

R; an Overview

Outline

- Why R, and R Paradigm
- References, Tutorials and links
- R Overview
- R Interface
- R Workspace
- Help
- R Packages
- Input/Output
- Reusing Results

Why R?

It's free!

It runs on a variety of platforms including Windows, Unix and MacOS.

It provides an unparalleled platform for programming new statistical methods in an easy and straightforward manner.

It contains advanced statistical routines not yet available in other packages.

It has state-of-the-art graphics capabilities.

R has a steep Learning Curve

First, while there are many introductory tutorials (covering data types, basic commands, the interface), none alone are comprehensive. In part, this is because much of the advanced functionality of R comes from hundreds of user contributed packages. Hunting for what you want can be time consuming, and it can be hard to get a clear overview of what procedures are available.

R has a Learning Curve

The **second** reason is more transient. As users of statistical packages, we tend to run one controlled procedure for each type of analysis. Think of PROC GLM in SAS. We can carefully set up the run with all the parameters and options that we need. When we run the procedure, the resulting output may be a hundred pages long. We then sift through this output pulling out what we need and discarding the rest.

R paradigm is different

Rather than setting up a complete analysis at once, the process is highly interactive. You run a command (say fit a model), take the results and process it through another command (say a set of diagnostic plots), take those results and process it through another command (say cross-validation), etc. The cycle may include transforming the data, and looping back through the whole process again. You stop when you feel that you have fully analyzed the data.

How to download?

- Google it using R or CRAN
(Comprehensive R Archive Network)
- <http://www.r-project.org>

tutorials

Each of the following tutorials are in PDF format.

- P. Kuhnert & B. Venables, An Introduction to R: Software for Statistical Modeling & Computing
- J.H. Maindonald, Using R for Data Analysis and Graphics
- B. Muenchen, R for SAS and SPSS Users
- W.J. Owen, The R Guide
- D. Rossiter, Introduction to the R Project for Statistical Computing for Use at the ITC

Web links

- Paul Geissler's [excellent R tutorial](#)
- Dave Robert's [Excellent Labs on Ecological Analysis](#)
- [Excellent Tutorials by David Rossitier](#)
- [Excellent tutorial on nearly every aspect of R](#) **MOST of these notes follow this web page format**
- [Introduction to R by Vincent Zoonekynd](#)
- [R Cookbook](#)
- [Data Manipulation Reference](#)

Web links

- [R time series tutorial](#)
- [R Concepts and Data Types](#)
- [Interpreting Output From lm\(\)](#)
- [The R Wiki](#)
- [An Introduction to R](#)
- [Import / Export Manual](#)
- [R Reference Cards](#)

Web links

- [KickStart](#)
- [Hints on plotting data in R](#)
- [Regression and ANOVA](#)
- [Appendices to Fox Book on Regression](#)
- [JGR a Java-based GUI for R \[Mac|Windows|Linux\]](#)
- [A Handbook of Statistical Analyses Using R\(Brian S. Everitt and Torsten Hothorn\)](#)

R Overview

R is a comprehensive statistical and graphical programming language and is a dialect of the S language:

- 1988 - S2: RA Becker, JM Chambers, A Wilks
- 1992 - S3: JM Chambers, TJ Hastie
- 1998 - S4: JM Chambers

R: initially written by Ross Ihaka and Robert Gentleman at Dep. of Statistics of U of Auckland, New Zealand during 1990s.

Since 1997: international “R-core” team of 15 people with access to common CVS archive

R Overview

You can enter commands one at a time at the command prompt (>) or run a set of commands from a source file.

There is a wide variety of data types, including vectors (numerical, character, logical), matrices, data frames, and lists.

To quit R, use

```
>q()
```

R Overview

Most functionality is provided through built-in and user-created functions and all objects are kept in memory during an interactive session.

Basic functions are available by default. Other functions are contained in packages that can be attached to a current session as needed

R Overview

A key skill to using **R** effectively is learning how to use the built-in help system. Other sections describe the working environment, inputting programs and outputting results, installing new functionality through packages and etc.

A fundamental design feature of **R** is that the output from most functions can be used as input to other functions. This is described in reusing results.

R Interface

Start the R system, the main window (RGui) with a sub window (R Console) will appear

In the `Console' window the cursor is waiting for you to type in some commands.

Your First R Session

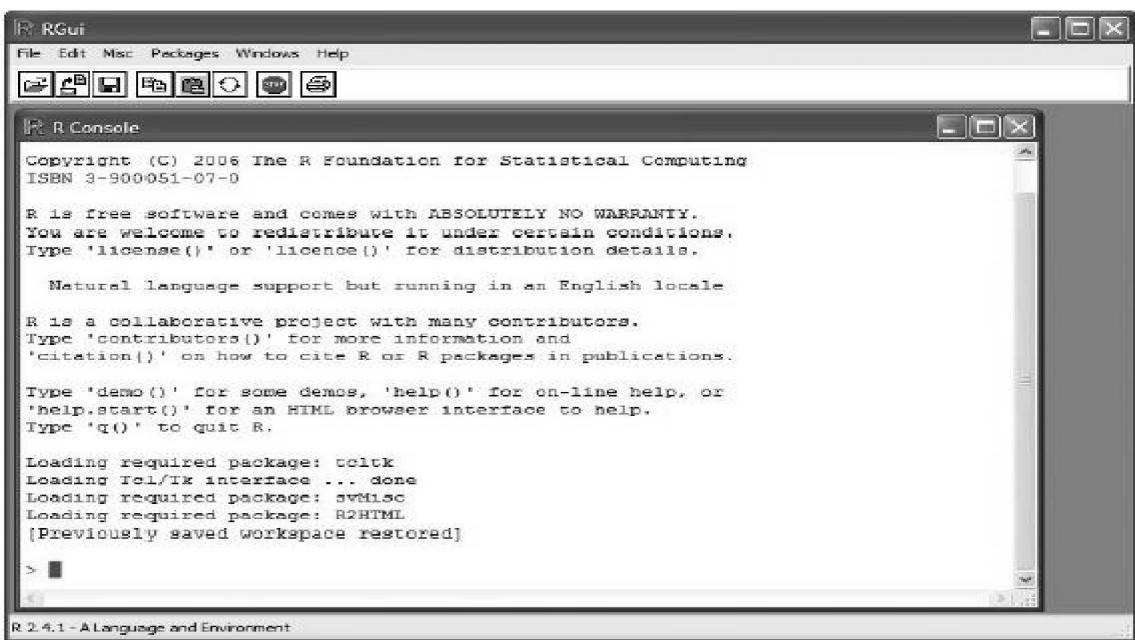


Figure 1.1: The R system on Windows

R Introduction

- Results of calculations can be stored in objects using the assignment operators:
 - An arrow (<-) formed by a smaller than character and a hyphen without a space!
 - The equal character (=).

R Introduction

- These objects can then be used in other calculations
To print the object just enter the name of the object
There are some restrictions when giving an object name:
 - Object names cannot contain 'strange' symbols like !, +, #.
 - A dot (.) and an underscore () are allowed, also a name starting with a dot.
 - Object names can contain a number but cannot start with a number.
 - R is case sensitive, X and x are two different objects, as well as temp and temP.

An example

```
> # An example
> x <- c(1:10)
> x[(x>8) | (x<5)]
> # yields 1 2 3 4 9 10
> # How it works
> x <- c(1:10)
> x
> 1 2 3 4 5 6 7 8 9 10
> x > 8
> F F F F F F F F T T
> x < 5
> T T T T F F F F F F
> x > 8 | x < 5
```

R Introduction

- To list the objects that you have in your current R session use the function `ls` or the function `objects`.

```
> ls()  
[1] "x" "y"
```
- So to run the function `ls` we need to enter the name followed by an opening (and a closing). Entering only will just print the object, you will see the underlying R code of the the function `ls`. Most functions in R accept certain arguments. For example, one of the arguments of the function `ls` is `pattern`. To list all objects starting with the letter x:

```
> x2 = 9  
> y2 = 10  
> ls(pattern="x")  
[1] "x" "y"
```

R Introduction

- If you assign a value to an object that already exists then the contents of the object will be overwritten with the new value (without a warning!). Use the function `rm` to remove one or more objects from your session.
`> rm(x, x2)`
- Lets create two small vectors with data and a scatterplot.

```
z2 <- c(1,2,3,4,5,6)
z3 <- c(6,8,3,5,7,1)
plot(z2,z3)
title("My first scatterplot")
```

R Warning !

R is a case sensitive language.

FOO, Foo, and foo are three different objects

R Introduction

```
> x = sin(9)/75
> y = log(x) + x^2
> x
[1] 0.005494913
> y
[1] -5.203902
> m <- matrix(c(1,2,4,1), ncol=2)
> m
> [,1] [,2]
[1,] 1 4
[2,] 2 1
> solve(m)
- - - - -
```

R Workspace

Objects that you create during an R session are held in memory, the collection of objects that you currently have is called the workspace. This workspace is not saved on disk unless you tell R to do so. This means that your objects are lost when you close R and not save the objects, or worse when R or your

R Workspace

When you close the RGui or the R console window, the system will ask you want to save the workspace image. If you select to save the workspace image then all the objects in your current R session are saved in a file .RData. This is a binary file located in the working directory of R which is by default the installation ..

R Workspace

- During your R session you can also explicitly save the workspace image. Go to the ‘File’ menu and then select ‘Save Workspace...’, or use the `save.image` function.

```
## save to the current working directory  
save.image()  
## just checking what the current working directory is  
getwd()  
## save to a specific file and location  
save.image("C:\\Program Files\\R\\R-2.5.0\\bin\\.RData")
```

R Workspace

If you have saved a workspace image and you start R the next time, it will restore the workspace. So all your previously saved objects are available again. You can also explicitly load a saved workspace, that could be the workspace image of someone else. Go the 'File' menu and select 'Load workspace...'.

R Workspace

Commands are entered interactively at the **R** user prompt. **Up** and **down arrow keys** scroll through your command history.

You will probably want to keep different projects in different physical directories

R Workspace

R gets confused if you use a path in your code like

c:\mydocuments\myfile.txt

This is because R sees "\\" as an escape character. Instead, use

c:\\my documents\\\\myfile.txt

or

c:/mydocuments/myfile.txt

R Workspace

```
getwd() # print the current working directory
```

```
ls() # list the objects in the current workspace  
setwd(mydirectory) # change to mydirectory  
setwd("c:/docs/mydir")
```

R Workspace

```
#view and set options for the session
  help(options) # learn about available options
  options() # view current option settings
  options(digits=3) # number of digits to print on output
# work with your previous commands
  history() # display last 25 commands
  history(max.show=Inf) # display all previous commands
```

R Workspace

```
# save your command history  
  savehistory(file="myfile") # default is ".Rhistory"  
  
# recall your command history  
  loadhistory(file="myfile") # default is ".Rhistory"
```

R Help

Once **R** is installed, there is a comprehensive built-in help system. At the program's command prompt you can use any of the following:

```
help.start() # general help  
help(foo)   # help about function foo  
?foo        # same thing  
apropos("foo") # list all function containing string foo  
example(foo) # show an example of function foo
```

R Help

```
# search for foo in help manuals and archived mailing lists
RSiteSearch("foo")
# get vignettes on using installed packages
vignette()    # show available vignettes
vignette("foo") # show specific vignette
```

R Datasets

R comes with a number of sample datasets that you can experiment with.

> data()

to see the available datasets. The results will depend on which packages you have loaded. Type

help(datasetname)

for details on a sample dataset.

R Packages

- One of the strengths of R is that the system can easily be extended. The system allows you to write new functions and package those functions in a so called 'R package' (or 'R library'). The R package may also contain other R objects, for example data sets or documentation. There is a lively R user community and many R packages have been written and made available on CRAN for other users. Just a few examples, there are packages for portfolio optimization, drawing maps exporting objects to html, time series analysis, spatial statistics and the list goes on and on.

R Packages

To attach another package to the system you can use the menu or the library function. Via the menu:

Select the 'Packages' menu and select 'Load package...', a list of available packages on your system will be displayed. Select one and click 'OK', the package is now attached to your current R session. Via the library function:

```
> library(MASS)
> shoes
$A
[1] 13.2 8.2 10.9 14.3 10.7 6.6 9.5 10.8 8.8 13.3
$B
[1] 14.0 8.8 11.2 14.2 11.8 6.4 9.8 11.3 9.3 13.6
```

R Packages

- The function library can also be used to list all the available libraries on your system with a short description. Run the function without any arguments

```
> library()
```

Packages in library 'C:/PROGRA~1/R/R-25~1.0/library':

base	The R Base Package
Boot	Bootstrap R (S-Plus) Functions (Canty)
class	Functions for Classification
cluster	Cluster Analysis Extended Rousseeuw et al.
codetools	Code Analysis Tools for R
datasets	The R Datasets Package
DBI	R Database Interface
foreign	Read Data Stored by Minitab, S, SAS, SPSS

R Packages

```
install = function() {  
  install.packages(c("moments","graphics","Rcmdr","hexbin"),  
    repos="http://lib.stat.cmu.edu/R/CRAN")  
}  
install()
```

R Conflicting objects

- It is not recommended to do, but R allows the user to give an object a name that already exists. If you are not sure if a name already exists, just enter the name in the R console and see if R can find it. R will look for the object in all the libraries (packages) that are currently attached to the R system. R will not warn you when you use an existing name.

```
> mean = 10  
> mean  
[1] 10
```

- The object mean already exists in the base package, but is now masked by your object mean. To get a list of all masked objects use the function conflicts

R Conflicting objects

The object mean already exists in the base package, but now masked by your object mean. To get a list of all masked objects use the function conflicts.

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

You can safely remove the object mean with the function rm() without risking deletion of the mean function.

Calling rm() removes only objects in your working environment by default

Source Codes

You can have input come from a script file (a file containing R commands) and direct it to a variety of destinations.

Input

The **source()** function runs a script in the current session. If the filename does not include a path, the file is taken from the current working directory.

```
# input a script
```

```
source("myfile")
```

Output

Output

The **sink()** function defines the direction of the output.

direct output to a file

```
  sink("myfile", append=FALSE, split=FALSE)
```

return output to the terminal

```
  sink()
```

Output

The **append** option controls whether output overwrites or adds to a file.

The **split** option determines if output is also sent to the screen as well as the output file.

Here are some examples of the **sink()** function.

```
# output directed to output.txt in c:\projects directory.
```

```
# output overwrites existing file. no output to terminal.
```

```
sink("myfile.txt", append=TRUE, split=TRUE)
```

Graphs

To redirect graphic output use one of the following functions. Use **dev.off()** to return to the terminal.

Function	Output to
<code>pdf("mygraph.pdf")</code>	pdf file
<code>win.metafile("mygraph.wmf")</code>	windows metafile
<code>png("mygraph.png")</code>	png file
<code>jpeg("mygraph.jpg")</code>	jpeg file
<code>bmp("mygraph.bmp")</code>	bmp file
<code>postscript("mygraph.ps")</code>	postscript file

Redirecting Graphs

```
# example - output graph to jpeg file  
jpeg("c:/mygraphs/myplot.jpg")  
plot(x)  
dev.off()
```

Reusing Results

One of the most useful design features of **R** is that the output of analyses can easily be saved and used as input to additional analyses.

Example 1

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

This will run a simple linear regression of miles per gallon on car weight using the data frame mtcars. Results are sent to the screen. Nothing is saved.

Reusing Results

Example 2

```
fit <- lm(mpg~wt, data=mtcars)
```

This time, the same regression is performed but the results are saved under the name `fit`. The output is sent to the screen. However, you now can manipulate the results.

```
str(fit) # view the contents/structure of "fit"
```

The assignment has actually created a list called "fit" that contains a wide range of information (including the predicted values, residuals, coefficients, and more).

Reusing Results

```
# plot residuals by fitted values  
plot(fit$residuals, fit$fitted.values)
```

To see what a function returns, look at the **value** section of the online help for that function.
Here we would look at **help(lm)**.

The results can also be used by a wide range of other functions.

```
# produce diagnostic plots  
plot(fit)
```

Lecture 2: Data Input

Outline

- Data Types
- Importing Data
- Keyboard Input
- Database Input
- Exporting Data
- Viewing Data
- Variable Labels
- Value Labels
- Missing Data
- Date Values

Data Types

R has a wide variety of data types including scalars, vectors (numerical, character, logical), matrices, dataframes, and lists.

Vectors

```
a <- c(1,2,5.3,6,-2,4) # numeric vector  
b <- c("one","two","three") # character vector  
c <- c(TRUE,TRUE,TRUE,FALSE,TRUE,FALSE)  
#logical vector  
Refer to elements of a vector using subscripts.  
a[c(2,4)] # 2nd and 4th elements of vector
```

Matrices

All columns in a matrix must have the same mode(numeric character, etc.) and the same length.

The general format is

```
mymatrix <- matrix(vector, nrow=r, ncol=c,  
  byrow=FALSE,dimnames=list(char_vector_rownames,  
  char_vector_colnames))
```

byrow=TRUE indicates that the matrix should be filled by rows. **byrow=FALSE** indicates that the matrix should be filled by columns (the default). **dimnames** provides optional labels for the columns and rows.

Matrices

```
# generates 5 x 4 numeric matrix  
y<-matrix(1:20, nrow=5,ncol=4)  
  
# another example  
cells <- c(1,26,24,68)  
rnames <- c("R1", "R2")  
cnames <- c("C1", "C2")  
mymatrix <- matrix(cells, nrow=2, ncol=2, byrow=TRUE,  
dimnames=list(rnames, cnames))  
  
#Identify rows, columns or elements using subscripts.  
x[,4] # 4th column of matrix  
x[3,] # 3rd row of matrix
```

Arrays

Arrays are similar to matrices but can have more than two dimensions. See **help(array)** for details.

Data frames

A data frame is more general than a matrix, in that different columns can have different modes (numeric, character, factor, etc.).

```
d <- c(1,2,3,4)
e <- c("red", "white", "red", NA)
f <- c(TRUE,TRUE,TRUE,FALSE)
mydata <- data.frame(d,e,f)
names(mydata) <- c("ID","Color","Passed") #variable names
```

Data frames

There are a variety of ways to identify the elements of a dataframe .

```
myframe[3:5] # columns 3,4,5 of dataframe
```

```
myframe[c("ID","Age")] # columns ID and Age from dataframe
```

```
myframe$X1 # variable x1 in the dataframe
```

Lists

An ordered collection of objects (components). A list allows you to gather a variety of (possibly unrelated) objects under one name.

```
# example of a list with 4 components -
```

```
# a string, a numeric vector, a matrix, and a scalar
```

```
w <- list(name="Fred", mynumbers=a, mymatrix=y, age=5.3)
```

```
# example of a list containing two lists
```

```
v <- c(list1,list2)
```

Lists

Identify elements of a list using the [] convention.

mylist[[2]] # 2nd component of the list

Factors

Tell **R** that a variable is **nominal** by making it a factor. The factor stores the nominal values as a vector of integers in the range [1... k] (where k is the number of unique values in the nominal variable), and an internal vector of character strings (the original values) mapped to these integers.

```
# variable gender with 20 "male" entries and  
# 30 "female" entries  
gender <- c(rep("male",20), rep("female", 30))  
gender <- factor(gender)  
# stores gender as 20 1s and 30 2s and associates  
# 1=female, 2=male internally (alphabetically)
```

Useful Functions

`length(object) # number of elements or components`

`str(object) # structure of an object`

`class(object) # class or type of an object`

`names(object) # names`

`c(object,object,...) # combine objects into a vector`

`cbind(object, object, ...) # combine objects as columns`

`rbind(object, object, ...) # combine objects as rows`

`ls() # list current objects`

`rm(object) # delete an object`

`newobject <- edit(object) # edit copy and save a newobject`

Importing Data

Importing data into **R** is fairly simple.

For Stata and Systat, use the [foreign](#) package.

For SPSS and SAS I would recommend the [Hmisc](#) package for ease and functionality.

See the [**Quick-R**](#) section on [packages](#), for information on obtaining and installing the the packages.

Example of importing data are provided below.

From A Comma Delimited Text File

```
# first row contains variable names, comma is separator  
# assign the variable id to row names  
# note the / instead of \ on mswindows systems
```

```
mydata <- read.table("c:/mydata.csv", header=TRUE, sep=",", row.names="id")
```

From Excel

The best way to read an Excel file is to export it to a comma delimited file and import it using the method above.

On windows systems you can use the **RODBC** package to access Excel files. The first row should contain variable/column names.

```
# first row contains variable names
# we will read in workSheet mysheet
library(RODBC)
channel <- odbcConnectExcel("c:/myexcel.xls")
mydata <- sqlFetch(channel, "mysheet")
odbcClose(channel)
```

From SAS

- ```
save SAS dataset in trasport format
libname out xport 'c:/mydata.xpt';
data out.mydata;
set sasuser.mydata;
run;
```
- ```
library(foreign)
#bsl=read.xport("mydata.xpt")
```

Keyboard Input

Usually you will obtain a dataframe by importing it from **SAS, SPSS, Excel, Stata**, a database, or an ASCII file. To create it interactively, you can do something like the following.

```
# create a dataframe from scratch  
age <- c(25, 30, 56)  
gender <- c("male", "female", "male")  
weight <- c(160, 110, 220)  
mydata <- data.frame(age, gender, weight)
```

Keyboard Input

You can also use **R**'s built in spreadsheet to enter the data interactively, as in the following example.

```
# enter data using editor  
mydata <- data.frame(age=numeric(0), gender=character(0), weight=numeric(0))  
mydata <- edit(mydata)  
# note that without the assignment in the line above,  
# the edits are not saved!
```

Exporting Data

There are numerous methods for exporting **R** objects into other formats . For SPSS, Stata. you will need to load the [foreign](#) packages. For Excel, you will need the [xlsReadWrite](#) package.

Exporting Data

To A Tab Delimited Text File

```
write.table(mydata, "c:/mydata.txt", sep="\t")
```

To an Excel Spreadsheet

```
library(xlsReadWrite)  
write.xls(mydata, "c:/mydata.xls")
```

To SAS

```
library(foreign)  
write.foreign(mydata, "c:/mydata.txt", "c:/mydata.sas", package="SAS")
```

Viewing Data

There are a number of functions for listing the contents of an object or dataset.

```
# list objects in the working environment
```

```
ls()
```

```
# list the variables in mydata
```

```
names(mydata)
```

```
# list the structure of mydata
```

```
str(mydata)
```

```
# list levels of factor v1 in mydata
```

```
levels(mydata$v1)
```

```
# dimensions of an object
```

```
dim(object)
```

Viewing Data

There are a number of functions for listing the contents of an object or dataset.

```
# class of an object (numeric, matrix, dataframe, etc)
  class(object)

# print mydata
  mydata

# print first 10 rows of mydata
  head(mydata, n=10)

# print last 5 rows of mydata
  tail(mydata, n=5)
```

Variable Labels

R's ability to handle variable labels is somewhat unsatisfying.

If you use the Hmisc package, you can take advantage of some labeling features.

```
library(Hmisc)
```

```
label(mydata$myvar) <- "Variable label for variable myvar"
```

```
describe(mydata)
```

Variable Labels

Unfortunately the label is only in effect for functions provided by the **Hmisc** package, **describe()**. Your other option is to use the variable label as the variable name and refer to the variable by position index.

```
names(mydata)[3] <- "This is the label for variable 3"  
mydata[3] # list the variable
```

Value Labels

To understand value labels in **R**, you need to understand the data structure [factor](#).

You can use the factor function to create your own value labels.

```
# variable v1 is coded 1, 2 or 3
```

```
# we want to attach value labels 1=red, 2=blue,3=green
```

```
mydata$v1 <- factor(mydata$v1,  
levels = c(1,2,3),  
labels = c("red", "blue", "green"))
```

```
# variable y is coded 1, 3 or 5
```

```
# we want to attach value labels 1=Low, 3=Medium, 5=High
```

Value Labels

```
mydata$v1 <- ordered(mydata$y,  
                      levels = c(1,3, 5),  
                      labels = c("Low", "Medium", "High"))
```

Use the **factor()** function for **nominal data** and the **ordered()** function for **ordinal data**. Statistical and graphic functions will then treat the data appropriately.

Note: factor and ordered are used the same way, with the same arguments. The former creates factors and the latter creates ordered factors.

Missing Data

In **R**, missing values are represented by the symbol **NA** (not available) . Impossible values (e.g., dividing by zero) are represented by the symbol **NaN** (not a number). Unlike **S** and **S+**, **R** uses the same symbol for character and numeric data.

Testing for Missing Values

```
is.na(x) # returns TRUE if x is missing  
y <- c(1,2,3,NA)  
is.na(y) # returns a vector (F F F T)
```

Missing Data

Recoding Values to Missing

```
# recode 99 to missing for variable v1  
# select rows where v1 is 99 and recode column v1  
mydata[mydata$v1==99,"v1"] <- NA
```

Excluding Missing Values from Analyses

Arithmetic functions on missing values yield missing values.

```
x <- c(1,2,NA,3)  
mean(x)      # returns NA  
mean(x, na.rm=TRUE) # returns 2
```

Missing Data

The function **complete.cases()** returns a logical vector indicating which cases are complete.

```
# list rows of data that have missing values  
mydata[!complete.cases(mydata),]
```

The function **na.omit()** returns the object with listwise deletion of missing values.

```
# create new dataset without missing data  
newdata <- na.omit(mydata)
```

Missing Data

Advanced Handling of Missing Data

Most modeling functions in **R** offer options for dealing with missing values. You can go pairwise or listwise deletion of missing values through methods such as multiple imputation. Good implementations that can be accessed through **R** include [Amelia](#), [Mice](#), and [mitools](#).

Date Values

Dates are represented as the number of days since 1970-01-01, with negative values for earlier dates.

```
# use as.Date( ) to convert strings to dates  
mydates <- as.Date(c("2007-06-22", "2004-02-13"))  
# number of days between 6/22/07 and 2/13/04  
days <- mydates[1] - mydates[2]
```

Sys.Date() returns today's date.

Date() returns the current date and time.

Date Values

The following symbols can be used with the format() function to print dates.

Symbol	Meaning	Example
%d	day as a number (0-31)	01-31
%a %A	abbreviated weekday unabbreviated weekday	Mon Monday
%m	month (00-12)	00-12
%b %B	abbreviated month unabbreviated month	Jan January
%y %Y	2-digit year 4-digit year	07 2007

Date Values

```
# print today's date
today <- Sys.Date()
format(today, format="%B %d %Y")
"June 20 2007"
```

Lecture 3: Data Manipulation

Outline

- Creating New Variable
- Operators
- Built-in functions
- Control Structures
- User Defined Functions
- Sorting Data
- Merging Data
- Aggregating Data
- Reshaping Data
- Sub-setting Data
- Data Type Conversions

Introduction

Once you have access to your data, you will want to massage it into useful form. This includes creating new variables (including recoding and renaming existing variables), sorting and merging datasets, aggregating data, reshaping data, and subsetting datasets (including selecting observations that meet criteria, random sampling observation, and dropping or keeping variables).

Introduction

Each of these activities usually involve the use of R's built-in operators (arithmetic and logical) and functions (numeric, character, and statistical). Additionally, you may need to use control structures (if-then, for, while, switch) in your programs and/or create your own functions. Finally you may need to convert variables or datasets from one type to another (e.g. numeric to character or matrix to dataframe).

Creating new variables

- Use the assignment operator `<-` to create new variables
A wide array of operators and functions are available here.
- # Three examples for doing the same computations

```
mydata$sum <- mydata$x1 + mydata$x2  
mydata$mean <- (mydata$x1 + mydata$x2)/2
```

```
attach(mydata)  
mydata$sum <- x1 + x2  
mydata$mean <- (x1 + x2)/2  
detach(mydata)
```

- `mydata <- transform(mydata,
sum = x1 + x2,
mean = (x1 + x2)/2`

Creating new variables

Recoding variables

- In order to recode data, you will probably use one or more of R's control structures.
- # create 2 age categories
mydata\$agecat <- ifelse(mydata\$age > 70,
c("older"), c("younger"))
another example: create 3 age categories
attach(mydata)
mydata\$agecat[age > 75] <- "Elder"
mydata\$agecat[age > 45 & age <= 75] <- "Middle Aged"
mydata\$agecat[age <= 45] <- "Young"
detach(mydata)

Creating new variables

Recoding variables

- In order to recode data, you will probably use one or more of R's control structures.
- # create 2 age categories
mydata\$agecat <- ifelse(mydata\$age > 70,
c("older"), c("younger"))

```
# another example: create 3 age categories  
attach(mydata)  
mydata$agecat[age > 75] <- "Elder"  
mydata$agecat[age > 45 & age <= 75] <- "Middle Aged"  
mydata$agecat[age <= 45] <- "Young"
```

Creating new variables

Renaming variables

- You can rename variables programmatically or interactively.
- ```
rename interactively
fix(mydata) # results are saved on close
```

```
rename programmatically
library(reshape)
mydata <- rename(mydata, c(oldname="newname"))
```

```
you can re-enter all the variable names in order
changing the ones you need to change. The limitation
```

# Arithmetic Operators

| Operator             | Description                 |
|----------------------|-----------------------------|
| <code>+</code>       | addition                    |
| <code>-</code>       | subtraction                 |
| <code>*</code>       | multiplication              |
| <code>/</code>       | division                    |
| <code>^ or **</code> | exponentiation              |
| <code>x %%</code> y  | modulus (x mod y) 5%%2 is 1 |
| <code>x %/% y</code> | integer division 5%/%2 is 2 |

# Logical Operators

| Operator  | Description              |
|-----------|--------------------------|
| <         | less than                |
| <=        | less than or equal to    |
| >         | greater than             |
| >=        | greater than or equal to |
| ==        | exactly equal to         |
| !=        | not equal to             |
| !x        | Not x                    |
| x   y     | x OR y                   |
| x & y     | x AND y                  |
| isTRUE(x) | test if x is TRUE        |

## Control Structures

- R has the standard control structures you would expect. **expr** can be multiple (compound) statements by enclosing them in braces { }. It is more efficient to use built-in functions rather than control structures whenever possible.

# Control Structures

- **if-else**
- `if (cond) expr`  
`if (cond) expr1 else expr2`
- **for**
- `for (var in seq) expr`
- **while**
- `while (cond) expr`
- **switch**
- `switch(expr, ...)`
- **ifelse**
- `ifelse(test, yes, no)`

## Control Structures

- # transpose of a matrix  
# a poor alternative to built-in t() function

```
mytrans <- function(x) {
 if (!is.matrix(x)) {
 warning("argument is not a matrix: returning NA")
 return(NA_real_)
 }
 y <- matrix(1, nrow=ncol(x), ncol=nrow(x))
 for (i in 1:nrow(x)) {
 for (j in 1:ncol(x)) {
 y[j,i] <- x[i,j]
 }
 }
}
```

## Control Structures

- # try it  
z <- matrix(1:10, nrow=5, ncol=2)  
tz <- mytrans(z)

# R built-in functions

Almost everything in R is done through functions. Here I'm only referring to numeric and character functions that are commonly used in creating or recoding variables.

Note that while the examples on this page apply functions to individual variables, many can be applied to vectors and matrices as well

# Numeric Functions

| Function                                                             | Description                                       |
|----------------------------------------------------------------------|---------------------------------------------------|
| <b>abs(<math>x</math>)</b>                                           | absolute value                                    |
| <b>sqrt(<math>x</math>)</b>                                          | square root                                       |
| <b>ceiling(<math>x</math>)</b>                                       | ceiling(3.475) is 4                               |
| <b>floor(<math>x</math>)</b>                                         | floor(3.475) is 3                                 |
| <b>trunc(<math>x</math>)</b>                                         | trunc(5.99) is 5                                  |
| <b>round(<math>x</math>, digits=<math>n</math>)</b>                  | round(3.475, digits=2) is 3.48                    |
| <b>signif(<math>x</math>, digits=<math>n</math>)</b>                 | signif(3.475, digits=2) is 3.5                    |
| <b>cos(<math>x</math>), sin(<math>x</math>), tan(<math>x</math>)</b> | also acos( $x$ ), cosh( $x$ ), acosh( $x$ ), etc. |
| <b>log(<math>x</math>)</b>                                           | natural logarithm                                 |
| <b>log10(<math>x</math>)</b>                                         | common logarithm                                  |
| <b>exp(<math>x</math>)</b>                                           | $e^x$                                             |

# Character Functions

| Function                                                            | Description                                                                                                                                                                                                                                    |
|---------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>substr(x, start=n1, stop=n2)</b>                                 | Extract or replace substrings in a character vector.<br>x <- "abcdef"<br>substr(x, 2, 4) is "bcd"<br>substr(x, 2, 4) <- "22222" is "a222ef"                                                                                                    |
| <b>grep(pattern, x, ignore.case=FALSE, fixed=FALSE)</b>             | Search for <i>pattern</i> in <i>x</i> . If fixed =FALSE then <i>pattern</i> is a <u>regular expression</u> . If fixed=TRUE then <i>pattern</i> is a text string. Returns matching indices.<br>grep("A", c("b","A","c"), fixed=TRUE) returns 2  |
| <b>sub(pattern, replacement, x, ignore.case=FALSE, fixed=FALSE)</b> | Find <i>pattern</i> in <i>x</i> and replace with <i>replacement</i> text. If fixed=FALSE then <i>pattern</i> is a regular expression. If fixed = T then <i>pattern</i> is a text string.<br>sub("\s",".", "Hello There") returns "Hello.There" |
| <b>strsplit(x, split)</b>                                           | Split the elements of character vector <i>x</i> at <i>split</i> .<br>strsplit("abc", "") returns 3 element vector "a", "b", "c"                                                                                                                |
| <b>paste(..., sep="")</b>                                           | Concatenate strings after using <i>sep</i> string to seperate them.<br>paste("x",1:3,sep="") returns c("x1","x2" "x3")<br>paste("x",1:3,sep="M") returns c("xM1","xM2" "xM3")<br>paste("Today is", date())                                     |
| <b>toupper(x)</b>                                                   | Uppercase                                                                                                                                                                                                                                      |
| <b>tolower(x)</b>                                                   | Lowercase                                                                                                                                                                                                                                      |

and Graphics

# Stat/Prob Functions

- The following table describes functions related to probability distributions. For random number generators below, you can use `set.seed(1234)` or some other integer to create reproducible pseudo-random numbers.

| Function                                                                                                                         | Description                                                                                                                                                                                                                                                     |
|----------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>dnorm(x)</b>                                                                                                                  | normal density function (by default m=0 sd=1)<br># plot standard normal curve<br>x <- pretty(c(-3,3), 30)<br>y <- dnorm(x)<br>plot(x, y, type="l", xlab="Normal Deviate", ylab="Density", yaxs="i")                                                             |
| <b>pnorm(q)</b>                                                                                                                  | cumulative normal probability for q<br>(area under the normal curve to the right of q)<br>pnorm(1.96) is 0.975                                                                                                                                                  |
| <b>qnorm(p)</b>                                                                                                                  | normal quantile.<br>value at the p percentile of normal distribution<br>qnorm(.9) is 1.28 # 90th percentile                                                                                                                                                     |
| <b>rnorm(n, m=0,sd=1)</b>                                                                                                        | n random normal deviates with mean m<br>and standard deviation sd.<br>#50 random normal variates with mean=50, sd=10<br>x <- rnorm(50, m=50, sd=10)                                                                                                             |
| <b>dbinom(x, size, prob)</b><br><b>pbinom(q, size, prob)</b><br><b>qbinom(p, size, prob)</b><br><b>rbinom(n, size, prob)</b>     | binomial distribution where size is the sample size<br>and prob is the probability of a heads (pi)<br># prob of 0 to 5 heads of fair coin out of 10 flips<br>dbinom(0:5, 10, .5)<br># prob of 5 or less heads of fair coin out of 10 flips<br>pbinom(5, 10, .5) |
| <b>dpois(x, lamda)</b><br><b>ppois(q, lamda)</b><br><b>qpois(p, lamda)</b><br><b>rpois(n, lamda)</b>                             | poisson distribution with m=std=lamda<br>#probability of 0,1, or 2 events with lamda=4<br>dpois(0:2, 4)<br># probability of at least 3 events with lamda=4<br>1- ppois(2,4)                                                                                     |
| <b>dunif(x, min=0, max=1)</b><br><b>punif(q, min=0, max=1)</b><br><b>qunif(p, min=0, max=1)</b><br><b>runif(n, min=0, max=1)</b> | uniform distribution, follows the same pattern<br>as the normal distribution above.<br>#10 uniform random variates<br>x <- runif(10)                                                                                                                            |

## and Graphics

|                                                                                                                                                                                | <b>Description</b>                                                                                                                                                                                          |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>mean</b><br>( <b>trim</b> =0,<br>FALSE)                                                                                                                                     | mean of object x<br># trimmed mean, removing any missing values and<br># 5 percent of highest and lowest scores<br>mx <- mean(x,trim=.05,na.rm=TRUE)                                                        |
|                                                                                                                                                                                | standard deviation of object(x). also look at var(x) for variance and mad(x) for median absolute deviation.                                                                                                 |
| <b>median</b><br>( <i>x</i> ,<br><i>probs</i> )                                                                                                                                | median<br>quantiles where x is the numeric vector whose quantiles are desired and probs is a numeric vector with probabilities in [0,1].<br># 30th and 84th percentiles of x<br>y <- quantile(x, c(.3,.84)) |
| <b>range</b><br><b>sum</b><br><b>lagged differences</b> , with lag indicating which lag to use<br><b>minimum</b><br><b>maximum</b><br><b>center</b> =TRUE, <b>scale</b> =TRUE) | range<br>sum<br>lagged differences, with lag indicating which lag to use<br>minimum<br>maximum<br>column center or standardize a matrix.                                                                    |

# Other Useful Functions

| Function                                           | Description                                                                      |
|----------------------------------------------------|----------------------------------------------------------------------------------|
| <b>seq</b> ( <i>from</i> , <i>to</i> , <i>by</i> ) | generate a sequence<br>indices <- seq(1,10,2)<br>#indices is c(1, 3, 5, 7, 9)    |
| <b>rep</b> ( <i>x</i> , <i>ntimes</i> )            | repeat <i>x</i> <i>n</i> times<br>y <- rep(1:3, 2)<br># y is c(1, 2, 3, 1, 2, 3) |
| <b>cut</b> ( <i>x</i> , <i>n</i> )                 | divide continuous variable in factor with <i>n</i> levels<br>y <- cut(x, 5)      |

# Sorting

- To sort a dataframe in R, use the **order( )** function. By default, sorting is ASCENDING. Prepend the sorting variable by a minus sign to indicate DESCENDING order. Here are some examples.
- # sorting examples using the mtcars dataset  
data(mtcars)  
# sort by mpg  
newdata = mtcars[order(mtcars\$mpg),]  
# sort by mpg and cyl  
newdata <- mtcars[order(mtcars\$mpg, mtcars\$cyl),]  
#sort by mpg (ascending) and cyl (descending)  
newdata <- mtcars[order(mtcars\$mpg, -mtcars\$cyl),]

# Merging

To merge two dataframes (datasets) horizontally, use the **merge** function. In most cases, you join two dataframes by one or more common key variables (i.e., an inner join).

```
merge two dataframes by ID
total <- merge(dataframeA,dataframeB,by="ID")

merge two dataframes by ID and Country
total <-
merge(dataframeA,dataframeB,by=c("ID","Country"))
```

# Merging

## ADDING ROWS

To join two dataframes (datasets) vertically, use the **rbind** function. The two dataframes **must** have the same variables, but they do not have to be in the same order.

```
total <- rbind(dataframeA, dataframeB)
```

If `dataframeA` has variables that `dataframeB` does not, then either:

Delete the extra variables in `dataframeA` or

Create the additional variables in `dataframeB` and set them to NA (missing)

before joining them with `rbind`.

# Aggregating

- **It is relatively easy to collapse data in R using one or more BY variables and a defined function.**
- # aggregate dataframe mtcars by cyl and vs, returning means  
# for numeric variables  
attach(mtcars)  
aggdata <-aggregate(mtcars, by=list(cyl),  
FUN=mean, na.rm=TRUE)  
print(aggdata)
- OR use apply

# Aggregating

- When using the `aggregate()` function, the `by` variables must be in a list (even if there is only one). The function can be built-in or user provided.
- See also:
- `summarize()` in the [Hmisc package](#)
- `summaryBy()` in the [doBy package](#)

## Data Type Conversion

- Type conversions in R work as you would expect. For example, adding a character string to a numeric vector converts all the elements in the vector to character.
- Use `is.foo` to test for data type *foo*. Returns TRUE or FALSE  
Use `as.foo` to explicitly convert it.
- `is.numeric()`, `is.character()`, `is.vector()`, `is.matrix()`, `is.data.frame()`  
`as.numeric()`, `as.character()`, `as.vector()`, `as.matrix()`, `as.data.frame()`