

Introduction to R

June 2006

Introduction	3
What is R?.....	3
Availability & Installation.....	3
Documentation and Learning Resources.....	3
The R Environment	4
The R Console	4
Understanding R Basics.....	5
Managing your R Workspace	6
R Object Naming Conventions.....	6
Packages/ Libraries.....	6
Online Help	7
Browser Based.....	7
Help Commands	7
Interpreting Help for a Function.....	8
Search paths	8
Data Classes.....	9
Vectors	10
Factors	10
Dates.....	11
Matrices.....	12
Data Frames	12
Information about Objects	13
Sample Data	14
Viewing and Editing Data	15
Operations on Objects	15
Common Operators.....	15
Arithmetic Operations.....	16
Boolean Operations	16
Missing Values – NA, NaN.....	17
Referencing elements of an object.....	17
Vectors and Matrices	17
Elements	17
Rows or Columns of Matrices	18
Dataframes.....	18
Columns	18
Rows	18
Cells	19
Adding Columns or Rows to a Dataframe.....	19
Subset an Object.....	19
R Scripts.....	20
Import Data from External Files	21
Working with a Dataframe.....	22
Change Class of Dataframe Columns.....	23
Assign Missing Values	24
Data Analysis	24
Descriptive Statistics	24

Contingency Tables	25
One-Way Absolute Frequencies	25
One-Way Relative Frequencies	25
Multi-way Contingency Tables	25
Pearson's Chi-squared test	26
Selecting Observations (Rows).....	26
Working with Function Results.....	27
Paired T-Test	27
2-sample t-test	28
Formula Syntax.....	28
Graphics.....	28
High-Level Graphics Functions	28
Plot Function	29
Histograms	31
Other High Level Graphics Functions	32
Useful Parameters for High Level Graphics Functions	32
Function Plots	32
Graphics Window Management	33
Low-Level Graphics	33
Regression Line	34
Title	35
Legends	36
Subsetting Data and presenting side-by-side graphs	37
Summary of Frequently Used R functions	40
Index	42

Introduction

What is R?

R is a statistical programming environment rather than a statistical package such as Mintab, Stata, SAS or SPSS. Consequently, the user has greater flexibility to get precisely the desired analysis. This flexibility results in the increased responsibility to ensure that the results are correct, and a steeper learning curve.

User/third-party packages for a wide variety of statistical procedures are readily available. Since these packages are written by a variety of third parties, it is important for you to understand their methods and check their accuracy.

R is a powerful language that provides an extensive graphics capability and access to some specific statistical routines, possibly through the R user community, that may not be available in traditional statistical packages.

Availability & Installation

The R language and programming environment is available as a free download. R is available for a variety of platforms including Microsoft Windows, Unix, Linux, and Macintosh OS. Although this document focuses on the implementation for MS Windows, the syntax for most operations is consistent across platforms.

To download R go to the website: <http://www.r-project.org/>. This site also provides detailed information concerning the free software distribution and the organizations that manage the R resources. Select one of the listed US-based download sites and follow the links to your operating system's download. For Windows, you will download a setup program named `R-n.n.n-win32.exe`. (R distributions are updated frequently, so the specific version number will change. In this document we use `R-n.n.n` to refer to any specific version, e.g. `R-2.2.1`).

Double-click the downloaded setup file to start the Installation Wizard. On the "Select Components" dialog, add PDF Reference Manual to the selected components (in addition to "On-line PDF Manuals", which are selected by default).

Documentation and Learning Resources

R Reference Index is at <http://www.r-project.org/> under Manuals. The PDF Reference Manual that you installed with R is Section I of this Reference Index. Section II covers additional packages. This document is approximately 2400 pages.

If you selected the Reference Manual during R installation, it will be available from the R Help menu's "Manuals in PDF" item. The Reference Index does not appear on R's Help menu. You can open it outside R; or you can replace the Reference Manual file, `refman.pdf` with the Reference Index in the R installation manual folder (`C:\Program Files\R\R-n.n.n\doc>manual`).

Additional resources:

*Eva Goldwater & Mira Shapiro
Biostatistics Consulting Center
University of Massachusetts School of Public Health
June 2006*

The University of Massachusetts Statistical Software Information website Resources page, <http://www-unix.oit.umass.edu/~statdata/resources.html>, has links to a selection of sites with information about R, including all of the ones listed below. A web search will yield many other sites...

<http://www.ats.ucla.edu/stat/r/> is a helpful site maintained by UCLA. On this site you will find links to classes, seminars, and downloadable books describing many of the facilities within R.

An Introduction to R - Venables & Smith

<http://cran.r-project.org/doc/manuals/R-intro.pdf>

Using R for Data Analysis & Graphics – JH Maindonald

<http://wwwmaths.anu.edu.au/~johnm/r/usingR.pdf>

To load the workspace (data) that goes with Maindonald book, open R and type:

```
load(url("http://wwwmaths.anu.edu.au/~johnm/r/dsets/usingR.RData"))
```

R "Contributed Documentation" has numerous other guides focusing on various themes:

<http://cran.r-project.org/other-docs.html>

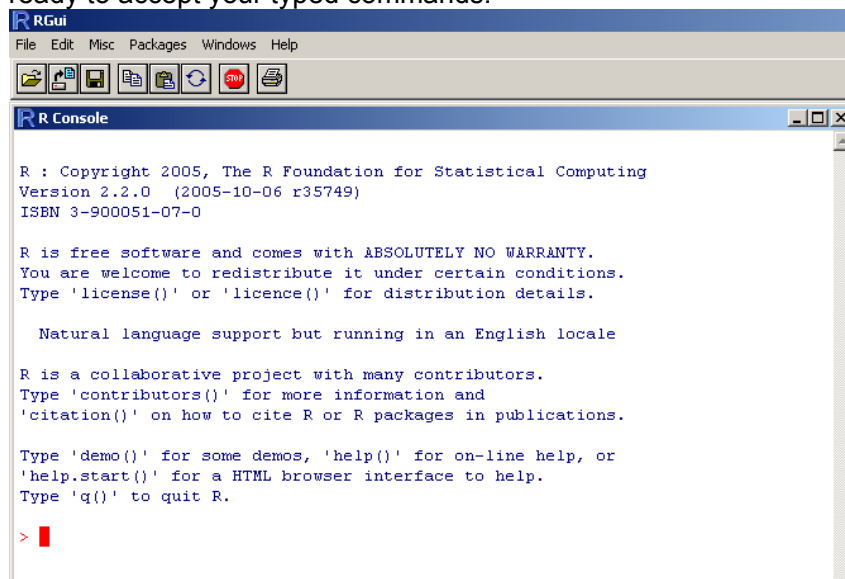
R-Help Mailing List – check the List Archive, or subscribe to the Mailing list to post questions – please observe etiquette. To subscribe, go to:

<http://www.r-project.org/> select Mailing Lists. The list email address is r-help@lists.R-project.org.

The R Environment

The R Console

To start the R environment, go to Start → All Programs → R → R n.n.n. The R console opens with copyright information. The “>” symbol is your prompt, indicating that R is ready to accept your typed commands:



```
RGui
File Edit Misc Packages Windows Help
[Icons]
R Console
R : Copyright 2005, The R Foundation for Statistical Computing
Version 2.2.0 (2005-10-06 r35749)
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for a HTML browser interface to help.
Type 'q()' to quit R.

> █
```

If a command is incomplete when you press Return, R waits for you to enter the rest of the command. The "+" prompt indicates R is waiting for you to complete a command.

Use the Up Arrow (↑) to recall commands from earlier in the session. You can then edit the command and press Return to run it again.

If you type a command and do not assign the result to a variable, the result is displayed at the console, but not saved. Here, the command is 5, the result is 5 (amazing!), which has not been assigned to a variable.

```
> # command without assignment displays result
> 5
[1] 5
>
```

Here we assign the value 5 to variable x. There is no response, other than the prompt for the next command. Assignment is done with either <- or =. <- is the 'gold standard'.

```
> # command with assignment does not display anything
> x<-5
>
```

To see the value of x, we must request it.

```
> # display value of x
> x
[1] 5
>
```

Understanding R Basics

In the R programming language

- all actions other than assignment and arithmetic operations are accomplished using **functions**;
- everything in R is an **object**, including data and functions;
- all objects you create become part of your **R workspace**.
- File references within R must use either forward slash (/) or double backslashes (\\), NOT the usual Windows backslash (\). For example, within R, the file 'C:\rwork\mydata.R', must be specified as either 'C:/rwork/mydata.R' or 'C:\\rwork\\mydata.R'

All R data objects have a data type or class. The data types we will discuss here are vectors, matrices, factors, ordered factors, and data frames.

Function calls require an argument list, though the list may at times be empty. For example, the q function used to quit (exit) R has an empty argument list:

```
> q()
```

If you leave out the parentheses on a function, R lists the function, rather than calling it:

```
> q
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
<environment: namespace:base>
```

If you leave off the ending parenthesis, the command is incomplete, and R issues the + prompt to indicate that it is waiting for you to complete your command:

```
> q(  
+ )
```

Managing your R Workspace

The operating environment within the R console is called the R workspace. When a workspace is saved all objects (data and functions) that were created during that session are saved unless they have been explicitly deleted using the `rm` (remove) function. If you are trying out a lot of things, your workspace can quickly become a mess of objects whose purpose you no longer recall. For this reason, it might be better to collect the commands that you need in a file, and save those rather than the workspace. This will be discussed in the section on Script files.

A saved workspace has a file extension of `RData`. For example, `hw1.RData` is a saved R workspace.

To save a workspace, select `File → Save Workspace`. Select the location where you would like the workspace to be saved, and give it a name. The file type should be `*.RData`.

To open a saved workspace, select `File → Load Workspace`. Select the location where the workspace was saved, with File type `*.RData`.

Alternatively you can save and retrieve workspaces using the commands:

```
> save.image("C:/rwork/hw1.RData") #saves a workspace  
> load("C:/rwork/hw1.RData")      #retrieves a workspace
```

Note the use of the forward slash (/) or double backslash (\\) rather than the backslash(\) normally used in Windows when specifying filenames.

R Object Naming Conventions

R is case sensitive! For example, *NAME*, *Name*, *NaMe* and *name* all refer to different objects.

An object name in R:

- must start with a letter, and can contain letters, numbers and the period.
- Can be longer than you'd want to type, so for all practical purposes the length of names is not limited
- should not include the underscore (`_`). Although the underscore is legal in the current release of R, earlier releases of did not permit use of the underscore, nor does S-Plus. Thus, for compatibility reasons, avoid using the underscore.

Packages/ Libraries

An R package, also called a library, contains functions and data that can be used in the R environment. In order to use a package, it must be

- Installed on your computer .
- Loaded in your working environment .

Libraries only need to be *installed once*. The `library()` function with an empty argument list displays the libraries installed on your computer.

```
library()
```

The `search()` function tells you which libraries are loaded in your working environment, and the order in which R will look for objects in these libraries. This will be described in more detail in the "Search paths" section of this document.

```
search()
```

To load a package that is installed but not loaded, use the `library()` function with the package name as the argument. If the library is not among those loaded by default, you will need to *load it in each R session* in which you wish to use it:

```
library("package_name")
```

Many additional packages are available for installation from CRAN sites by selecting Packages → Install Packages from the menu bar.

Note: Although you can install additional packages in SPPHS computer lab, the Help files for these additional packages cannot be installed in the lab at this time.

Online Help

Browser Based

To start HTML (browser-based) help, select Help → Html Help, or type

```
help.start()
```

From the html help screen, 'packages' will display the installed (not necessarily loaded) libraries. Click on a library to see the objects in the library; click on an object to find out what it is and how to use it.

Help Commands

There are various Help commands you can type at the console.

To find out what's in a library (in this example the stats library):

```
help(package=stats)
```

The package must be installed, but need not be loaded.

To find a function based on keywords (in this example, search for functions to compute the mean):

```
help.search("mean")
```

The results of the search show function names, which library the function is in (in parentheses), and a brief description of the purpose of the function. The library that the function is in must be installed, but not necessarily loaded.

To find out how to use a specific function whose name you know (for example: mean):

```
help(mean)
```

or

```
?mean
```

The latter will not work if the package containing the function is in is not loaded. Either load the package, or use the help command and specify the package name:

```
help(mean, package=base)
```

Note: The base package is always loaded. The above command is only intended to show the general form for specifying a package.

Interpreting Help for a Function

Help tells you the names of the arguments a function accepts, the order in which it expects them, and default values of the arguments, if any. When you call the function, you can specify arguments by name, by order, or, if you want the default value, not at all.

For example, the help for mean says:

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

There is some explanation of each item under "Arguments". The arguments are `x`, `trim`, and `na.rm`, in that order. Default values, if any, are shown following the equal sign next to an argument. Note that the default value `na.rm=FALSE` means that the function will NOT exclude missing values. With this option, the mean function will fail if the argument `x` contains any missing values.

Thus, to get a simple mean, excluding missing data (NA):

```
mean(x, na.rm=TRUE)
```

Or, rather more cryptically, using argument order:

```
mean(x,, TRUE)
```

Note that the value TRUE must be capitalized. It can be abbreviated as T.

Search paths

The `search()` function shows not only what packages are loaded, but the order in which they are searched when you refer to an object. If there are objects with the same name in different packages, then R will use the object with that name that is highest on the search path. The other object is "masked".

```
#Example: Results of search() show the hierarchy
search()
[1] ".GlobalEnv"          "package:methods"    "package:stats"
[4] "package:graphics"    "package:grDevices"  "package:utils"
[7] "package:datasets"    "Autoloads"          "package:base"
```

When you load a library, it is added to the search path in the second position. Therefore functions and data objects in the newly loaded package take precedence over objects with the same in other packages.

```
> library(MASS)
> search()
[1] ".GlobalEnv"          "package:MASS"        "package:methods"
[4] "package:stats"       "package:graphics"    "package:grDevices"
[7] "package:utils"       "package:datasets"    "Autoloads"
[10] "package:base"
```

The `pos` argument of the `library` function lets you control where in the search path a package is inserted:

```
> library(MASS, pos=11)
```

Objects you create in your workspace are highest in the search path. Therefore, if you accidentally create an object with the same name as one in a library, the library object becomes "masked". This obviously will cause serious problems if you need to use the masked object.

Example:

```
> #generate 10 random normal numbers with mean=90, std.dev=5
> x<-rnorm(10,90,5)
> # mean of x.
> mean(x)
[1] 91.9769
```

Now, we define a new function called mean, which trivially always returns 5:

```
> mean<-function(y) {5}
> mean(x)
[1] 5
```

The original mean function is no longer available. The `conflicts()` function displays any objects' names that are in conflict with (i.e. masking) another object:

```
> conflicts()
[1] "body<-" "mean"
```

To correct the situation, remove the conflicting object. If you really need the new object, copy it to another name, then remove it. ("body<-" is the one item that you need not be concerned with as the output of `conflicts()`.)

```
> mean5<-mean
> rm(mean)
> conflicts()
[1] "body<-"
> mean(x)
[1] 91.9769
```

Data Classes

A **vector** is a collection of data stored as a single column. It is further classified according to the type of values it contains - *numeric, character, factor, Date, logical*, etc.

A **matrix** is an n x n data structure whose elements are all of the same data type – character or numeric. The rows and/or columns of a matrix may or may not be named.

A **factor** is a vector that contains a categorical data. The displayed are labels associated with underlying integer values. An **ordered factor** is a factor whose levels have some natural order.

A **data frame** is a structure of rows and columns like a matrix, but the columns can be of different types, and are always named. A data frame is analogous to the datasets you would use in a statistical package such as SAS, SPSS, etc. with columns of variables and rows of observations. This is the structure we will be using for all data analysis examples.

A **list** is a composite of smaller objects of arbitrary types. Each element of a list is an object, possibly of different kinds. For example, vectors of different lengths or different types could

be combined as elements of a list, as could matrices, data frames, or other lists. Functions often return lists as their output.

To find out what class an object belongs to, use the `data.class()` function.

Vectors

We will be using some simple functions for making vectors to "play" with:

The `c()` function concatenates its arguments to form a vector:

```
> c(1,8)
[1] 1 8
```

`rep()` replicates an argument:

```
> rep(1,3)
[1] 1 1 1
```

`seq()` generates a vector of sequential numbers

```
> seq(1,3)
[1] 1 2 3
```

`rnorm()` and `runif()` generate random normal and uniform numbers, respectively. See usage example of `rnorm` under Search Paths.

Note: The elements of a vector must be of one type. If you attempt to make a vector with a mix of numeric and character values, R will "coerce" the numbers to characters, and make the resulting vector character.

Examples: Creating vectors

```
> #create a numeric vector called age
> age<-c(12,9,4,35)
> #display the vector age
> age
[1] 12 9 4 35

> #create a character vector of names
> name<-c('Peter','Sally','Randy','Jan')

> #create a numeric vector representing gender,
> #where 1=female and 2=male
> sex<-rep(c(2,1),2)
> # check the class of vector sex.
> data.class(sex)
[1] "numeric"
> # list the values of vector sex.
> sex
[1] 2 1 2 1
>
```

Factors

In order to define a vector as a **factor**, you specify the levels and their (optional) associated labels. R assigns sequential integers to the levels, in the order in which you list them in the

levels argument. If the vector contains any values not listed on the levels argument, that value will be set to missing (NA). The original level values are discarded!

In many R statistical functions, if the intention is for a particular set of values to be used as a categorical variable, it must be defined as a factor.

Example: We have a numeric vector, `sex`, with values 1 and 2. In order to use the variable `sex` as a categorical variable, use the `factor()` function to change `sex` to a **factor**. After it is defined as a factor, the values of `sex` are no longer listed as numbers, but as the labels associated with the factor levels. It can now be used as, for example, the grouping variable of a 2-sample t-test.

```
> #create a factor from the numeric vector sex.
> # anything other than 1 or 2 becomes NA.
> sex<-factor(sex,levels=c(1,2),labels=c('female', 'male'))
> data.class(sex)
[1] "factor"
> # list the values of vector sex
> sex
[1] male   female male   female
Levels: female male
```

An **ordered factor** is one in which the underlying levels have an order. The syntax for creating an ordered factor is the same as for any factor, but with the additional argument `ordered=TRUE`. *Be sure to list the levels in their natural order, lowest to highest.* R uses this to assign the ordering of the levels.

Example: Make an ordered vector, `activity`, with values 1,2,3.

```
> # generate vector for a 3-level ordered factor and list it.
> activity<-rep(seq(1,3),4)
> activity
[1] 1 2 3 1 2 3 1 2 3 1 2 3
> # make activity an ordered factor, check its class, and list it.
> activity<-factor(activity,levels=c(1,2,3),
+ labels=c("low","medium", "high"), ordered=TRUE)
> data.class(activity)
[1] "ordered"
> activity
[1] low   medium high   low   medium high   low   medium high   low
[11] medium high
Levels: low < medium < high
```

Dates

Dates are stored internally as number of days from January 1, 1970. The `as.Date()` function is used to convert a character string into a date. The second argument of `as.Date()` is the template for interpreting the character string. Use `%Y` for strings with 4-digit years; `%y` for 2-digit years. Dates are displayed in the format `yyyy-mm-dd`:

```
> #dates are stored as days from Jan 1, 1970.
> d=as.Date(c('12/31/1969','01/01/1970','01/02/1970'), '%m/%d/%Y')
> d
[1] "1969-12-31" "1970-01-01" "1970-01-02"
> data.class(d)
[1] "Date"
```

```

> # display underlying numeric value of date values.
> as.numeric(d)
[1] -1 0 1

```

Matrices

The `matrix()` function arranges the elements of a vector into a rectangular matrix structure. By default, the matrix is filled by columns:

```

> # create numeric matrix m
> m=matrix(seq(1,12), nrow=3, ncol=4)
> m
      [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

```

Use `byrows=T` to fill the matrix by rows, and `dimnames` to name the rows and columns:

```

> # create character matrix A, by rows, with row & column names.
> A=matrix(c('a','b','c','d','e','f'), nrow=2, ncol=3, byrow=T,
+ dimnames=list(c('R1','R2'),c('c1','c2','c3')))
> A
      c1 c2 c3
R1 "a" "b" "c"
R2 "d" "e" "f"
>

```

Data Frames

The `data.frame` function can be used to combine vectors into a dataframe, or to interpret a matrix as a dataframe. The following R command creates a dataframe from the three vectors that we created earlier. The names of the vectors become the names of the dataframe's columns:

```

> #create a dataframe from vectors name, age and sex
> df <-data.frame(name, age, sex)
> #list the contents of data.frame df on the console
> df
  name age  sex
1 Peter 12  male
2 Sally  9 female
3 Randy  4  male
4  Jan 35 female

```

Here the `data.frame` function interprets the previously created matrix `m` as a data frame. Since the matrix did not have column names, the default names `X1`, `X2`, `X3` and `X4` are assigned to the data frame columns:

```

> data.frame(m)
  X1 X2 X3 X4
1  1  4  7 10
2  2  5  8 11
3  3  6  9 12

```

Use `dataframe$columnname` to refer to a column of a dataframe.

```

> df$sex
[1] male  female male  female

```

```
Levels: female male
```

You can also refer to dataframe columns by their number in double brackets:

```
> df[[3]]
[1] male    female male    female
Levels: female male
```

Information about Objects

A variety of functions can be used to find out the attributes of objects in your workspace. Here are a few of the functions most commonly used for this purpose:

objects() lists the objects currently in the workspace

```
> # what objects are in the workspace now?
> objects()
[1] "A"      "activity" "age"      "d"        "df"       "m"        "name"
[8] "sex"
```

Note that the vectors `name`, `age` and `sex` remain separate objects from the dataframe `df` which was made by combining those three vectors.

data.class() returns the class of an object.

```
> #check the class of dataframe df and some of its columns.
> data.class(df)
[1] "data.frame"
> data.class(df$age)
[1] "numeric"
> data.class(df$sex)
[1] "factor"
```

Note that when a character vector is added to a dataframe, it is converted to factor. For example, compare the data class of vector `name` to the class of dataframe column `df$name`:

```
> data.class(df$name)
[1] "factor"
> data.class(name)
[1] "character"
```

This feature is not always desirable. In this case, it makes sense for `df$sex` to be a factor, but we might prefer `df$name` to remain character.

To prevent `data.frame` from converting a character vector to factor, use `I()` to "inhibit interpretation":

```
> df<-data.frame(I(name), age, sex) # preserve name as character
> data.class(df$name)
[1] "AsIs"
```

length() returns the number of elements in a vector or matrix. For other types of objects it may be defined differently. For example, for a dataframe, it returns the number of columns, not the number of elements:

```
> # length returns number of elements in a vector or matrix.
> length(name)
[1] 4
> length(df$name)
[1] 4
> length(m)
```

```
[1] 12
> # length returns number of columns in a dataframe.
> length(df)
[1] 3
```

dim() returns the number of rows and columns of a dataframe or matrix:

```
> dim(df)
[1] 4 3
> dim(m)
[1] 3 4
```

names() returns the column names of a dataframe. **dimnames()** returns row and column names of a dataframe or of a named matrix. The rownames of a dataframe are just the row number, and are not usually of interest:

```
> #names and dimnames.
> names(df)
[1] "name" "age" "sex"
> dimnames(A)
[[1]]
[1] "R1" "R2"
[[2]]
[1] "c1" "c2" "c3"
```

levels() returns the names of a factor's levels. The underlying numeric codes are sequential integers 1 to n, regardless of the original values of the vector (before defining it as a factor).

```
> levels(sex)
[1] "female" "male"
> levels(activity)
[1] "low" "medium" "high"
```

Sample Data

Many R libraries include data (usually dataframes) that you can use for practice. In particular, the `datasets` library is a collection of such data. It is on your default Search Path. If a library of interest is not on your Search Path, don't forget to load it with the `library()` command. To see what's in the `datasets` library, type:

```
help(package=datasets)
```

To find out more about any particular data in the `datasets` package, e.g. `chickwts`:

```
help(chickwts)
```

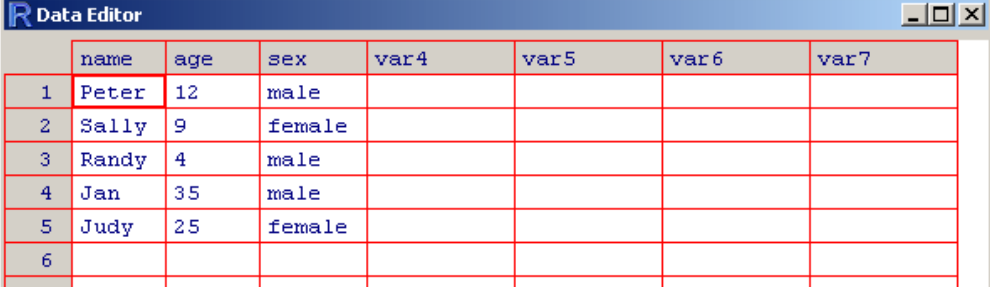
Although these sample datasets are not listed as objects in your workspace, you can use them just like your own objects

```
> data.class(chickwts)
[1] "data.frame"
> dim(chickwts)
[1] 71 2
> names(chickwts)
[1] "weight" "feed"
> data.class(chickwts$weight)
[1] "numeric"
```

Viewing and Editing Data

The `edit()` function invokes a spreadsheet-like data viewer. This function is only for browsing data. Any changes that you make are lost when you close the edit window (even though R asks whether to save the changes). If you need to change the data, use `fix()`.

```
edit(df)
```



	name	age	sex	var4	var5	var6	var7
1	Peter	12	male				
2	Sally	9	female				
3	Randy	4	male				
4	Jan	35	male				
5	Judy	25	female				
6							

Note that you cannot run any R commands when the data editor window is open.

Operations on Objects

Common Operators

The following table lists the most commonly used operators. It is by no means exhaustive.

?	Help
<-	Left assignment
->	Right assignment
-	Minus
+	Plus
*	Multiplication
/	Division
^	Exponentiation
%%	Integer divide
%%*	Matrix product
%%o	Outer product, binary
<	Less than
>	Greater than
==	Equal to
>=	Greater than or equal to
<=	Less than or equal to
&	And, vectorized
	or
!	not
~	Tilde, used for model formulae
:	Sequence, (in model formulae: interaction)

`%in%` Matching operator (in model formulae: nesting)

Arithmetic Operations

Simple arithmetic operations are applied to each element of a vector or matrix.

```
> age
[1] 12 9 4 35
> age+10 # added to each element.
[1] 22 19 14 45
```

Unless you use parentheses, standard order of operations is used – exponentiation precedes multiplication/division, which precedes addition/subtraction:

```
> age+10/2 # standard order of operations (*,/ before +,-)
[1] 17 14 9 40
> (age+10)/2
[1] 11.0 9.5 7.0 22.5
```

If the operands are of different lengths, the shorter one is cycled to match the length of the longer one:

```
> age*c(1,2,3) # different length vectors are cycled.
[1] 12 18 12 35
Warning message:
longer object length
is not a multiple of shorter object length in: age * c(1, 2, 3)
```

Vector operands are applied to matrices by columns:

```
> m
      [,1] [,2] [,3] [,4]
[1,] 1    4    7    10
[2,] 2    5    8    11
[3,] 3    6    9    12
> m*c(1,2,3) # values are cycled through matrix by columns.
      [,1] [,2] [,3] [,4]
[1,] 1    4    7    10
[2,] 4   10   16   22
[3,] 9   18   27   36
```

For matrices, `*` means element-wise multiplication. Use `%*%` for matrix multiplication:

```
> m*m # elementwise multiplication
      [,1] [,2] [,3] [,4]
[1,] 1   16  49  100
[2,] 4   25  64  121
[3,] 9   36  81  144
> t(m)%*%m # matrix (inner product) multiplication
      [,1] [,2] [,3] [,4]
[1,] 14   32   50   68
[2,] 32   77  122  167
[3,] 50  122  194  266
[4,] 68  167  266  365
```

Boolean Operations

Logical operators return values TRUE or FALSE. They are of class "logical".

```
> age==10 # use == for test of equality; = is assignment
[1] FALSE FALSE FALSE FALSE
> data.class(age==10)
```

```
[1] "logical"
```

In interpreting logical statements, & (and) precedes | (or). Use parentheses to make sure logic is interpreted correctly.

```
> age < 15 | age >30 & age !=12 # order of operations: OR, AND
[1] TRUE TRUE TRUE TRUE
> (age < 15 | age >30) & age !=12
[1] FALSE TRUE TRUE TRUE
```

Missing Values – NA, NaN

R uses two "codes" to indicate missing data – NA (not applicable) and NaN (not a number). When you apply a function to a data object some of whose elements are missing, the function may return a missing result. Many functions have an option to specify how to treat missing elements.

```
> ageNA=c(age,NA) # add a missing value to age
> ageNA
[1] 12 9 4 35 NA
> mean(ageNA) # mean function does not exclude missing
[1] NA
> mean(ageNA, na.rm=TRUE) # use na.rm option to exclude missing
[1] 15
```

Mathematical operations that have no defined value are returned as NaN:

```
> log(-1) # not a number
[1] NaN
Warning message:
NaNs produced in: log(x)
```

All operations involving NA return NA.

```
> ageNA > 10
[1] TRUE FALSE FALSE TRUE NA
```

`is.na()` is very useful in locating missing values.

```
> is.na(ageNA) # locate missing values
[1] FALSE FALSE FALSE FALSE TRUE
```

Referencing elements of an object

Vectors and Matrices

Elements

Most generally, elements of vectors and matrices are referenced by their sequential index in square brackets. For matrices, the first index is the row, the second is the column. Matrix elements can be indicated either with the row and column index, or a single index representing the matrix elements counted row-wise. If the rows or columns of a matrix are named, the names can be used as the indices for an element. Some examples:

```
> name[3] # 3rd element of vector name.
[1] "Randy"
> m[1,3] # element in row 1, column 3 of matrix m.
[1] 7
> m[5] # 5th element of matrix is row2, col1.
[1] 5
> A['R1','c2'] # element of a matrix with named rows & columns.
```

```
[1] "b"
> data.class(m[2,])      # a row or column of a matrix is a vector.
[1] "numeric"
```

Rows or Columns of Matrices

An entire row or column of a matrix can be referenced by leaving the other index blank. Again, if the matrix rows/columns are named, you can use either the name or the number of any index:

```
> m[2,]                #second row of m
[1] 2 5 8 11
> A['c1']              # column 'c1' of A
  R1 R2
"a" "d"
```

Dataframes

When working with dataframes, the columns are generally the object of interest. Although rows, columns and cells of a dataframe can be referenced using the same notation as for matrices, there are also some dataframe-specific ways of referencing dataframe columns. In particular, a dataframe reference with only one dimension specified always refers to a column.

Columns

Using single square brackets with a column name or number returns that column of a dataframe as a dataframe, NOT as a vector:

```
> df[1]                #first column as dataframe (output cut out)
> df['name']           # by column name as dataframe
  name
1 Peter
2 Sally
3 Randy
4 Jan
> data.class(df['name'])
[1] "data.frame"
```

Using double square brackets with a column name or number returns that column of the dataframe as a vector. This is what you need in most situations. Combining the dataframe and column names with a \$ is another way to get the same result:

```
> df[[1]]              #first column as vector (output cut out)
> df[['name']]         # by column name as vector (output cut out)
> df$name              # by column name as vector
[1] Peter Sally Randy Jan
Levels: Jan Peter Randy Sally
> data.class(df$name)
[1] "AsIs"
```

Rows

Rows of dataframes are referenced the same way as rows of matrices. The result is a dataframe:

```
> df[2,]               # second row
  name age sex
2 Sally 9 female
> data.class(df[2,])
[1] "data.frame"
```

Cells

Cells of dataframes can be referenced like cells of a matrix. They can also be referenced using vector notation, by first extracting the column as a vector:

```
> df[2,1] # using matrix notation [row, col]
[1] Sally
Levels: Jan Peter Randy Sally
> df[[1]][2] # using vector notation [[col]][row]
[1] Sally
Levels: Jan Peter Randy Sally
> df[['name']][2] # same - output cut out
> df$name[2] # same - output cut out
```

Adding Columns or Rows to a Dataframe

The `cbind()` and `rbind()` functions provide the capability to combine two objects. As the names imply `cbind()` adds columns and `rbind()` adds rows. The appropriate dimension (row or column) of the objects must match for the operation to be successful. `cbind()` and `rbind()` can be used to combine vectors or matrices as well as dataframes. In the following example, `cbind()` is used to add an id number to the dataframe, `df`.

```
> #adding columns to a dataframe
> idnum=seq(1:4) # vector of ID numbers.
> df<-cbind(idnum, df) # add a first column of df
> df
  idnum name age sex
1     1 Peter 12 male
2     2 Sally 9 female
3     3 Randy 4 male
4     4 Jan 35 female
>
```

Subset an Object

Using negative indices excludes rows, columns, or cells of an object.

```
# excluding elements, rows, columns
age[-3] # an element of a vector
m[,-1] # a column of a matrix
m[-c(1,2),] # two rows of a matrix
df[-1] # a column of a dataframe
df[-seq(2,4),] # three rows of a dataframe
```

To select elements of an object based on a logical criterion, you apply a logical (TRUE/FALSE) vector to the object. Elements corresponding to TRUE are selected. All other elements are discarded:

```
> age[age>10] # applies T,F,F,T to vector age
[1] 12 35
```

Using the `is.na()` function you can select the non-missing elements of an object:

```
> ageNA
[1] 12 9 4 35 NA
> ageNA[!is.na(ageNA)]
[1] 12 9 4 35
```

For dataframes, you can select rows based on some logical condition. In the following example, the rows that have a value of age that is greater than 10 are selected.

```

> df[df$age>10,]           # select rows based on age, all columns.
  name age  sex
1 Peter 12  male
4   Jan 35 female

> df[df[[2]]>10,]         # same as above, using column number.
  name age  sex
1 Peter 12  male
4   Jan 35 female

```

To select dataframe columns and rows at the same time, specify the column numbers to include (or exclude) along with the desired logical condition for the rows to select.

```

> df[df[[2]]>10,c(2,3)]  # select rows based on age, columns 2 & 3.
  age  sex
1 12  male
4 35 female

```

For matrices, since all columns are of the same type, you can apply logical conditions to rows, columns, or both:

```

> m
  [,1] [,2] [,3] [,4]
[1,]   1   4   7  10
[2,]   2   5   8  11
[3,]   3   6   9  12
> m[m[,2]>4,]           #rows where column 2 is >4, all columns.
  [,1] [,2] [,3] [,4]
[1,]   2   5   8  11
[2,]   3   6   9  12
> m[m[1,]<8]           # columns where row1 is <8.
  [,1] [,2] [,3]
[1,]   1   4   7
[2,]   2   5   8
[3,]   3   6   9
> m[m[,2]>4,m[1,]<8]   # rows where col2>4 and columns where row1<8
  [,1] [,2] [,3]
[1,]   2   5   8
[2,]   3   6   9

```

R Scripts

Each line you type on the R console is interpreted and executed when you press Enter. If you plan to run the same or similar commands again, or if you want a record of what you did, you can collect the commands in a file. A file of R commands is called a "script".

When you are developing an R program, your R workspace will typically have many objects that you may have tried during testing, but which are no longer needed. Further, it can be difficult to remember what all the objects in a workspace are, how and why they were created, etc. Rather than saving the workspace, you may prefer to save a script of the "successful" commands. This will enable you to re-create the objects you need, and at the same time will document each object. The script can (and should) also contain comments to further document the code.

To start an R script, select `File` → `New script` from the R console menu bar. Type your R programming statements in the script window, or copy and paste commands from the Console window. Adds lots of comments! Comments begin with the pound sign (#) and end at the end of the line.

To run commands in your script file, select `Edit` → `Run all`. Or select a portion of the script and choose `Edit` → `Run line or selection`.

To save your script, select `File` → `Save As`. Browse to the directory where you want to save the script, and assign it a name. The extension will be `.R`.

Note: The File and Edit menu choices change depending on whether the Console or Script window is active. When the Console is active, the File menu lets you save the Workspace or the History, but not the script. With the Script window active, you can save scripts (but not the Workspace or History). The `Run all` and `Run line or selection` items appear on the Edit menu only when the Script window is active.

Example: This R Script creates three vectors, defines a factor, and combines the three vectors into a dataframe.

```
#create a character vector of names
name<-c('Peter','Sally','Randy','Jan')
#create a vector with ages
age<-c(12,9,4,35)
#create a numeric vector indicating the gender of four subjects,
# where 1=female and 2=male
sex<-rep(c(2,1),2)
#create a factor from the numeric vector sex
# anything other than 1 or 2 becomes NA.
sex<-factor(sex,levels=c(1,2),labels=c('female','male'))
#create a dataframe from vectors name, age and sex
df<-data.frame(name, age, sex)
#list the contents of data.frame df on the console
df
```

Import Data from External Files

For the remaining examples we will use data from Appendix A of Minitab Handbook, Second Edition, Ryan, Joiner and Ryan, PWS-KENT Publishing Company, 1985 p. 318. A date variable was added to the original data, and a few missing values introduced for instructional purposes. The data is available at

<http://www-unix.oit.umass.edu/~statdata/statdata>. Go to Index of Local Datasets, and download "Minitab Pulse Data with Dates" dataset.

Each subject measured his/her pulse rate. The subjects were then randomly divided into two groups. One group ran in place for two minutes; the other did nothing. At the end of the two minutes, everyone measured his/her pulse again.

<u>Variable</u>	<u>Description</u>
BDATE	Date of Birth (mm/dd/yyyy)
PULSE1	First Pulse (0 = Missing Value)
PULSE2	Second Pulse (0 = Missing Value)
GROUP	Group (1=Ran in place, 2 = Did not run in place)
SMOKE	Smokes (1= Yes, 2 = No; 9 = Missing Value)
GENDER	Gender (M or F)
HEIGHT	Height in inches

*Eva Goldwater & Mira Shapiro
Biostatistics Consulting Center
University of Massachusetts School of Public Health
June 2006*


```

6  4/18/1986    74    84    1    2    M    73    165    1
7  6/12/1985    84    84    1    9    M    72     0    3..

```

Recall that you refer to a column in your data frame using `dataframe$column`, and that R names are case sensitive. Thus, the columns of this dataframe are referenced as `minidat$BDATE`, `minidat$PULSE1`, etc.

First, let's check the data types that R assigned to the columns of the dataframe. We can do this one column at a time:

```

> data.class(minidat$BDATE)
[1] "character"

```

More conveniently, the `apply()` function can be used to apply a function to every column of a dataframe:

```

> sapply(minidat, data.class)
      BDATE      PULSE1      PULSE2      GROUP      SMOKE      GENDER
"character" "numeric"  "numeric"  "numeric"  "numeric"  "factor"
      HEIGHT      WEIGHT      ACTIVITY
"numeric"  "numeric"  "numeric"

```

Change Class of Dataframe Columns

Observe that `minidat$GROUP`, and `minidat$SMOKE` are numeric, but should be factors, and that `minidat$ACTIVITY` should be an ordered factor. We can re-interpret these columns and store the result in the same column:

```

# make group, smoke, activity factors;
> minidat$GROUP <- factor(minidat$GROUP, levels=c(1,2),
+ labels=c('Ran', 'Did not Run'))
> minidat$SMOKE <- factor(minidat$SMOKE, levels=c(1,2),
+ labels=c('Smoker', 'Non-Smoker'))
> minidat$ACTIVITY <- factor(minidat$ACTIVITY, levels=c(1:3),
+ labels=c('Slight', 'Moderate', 'Very'), ordered=TRUE)

```

Any levels not defined in the factor function become NA. Thus, the missing value codes (9) for these variables become NA.

Also, `minidat$BDATE` is character string of form `mm/dd/yyyy`, which we need to interpret as a date.

```

# change character data to Date.
> minidat$BDATE <- as.Date(minidat$BDATE, "%m/%d/%Y")

```

The `as.Date()` function uses the pattern `"%m/%d/%Y"` to interpret `minidat$BDATE`, and stores the result back in `minidat$BDATE`. The date pattern is described using the following symbols:

```

%y    2 digit year
%Y    4 digit year
%m    1 or 2 digit numeric month (1-12)
%b    3 character month abbreviation
%B    full month name
%d    1 or 2 digit day (1-31)
-     any characters used to separate the parts of the date string

```

Examples:

*Eva Goldwater & Mira Shapiro
 Biostatistics Consulting Center
 University of Massachusetts School of Public Health
 June 2006*

```

> as.Date("9-Oct-1990", "%d-%b-%Y")
[1] "1990-10-09"

> as.Date("October 9, 1990", "%B %d, %Y")
[1] "1990-10-09"

> as.Date("13/30/2001", "%m/%d/%Y")      # invalid month
[1] NA

> as.Date("9/31/2001", "%m/%d/%Y")      # day 31 in 30-day month.
[1] "2001-10-01"

```

Assign Missing Values

The file that we read in contained "0"s for missing values in `minidat$PULSE1`, `minidat$PULSE2` and `minidat$WEIGHT`; and "9"s for missing values in `minidat$SMOKE` and `minidat$ACTIVITY`. The '9's were made missing when we defined the factors levels for `minidat$SMOKE` and `minidat$ACTIVITY` (see previous section). In order to make the "0"s missing, we need to replace them with NA:

```

> # Make 0 missing for these numeric variables
> minidat$PULSE1[minidat$PULSE1==0] <- NA
> minidat$PULSE2[minidat$PULSE2==0] <- NA
> minidat$WEIGHT[minidat$WEIGHT==0] <- NA

```

In these assignment statements, when the condition `minidat$PULSE1==0` is true, NA is assigned to the corresponding position in the vector. When the condition is false, the vector is unchanged.

If an external file with *non-blank* separators has numeric fields left blank, these fields are automatically coded as NA when the file is imported using the `read.table` function. Fields left blank in a blank separated file result in an error on import with `read.table`.

Data Analysis

Now that all columns are defined with the proper class, and missing values, we can begin to analyze the data.

Descriptive Statistics

The `summary()` function is a "generic" function, whose results depend on the data class of its argument. For a numeric argument it gives mean, median, quartiles; for a factor argument it gives frequencies:

```

> summary(minidat$PULSE1)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.   NA's
54.00  66.00  72.00  73.46  80.00 100.00   1.00

> summary(minidat$SMOKE)
  Smoker Non-Smoker    NA's
     28         62         1

```

Use `sapply()` to quickly get summaries of all columns in a dataframe. `sapply` will apply the specified function to each element of the first parameter (in this example, the columns of the data frame `minidat`).

```
sapply(minidat, FUN=summary)
```

The `summary()` function does not give you standard deviations. You can get standard deviations from the `sd` function, but you must use the `na.rm=TRUE` argument to exclude NA's:

```
> sd(minidat$WEIGHT)
Error in var(x, na.rm = na.rm) : missing observations in cov/cor
> sd(minidat$WEIGHT, na.rm=T)
[1] 22.93399
```

Contingency Tables

One-Way Absolute Frequencies

The `table()` function is another way to get absolute frequencies. Unlike the `summary` function, it does not display the number of missing values.

```
> # table function does not display number missing.
> table(minidat$SMOKE)
  Smoker Non-Smoker
     28       62
```

One-Way Relative Frequencies

One of the strengths of R is its ability to nest functions. The result of the inner most function is passed as a parameter to the next outer function. Here we use this technique to create one-way relative frequencies.

The `prop.table()` function returns the proportion that each number in a vector is of the total. We use `summary()` or `table()` to get a table of absolute frequencies (as above), then apply `prop.table` to the resulting table to get relative frequencies.

```
> prop.table(summary(minidat$SMOKE)) # relative freq with NA.
  Smoker Non-Smoker   NA's
0.30769231 0.68131868 0.01098901
> prop.table(table(minidat$SMOKE)) # relative freq without NA.
  Smoker Non-Smoker
0.3111111 0.6888889
```

Multi-way Contingency Tables

The `table` function can also produce multi-way tables of absolute frequencies:

```
> # 2-way contingency tables.
> table(minidat$SMOKE, minidat$GENDER)
```

```
      F  M
Smoker  9 19
Non-Smoker 27 35
```

To get row, column or total percents, apply the `prop.table` function to the results of the `table` function. The `margin` argument of `prop.table` specifies what kind of percents you want – `margin=1` for row percents; `2` for column percents. If you don't specify `margin`, you get total percents.

```
> # row percents (margin=1).
> prop.table(table(minidat$SMOKE, minidat$GENDER), margin=1)
```

	F	M
Smoker	0.3214286	0.6785714
Non-Smoker	0.4354839	0.5645161

Pearson's Chi-squared test

To get a chi-square test statistic for a table, use the `chisq.test` function. The default for this function uses the Yates continuity correction. To get a chi square test without the Yates continuity correction, use the `correct=FALSE` option.

Here is a one-sample chi-square test against the hypothesis that 1/3 of the group are smokers and 2/3 non-smokers. The default test is against equal probabilities. The `p` argument is used to specify a vector of non-equal probabilities:

```
> chisq.test(table(minidat$SMOKE), p=c(.33,.67))

Chi-squared test for given probabilities

data:  table(minidat$SMOKE)
X-squared = 0.1452, df = 1, p-value = 0.7031
```

The `chisq.test` function applied to a two-way table gives the chi-square test of independence. Here we request the test without Yates continuity correction:

```
> chisq.test(minidat$SMOKE,minidat$GENDER, correct=FALSE) # no Yates
correction.

Pearson's Chi-squared test

data:  minidat$SMOKE and minidat$GENDER
X-squared = 1.0455, df = 1, p-value = 0.3065
```

Selecting Observations (Rows)

You can select specific rows, based on the values of any column, then apply the function of interest to the selected rows. In this example we request the one-way absolute frequency of SMOKE for females.

```
#Use table function on selected rows (females).
> table(minidat$SMOKE[minidat$GENDER=="F"])

Smoker Non-Smoker
     9         27
```

Here we get the number of smokers and non-smokers among those observations where `pulse1` is greater than 80.

```
# use the table function on rows with pulse1 greater than 80.
> table(minidat$SMOKE[minidat$PULSE1>80])

Smoker Non-Smoker
     10         11
>
```

Working with Function Results

The above `chisq.test` function displayed the results of the test of independence. If you assign a function to a new variable you can use the results of the function for additional work.

Some functions also return additional results that are not obvious from the display:

```
> csq=chisq.test(minidat$SMOKE,minidat$GENDER, correct=FALSE)
> data.class(csq)
[1] "htest"
> length(csq)
[1] 8
> names(csq)
[1] "statistic" "parameter" "p.value" "method" "data.name" "observed"
[7] "expected" "residuals"
```

The result of `chisq.test` is an object of type "htest", and has 8 elements. The 8 elements are of various data classes:

```
> sapply(csq, data.class)
statistic parameter p.value method data.name observed
"numeric" "numeric" "numeric" "character" "character" "table"
expected residuals
"matrix" "table"
```

Not all the elements are displayed by default:

```
> csq

Pearson's Chi-squared test

data: minidat$SMOKE and minidat$GENDER
X-squared = 1.0455, df = 1, p-value = 0.3065
```

To see the last 3 elements you need to request them explicitly. Here are the residuals:

```
> csq[8]
$residuals
      minidat$GENDER
minidat$SMOKE      F      M
  Smoker    -0.6573757  0.5367450
 Non-Smoker  0.4417706 -0.3607042
```

Paired T-Test

The `t.test()` function can be used to get either paired or independent samples t-tests. To get a paired t-test, use the following form of the `t.test()` function:

```
> # a paired t test
> t.test(minidat$PULSE1,minidat$PULSE2,paired=T)

Paired t-test

data: minidat$PULSE1 and minidat$PULSE2
t = -4.8662, df = 88, p-value = 0.000004957
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -10.016945 -4.207774
sample estimates:
mean of the differences
      -7.11236
```

2-sample t-test

To get a 2-sample t-test, use the following form of the `t.test()` function:

```
># two sample t-test
>t.test(minidat$PULSE1~minidat$GENDER)
```

Here the dependent variable is `minidat$PULSE1` and the two samples are determined by the factor variable `minidat$GENDER`. The grouping variable must be defined as a factor.

R returns the following results:

```
Welch Two Sample t-test

data:  minidat$PULSE1 by minidat$GENDER
t = 2.3705, df = 66.012, p-value = 0.02070
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.8733857 10.2006884
sample estimates:
mean in group F mean in group M
      76.77778      71.24074
```

Formula Syntax

In this example we do a Two-Way Analysis of variance of `minidat$PULSE1` as the outcome, with `minidat$GENDER` and `minidat$GROUP` as the factors. The formula to describe a full factorial model uses the following general syntax:

```
dependent~factor1 + factor2 + factor1:factor2
```

Tilda (`~`) separates the dependent variable from the model; plus sign (`+`) adds various effects to the model, and a colon (`:`) describes interaction effects.

```
>anova(lm(minidat$PULSE2~minidat$GENDER+minidat$GROUP+minidat$GENDER:minidat$GROUP))
Analysis of Variance Table

Response: minidat$PULSE2
          Df Sum Sq Mean Sq F value    Pr(>F)
minidat$GENDER      1  2898.1   2898.1   22.352 8.796e-06 ***
minidat$GROUP       1  9465.8   9465.8   73.007 4.163e-13 ***
minidat$GENDER:minidat$GROUP  1  2596.8   2596.8   20.028 2.326e-05 ***
Residuals          86 11150.5    129.7
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Graphics

High-Level Graphics Functions

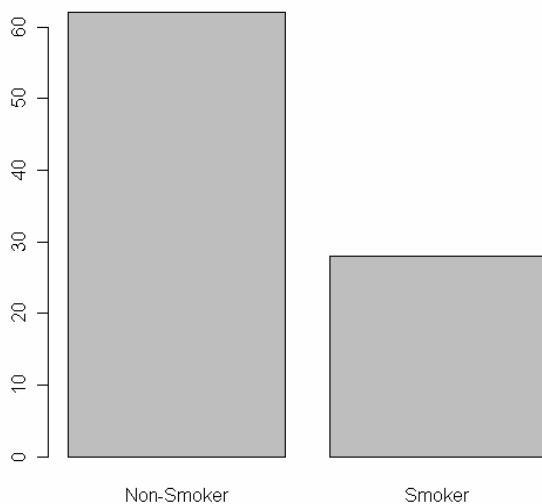
High level graphics functions create new plots. By default, there is only one graphics view window. A high-level graphics function opens the graphics view window, and places the latest graph in it, replacing any that may have been there previously.

Plot Function

The generic function `plot()` is the simplest way to generate a graph in R. The `plot` function will create the type of graph that its algorithm deems appropriate given the variables that are to be plotted. Additional options for the `plot` function and additional functions are available that are more specific in nature and allow for graph selection and customization. Here are examples of the `plot` function with different types of variables.

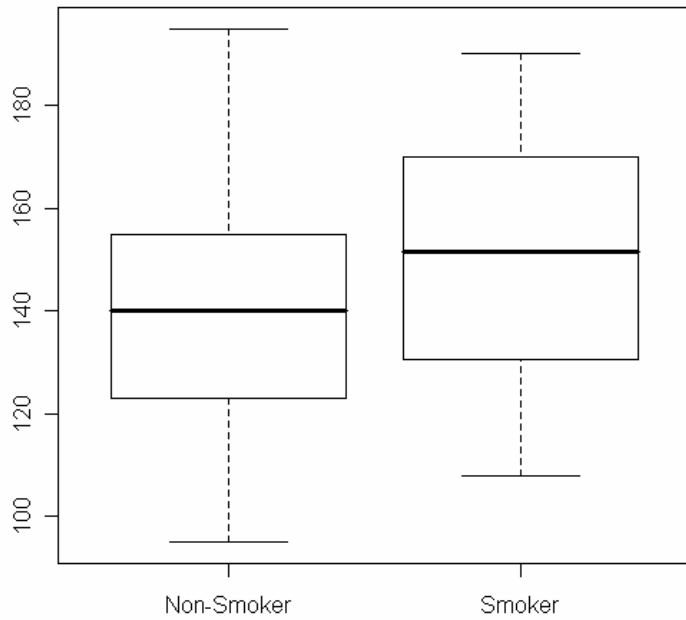
Bar Graph: Here `minidat$SMOKE` is a factor. Used with a factor, `plot()` makes a Bar Chart of the frequencies of each value of the factor.

```
> plot(minidat$SMOKE)
```



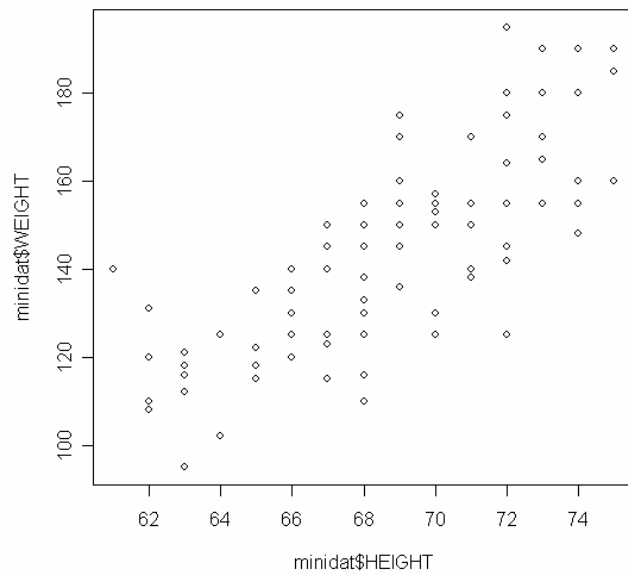
Box Plots: Here `minidat$SMOKE` is a factor and `minidat$WEIGHT` is a numeric vector. The `plot()` function makes a Boxplot of the `minidat$WEIGHT` at each value of the factor `minidat$SMOKE`.

```
> plot(minidat$SMOKE,minidat$WEIGHT)
```



Scatterplots: Here `minidat$WEIGHT` and `minidat$HEIGHT` are both numeric vectors. Plot makes a scatterplot of `minidat$HEIGHT` vs. `minidat$WEIGHT`.

```
> plot(minidat$HEIGHT,minidat$WEIGHT)
```

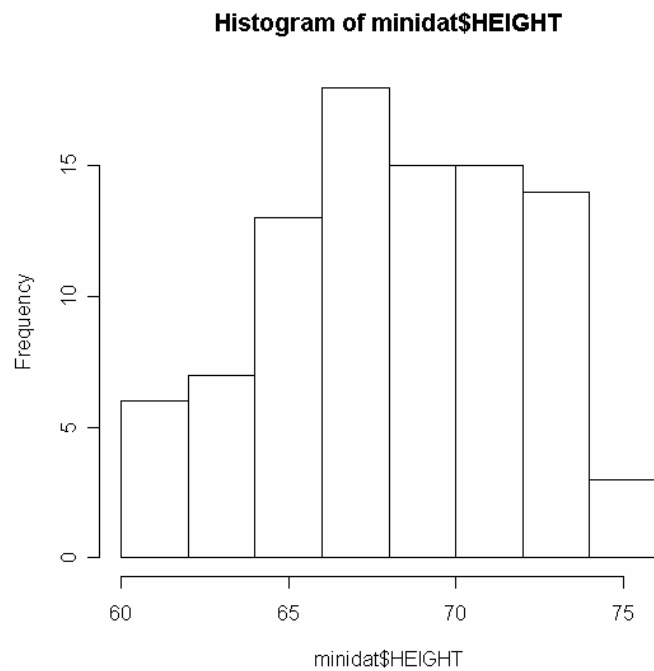


Histograms

The `hist()` function makes histograms. R selects the number of bars for the histogram. The `nclass` or `breaks` options can be used to suggest the desired number of bars or break points. Note that these are taken as only suggestions, and R may override these specifications

Example1: Using the `hist()` function.

```
> hist(minidat$WEIGHT)
```



Other High Level Graphics Functions

Other high-level graphics functions include `barplot`, `boxplot`, `dotchart`, `pie`, `qqnorm`, `contour`, `pairs` and `coplot`.

Useful Parameters for High Level Graphics Functions

Here are some parameters that can be used with most high-level graphics function:

`type='value'`

where `value` is a code for the type of plot. Some possible values (there are others) and their meanings are:

- `n` create the axis and coordinate scale but do not plot the data
- `p` plot the data points
- `l` connect data with a line (data points are not marked).
- `b` plot the points and connect them with lines

`xlab=string`

`ylab=string`

where `string` is the text to print for the x or y axis labels.

`main=string`

where `string` is a title to be printed at the top of the plot.

Function Plots

We demonstrate the use of these parameters with a plot of the sine function:

```
> x=seq(0,2*pi,by=.01)
> y=sin(x)
> plot(x,y, type='l', main='Sine Function', xlab='X (Radians)',
+ ylab='Sine(x)')
```

Graphics Window Management

Unless you tell it otherwise, high-level graphics functions always display their output graph in a single graphics window, overwriting any previously created graph in that window. In order to view more than one graph at the same time, open a new graphics window using the `windows()` function.

The following displays two histograms. The second will appear on top of the first, but it does not replace it; you can drag its window aside to see both at the same time.

```
> hist(minidat$HEIGHT)
> windows()
> hist(minidat$WEIGHT)
```

Graphics windows are referenced by number, called 'device' numbers. Numbering begins at 2. To see the device numbers of the graphics windows currently open, use `dev.list()`. Here we see that there are now two open graphics windows devices, numbered 2 & 3.

```
> dev.list()
windows windows
      2      3
```

Only one graphics window can be active at a time. Unless a new window is opened, any new graphics produced is added to the active graphics window. By default, the most recently created graphics window is the active window. To check which is the active window, use `dev.cur()`

```
> dev.cur()
windows
      3
```

To change the active window, use `dev.set()`. Here we make window (device) 2 active.

```
> dev.set(2)
windows
      2
```

Remember that all new graphics output goes to the active window.

Finally, you can close a graphics window using `dev.off()`

```
> dev.off(2)
windows
      3
```

Low-Level Graphics

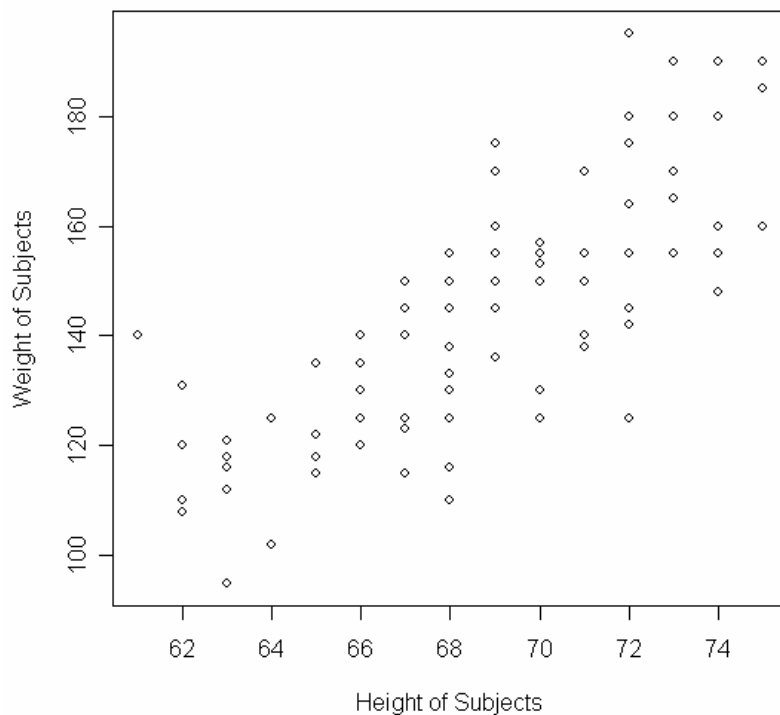
In contrast to high-level graphics functions, low-level graphics functions do not create new plots; rather, they are used to embellish existing graphs. It is often easiest to begin by creating a plot using high-level graphics functions, then add titles, lines, legends, colors and other features using low-level graphics functions.

R uses the active device window as the target for embellishments added with low-level graphics functions. Once added, a feature cannot be removed. You will need to regenerate the graph and add the desired features.

If you have more than one graphics window open, make sure that the one you want to add the embellishments to is the active window. (See previous section for how to do this.)

In the following example, a simple scatter plot is created containing axis labels. We then add a regression line and title.

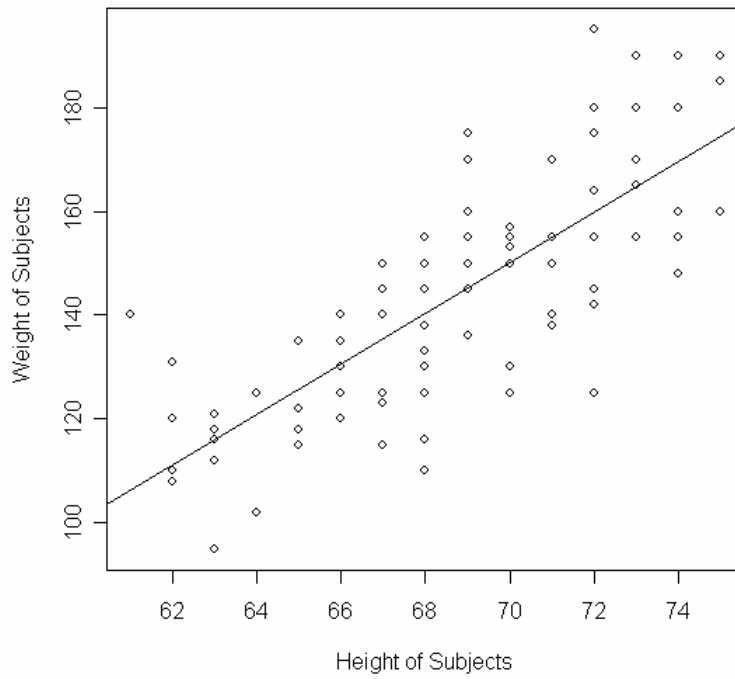
```
# start with a high-level plot
> plot(minidat$HEIGHT,minidat$WEIGHT,xlab="Height of
+ Subjects",ylab="Weight of Subjects")
```



Regression Line

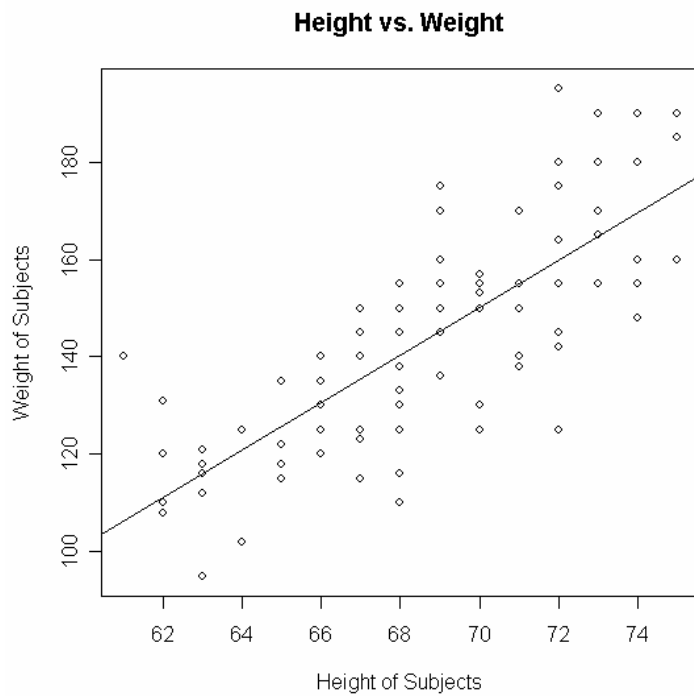
The `abline()` function to adds a regression line to a plot.

```
> # compute regression of height on weight. store results in wtht.
> wtht<-lm(minidat$WEIGHT~minidat$HEIGHT)
> # Add the regression line to the active graphics window
> abline(wtht)
```



Title

```
# add main title to current graph  
title("Height vs. Weight")
```

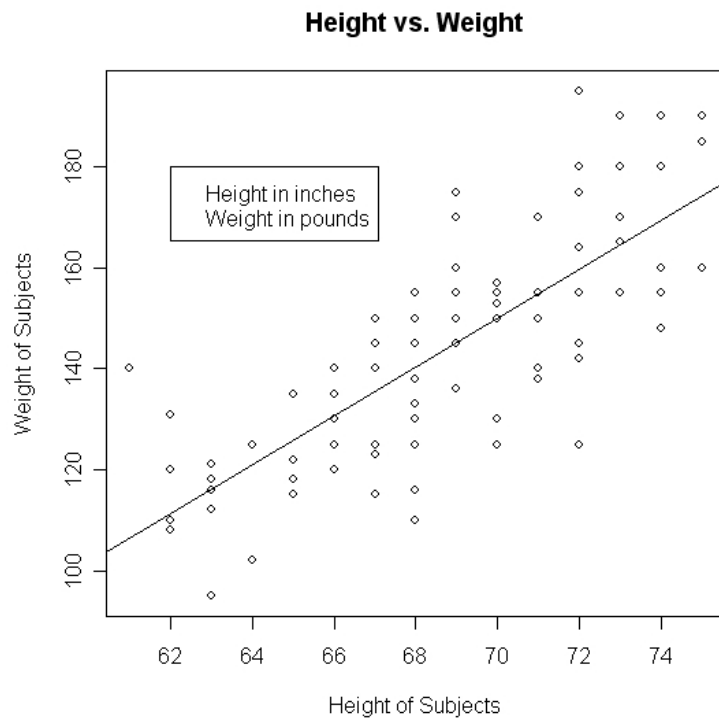


Legends

To add a legend to the active graphics window, use the `legend()` function. The `legend()` function has many parameters and options that can be examined via the R help facility. The following simple example creates a vector of labels and then adds the legend to our active graphics window.

```
> # add a legend
> leg<-c("Height in inches", "Weight in pounds")
> legend(62,180,leg)
```

Here the 62, 180 are the x and y coordinates of the left upper corner of the legend location; `leg` is a vector containing the text to be displayed.

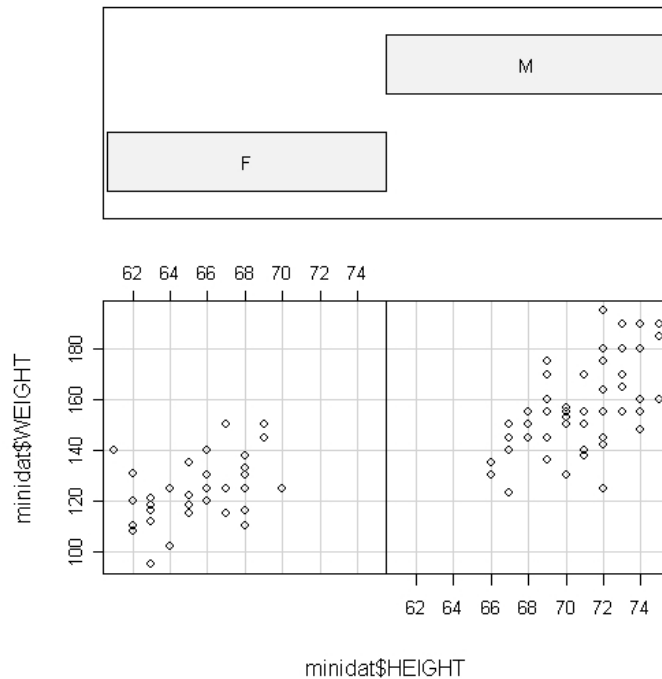


Subsetting Data and presenting side-by-side graphs

The `coplot()` function is a quick way to get graphs of subgroups. Here we compare the weight vs. height plots of males and females:

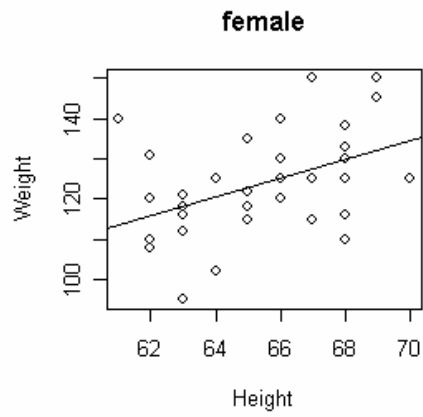
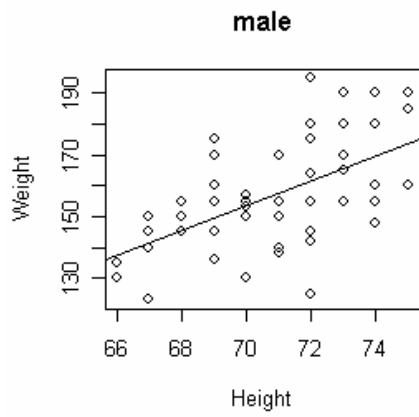
```
> coplot(minidat$WEIGHT~minidat$HEIGHT|minidat$GENDER)
```

Given : minidat\$GENDER



More generally, the `par()` function is used to arrange multiple graphs on a page. Here we use `par` function to set a template for 4 graphs per page, arranged in two rows and two columns. We then use a logical variable, `mf`, to select males. The `plot()` function creates a scatter plot of the selected subjects, and `abline` with `lm` adds the linear regression line to the plot. This plot is placed in the first available location of the 2x2 page template. The process is then repeated for females, placing the plot in the next available location. The remaining 2 locations are not used.

```
> par(mfrow=c(2,2))
> # mf is a logical variable that is used to select males
> mf<- minidat$GENDER=="M"
> plot(minidat$WEIGHT[mf] ~ minidat$HEIGHT[mf], xlab = 'Height', ylab =
+ 'Weight')
> title("Male")
> abline(lm(minidat$WEIGHT[mf] ~ minidat$HEIGHT[mf]))
>
> #repeat for females
> mf<- minidat$GENDER=="F"
> plot(minidat$WEIGHT[mf] ~ minidat$HEIGHT[mf], xlab = 'Height', ylab =
+ 'Weight')
> title("Female")
> abline(lm(minidat$WEIGHT[mf] ~ minidat$HEIGHT[mf]))
```



Summary of Frequently Used R functions

Managing Workspaces and Libraries

load	Open an R workspace
search	Display the search path of currently loaded libraries
library	Add a library to the search path
save.image	Save an R workspace
q	Quit R

Information about Objects

help	Display help for a function
help.search	Search help for a phrase/keyword
conflicts	Display names of objects that are hidden by other objects with the same name.
dim	Display the dimensions of a matrix or dataframe
dimnames	Display the names of the rows and columns of a matrix or dataframe.
is.na	Show which elements of an object are missing (NA).
length	Display the length of a vector.
levels	Display the levels of a factor.
names	List names of the columns of a dataframe.
objects	List the objects in the workspace.

Creating and Modifying Objects

c	Combine elements into a vector
cbind	Combine vectors into a matrix columnwise
data.frame	Create a data frame
as.Date	Interpret characters string as a date.
edit	Open object in data editor
factor	creates a categorical variable with value labels if desired
fix	Open object in data editor
l	Treat an object 'as is' – i.e. do not interpret it.
matrix	Make a matrix object
rbind	Combine vectors into a matrix rowwise
read.table	read text file into a data frame
rep	Make a vector by replicating an object.
rnorm	Generate normally distributed random numbers
runif	Generate uniformly distributed random numbers.
sapply	apply a function to elements of a list
seq	Make a vector of sequential numbers or characters

Statistical Functions

anova	Display an anova table from an lm object
chisq.test	Compute chisquare test on a vector or matrix
fitted	extracts the fitted values of an lm object
lm	fit a linear model
mean	Compute the mean of a vector.
prop.table	Compute cell proportions of a table
resid	extract the residuals from a lm object

sd	Compute the standard deviation of a vector.
summary	generic function to display summary statistics of an object
t.test	t-tests, including one sample, two sample and paired
table	creates a table of absolute frequencies.
var	Compute the variance of a vector.

Graphics Functions

abline	add a straight line to an existing plot
coplot	Create side-by-side plots, conditional on other objects.
dev.cur	Display the currently active graphics device.
dev.list	List all currently open graphics devices.
dev.off	Close a graphics device
dev.set	Make a graphics device active.
hist	Create a histogram plot
legend	place a legend on a plot
title	Place a title on a plot
par	Set or display graphics parameters.
plot	Create a plot (generic function)
windows	Open a new graphics window.

Index

abline.....	34, 38	lm.....	28, 34, 38
anova.....	28	load.....	6
as.date.....	11, 23	matrix.....	12
c.....	10	mean.....	8, 9
cbind.....	19	names.....	14
chisq.test.....	26, 27	objects.....	13
conflicts.....	9	par.....	38
coplot.....	37	plot.....	29, 38
data.class.....	10, 13	prop.table.....	25
data.frame.....	12, 13	q.....	5
dev.cur.....	33	rbind.....	19
dev.list.....	33	read.table.....	22, 24
dev.off.....	33	rep.....	10
dev.set.....	33	resid.....	40
dim.....	14	rm.....	6, 9
dimnames.....	14	rnorm.....	9, 10
edit.....	15	runif.....	10
factor.....	10, 13, 23	sapply.....	23, 24
fitted.....	40	save.image.....	6
fix.....	15	sd.....	25
help.....	7	search.....	7, 8
hist.....	31	seq.....	10
l.....	13	summary.....	24, 25
is.na.....	17	t.test.....	27
legend.....	36	table.....	25
length.....	13	title.....	35
levels.....	14	windows.....	33
library.....	7, 8, 14		