograms on top of each other.

```
men<-rnorm(40,175,10)
women<-rnorm(40,145,12)

data<-c(men,women)

hist(men,col=2,breaks=seq(min(data),max(data)+10,by=10),
xlab="Weight",main="Histograms for Men and Women")

hist(women,col=3,add=T)
```

Pie Charts:

```
help(pie)
```

We pass in a vector of values. The pie chart is created with pieces whose area corresponds to the values. To name the pieces, we need to name the data.

```
values<-c(1,2,3,4)
names(values)<-c("Group1","Group2","Group3","Group4")
pie(values,col=c(4,5,6,7))
```

Boxplots:

```
help(boxplots)
```

Produces box-and-whisker plots of a passed-in vector of values.

```
boxplot(rnorm(100,0,1),col=3)
```

We can put multiple boxplots on one graph if we want to compare subgroups.

```
boxplot(men,women, names=c("Males", "Females"), col=c(2,3))
```

Legends:

If you have several groups on your scatterplot that you have labeled with different colors, point types, sizes, etc, you may want to include a legend on your plot that identifies each group with its characteristics.

```
> help(legend)
```

- The first two arguments are x and y, the location of the legend. (the upper left corner)

- The argument "legend" is a list of the text that you want associated with each group. For example, c("Group 1", "Group 2", "Group 3") – (note: text in quotes)
- col – a list of the colors you used
- pch – a list of the point types you used
- lty – a list of the line types you used
- lwd – a list of the line widths you used
- etc

One dataset in the MASS library of R is `birthwt`: Risk Factors Associated with Low Infant Birth Weight: 189 rows, 10 columns (we'll just look at a few variables)

```
> library(MASS)

> low.ind<-birthwt[,1]     (indicator of low birth weight: 1-yes, 2-no)
> mom.age<-birthwt[,2]     (age of the mother)
> mom.wt<-birthwt[,3]      (weight of the mother)
> mom.race<-birthwt[,4]    (race of mother: 1-white, 2-black, 3-other)
```

```
plot(mom.age,mom.wt)
plot(mom.age,mom.wt,xlab="Mother's Age",ylab="Mother's
Weight",type="n")
points(mom.age[low.ind==1],mom.wt[low.ind==1],col=2,pch=16)
points(mom.age[low.ind==0],mom.wt[low.ind==0],col=3,pch=16)

legend(35,250,c("Low Birth Wt", "Normal Birth
Wt"),col=c(2,3), pch=c(16,16))
```

If you know exactly where you want the legend, you can choose your x and y. But if you're not sure where your data are located or where you'll have enough room to put the legend, you can use the `locator()` function.

```
plot(mom.age,mom.wt,xlab="Mother's Age",ylab="Mother's
Weight",type="n")
points(mom.age[low.ind==1],mom.wt[low.ind==1],col=2,pch=16)
points(mom.age[low.ind==0],mom.wt[low.ind==0],col=3,pch=16)

legend(locator(1),c("Low Birth Wt","Normal Birth Wt"),
col=c(2,3),pch=c(16,16))
```

When you run the legend command, it will pause and wait for you to select a location. The graphics window will pop up (and your mouse arrow may turn into a cross); then click the spot on the graph where you want the left corner of your legend.

Identifying Points on Your Graph:

After you plot your data, you may want to know which subjects belong to some select data points (outliers, unexpected locations, etc).

You can plot the subject numbers instead of the plots.

```
> plot(mom.age,mom.wt,xlab="Mother's Age",ylab="Mother's
Weight",type="n")
> text(mom.age,mom.wt,labels=seq(1,length(mom.age)))
```

Side Note:
You can use text and locator together if you're interested in labeling curves, lines, etc.

```
plot(x<-rnorm(100,0,1),dnorm(x,0,1),pch=16,xlab="x",ylab="Density")
points(x2<-rnorm(100,0,2),dnorm(x2,0,2),pch=16,col=2)
points(x3<-rnorm(100,0,3),dnorm(x3,0,3),pch=16,col=3)
text(locator(1),"N(0,1)")
text(locator(1),"N(0,2)")
text(locator(1),"N(0,3)")
```

But what if you want to interact with your plot? Select points on your plot?

```
identify()
```

This function reads the location of the graphics pointer (your mouse) when you press the mouse button and returns the closest point.

Example:

```
x<-runif(50,0,1)
y<-runif(50,0,1)
plot(x,y)

identify(x,y,labels=seq(1,50),n=2)
```

Arguments:
x and y identify the data you're referencing (could pass in a two-column matrix);
labels = seq(1,50) – the text you're labeling each point (and also returning)
n = 2 – the maximum number of points you're selecting

```
identify(x,y,labels=seq(1,50),n=10)
```

You can plot information other than the label number.

```
plot(x,y)
identify(x,y,labels=x,n=2)
```

Maybe:

```
plot(x,y)
identify(x,y,labels=round(x,2),n=2)
```

`identify( )` returns the index of the points you identified. If you assign the identify statement to a variable, you can then analyze your selected points.

```
selected.pts<-identify(x,y,labels=seq(1,50),n=10)
points(x[selected.pts],y[selected.pts],pch=16,col=3)
```

```
mean(x[selected.pts]),etc.
```