*Statistics and Data Analysis Notes

October 20, 2013

Contents

- Peng is researcher on air pollutation and health effects.

1 Week One Lectures

1.1 What Makes R Different?

- hybrid of command line and programming interface - best and worst of both worlds.

1.2 How to get help

- People may not know you or what you mean

- Use resources:

– Search Forum – Search Web – Read the Manual – Read FAQ – Inpection and experimentation – Talk to a Friend – Read the source code

- Ask Questions

– let people know you looked at above – reproducable example (test data) – expected output to be (maybe you are wrong) and what comes out needs to be solved – version information and package information – Type of OS you are using

- Send email to forum

– Subject lines should be useful – Describe big picture and goal – Specific problem (minimum amount of information)

- Do not:

– Don't claim you found a bug – No groveling (lazy didn't do above) – No multiple mailing lists at once – Don't ask others to debug code

1.3 Background and Overview

1.3.1 History of R

- R is dialect of S
- S developed at Bell Labs (1976)
- Version 4 1998
- R is implimentation of S
- 1991 R created by Ross Ihaka and Robert Gentleman
- Announced in 1993 to public
- Martin Machler convinved Ross and Robert to release R under GNU GPL
- 2000 R 1.0.0 released
- 2012 R 2.15.1 released

1.3.2 Features of R

- will run on almost any OS (including PlayStation 3)
- functionality is divided into packages
- strong graphics
- GPL (4 freedoms)

1.3.3 Drawbacks

- 40 year old technology is platform
- little support for 3d graphics (but have improved)
- no corporate help line or contact for feature (you have to build it)
- objects stored in physical memory (some advancements made)
- not ideal for all situations

1.3.4 Design of system

- Base download

– Base system (base, utils etc) – Recommend packages (boot, nlme etc.)

- 4,000 User contributed packages on CRAN (but must meet certain level of quality)
- http://bioconductor.org project (genomic and biological data analysis)
- Others

1.3.5 Documentation

- An introduction to R
- Writing R extensions
- R Data Import/Export
- R installation and Admistration
- R Internals

1.4 Data Types

Everything in R is an object:

1.4.1 five basic 'atomic' classes of objects

- character
- numeric
- integer
- complex numbers
- logical

1.4.2 vector

- contain only objects of same class
- but a list can have different classes
- empty vectors created with vector() function with args: class and length

1.4.3 numbers

- generally as double precision
- can explicitly define an integer with L after number
- Inf is infinity can be plus or minus
- NaN is undefined (not a number)

1.4.4 Attributes

- can be part of an object in R:
- names, dimnames
- dimensions
- class
- length (length of vector etc.)

- other user-defined attributed/metadata
- General function attributes() can set or modify attributes

1.4.5 Expressions

```
 <- assignment operator
# Error: needs things on both sides
print(x)
[1] "1"
> msg <- "hello"
# hash is comment character
```

-Evaluation by R engine may or may not show anything. – assignment does not show anything – but putting a variable in the engine will autoprint – same as calling print function – with print the double square brackets shows what element of vector is being shown

```
x <- 1:20 the : creates a sequence 1 to 20
```

- c() concatnates objects to create vectors
- vector can initialize vectors:

```
x <- vector("numeric", length=10) # initializes vector with 0's
```

- concatnating will coerce classes:

```
y <- c(1.7, "a") ## character is least common denominator
```

– logical concatnated to numeric is coerced to numeric – logical and character coerced to character – coercsion will happen behind the scenes – you can coerce explicitly (as. character, as.numeric etc) – nonsensical coercion will result in NA's

1.4.6 Matrices

- special type of vectors
- has dimension attribute
- created by matrix function
- by default matrix is filled by columnwise (can be switched)
- can create matrix by assigning dim attribute eg. dim(x) $<=$ c(5,2)
- column binding (cbind) and row binding (rbind) can create matrix

1.4.7 List

- like vector but each object can be different
- used to carry around data in functions

1.4.8 Factors

- used to represent catagorical data
- treated special by models
- can be ordered (hierarchical) and unordered
- factors are better than integers that represent things (self explaining)
- levels is a special attribute of factors
- unclass function strips out levels - create an integer with key as to what they mean
- create an ordering using the levels= argument in the factor function

1.4.9 missing values

- NA or NaN. NA is missing, NaN is not a number
- NaN is also NA, but NA is not NaN.

1.4.10 Data frame

- stores tabular data
- special type of list, with each having the same length
- columns can be different types
- has attributes row.names

-data.matrix() can convert data.frame to matrix (with coercion)

1.4.11 Names

- can assign names to elements (self describing data!)
- matrix can have names (set with dimnames)

1.5 Sub-setting Data

1.5.1 vector operators

- single square bracket - always returns same type of object. Can be used to get more than one object

```
x[x>4], x[1:4], c[c=="fruit"] #etc
```

– NEAT can use lexical orders

```
x[x>"a"] # returns "b","c", "d", ...
```

– logical indexing

```
u <- x > "a" ; x [u]
```

- double square bracket - list of data frame. Object returned may not be same class as parent object.
- dollar sign - list or data frame that have name.

1.5.2 matrix

- matrices can be subsetted with i,j notation. leave i or j to get row or columns.
- subset single number or row of matrix you get a vector (not a 1 by 1 matrix) turn off default with drop=FALSE

1.5.3 list

- single square bracket -

```
x <- list(foo=1:4, bar=0.6)
x[1] #returns list can also use ["name"]
```

- double bracket reutrns sequence 1,2,3,4
- can not use double bracket for multiple objects
- can use double bracket with computed index:

```
name <- "foo"
x[[[[name]]]]  #returns "x[[[["foo"]]]"
x[[[[c(1,3)]]]] #extracts 3rd item of 1st object
```

- partial matching

```
x <- list(aardvark = 1:5) ; x$a matches x$aardvark but double
```

bracket does not partial match unless exact=FALSE argument is made

1.5.4 removing NA's

```
x[!is.na(x)] subsets non NA values
  good <- complete.cases(x,y) #gives non-missing across both vectors;
  x[good]; y[good]
  good <- complete.cases(airquality); airquality[good,][1:6]
```

1.6 Vecotorized Operations

- Feature in many languages - avoids looping
- two vectors added together (x and y)

– each element is added together when x and y are same length

- greater than less than operators return logical vector

-other arithemetic operations are vector operations

- Can work with matrix multiplication without the % sign around it it is element wise operation
- makes code easier to write

1.7 Reading and writing data in R

- read.table, read.csv
- readLines for reading lines of text file
- source R code files
- dget for reading R code files
- load for reading saved workspaces
- unserialize for reading single R objects in binary form

1.7.1 write data

- write.table
- writeLines
- dump
- dput
- save
- serialize

1.7.2 more on read.table

- file: connection (string)
- header: is first line a header
- sep: how columns are separated
- colClasses - a vector of column classes
- nrows - number of rows in dataset
- comment.char - character string indicating the comment character
- skip = number of rows to skip
- stringsAsFactors = should character variables be coded as a factor?
- read.table has some intuitive defaults - telling up front makes R more efficient
- read.csv is for csv files (defualt separator is command and header=TRUE)

1.7.3 large data

- read help page
- estimate memory size (if not enough RAM your done)
- set comment = “” if no comment lines

```
initial <- read.table("datatable.txt", nrows=100)
classes <- sapply(initial, class)
tabAll <- read.table("datatable.txt", colClasses = classes)
```

- Set nrows (if known) helps with memory usage - can overestimate

1.7.4 R with large datasets

- how much memory does my computer have?
- what other applications are in use?
- multiuser system?
- OS 32 or 64 bit?
- rough calc of memory = 1.5 million by 120 columns all numeric:

– 1.5 million x 120 x 8 bytes/numeric = 1.34 GB of physical memory – going to need a little more for overhead (twice as much needed for read.table)

<s and TAB will create code block!

1.7.5 dumping and dputting

- text format, but contain metadata (type of data in each object). Potentially recoverable.
- read data source and dget.
- Editable format. Can recover if corrupted. Longer lived.
- Version control is workable with textual data (track changes)
- Unix philosophy - store the data as text
- BIG

```
y <- data.frame(a=1, b="a))
dput(y)

structure(list(a=1, b=structure(1L, .Label ="a",
class -"factor")), ..

dput(y, file="y.R)

# puts file y.R with y in it.

dump(c("x","y"), file="data.R")

source("data.R") # reconstructs objects
```

1.7.6 Connections

- *file* opens a connection (to the file of course!)
- *gzfile* opens gzip file connection
- *bzfile* opens connection to compressed bzip2
- *url* opens a connection to a webpage

file arguments *open* = 'r' is read only; 'w' is writing; 'a' is appending; 'rb' 'wb' 'ab' is reading writing or appending binary mode (Windows).

- Gzip and file Connections

```
con <- file("foo.txt", "r")
data <- read.csv(con)
close(con)

# smae as

data <- read.csv(foo.txt")

con <- gzfile("words.gz")
x <- readLines(con,10)
x
close(con)
```

- HTML FILE Connections

```
con <- url("http//www.jhsph.edu", "r")
x <- readLines(con)
head(x)
close(con)
```

1.7.7 Edit Code and Setting up Working Directory

```
getwd()
read.csv("mydata.csv") # can return error if file is not there.

setwd() # will change directory
```

When you read or write data they will be sent to the working directory. Peng suggests creating one directory.

- Editing script
  - can use R's script editor
  - move between editor and R using Cntrl-A Cntrl -C Cntrl-V (select all - copy-paste)
  - save file in coursera folder you should create then use

  ```
  source("mycode.R")
  ```

  assuming you've got the right directory. He did not go into hotkeys (to send code to R) or any IDE's such as R Studio or Emacs (insert boo's here)

1.7.8 Lecture on Macs (I will never afford one I suppose!)

1.7.9 Structure of R Object

```
str()
```

Is the most important function in R according to Peng. Diagnostic function and an alternative to summary.

- useful for large lists
- 1 line of output per object

```
str(str)
#function (object, ...)

str(lm) # etc.

x <- rnorm(100,2,4)
summary(x) # five num summary
str(x) # gives class, 100 elements, first 5 numbers

f <- gl(40,10)

str(f) # returns information of factor levels and first couple
```

```
of numbers.

summary(f) # counts on each levels

data(airquality)
head(airquality)
str(airquality) # tells us its a data.frame, first observations on
each of the vectors (and their classes).

m <- matrix(rnorm(100),10,10)
str(m) will tell us its a matrix (10 by 10) and first couple of obs.

s <- split(airquality, airquality$Month)

str(s) # givens the list of data frames each data.frame str() applied
to it. (nice!)
```

Yep its definitely useful!

2 Week Two Lectures

2.1 Control Structures

typically used in functions

- if, else
- for
- while
- repeat
- break
- next
- return

2.1.1 if, else and if else

```
rm(list=ls())
x <- 2
```

```
if ( x > 3 ){
    y <- 3
} else {
    y <- 0
}
print(y)

# or can value to assign to y
x <- 10
y <- if ( x >3 ){
    10
} else {
    0
}
print(y)

# if else
x <-3
y <- if (x<3){
10
} else if (x>3){
11
} else {
12
}
print(y)
```

```
 [1] 0
 [1] 10
 [1] 12
```

2.1.2 for loops

```
for ( i in 1:10){
    print(i)
}
```

```
[1] 1
[1] 2
```

```
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Loop index i, cycles through. Don't overwright.

Different ways of using for loop (all do the same thing):

```
x <- c("a", "b","c","d")

for(i in 1:4){
print(x[i])
}

for (i in seq_along(x)) {
print(x[i])
}
# don't need numbers can use letters
for(letter in x){
print(letter)
}

for (i in 1:4) print(x[i]) # single expression, compact style

# Nested

x <- matrix(1:6,2,3)
for(i in seq_len(nrow(x))){

for(j in seq_len(ncol(x))){
print(x[i,j])
}
}

[1] "a"
[1] "b"
```

```
[1] "c"
[1] "d"
[1] "a"
[1] "b"
[1] "c"
[1] "d"
[1] "a"
[1] "b"
[1] "c"
[1] "d"
[1] "a"
[1] "b"
[1] "c"
[1] "d"
[1] 1
[1] 3
[1] 5
[1] 2
[1] 4
[1] 6
```

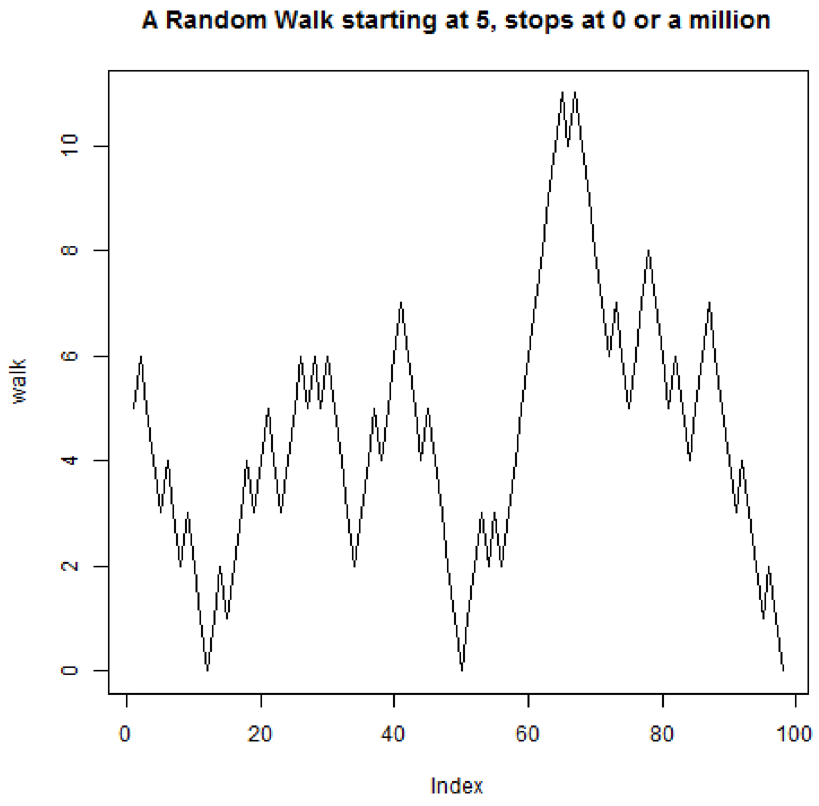
Nested loops beyond 2 levels is difficult to comprehend can ususally use functions to get by without.

2.1.3 While

Other looping. Takes a logical expression - while TRUE will run loop.

```
count <- 0
while(count<10){
print(count)
count <- count +1
}

#Random Walk
walk <- NULL
z <- 5
while(z >=0 && z <=100000){
walk <- c(walk, z)
coin <- rbinom(1,1,0.5)
if(coin==1) { ## random walk
```

```
    z <- z +1
} else {
    z <- z -1
}
}

plot(walk, type="l", main="A Random Walk starting at 5, stops at 0 or a million")
```



- Can produce an infinite loop
- Conditions evaluated from left to right
- && single logical value is used

2.1.4 repeat loops

- need to call break or it runs forever

```
x0 <-1
tol <- 1e-8

repeat{
x1 <- computeEstimate()

if(abs(x1-x0) < tol){
    break
}else{
    x0 <-x1
}
}
```

- Uses tolerance to keep looking at algorythm until tolerance value is met. BUT sometimes will not converge - so we should have a max iterations argument - use a for loop and it will eventually reached limit and stop.

2.1.5 next

- used to skip a certain part of loop:

```
for(i in 1:25){
if (i<=20){
        ## skip the first 20 iterations
        next
}
print(i)
}
```

```
 [1] 21
 [1] 22
 [1] 23
 [1] 24
 [1] 25
```

2.1.6 return

- Exit entire function and return a value that you pass it.
- Interupts everything

2.1.7 Final Notes

- Infinite loops should be looked out for
- Looping functions are generally used with apply (especially when in-eracting with data).

2.2 Control Structures

- Transition from User to Programmer.

2.2.1 Basics of writing functions

- created using functions directive
- R functions are class "function":

```
f <- function(<args>){

## DO anything
}
```

- Functions are first class functions. They can be treated like any other objects
- can be passed to other functions
- nested so you can define a function inside another function
- the return value of the function = last expression in the body to be evaluated
- Functions have *names arguments* which can have *default values*
- *Formal arguments* are arguments that are included in the function definition
- The *formals* function: returns list of (formal) arguments of a function (included inside the ())
- Not all functions have formal arguments ?
- Functions can have missing arguments or might have defaults
- R function arguments can be matched by position or name:

```
mydata <- rnorm(1000)
formals(sd)

sd(mydata)
sd(mydata, na.rm=FALSE)

# OR
sd(mydata, FALSE)

# or
sd(x=mydata, na.rm=FALSE)

# or

sd(na.rm=FALSE, mydata)

# first argument is matched to mydata (after assigning mydata)
# Reversing arguments is a bad idea

#equivalent:
lm(data=mydata, y~x, model=FALSE, 1:100)
lm(y ~x, mydata, 1:100,  model=FALSE)
```

- names arguments are used on the command line (order not so important)
- functions can be partially matched (if it has a long name) looks for unique match. The order of preference is:
- check for exact match
- partial match
- positional match

2.2.2 LAZY EVALUATION

- Arguments are evaluated only as needed.

```
f <- function(a,b){
     a^2
}
f(2)

 [1] 4
```

- This function never actually uses the argument b, so called f(2) will NOT produce an error because a=2 (and it does not care about b).
- BUT:

```
f <- function(a,b){
print(a)
print(b)
}

f(45)

 [1] 45
 Error in print(b) (from #3) : argument "b" is missing, with no default
```

- Produces this:
- up to print(a) we are ok, but the print(b) throws the error (LAZY EVALUATION) - executes until it bonks on an error.

2.2.3 "..." argument.

- Often used to extend a function.
- so if you were calling another function inside your function, you can use ... to pass the arguments to the extended function
- Generic functions (we will talk about this later).
- dispatch methods for different types of data
- ... is handy if known number of arguments can not be known.
- paste is example.

```
args(paste)
args(cat)
```

```
function (..., sep = " ", collapse = NULL)
NULL
function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
    append = FALSE)
NULL
```

- One catch is that anything after ... must be EXPLICIT and CANNOT be PARTIALLY MATCHED

2.2.4 A Diversion on Binding Values to a Symbol

```
lm <- function(x) {x *x}
lm
```

- how does R know what you are talking about?
- *lm* in *stats package*
- *lm* in you just defined.
- R binds a value to a symbol.
- Searches through series of *Environments* for a match.
- Search .GlobalEnv environment first (users workspace)
- Then search namespaces of search list (all R packages loaded in R)
- *Base* package is last

```
search()
```

```
function(x) {x *x}

 [1] ".GlobalEnv"        "package:stats"     "package:graphics"
 [4] "package:grDevices" "ESSR"              "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

- list of packages are dynamic depending on session - library installs namespace right behind global environment.

```
library(NADA)
search()
```

```
Loading required package: survival
Loading required package: splines

Attaching package: 'NADA'

The following object is masked from 'package:stats':

    cor
 [1] ".GlobalEnv"        "package:NADA"      "package:survival"
 [4] "package:splines"   "package:stats"     "package:graphics"
 [7] "package:grDevices" "ESSR"              "package:utils"
[10] "package:datasets"  "package:methods"   "Autoloads"
[13] "package:base"
```

- you can have a vector names ‘c’ (and it would not interfere normally with function ‘c’) (separate namespaces for functions and non-functions)

2.2.5 Scoping Rules (Lexical Scoping)

- makes it different from S
- Scoping rules determine how value is bound to variable.
- Useful for simplifying calculations
- Sometimes called *static scoping* and alternative called *dynamic scoping*
- R uses search list to bind a value to a symbol
- Consider this:

```
f <- function(x,y){
x^2 + y / z
}
```

This function has 2 formal args, x and y. There is a symbol z in the body. z is a free variable... how assign value to z is the scoping rules.

- lexical scoping looks for value in the environment in which the function was defined.
- x <- 3.14

- y <- data.frame(x=rnorm(100), y=rnorm(100))
- and environment is a symbol-value pair
- every environment has a parent environment
- create a function and assign to an environment it creates a closure
- if a free variable is encountered, R looks in environment the function was defined in. If not. Search then looks in the parent environment until it hits the top-level environment. If can't find anything it throws an error
- Possible to define a function outside .GlobalEnv

-WHY DOES THIS MATTER?

- DEFINE GLOBAL VARIABLES
- DEFINE FUNCTIONS INSIDE OTHER FUNCTIONS
- EXAMPLE constructor functions that construct other functions

make power returns a function (one function can make many functions)

```
make.power <- function(n){
pow <- function(x){
x^n
}
pow
}

cube <- make.power(3)
square <- make.power(2)

cube(3)
square(3)
```

```
 [1] 27
 [1] 9
```

-Produce some different results

- how do you know what is in the functions environment>

```
ls(environment(cube))
ls(environment(square))

get("n", environment(cube))
get("n", environment(square))
```

```
 [1] "n"   "pow"
 [1] "n"   "pow"
 [1] 3
 [1] 2
```

- how the new function knows what to do (each has its own environment with things definitions
- *dynamic scoping* would do this - a free variable looks up value in environment where the function was defined.
- When a function is defined in the .GlobalEnv it will appear to be *dynamic scoping*

```
rm(list=ls())
g <- function(x){
a <- 3
x + a + y
}

g(2)
y<-3
g(2)
```

```
 Error in g(2) (from #3) : object 'y' not found
 [1] 8
```

- Other languages with *lexical scoping*
- Scheme
- Perl
- Python
- Common Lisp (*theorem: all languages converge to Lisp*)

- Consequence of Lexical Scoping
  - ALL OBJECTS GET STORED IN PHYSICAL MEMORY!
  - Limits big data
  - Every function has to have a pointer to its defining environment
  - in S+ free variable looked up in .GlobalEnv

2.3 Optimization

- optim and nl, and optimize require pass a function to them, whose argument is a vector of parameters
- finds minimum or maximize (usually log-likelihood)
- lexical scoping makes it easy
- create constructor function that constructs the objective function
- have data etc. in environment (like baggage)
- example:

```
rm(list=ls())
make.NegLogLik <- function(data, fixed=c(FALSE,FALSE)){
    params <- fixed
    function(p){
        params[!fixed] <- p
        mu <- params[1]
        sigma <- params[2]
        a <- -0.5 * length(data) * log(2*pi*sigma^2)
        b <- -0.5 * sum((data -mu)^2)/(sigma^2)
        -(a+b)
        }
}
```

- Note Optimization functions in R *minimize* functions, so you need to use the negative log-likelihood
- fit normal distribution.

```
set.seed(1); normals <- rnorm(100,1,2)
```

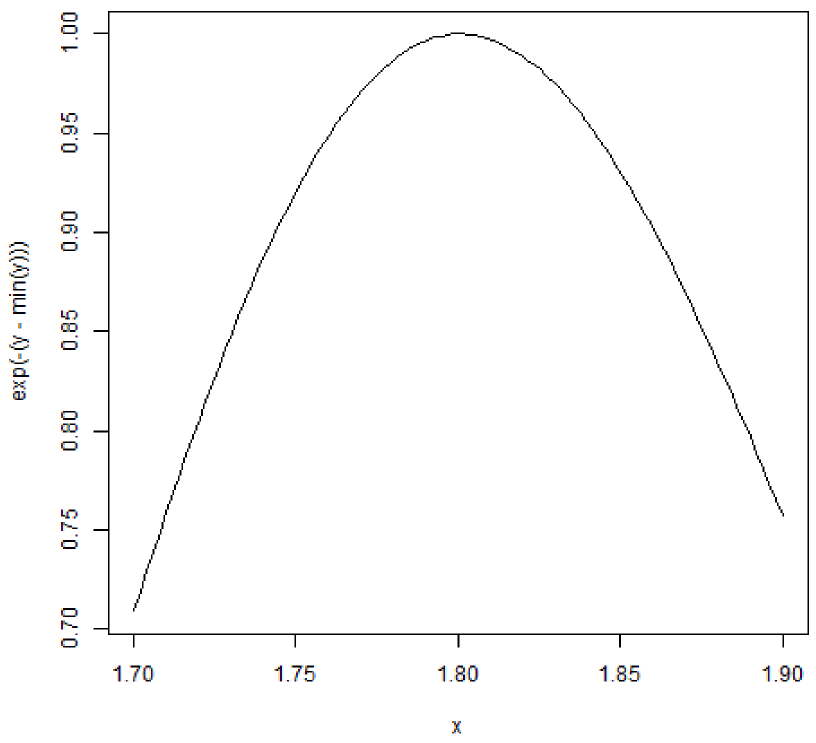
```
nLL <- make.NegLogLik(normals)
nLL
ls(environment(nLL))

function(p){
        params[!fixed] <- p
        mu <- params[1]
        sigma <- params[2]
        a <- -0.5 * length(data) * log(2*pi*sigma^2)
        b <- -0.5 * sum((data -mu)^2)/(sigma^2)
        -(a+b)
        }
<environment: 0x04add384>
[1] "data"   "fixed"  "params"
```

- environment is some fancy hex. number e.g. 0x165b1a4
- data variable is a free variable. But data can look up in the parent environment.
- now can call optim.

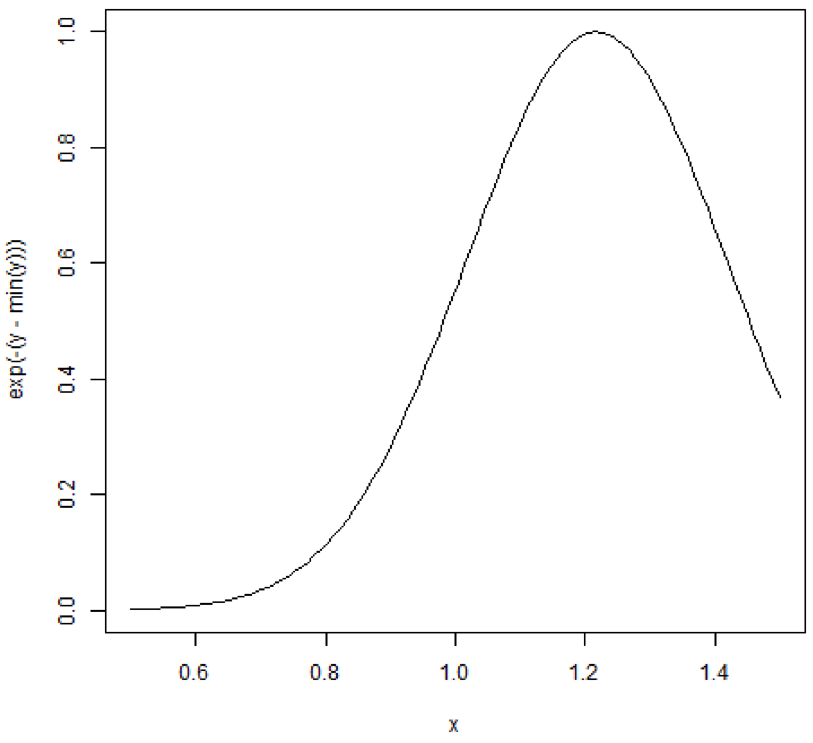
```
optim(c(mu=0, sigma=1), nLL)$par

print("fix sigma = 2")
nLL <- make.NegLogLik(normals, c(FALSE, 2))
optimize(nLL, c(-1,3))$minimum

print("fix mu =1")
nLL <- make.NegLogLik(normals, c(1, FALSE))
optimize(nLL, c(1e6,10))$minimum

       mu    sigma
 1.218239 1.787343
 [1] "fix sigma = 2"
 [1] 1.217775
 [1] "fix mu =1"
 [1] 10.00005
```

- plotting a log-liklihood

```
nLL <- make.NegLogLik(normals, c(1, FALSE))
x <- seq(1.7, 1.9, len=100)
y <- sapply(x,nLL)
plot(x, exp(-(y-min(y))), type="l")
```



```
nLL <- make.NegLogLik(normals, c(FALSE, 2))
x <- seq(0.5, 1.5, len=100)
y <- sapply(x,nLL)
plot(x, exp(-(y-min(y))), type="l")
```



```
x <- rlorm(1099)
summary(x)
function(x){
    rnorm(n=x)
}
```

2.4 loops lapply

- lapply - loop over a list and evaluation a function for each element in the list
- sapply - same as sapply but try and simplify result (Hadly was not a found fan of this)
- apply - apply function over margins of array
- tapply - apply function over subset of vector

- mapply- multivariate version of lapply
- split is an aux. function and is useful in conjunction with lapply (splits into a list of sub-pieces)

2.4.1 lapply

- takes 3 arguments
- X (a list). If X is not a list it will be coerced into a list
- FUN (a function)
- ... (extra arguments)
- actual looping is done in C

```
  x <- list(a=1:5, b=rnorm(10))
  lapply(x, mean)

x <- list(a=1:4, b=rnorm(10), c=rnorm(20,1), d= rnorm(100,5))
lapply(x,mean)

x <- 1:4

lapply(x,runif)

#suppose wanted nondefault behavior

lapply(x, runif, min=0, max=10) # using the ... part of lapply

$a
[1] 3

$b
[1] 0.3474802
$a
[1] 2.5

$b
[1] -0.1198232
```

```
$c
[1] 0.782726

$d
[1] 4.964196
[[1]]
[1] 0.7603133

[[2]]
[1] 0.1554012 0.8494571

[[3]]
[1] 0.9468178 0.5884192 0.5022508

[[4]]
[1] 0.189779918 0.001836858 0.877578062 0.134111338
[[1]]
[1] 0.2274122

[[2]]
[1] 9.391367 2.929487

[[3]]
[1] 1.643266 3.991026 4.595754

[[4]]
[1] 4.3403085 5.1700983 8.4624575 0.5516429
```

- what goes in is coerced to a list and what comes out *always* is a list.

```
x <- list(a=matrix(1:4, 2,2), b= matrix(1:6, 3,2))
x
# extract first column from matrix (need to create function to do this):
lapply(x, function(elt) elt[,1])

# function is gone at the end of lapply (an anonymous functions)

$a
     [,1] [,2]
[1,]    1    3
```

```
[2,]    2    4

$b
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
$a
[1] 1 2

$b
[1] 1 2 3
```

2.4.2 sapply

- will try and simplyfy results
- for instance, if all elements in list that comes back has same length - it will create vector (length 1) or matrix (if greater than 1).
- if can't it will return a list

```
x <- list(a=1:4, b=rnorm(10), c=rnorm(20,1), d= rnorm(100,5))
sapply(x,mean)
# returns vector
class(sapply(x,mean))

# error:

mean(x) # mean can not be directly applied to lists

What will this produce?
x <- list(rnorm(100), runif(100), rpois(100, 1))
sapply(x, quantile, probs = c(0.25, 0.75))

         a          b          c          d
2.50000000 0.01869495 1.07583942 5.06185108
[1] "numeric"
[1] NA
Warning message:
In mean.default(x) : argument is not numeric or logical: returning NA
```

```
Error: unexpected symbol in "What will"
          [,1]      [,2] [,3]
25% -0.7070224 0.2235743 0.00
75%  0.6800490 0.6987254 1.25
```

2.4.3 apply function

- it is common to apply to rows or columns of a matrix
- apply is not really faster (but used to be true).
- BUT it involves less typing. Good programers are always lazy.
- X is an array
- MARGIN is an integer that indicated what should be 'retained'
- ...

```
print("apply mean to margin 2")
x <- matrix(rnorm(200),20,10)
apply(x,2,mean)

print("apply to sum on margin 1")
apply(x,1,sum)
```

```
 [1] "apply mean to margin 2"
  [1]  0.041597801 -0.007965326 -0.067736108 -0.442325546 -0.319348298
  [6] -0.160523432  0.054901735  0.037705261 -0.157675375  0.164261296
 [1] "apply to sum on margin 1"
  [1]  -5.4715707  -0.2029979   5.6390271   2.9712664  -4.7227922  -4.6296715
  [7]  -3.4401373   0.7085447   7.1666165   3.7750684  -2.2374511  -0.3080558
 [13]  -2.0090229  -6.1822947  -0.9802599 -10.4664555  -0.1741214   1.2001970
 [19]   1.2206894   1.0012615
```

- column in margin 2 (the row dimension has been eliminated)
- row is margin 1 (collapse the columns and preserve the rows)

2.4.4 simple shortcuts

Quicker than apply (use these where you can)

- rowSums

- rowMeans
- colSums
- colMeans
- apply in applying a function

```
x <- matrix(rnorm(200),20,10)
apply(x,1,quantile, probs=c(0.25,0.75))

          [,1]       [,2]       [,3]       [,4]       [,5]       [,6]
25% -0.9042243 0.03361393 -0.6115209 -0.2969511 -0.5633196 -1.0058142
75%  0.5425005 0.98014390  0.7370544  0.2220907  0.5935485  0.6515029
          [,7]       [,8]       [,9]      [,10]     [,11]        [,12]
25% -0.8218577 -0.1700024 -0.7734633 -0.6774506 0.1271081 -0.001441062
75%  0.8648419  1.4822250  0.7245645  0.3245205 0.7244844  0.906033317
         [,13]        [,14]      [,15]      [,16]      [,17]      [,18]
25% -0.8746104 -1.954728372 -0.3022716 -0.8702418 -0.1525945 -1.1571882
75%  0.6324104  0.004634306  0.5467537  0.9206052  0.6926430  0.6431997
          [,19]      [,20]
25% -1.50837818 -0.8324263
75% -0.06477856  0.3844889
```

- Average of matrix in an array

```
a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
apply(a,c(1,2), mean)

print("rowMeans")
rowMeans(a, dims=2)

             [,1]       [,2]
 [1,] -0.09143177 -0.4493842
 [2,] -0.08272792 -0.1027398
 [1] "rowMeans"
             [,1]       [,2]
 [1,] -0.09143177 -0.4493842
 [2,] -0.08272792 -0.1027398
```

  - average of a bunch of 2 by 2 matrices (collapsing 3rd dimension)
  - rowMeans can work as well (using dims=2).

```
x <- matrix(rnorm(200), 50, 4)
apply(x,1,sum)
apply(x,3,mean)
apply(x,2,min)
apply(x, c(1,2), mean)

 [1]  0.05391193 -0.98725795  0.43950476  1.98447766  4.24731610 -2.22369827
 [7] -3.15906582 -2.72183522 -2.40786140 -0.17309768  3.96631043 -1.51406148
[13] -1.57396399 -1.98776564  1.85180050  3.09105187  0.24316866 -0.22275780
[19]  5.76285976 -1.30485739  4.06905865  0.95759103 -3.64273739  1.49945830
[25]  0.78987460 -2.00602076  1.55099746  0.37871842 -1.10517239 -0.67178684
[31] -3.14734641  0.87092184 -0.60548329 -1.11226137  1.44527850 -0.16108587
[37]  0.13958292 -2.05200753  2.35397435  0.95810458 -4.54757894  4.24600726
[43]  1.76835899 -0.75520723 -4.76132526 -1.49331871  0.98204482 -2.99763275
[49] -1.35146542 -1.62787599
Error in if (d2 == 0L) { : missing value where TRUE/FALSE needed
[1] -2.321491 -2.090846 -3.213189 -2.106118
              [,1]         [,2]        [,3]        [,4]
 [1,] -0.326489593  0.855519222 -0.09953695 -0.37558075
 [2,]  0.774005212 -0.819963127 -0.43985764 -0.50144240
 [3,]  0.785006401 -0.123602760 -0.71851145  0.49661258
 [4,]  0.763246080  0.254948236 -0.55459760  1.52088094
 [5,]  0.294808760  1.718926338  1.24548918  0.98809183
 [6,] -1.252355924 -0.958543528 -1.25892135  1.24612253
 [7,] -1.009503753 -1.604310262 -0.21538448 -0.32986733
 [8,]  0.751391195 -1.845609422 -2.47196171  0.84434471
 [9,] -1.308353513  0.555737185 -0.67416932 -0.98107576
[10,]  0.527540097 -0.060119191 -0.50129719 -0.13922141
[11,] -0.533539574  0.772086304  1.54232579  2.18543791
[12,] -0.398376014 -0.140839387 -0.96201807 -0.01282801
[13,] -0.789569450  0.393093926 -0.87217954 -0.30530893
[14,] -0.230141136  0.224218574 -1.39762962 -0.58421346
[15,]  0.877184842  0.023541985  0.17980517  0.77126850
[16,]  0.453733178 -0.622962660  1.15409199  2.10618936
[17,] -0.232464148  1.262009381 -1.19853361  0.41215704
[18,]  0.870005525 -0.405774043 -0.42572440 -0.26126488
```

```
[19,]  1.656003734  0.666763771  1.36630861  2.07378365
[20,] -0.006368929  0.164639155 -0.68429739 -0.77883022
[21,]  0.470489453  1.781524475  0.68551221  1.13153251
[22,]  0.278218649  0.711213964  0.38950354 -0.42134513
[23,] -0.977902941 -0.337691156 -1.30539592 -1.02174737
[24,] -0.926586142 -0.009148952  1.21688801  1.21830538
[25,]  1.919770463 -0.125309208  0.79517402 -1.79976067
[26,]  0.881277788 -2.090846097 -0.48820251 -0.30824994
[27,]  0.742081772  1.697393895 -0.90399345  0.01551524
[28,]  0.147573404  1.063881154 -0.39041842 -0.44231772
[29,]  0.485388565 -0.766616636  0.81406342 -1.63800773
[30,]  0.151856040  0.382007559 -0.56424928 -0.64140116
[31,]  0.041998754  0.241895904 -1.87420532 -1.55703574
[32,]  0.223422312 -1.132759411 -0.14290471  1.92316365
[33,] -1.010465086  1.489907414  0.77190401 -1.85682963
[34,]  2.401222102 -0.248247105 -1.15911793 -2.10611844
[35,]  0.801961790  0.183583708 -0.23791553  0.69764853
[36,] -0.251207959  0.404871009 -1.22219333  0.90744441
[37,]  1.212889371 -0.994124469  0.11680621 -0.19598820
[38,] -0.627258086 -1.085429330 -0.13249963 -0.20682049
[39,]  1.711158507 -0.048542555 -0.03368478  0.72504317
[40,] -0.394373553  0.576085601 -0.62232642  1.39871896
[41,] -2.321490856  0.073830532 -0.70936346 -1.59055515
[42,]  1.364119195  0.705945571  0.87144541  1.30449708
[43,]  1.132229133  0.334980103  0.10513802  0.19601173
[44,] -0.774316319  0.545387806 -0.18693527 -0.33934345
[45,] -1.410374966 -1.402905906 -3.21318853  1.26514414
[46,] -1.834527581  0.677053891 -1.27561870  0.93977369
[47,] -0.269013538 -0.789800446  0.76290632  1.27795249
[48,] -1.833928577 -0.465728889 -0.40681514 -0.29116014
[49,] -0.814468019 -0.104852065 -1.20831778  0.77617245
[50,]  0.163572122 -1.647851087 -0.43932266  0.29572563
```

2.4.5 tapply

- function over a vector (pieces need summary statistic over)
- X is a vector
- INDEX is a factor or list of factors (coerced into factors)
- FUN is the function to be applied

- ... contains other arguments to be passed to FUN
- simplify (TRUE) like sapply simplification

```
x <- c(rnorm(10), runif(10), rnorm(10,1))
f <- gl(3,10) # generate levels
f
tapply(x,f,mean)
```

```
  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
 Levels: 1 2 3
         1         2         3
 0.4040684 0.4625386 0.8182498
```

- if you don't simplify the results you will get back a list.

Can get a complicated thing back with this as well:

```
tapply(x,f,range)
```

```
 $`1`
 [1] -0.7057714  2.6064445

 $`2`
 [1] 0.1394657 0.9299194

 $`3`
 [1] -0.6521103  3.3905767
```

2.4.6 split

- takes a vector or other objects and splits them by a factor or list of factors.
- x is a vector
- f is a factor or list of factors
- drop indicated if empty factors will be dropped.

```
x <- c(rnorm(10), runif(10), rnorm(10,1))
f <- gl(3,10)
split(x,f)

# returns a list of length 3 with the 3 distributions in each

$`1`
 [1] -0.36513161  1.38145425 -0.15331726 -0.25557163 -1.28862710  0.06526642
 [7]  1.03532642  2.26021579  1.31469628 -0.87002335

$`2`
 [1] 0.30743642 0.52829125 0.72822568 0.95355653 0.49599413 0.13201725
 [7] 0.60846876 0.99186858 0.09470967 0.89528346

$`3`
 [1] 0.57347013 0.07359575 0.07144208 0.28109940 0.58520943 0.99433211
 [7] 1.63344892 0.49081923 0.14299138 2.61616758
```

- can also do some real fun here with plots:

```
lapply(split(x,f), mean)

lapply(split(x,f), hist)

print("Some more things to do")
library(datasets)
head(airquality)

# calculate mean for each month of all the columns

s <- split(airquality, airquality$Month)
lapply(s, function(x) colMeans(x[c("Ozone", "Solar.R", "Wind")], na.rm=TRUE))

$`1`
[1] 0.3124288

$`2`
[1] 0.5735852

$`3`
```

```
[1] 0.7462576
$`1`
$breaks
[1] -2 -1  0  1  2  3

$counts
[1] 1 4 1 3 1

$density
[1] 0.1 0.4 0.1 0.3 0.1

$mids
[1] -1.5 -0.5  0.5  1.5  2.5

$xname
[1] "X[[1L]]"

$equidist
[1] TRUE

attr(,"class")
[1] "histogram"

$`2`
$breaks
[1] 0.0 0.2 0.4 0.6 0.8 1.0

$counts
[1] 2 1 2 2 3

$density
[1] 1.0 0.5 1.0 1.0 1.5

$mids
[1] 0.1 0.3 0.5 0.7 0.9

$xname
[1] "X[[2L]]"

$equidist
```

```
[1] TRUE

attr(,"class")
[1] "histogram"

$`3`
$breaks
[1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0

$counts
[1] 5 3 0 1 0 1

$density
[1] 1.0 0.6 0.0 0.2 0.0 0.2

$mids
[1] 0.25 0.75 1.25 1.75 2.25 2.75

$xname
[1] "X[[3L]]"

$equidist
[1] TRUE

attr(,"class")
[1] "histogram"
[1] "Some more things to do"
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
$`5`
    Ozone   Solar.R      Wind
 23.61538 181.29630  11.62258

$`6`
    Ozone   Solar.R      Wind
 29.44444 190.16667  10.26667

$`7`
     Ozone    Solar.R       Wind
 59.115385 216.483871   8.941935

$`8`
     Ozone    Solar.R       Wind
 59.961538 171.857143   8.793548

$`9`
    Ozone   Solar.R      Wind
 31.44828 167.43333  10.18000
```

- Or you can use sapply

```
sapply(s, function(x) colMeans(x[,c("Ozone", "Solar.R", "Wind")], na.rm=TRUE))
```

```
                 5         6          7          8         9
 Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
 Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
 Wind     11.62258  10.26667   8.941935   8.793548  10.18000
```

- splitting on more than one level
  - more than one factor
  - combinations

```
x <- rnorm(10)
f1 <- gl(2,5)
f2 <- gl(5,2)
f1
f2
interaction(f1,f2)
# 10 different levels

#interactions can create empty levels!

str(split(x, list(f1,f2))) # automatically creates interaction

str(split(x, list(f1,f2), drop=TRUE)) # automatically creates interaction
```

```
 [1] 1 1 1 1 1 2 2 2 2 2
Levels: 1 2
 [1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5
 [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 2.4 1.5 2.5
List of 10
 $ 1.1: num [1:2] 0.994 0.697
 $ 2.1: num(0)
 $ 1.2: num [1:2] 1.7 -0.978
 $ 2.2: num(0)
 $ 1.3: num 2.05
 $ 2.3: num 1.23
 $ 1.4: num(0)
 $ 2.4: num [1:2] 0.307 0.624
 $ 1.5: num(0)
 $ 2.5: num [1:2] 0.0613 -0.1108
List of 6
 $ 1.1: num [1:2] 0.994 0.697
 $ 1.2: num [1:2] 1.7 -0.978
 $ 1.3: num 2.05
 $ 2.3: num 1.23
 $ 2.4: num [1:2] 0.307 0.624
 $ 2.5: num [1:2] 0.0613 -0.1108
```

2.4.7 mapply

- loop function multivariate apply
- where to use - what if you have 2 lists - 1 for each arg of function
- can use for loop.
- or can use mapply
- ARGS
- FUN is function
- ... is arguments to apply over (must equal the number of functions)
- MoreArgs is a list of other arguments to FUN

- SIMPLIFY indicates whether the result should be simplified.

```
mapply(rep, 1:4, 4:1)

[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4
```

- mapply can be used for a lot of arguments.

```
  noise <- function(n, mean, sd){
      rnorm(n,mean, sd)
  }

  noise(5,1,2)
  # this does not do what he wants:
# what he wants 1 normal with mean 1, 2 normals with mean 2 etc.

  noise(1:5,1:5,2)


# but this works:

mapply(noise, 1:5, 1:5, 2)

[1] -2.152467  2.485180  5.284647  5.179546  1.339589
[1] 0.7842444 2.3639890 5.2918564 6.9545976 5.8158883
[[1]]
[1] 3.228557

[[2]]
[1] 1.945848 2.995444
```

```
[[3]]
[1]  5.371280 10.279147  2.891995

[[4]]
[1] 2.663705 4.891616 3.188350 5.270566

[[5]]
[1] 5.682497 7.632334 3.080447 2.588850 8.135146
```

- **instantly vectorize the function!**

2.5 Debugging tools

- built in with R
- figure out *what is wrong* after you find a problem
- `message` Notification/ FYI
- `warning` indication is unexpected event (may not be a problem)
- `error` stops execution of function - and prints a message (produced by stop function
- `condition` a generic event that can be created by a function (generic)

2.5.1 WARNING

```
log(-1)
```

```
 [1] NaN
 Warning message:
 In log(-1) : NaNs produced
```

- may be fine or not. . .

```
  printmessage <- function(x){
      if(x>0)
          print("X is greater than zero")
      else
          print("X is less than or equal to zero")
      invisible(x) # return object but will not autoprint
  }

printmessage(1)

printmessage(NA)
```

```
 [1] "X is greater than zero"
 Error in if (x > 0) print("X is greater than zero") else print("X is less than or equa
   missing value where TRUE/FALSE needed
```

- has to error out - missing value(NA) was expecting T/F and it got NA which is neither.
- printmessage 2:

```
printmessage2 <- function(x){
    if(is.na(x))
       print("X is missing value")

       else if(x>0)
        print("X is greater than zero")

       else
        print("X is less than or equal to zero")
    invisible(x) # return object but will not autoprint
}

    x <- log(-1)
    printmessage2(x)
```

```
 Warning message:
 In log(-1) : NaNs produced
 [1] "X is missing value"
```

- not an error but not what might be expected...

```
 Warning message:
 In log(-1) : NaNs produced
 [1] "X is missing value"
```

- when you think something has gone wrong:

- what is your input? how did you call the function?
- what were you expecting? output, messages or other results?
- what were the results?
- how does what you get differ from the expectation?
- were your expectations correct to begin with?
- can you reproduce the problem?
- can you reproduce the problem that you had (could be a real problem over the web or on a network machine)?

2.5.2 Debugging tools

- `traceback` prints out the function call stack after an error occurs; does nothing if there's no error.
- `debug` flags a function for "debug" mode which allows you to step through execution one line at a time
- `browser` suspends execution of a function wherever it is called and puts things into debug mode
- `trace` allows you to insert debugging code into a specific place of your function
- `recover` allows you to modify the behavior so that you can browse the function call stack

These are interactive tools specifically designed to pick through a function.

```
rm(list=ls())
mean(x)
traceback()

 Error in mean(x) :
   error in evaluating the argument 'x' in selecting a method for function 'mean': Err
 1: mean(x)
```

- mean was where the error occurred

- must execute immediately after error.

```
lm(y ~ x)
traceback()
```

=Error in eval(expr, envir, enclos) : object ‘y’ not found 7: eval(expr, envir, enclos) 6: eval(predvars, data, env) 5: model.frame.default(formula = y ˜ x, drop.unused.levels = TRUE) 4: model.frame(formula = y ˜ x, drop.unused.levels = TRUE) 3: eval(expr, envir, enclos) 2: eval(mf, parent.frame()) 1: lm(y ˜ x) =- could not evaluate the formula (y ˜x) ORG-LIST-END-MARKER

2.5.3 debug function

- can debug lm function
- prints out whole function body
- give you expression you gave
- put you in browser
- workspace environment is inside the function environment
- press ‘n’ for next etc. for each line until you get to line with error
- if you need a value that is n use print(n) to get it

2.5.4 recover function

- `options(error=recover)`
- on error get a function call stack and can select which function you want to enter browser with.

2.5.5 Summary

- message warning, error are indications of a problem
- reproduce problems and understand what the expectations are
- use interactive tools to poke around
- use your head

3 Week Three Lectures

3.1 Simulation

- Important for statistics and other applications

3.1.1 Distribution Funtions

- rnorm
  - generate normal variates with a mean and standard deviation
- dnorm
  - evaluation normal probability denisty at a point or vector of points
- pnorm
  - cummulative probability distribution
- qnorm
  - quantile for normal
- rpois
  - random variate from poison distribution
- Notation
  - r (like rnorm) indicates a random variate
  - d (like dnorm) indicates a density function
  - p (like pnorm) indicates a cummulative probability function
  - q (like qnorm) indicates a quantile (given a probability)

```
dnorm(x, mean = 0, sd = 1, log=FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail=TRUE, log.p=FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail=TRUE, log.p=FALSE)
rnorm(n, mean = 0, sd = 1)
```

  - all require specification of mean and sd and have default mean 0 and sd = 1

If Φ is a cumulative distribution function for a standard Normal distribution, then `pnorm(q)` $= \Phi(q)$ and `qnorm(q)` $= \Phi^-1(q)$.

```
x <- rnorm(10)
print(x)

x<- rnorm(10,20,2)
print(x)
summary(x)
```

```
  [1]  0.22528580 -0.92241066 -1.07377241 -0.55235829  0.59046140 -0.52727541
  [7]  1.34184616  0.24384463  0.19588079 -0.01551973
  [1] 19.40485 19.66464 22.31296 17.32247 22.17354 19.29863 21.46945 19.88317
  [9] 17.30837 20.69381
    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   17.31   19.33   19.77   19.95   21.28   22.31
```

3.2 Graphics

3.2.1 Base

3.2.2 Lattice

3.2.3 ggplot2

3.3 Reproducable Research

4 Week 4 Notes

4.1 Colors

4.2 Regular Expressions and R

4.3 Classes and Methods

- both an interactive language and programming language
- much of the code was writen by John Chambers (who created S)
- much is documented in the green book (Programning with Data: A Guide to the S Language)
- OO was designed to allow to cross from user to programmer.
- Classes and methods are for the programmer.

4.3.1 S3 classes/methods

- informal, new classes of data did not have a formal definition
- easier to implement

4.3.2 S4 classes/methods

- formal definitions of data
- harder to implement
- for now S3 and S4 will exist in R and can be mixed.
- A \ is a description of a thing, created using `setClass()`
- An *object* is an instance of a class. Objects can be created using `new()`
- A *method* is a function that operates only a certain class of objects
- A *generic function* figures out the class and finds the method and calls the *method* for that object

4.3.3 Documentation

- help page for classes and methods (long)
- `?setClass ?setMethod ?setGeneric` very technical
- assumes you are at programming level

4.3.4 Determining class

```
class(1)
class(TRUE)
class(rnorm(100))
class(NA)
class("foo")
```

```
 [1] "numeric"
 [1] "logical"
 [1] "numeric"
 [1] "logical"
 [1] "character"
```

- can go farther

```
x <- rnorm(100)
y <- x + rnorm(100)
fit <- lm(y ~x)
class(fit)
```

```
 [1] "lm"
```

- you might want to customize the output of the function
- define a method for the class `lm` that provides the functionality
- can define new generic

```
mean

print

standardGeneric for "mean" defined from package "base"

function (x, ...)
standardGeneric("mean")
<environment: 0x04a8dfd4>
Methods may be defined for arguments: x
Use  showMethods("mean")  for currently available ones.
standardGeneric for "print" defined from package "base"

function (x, ...)
standardGeneric("print")
<environment: 0x05270ce8>
Methods may be defined for arguments: x
Use  showMethods("print")  for currently available ones.
```

- `UseMethod("mean")` dispatched method for given data type.

4.3.5 finding methods in S3 and S4

- give name of generic will call a function that returns the methods available in S3.

```
methods("mean")
show # S4 equivalent to print

showMethods("show")

[1] mean.Date     mean.default  mean.difftime mean.POSIXct  mean.POSIXlt
standardGeneric for "show" defined from package "methods"

function (object)
standardGeneric("show")
<bytecode: 0x04b617c8>
<environment: 0x04226280>
Methods may be defined for arguments: object
Use  showMethods("show")  for currently available ones.
(This generic function excludes non-simple inheritance; see ?setIs)
Function: show (package methods)
object="ANY"
object="cenfit"
object="cenken"
object="cenmle"
object="cenreg"
object="censummary"
object="classGeneratorFunction"
object="classRepresentation"
object="envRefClass"
object="function"
    (inherited from: object="ANY")
object="genericFunction"
object="genericFunctionWithTrace"
object="MethodDefinition"
object="MethodDefinitionWithTrace"
object="MethodSelectionReport"
object="MethodWithNext"
object="MethodWithNextWithTrace"
object="NADAList"
object="namedList"
object="ObjectsWithPackage"
object="oldClass"
object="refClassRepresentation"
object="refMethodDef"
```

```
object="refObjectGenerator"
object="ros"
object="signature"
object="sourceEnvironment"
object="standardGeneric"
    (inherited from: object="genericFunction")
object="summary.cenreg"
object="traceable"
```

- S3 will be method.class notation
- S4 is not

-define generic function as a function that checks for a method for the class

- call the method and execute
- if a method does not exist search for a default method (and call if it is exist)
- if neither exist return an error

4.3.6 looking at code for the method

- need to specify generic and the class

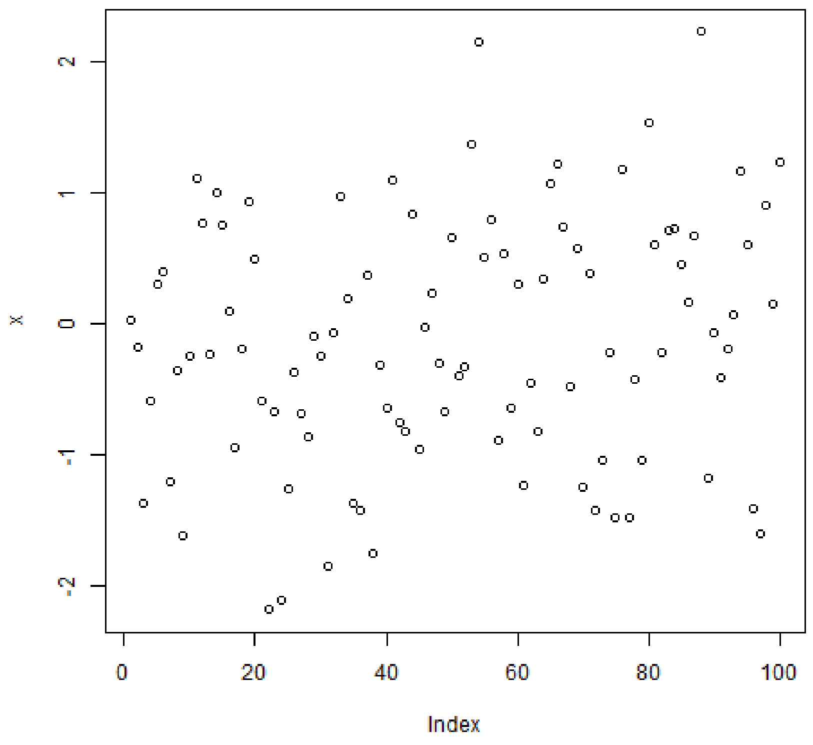
```
getS3method() # is S3
getMethod() # is S4

###### EXAMPLE

set.seed(2)
x <- rnorm(100)
mean(x)

head(getS3method("mean","default"))
tail(getS3method("mean","default"))

set.seed(3)
df <- data.frame(x=rnorm(100), y=1:100)
sapply(df, mean)
```

- the data frame class is `df`
- apply the mean function over the data frame (which has integer and numeric data)
- in each column the function checks the appropriate method
- in both cases since there is not a specific method `mean` calls the default method
- notice some methods in S3 are visible (can be called directly) BUT you should **NEVER** call methods directly. If the methods change or whatever you can always keep up to date if the methods change.

4.3.7 Example of class and method

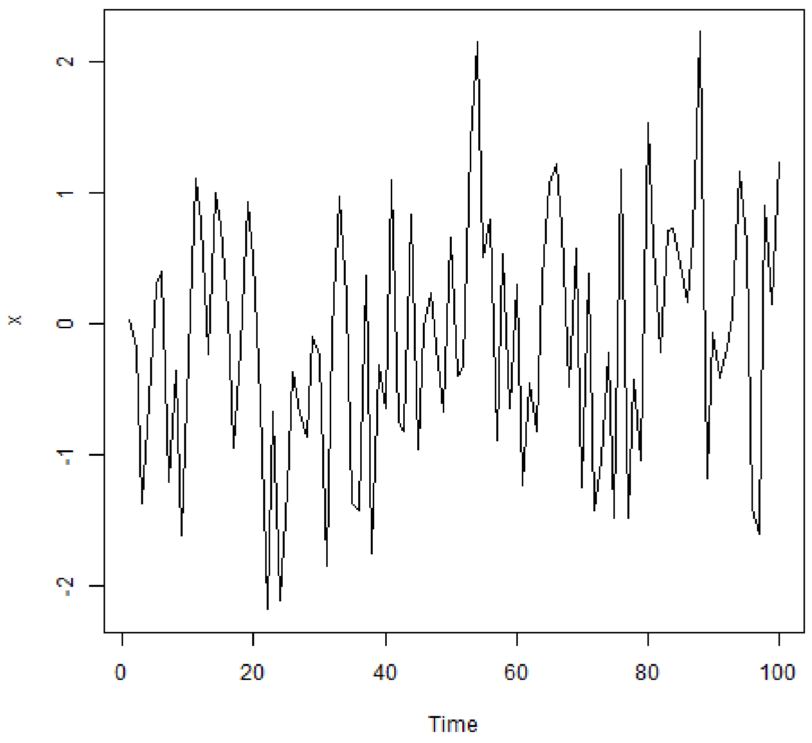
- this dispatches plot method of numeric data

```
set.seed(10)
x <- rnorm(100)
plot(x)
```



- but if it is a time series (using `as.ts` to convert)
- a different plotting function is dispatched

```
rm(list=ls())
set.seed(10)
x <- rnorm(100)
x <- as.ts(x)
plot(x)
```



4.3.8 new classes and methods

- Why?
- You have new data (new to R anyways) without built in ways to manipulate them.
- new ideas not implimented yet
- typically write methods for
- print/show
- summary
- plot
- extend R system via classes and methods

- write a new class but for existing generic (e.g. `print`)
- write new generics function and new methods for those generics

4.3.9 S4 methods

- explicity definition for every class
- use setClass function
- specify the name of the class (at minimum)
- data elements (*slots*) which are elements that store data
- `setMethod` to define methods
- `showClass` gives you information about the class
- Creating new classes/methods
  - say we want to create a class for `polygon` explicitly (this is new)
  - usually stored in separate file (sourced into R)

```
setClass("polygon", representation(x="numeric", y="numeric"))
```

  - the slots for the class `polygon` are `x` and `y`
  - the slots can be accessed with the `@` operator

A plot method for the `polygon`

```
setMethod("plot", "polygon", function(x,y,...){
    plot(x@x, x@y, type="n", ...)
    xp <- c(x@x, x@x[1])
    yp <- c(x@y, x@y[1])
    lines(xp, yp)

})
```

```
 [1] "plot"
```

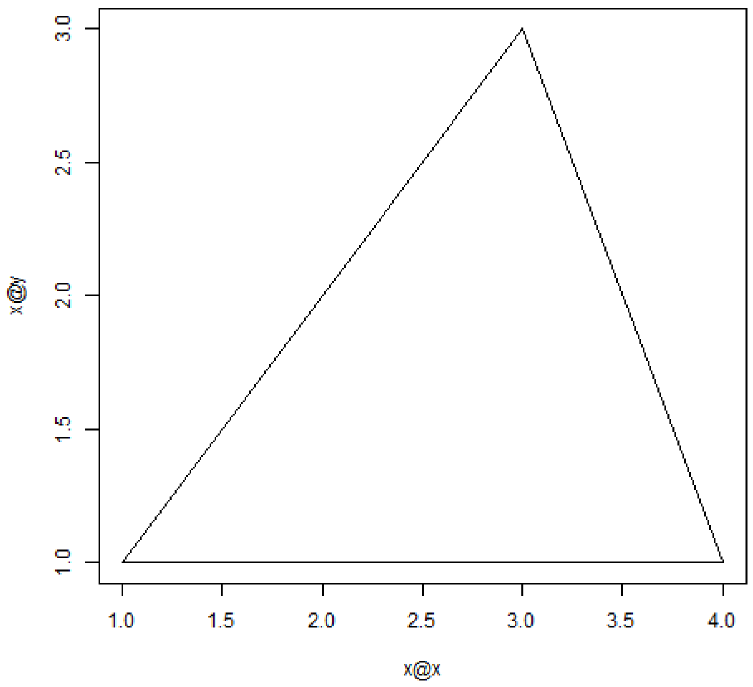
  - notice the `@` operator to access the slots
  - when run, the setMethod 'registers' the method with the system

- if you close R out - you will have to rededine method

```
showMethods("plot")

 Function: plot (package graphics)
 x="ANY", y="ANY"
 x="cenfit", y="ANY"
 x="cenmle-gaussian", y="ANY"
 x="cenmle-lognormal", y="ANY"
 x="cenreg", y="ANY"
 x="polygon", y="ANY"
 x="ros", y="missing"
```

- will show that the ploygon method is registered.

```
p <- new("polygon", x=c(1,2,3,4), y=c(1,2,3,1))
plot(p)
```



4.3.10 places to go to get help

- CRAN (packages that use S4)
- SparseM, flexm, lme3
- bioconductor site
- stats4 (comes with R) has mle methods in S4