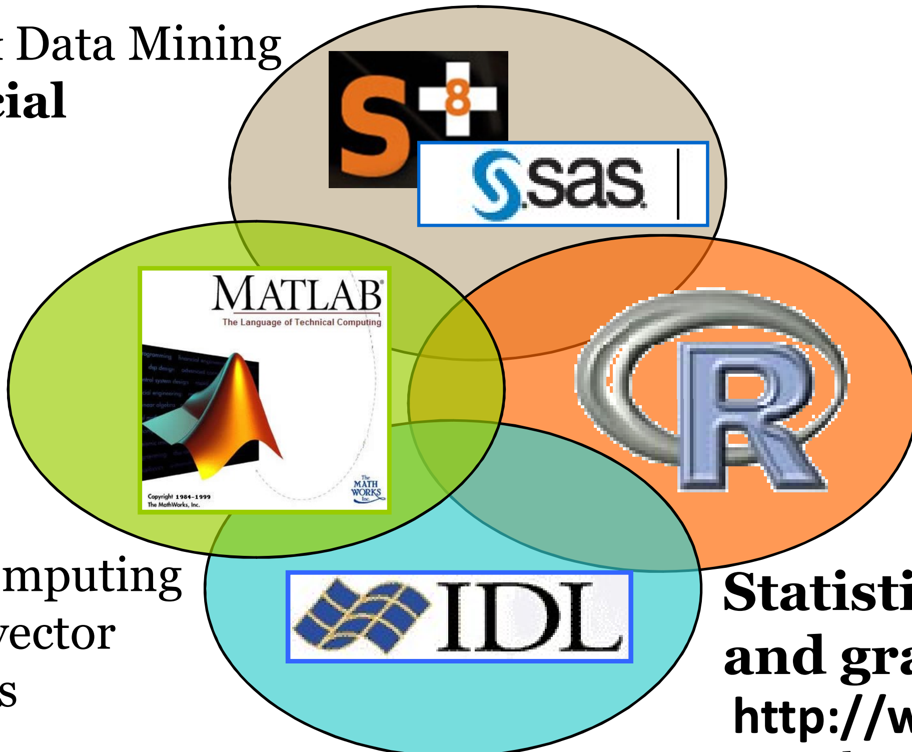
Introduction to R

Dr. S. DEVAARUL
Asst. Professor,
Department of Statistics,
Government Arts college, Coimbatore.

Why R?

- Statistics & Data Mining
- **Commercial**



- Technical computing
- Matrix and vector formulations

- Data Visualization and analysis platform
- Image processing, vector computing

Statistical computing and graphics

http://www.r-project.org

- Developed by **R**. Gentleman & **R**. Ihaka
- Expanded by community as **open source**
- Statistically rich

Features of R

R is an integrated suite of software for data manipulation, calculation, and graphical display

- Effective data handling
- Various operators for calculations on arrays/matrices
- Graphical facilities for data analysis
- Well-developed language including conditionals, loops, recursive functions and I/O capabilities.

Basic usage: arithmetic in R

- You can use R as a calculator
- Typed expressions will be evaluated and printed out
  - Main operations: +, -, *, /, ^
  - Obeys order of operations
  - Use parentheses to group expressions
- More complex operations appear as *functions*
  - sqrt(2)
  - sin(pi/4), cos(pi/4), tan(pi/4), asin(1), acos(1), atan(1)
  - exp(1), log(2), log10(10)

Getting help

- *help(function_name)*
  - *help(prcomp)*
- *?function_name*
  - *?prcomp*
- *help.search("topic")*
  - *??topic* or *??"topic"*
- Search CRAN
  - http://www.r-project.org
- From R GUI: Help → Search help…
- CRAN Task Views (for individual packages)
  - http://cran.cnr.berkeley.edu/web/views/

Variables and assignment

- Use variables to store values
- Three ways to assign variables
    - a = 6
    - a <- 6
    - 6 -> a
- Update variables by using the current value in an assignment
    - x = x + 1
- Naming rules
    - Can include letters, numbers, ., and _
    - Names are case sensitive
    - Must start with . or a letter

R Commands

- Commands can be *expressions* or *assignments*
    - Separate by semicolon or new line
- Can split across multiple lines
    - R will change prompt to + if command not finished
- Useful commands for variables
    - *ls()*: List all stored variables
    - *rm(x)*: Delete one or more variables
    - *class(x)*: Describe what type of data a variable stores
    - *save(x,file="filename")*: Store variable(s) to a binary file
    - *load("filename")*: Load all variables from a binary file
        - Save/load in current directory or My Documents by default

Vectors and vector operations

To create a vector:

```
# c() command to create vector x
x=c(12,32,54,33,21,65)
# c() to add elements to vector x
x=c(x,55,32)
```

```
# seq() command to create sequence of number
years=seq(1990,2003)
# to contain in steps of .5
a=seq(3,5,.5)
# can use : to step by 1
years=1990:2003;
```

```
# rep() command to create data that follow a regular pattern
b=rep(1,5)
c=rep(1:2,4)
```

To access vector elements:

```
# 2nd element of x
x[2]
# first five elements of x
x[1:5]
# all but the 3rd element of x
x[-3]
# values of x that are < 40
x[x<40]
# values of y such that x is < 40
y[x<40]
```

To perform operations:

```
# mathematical operations on vectors
y=c(3,2,4,3,7,6,1,1)
x+y; 2*y; x*y; x/y; y^2
```

Matrices & matrix operations

To create a matrix:

```
# matrix() command to create matrix A with rows and cols
A=matrix(c(54,49,49,41,26,43,49,50,58,71),nrow=5,ncol=2))
B=matrix(1,nrow=4,ncol=4)
```

To access matrix elements:

```
# matrix_name[row_no, col_no]
A[2,1] # 2nd row, 1st column element
A[3,]  # 3rd row
A[,2]  # 2nd column of the matrix
A[2:4,c(3,1)]  # submatrix of 2nd-4th elements of the 3rd and 1st columns
A["KC",] # access row by name, "KC"
```

Statistical operations:

```
rowSums(A)
colSums(A)
rowMeans(A)
colMeans(A)
# max of each columns
apply(A,2,max)
# min of each row
apply(A,1,min)
```

Element by element ops:

```
2*A+3; A+B; A*B; A/B;
```

Matrix/vector multiplication:

```
A %*% B;
```

Useful functions for vectors and matrices

- Find # of elements or dimensions
    - *length(v), length(A), dim(A)*
- Transpose
    - *t(v), t(A)*
- Matrix inverse
    - *solve(A)*
- Sort vector values
    - *sort(v)*
- Statistics
    - *min(), max(), mean(), median(), sum(), sd(), quantile()*
    - Treat matrices as a single vector (same with *sort()*)

Graphical display and plotting

- Most common plotting function is *plot()*
    - *plot(x,y)* plots *y* vs *x*
    - *plot(x)* plots *x* vs *1:length(x)*
- *plot()* has many options for labels, colors, symbol, size, etc.
    - Check help with *?plot*
- Use *points()*, *lines()*, or *text()* to add to an existing plot
- Use *x11()* to start a new output window
- Save plots with *png()*, *jpeg()*, *tiff()*, or *bmp()*

R Packages

- R functions and datasets are organized into packages
  - Packages *base* and *stats* include many of the built-in functions in R
  - CRAN provides thousands of packages contributed by R users
- Package contents are only available when loaded
  - Load a package with *library(pkgname)*
- Packages must be installed before they can be loaded
  - Use *library()* to see installed packages
  - Use *install.packages(pkgname)* and *update.packages(pkgname)* to install or update a package
  - Can also run *R CMD INSTALL pkgname.tar.gz* from command line if you have downloaded package source

Exploring the *iris* data

- **Load iris data into your R session:**
  - data (iris);
  - help (data);
- **Check that iris was indeed loaded:**
  - ls ();
- **Check the class that the iris object belongs to:**
  - class (iris);
- **Print the content of iris data:**
  - iris;
- **Check the dimensions of the iris data:**
  - dim (iris);
- **Check the names of the columns:**
  - names (iris);

Exploring the *iris* data (cont.)

- **Plot Petal.Length vs. Petal.Width:**
  - plot (iris[, 3], iris[, 4]);
  - example(plot)
- **Exercise: create a plot <u>similar</u> to this figure:**

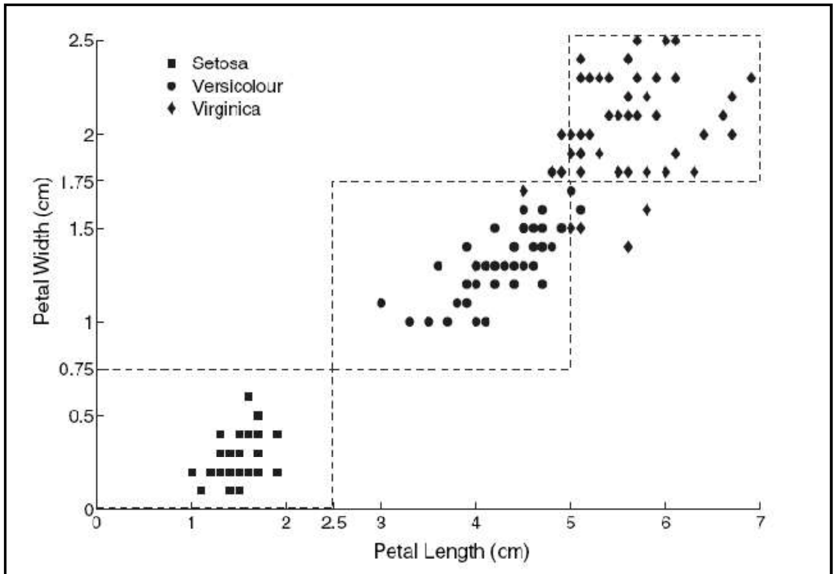


Figure 1.4. Petal width versus petal length for 150 Iris flowers.

Src: Figure is from *Introduction to Data Mining* by Pang-Ning Tan, Michael Steinbach, and Vipin Kumar

Reading data from files

- Large data sets are better loaded through the file input interface in R
- Reading a table of data can be done using the *read.table()* command:
    - *a <- read.table(“a.txt”)*
- The values are read into R as an object of type data frame (a sort of matrix in which different columns can have different types). Various options can specify reading or discarding of headers and other metadata.
- A more primitive but universal file-reading function exists, called *scan()*
    - b = scan(“input.dat”);
    - *scan()* returns a vector of the data read

Programming in R

- The following slides assume a basic understanding of programming concepts

- For more information, please see chapters 9 and 10 of the R manual:

http://cran.r-project.org/doc/manuals/R-intro.html

Additional resources

- *Beginning R: An Introduction to Statistical Programming* by Larry Pace
- Introduction to R webpage on APSnet:

http://www.apsnet.org/edcenter/advanced/topics/ecologyandepidemiologyinr/introductiontor/Pages/default.aspx

- The R Inferno:

http://www.burns-stat.com/pages/Tutor/R_inferno.pdf

Conditional statements

- Perform different commands in different situations
- *if (condition) command_if_true*
    - Can add *else command_if_false* to end
    - Group multiple commands together with braces {}
    - *if (cond1) {cmd1; cmd2;} else if (cond2) {cmd3; cmd4;}*
- Conditions use relational operators
    - ==, !=, <, >, <=, >=
    - Do not confuse = (assignment) with == (equality)
        - = is a <u>command</u>, == is a <u>question</u>
- Combine conditions with *and* (&&) and *or* (||)
    - Use & and | for vectors of length > 1 (element-wise)

Loops

- Most common type of loop is the *for* loop
  - *for (x in v) { loop_commands; }*
  - *v* is a vector, commands repeat for each value in *v*
  - Variable *x* becomes each value in *v*, in order
  - **Example:** adding the numbers 1-10
    - *total = 0; for (x in 1:10) total = total + x;*
- Other type of loop is the *while* loop
  - *while (condition) { loop_commands; }*
  - Condition is identical to *if* statement
  - Commands are repeated until condition is false
    - Might execute commands 0 times if already false
- *while* loops are useful when you don't know number of iterations

Scripting in R

- A script is a sequence of R commands that perform some common task
    - E.g., defining a specific function, performing some analysis routine, etc.
- Save R commands in a plain text file
    - Usually have extension of .R
- Run scripts with *source()* :
    - *source("filename.R")*
- To save command output to a file, use *sink()*:
    - *sink("output.Rout")*
    - *sink()* restores output to console
    - Can be used with or outside of a script

Lists

- Objects containing an ordered collection of objects
- Components do not have to be of same type
- Use *list()* to create a list:
  - *a <- list(“hello”,c(4,2,1),“class”);*
- Components can be named:
  - *a <- list(string1=“hello”,num=c(4,2,1),string2=“class”)*
- Use [[position#]] or $name to access list elements
  - E.g., a[[2]] and a$num are equivalent
- Running the *length()* command on a list gives the number of higher-level objects

Writing your own functions

- Writing functions in R is defined by an assignment like:
  - *a <- function(arg1,arg2) { function_commands; }*
- Functions are R objects of type "function"
- Functions can be written in C/FORTRAN and called via *.C()* or *.Fortran()*
- Arguments may have default values
  - Example: *my.pow <- function(base, pow = 2) {return base^pow;}*
  - Arguments with default values become optional, should usually appear at end of argument list (though not required)
- Arguments are untyped
  - Allows multipurpose functions that depend on argument type
  - Use *class()*, *is.numeric()*, *is.matrix()*, etc. to determine type

How do I get started with R (Linux)?

- **Step 1:** Download R
  - mkdir for RHOME; cd $RHOME
  - wget http://cran.cnr.berkeley.edu/src/base/R-2/R-2.9.1.tar.gz
- **Step 2:** Install R
  - tar –zxvf R-2.9.1.tar.g
  - ./configure *--prefix=<RHOME> --enable-R-shlib*
  - make
  - make install
- **Step 3:** Run R
  - Update env. variables in $HOME/.bash_profile:
    - *export PATH=<RHOME>/bin:$PATH*
    - *export R_HOME=<RHOME>*
  - R

Useful R links

- **R Home: http://www.r-project.org/**
- **R's CRAN package distribution: http://cran.cnr.berkeley.edu/**
- **Introduction to R manual: http://cran.cnr.berkeley.edu/doc/manuals/R-intro.pdf**
- **Writing R extensions: http://cran.cnr.berkeley.edu/doc/manuals/R-exts.pdf**
- **Other R documentation: http://cran.cnr.berkeley.edu/manuals.html**

Lecture 1: R Basics

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
> T T T T F F F F T T
> x[c(T,T,T,T,F,F,F,F,T,T)]
> 1 2 3 4 9 10
```

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 ls 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" "x2"
```

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)
[,1] [,2]
[1,] -0.1428571 0.5714286
[2,] 0.2857143 -0.1428571
```

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
    x[2:4,1:3] # rows 2,3,4 of columns 1,2,3
```

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 scaler
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)
# R now treats gender as a nominal variable
summary(gender)
```

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
fix(object)               # edit in place
```

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 these 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:/myexel.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, SAS and 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, such as **describe()**. Your other option is to use the variable label as the variable name and then 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 lables.

```
# 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**. **R** 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 later 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 SAS, **R** uses the same symbol for character and numeric data.

Testing for Missing Values

is.na(x) # returns TRUE of 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 beyond pairwise of listwise deletion of missing values through methods such as multiple imputation. Good implementations that can be accessed through **R** include **Amelia II**, **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	abbreviated weekday	Mon
%A	unabbreviated weekday	Monday
%m	month (00-12)	00-12
%b	abbreviated month	Jan
%B	unabbreviated month	January
%y	2-digit year	07
%Y	4-digit year	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, randomly 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"
  detach(mydata)

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
# is that you need to enter all of them!
names(mydata) <- c("x1","age","y", "ses")
```

Arithmetic Operators

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation
x %% y	modulus (x mod y) 5%%2 is 1
x %/% y	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]
    }
   }
  return(y)
  }
  ```
  ```

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

Character Functions

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

o

Computing and

Stat/Prob Functions

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

Function	Description

dnorm(*x*)	normal density function (by default m=0 sd=1)
	# plot standard normal curve
	x <- pretty(c(-3,3), 30)
	y <- dnorm(x)
	plot(x, y, type='l', xlab="Normal Deviate", ylab="Density", yaxs="i")

pnorm(*q*)	cumulative normal probability for q (area under the normal curve to the right of q) pnorm(1.96) is 0.975
qnorm(*p*)	normal quantile. value at the p percentile of normal distribution qnorm(.9) is 1.28 # 90th percentile
rnorm(*n*, m=0**,sd=**1**)**	n random normal deviates with mean m and standard deviation sd. #50 random normal variates with mean=50, sd=10 x <- rnorm(50, m=50, sd=10)

dbinom(*x*, *size*, *prob*)
pbinom(*q*, *size*, *prob*)
qbinom(*p*, *size*, *prob*)
rbinom(*n*, *size*, *prob*)
	binomial distribution where size is the sample size
	and prob is the probability of a heads (pi)
	# prob of 0 to 5 heads of fair coin out of 10 flips
	dbinom(0:5, 10, .5)
	# prob of 5 or less heads of fair coin out of 10 flips
	pbinom(5, 10, .5)

dpois(*x*, *lamda*)
ppois(*q*, *lamda*)
qpois(*p*, *lamda*)
rpois(*n*, *lamda*)
	poisson distribution with m=std=lamda
	#probability of 0,1, or 2 events with lamda=4
	dpois(0:2, 4)
	# probability of at least 3 events with lamda=4
	1- ppois(2,4)

dunif(*x*, min=0**, max=**1**)** **punif(*q*, min=0, max=**1**)** **qunif(*p*, min=**0**, max=**1**)** **runif(*n*, min=**0**, max=**1**)**	uniform distribution, follows the same pattern as the normal distribution above. #10 uniform random variates x <- runif(10)

Computing and

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

Other Useful Functions

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