

# R Basics

Naomi Altman  
Department of Statistics

# Help in R

If you do not know the name of the function (e.g. variance)

`help.search("variance")` finds all occurrences of the word in the documentation

If you know the name of the function (e.g. var):

`?var` or `help(var)` brings up the documentation

`help.start()` or `help>HTML help`

brings up the documentation system

`var`<sub>9/20/2006</sub>

shows you the code (usually)

# Example 1

We will

- generate some random normals
- compute some summary statistics
- draw a histogram, boxplot and Normal Probability plot
- see what variables are in our workspace
- save our workspace
- quit

# Example 1

```
help.search("normal")
```

```
?Normal
```

```
x = rnorm(100)
```

```
x
```

```
1:10
```

```
x[1:10]
```

```
y=c(2,4,7)
```

```
y
```

```
x[y]
```

```
length(x)
```

```
length(x[y])
```

# Example 1

```
help.search("histogram")
```

```
?hist
```

```
hist(x)
```

```
hist(x, nclass=20)
```

```
hist(x,xlab="Random Normals",main="Bioinformatics II")
```

```
hist(x,breaks=c(-4,-1,0,2))
```

```
?boxplot
```

```
boxplot(x)
```

```
hist(x,xlab="Random Normals",main="Bioinformatics II")
```

```
boxplot(x,add=T,horizontal=T,col=2)
```

# Example 1

```
help.search("normal")
```

```
?qqnorm
```

```
qqnorm(x)
```

```
qqline(x)
```

```
help.search("quantile")
```

```
par(mfrow=c(2,2))
```

```
hist(x,xlab="Random Normals",main="Bioinformatics II")
```

```
boxplot(x,horizontal=T,col=2,xlab="Random  
  Normals",main="Bioinformatics II")
```

```
qqnorm(x, main="Bioinformatics II")
```

```
plot(density(x), main="Bioinformatics II")
```

# Example 1

```
ls()  
history()  
history(5)
```

```
file>save workspace
```

```
q()
```

# Naming Conventions

- Names in R are made up of upper- and lower-case alphabets. 0-9 and period, '.', and be used except in the first position.
- R is case sensitive.

# Language Layout

- R commands are either expressions or assignments.

```
1 - pi +exp(1.7) #prints the output
```

```
a = 6           #saves "a"
```

- '#' marks the rest of line as a comment.
- If a command is not complete, R issues '+' rather than '>'.

# Vectors

- There are no scalars; vectors of length 1 are used instead.
- Vectors are made up of numeric, logical values, or character strings. But you can not mix them.
- Character strings can be entered with either double or single quotes, but will always be printed with double quotes. (Beware of "smart quotes" inserted by word processing programs.)

# Vector Arithmetic

- Arithmetic operations on vectors are performed element by element.
- If two vectors in the same expression have different lengths, the expression will produce a vector with the same length as the longest in the expression. The shorter vectors are recycled until they match the length of the longest.

# Logical Vectors

- Possible elements of a logical vector is T (True) and F (False).
- Logical operators are `>`, `>=`, `<`, `<=`, `==`, `!=`.
- `c1&c2` (intersection), `c1|c2` (union), `!c1` (negation).
- A logical vector can be used in ordinary arithmetic, in which case F and T become 0 and 1, respectively.

# Using Logical and Numerical Vectors

```
x=1:20
```

```
x^2
```

```
3*x
```

```
y=seq(0,1,.3)
```

```
y
```

```
x+y
```

```
z=x>15
```

```
z
```

```
2*z
```

```
z+y
```

```
x[z]
```

# Missing Value Marker, NA

- When an element of a vector is "not available" or "missing", the special value NA can be used to reserve the place e.g.  $\log(-5)$  yields NA
- In general, any operation on an NA becomes an NA.

# Session: NA

```
x = 1:5
```

```
is.na(x)
```

```
x[3] = NA
```

```
is.na(x)
```

```
x-2
```

```
log(x-2)
```

```
is.na(log(x-2))
```

# Session: Character Data

```
a = c("abc", "def", "ghi")  
paste("abc", "def")  
paste("abc", "def", sep="")  
paste(c("X", "Y"), 1:10, sep="")
```

# Index Vectors

- Subsets of elements of a vector can be selected by appending to a name of the vector an index vector in square brackets.
- An index expression can appear on the receiving side of an assignment expression.
- There are four kinds of index vectors:
  - a logical vector (in this case, the lengths of the vector and index vector must match.)
  - a positive integer vector
  - a negative integer vector
  - a character string vector.

# Session: Using Indices

```
x=1:20
```

```
x
```

```
x[c(3,5,6)]
```

```
x[-c(3,5,6)]
```

```
ind=x>14
```

```
x[ind]
```

```
x[x>14]
```

# Matrices and Dataframes

A matrix is a rectangular array. It can be viewed as a collection of column vectors all of the same length and the same type (i.e. numeric, character or logical).

A dataframe is a rectangular array. All of the columns must be the same length, but may be different types.

The rows and columns of a vector or dataframe can be given names.

We will store our array data in matrices and dataframes.

# Session: Matrices and Dataframes

```
x=1:10
y=rnorm(10,0,1)
z=paste("a",1:10)
x
y
z
mxy=cbind(x,y)
mxy
rownames(mat)
colnames(mat)
mat[1,]
mat[,1]
mat[, "x"]
mode(mat[,1])
```

```
tab=cbind(x,y,z)
tab
mode(tab[,1])
tab=data.frame(x,y,z)
mode(tab[,1])
mode(tab[,3])
rownames(tab)
colnames(tab)
rownames(tab)=
  paste("a",1:10)
```

# Lists

A list is a collection of objects that may be the same or different types.

The objects generally have names, and may be indexed either by name or component number.

e.g. the output of the `t.test` function is stored in a list that includes:

the t-value (numeric vector of length 1)

the null value (a numeric vector of length 1)

the alternative hypothesis (a character vector)

a confidence interval (a list including the ends of the interval and the confidence level)

A dataframe can be treated as a list of column vectors.

# Intrinsic Attributes: mode and length

- Every object in R has two attributes, *mode and length*.
- The mode of an object says what type of object it is.
- The function `attributes(x)` gives all non-intrinsic attributes of `x`.
- There is a special attribute called `class` which was introduced to enhance object-oriented programming. For example, the results of `plot(x)` depend on the class of `x`.

# Objects

A object is a vector, array or list along with a class.

The class is a flag that can be used for object oriented programming. It is used to create "methods".

e.g. "plot" could do a boxplot on a t-test output object and a residual plot on a regression output object.

# Session: Intrinsic Attributes

```
x=rnorm(100,3,2)
```

```
t.out=t.test(x,mu=1)
```

```
attributes(x)
```

```
class(x)
```

```
attributes(t.out)
```

```
length(t.out)
```

```
class(t.out)
```

```
length(t.out)
```

```
names(t.out)
```

```
t.out[[1]]
```

```
t.out$conf
```

```
t.out
```

```
#printing is affected by the class
```

```
unclass(t.out)
```

# Factors

- A factor is a special type of vector, normally used to hold a categorical variable in many statistical functions.
- Factors are primarily used in Analysis of Variance (ANOVA). When a factor is used as a predictor variable, the corresponding indicator variables are created.
- We need factors for differential expression analysis.

# Session: Factors

```
citizen <- c("uk", "us", "no", "au", "uk", "us",  
            "us", "no", "au")
```

```
citizenf <- factor(citizen)
```

```
citizen
```

```
citizenf
```

```
attributes(citizenf)
```

```
unclass(citizenf)
```

```
table(citizenf)
```

```
table(citizen)
```

# Control Structures

for (x in set){operations}

while (x in condition){operations}

repeat {operations, test, break}

if (condition) {operations}

    else {more operations}

switch(flag,dolist)

# "apply" and its relatives

Often we want to repeat the same function on all the rows or columns of a matrix, or all the elements of a list.

We could do this in a loop, but R has a more efficient operator the apply function

# Applying a function to the rows or columns of a matrix

If `mat` is a matrix and `fun` is a function (such as `mean`, `var`, `lm` ...) that takes a vector as its arguments, then

`apply(mat,1,fun)` applies `fun` to each  
row of `mat`

`apply(mat,2,fun)` applies `fun` to each  
column of `mat`

**In either case, the output is a vector.**

# Relatives of apply

`lapply(list,fun)` applies the function to every element of list

`tapply(x,factor,fun)` uses the factor to split x into groups, and then applies fun to each group

# Using apply and related functions

```
data()      #a list of internal data sets
```

```
?trees
```

```
class(trees)
```

```
class(trees$Girth)
```

```
class(trees$Height)
```

```
class(trees$Volume)
```

```
apply(trees,2,mean)
```

```
lapply(trees,mean)
```

?ChickWeight

```
tapply(ChickWeight$weight,ChickWeight$Chick,mean)
```

```
weight.byChick=split(ChickWeights$weight,ChickWeights$Chick)
```

```
weight.byChick
```

```
lapply(weight.byChick,mean)
```

# Functions

Functions are stored like data.

```
my.function=function(arguments){operations}
```

I write functions for just about anything I need to do more than once - I run through the commands once interactively, and then use the `history()` feature and an editor to create the function.

It is wise to include a comment at the start of each function to say what it does and to document functions of more than a few lines.

```
e.g. makeList=function(mat,i=2){  
#change a matrix into a list of rows or columns  
  mylist=list()  
  m=switch(i,t(mat),mat)  
  for (i in 1:ncol(m)){  
    mylist[[i]]=m[,i]  
  }  
  mylist  
}
```

#using makeList to change mxy to a list

```
makeList(mxy)  
makeList(mxy,i=2)  
makeList(mxy,i=1)
```

# Calling Conventions for Functions

- Arguments may be specified in the same order in which they occur in function definition, in which case the values are supplied in order.
- Arguments may be specified as name=value, when the order in which the arguments appear is irrelevant.
- Above two rules can be mixed.

```
t.test(x1, y1, var.equal=F, conf.level=.99)
```

```
t.test(var.equal=F, conf.level=.99, x1, y1)
```

# Default values

When creating a function, the programmer can supply default values.

```
makeList=function(mat,i=2){}
```

means that by default,  $i=2$ . The user can change the value when calling the function

```
myRowList=makeList(mymat,1)
```

# Missing Arguments

R functions can handle missing arguments two ways: either by providing a default expression in the argument list of definition, or by testing explicitly for missing arguments.

# Variable Number of Arguments

- The special argument name “...” in the function definition will match any number of arguments in the call.
- nargs() returns the number of arguments in the current call.

## Variable Number of Arguments

```
mean.of.all <- function(...) mean(c(...))  
mean.of.all(1:10,20:100,12:14)  
mean.of.means <- function(...){  
  means <- numeric()  
  for(x in list(...)) means <-  
    c(means,mean(x))  
  mean(means)  
}
```

## Session 2: Variable Number of Arguments

```
mean.of.means <- function(...){  
#computes the mean of the means of the  
# arguments  
  n <- nargs()  
  means <- numeric(n)  
  all.x <- list(...)  
  for(j in 1:n) means[j] <- mean(all.x[[j]])  
  mean(means)  
}
```

```
mean.of.means(1:10,10:100)
```

## Variables are local

Note that *any ordinary assignments done within the function are local and temporary and lost after exit from the function.*

```
fun1 <- function(){  
  a <- 1:10  
  rnorm(10)  
}  
a
```

# Output from a function

The last line of a function should be the output.

```
myfun=function(x){  
  y=3*x  
  y}
```

```
myfun=function(x){  
  y=3*x  
  list(x=x,y=y)  
}
```

# Edit

```
funname=edit(myfun)
```

brings up an edit window.

If funname is the same as myfun, you will save the edited function, overwriting myfun.

If you make a syntax error, when editing, R will send you an error message upon closing the edit window and will not save the changes.

```
funname=edit()
```

will bring up the most recent edit window

I use edit to see the code for R functions and to build new functions from existing functions

# Session 3: Editing a function

`read.table`

`my.readtable=edit(readtable)`

# Functions calling Functions

When a function calls another function, you need to be careful to understand which variables are local to which functions.

This is called "scope" and is discussed in the tutorial. (10.7)

One of the few differences between Splus and R pertain to "scope" and so it is important to understand this if you are trying to port functions between them.

# Garbage Collection

R does not always release memory after loading large data objects.

Releasing memory not currently in use is called garbage collection.

`gc()` will "do the trick".

So will quitting and restarting R.