Getting Started

March 14, 2017

Contents

1 Working with R

R is a case-sensitive, interpreted language. You can enter commands one at a time at the command prompt (>) or run a set of commands from a source file. There are a wide variety of data types, including vectors, matrices, data frames (similar to datasets), and lists (collections of objects). Most functionality is provided through built-in and user-created functions, and all data objects are kept in memory during an interactive session. Basic functions are available by default. Other functions are contained in packages that can be attached to a current session as needed.

R allows the = sign to be used for object assignments. However, you won't find many programs written that way, because it's not standard syntax, there are some situations in which it won't work, and R programmers will make fun of you. You can also reverse the assignment direction. For instance, rnorm(5) -> x is equivalent to the previous statement.

Comments are preceded by the # symbol . Any text appearing after the # is ignored by the R interpreter.

If you want to check the version of R, you just type 'R.version.string' on your console.

1.1 Workspace

The workspace is your current R working environment and includes any user-defined objects (vectors, matrices, functions, data frames, or lists). At the end of an R session, you can save an image of the current workspace that's automatically reloaded the next time R starts. You can find out what the current working directory is by using

```
getwd()

## [1] "C:/Users/XXXHHF/Desktop/Lecture Note_ADvanced/2. Getting Started/#FILE"
```

Also, you can set the current working directory by using the setwd() function.

If you plan to switch back to the default working directory during a session, then it is sensible to save the long default path before you change it, like this:

```
mine<-getwd()
setwd("c:\\temp")
...
setwd(mine)
```

If you want to view file names from R, then use the dir function like this:

```
dir("c:\\temp")
```

Other useful function for managing the R workspace are:

- check for memory assigned to R
  - memory.size() or memory.limit()
  - memory.limit(4000)
- List the objects in the current workspace. By default, it lists all objects in the global environment (i.e., the user's workspace)
  - ls() or objects()
  - object.size(x)
  - Calling objects("package:base") will show the names of more than a thousand objects defined in base, including the function objects() itself.

- Search for current packages
  - search()
- Remove (delete) one or more objects.
  - rm(objectlist)
- Remove all object
  - rm(list=ls(all=TRUE))
- Display your last # commands (default = 25).
  - history(#)
- Save the commands history to myfile (default = .Rhistory).
  - savehistory("myfile.Rhistory")
  - loadhistory("myfile.Rhistory")
- Save the workspace to myfile (default = .RData).
  - save.image("mydata.RData")
  - load("mydata.RData")
- Quit R. You'll be prompted to save the workspace.
  - q()

R can be used at various levels. Of course, standard arithmetic is available, and hence it can be used as a (rather sophisticated) calculator. It is also provided with a graphical system that writes on a large variety of devices. Furthermore, R is a full-featured programming language that can be employed to tackle all typical tasks for which other programming languages are also used. It connects to other languages, programs, and data bases, and also to the operating system; users can control all these from within R.

1.2 Packages

R comes with extensive capabilities right out of the box. But some of its most exciting features are available as optional modules that you can download and install. There are over 10000+ user-contributed modules called packages that you can download from http://cran.r-project.org/web/packages. Packages are collections of R functions, data, and compiled code in a well-defined format. The directory where packages are stored on your computer is called the library. The function

```
.libPaths()
```

```
## [1] "C:/Users/XXXHHF/Documents/R/win-library/3.3"
## [2] "C:/Program Files/R/R-3.3.2/library"
```

shows you where your library is located, and the function

```
library()
```

R comes with a standard set of packages (including base, datasets, utils, grDevices, graphics, stats, and methods). They provide a wide range of functions and datasets that are available by default. Other packages are available for download and installation. Once installed, they have to be loaded into the session in order to be used. The following command tells you which packages are loaded and ready to use.

```
search()

## [1] ".GlobalEnv"        "package:knitr"     "package:stats"
## [4] "package:graphics"  "package:grDevices" "package:utils"
## [7] "package:datasets"  "Autoloads"         "package:base"
```

There are a number of R functions that let you manipulate packages. To install a package for the first time, You can download and install the package with the command

```
install.packages("Rcmdr")
```

You only need to install a package once. But like any software, packages are often updated by their authors. Use the command update.packages() to update any packages that you've installed. Installing a package downloads it from a CRAN mirror site and places it in your library. To use it in an R session, you need to load the package using the library() command.

```
library("Rcmdr")
```

When you load a package, a new set of functions and datasets becomes available. Small illustrative datasets are provided along with sample code, allowing you to try out the new functionalities. The help system contains a description of each function (along with examples), and information on each dataset included. Entering help(package="package_name") provides a brief description of the package and an index of the functions and datasets included. For example,

```
help(package='Rcmdr')
```

An Example:

```
#1
install.packages("quantmod")
update.packages("quantmod")
#update.packages (ask=F)
library("quantmod")
getSymbols("^GSPC",src="yahoo",from="1990-1-1",to=Sys.Date())
print(head(GSPC));print(tail(GSPC))
View(GSPC)
getSymbols("BABA",src="yahoo",from="2015-1-1",to=Sys.Date())
print(head(BABA));print(tail(BABA))
View(BABA) rm(list=ls(all=TRUE))
save.image("baba.RData")
load("baba.RData")
detach("package:quantmod") #must be detached first
remove.packages("quantmod")
#2
help.start()
install.packages(c("Hmisc", "MASS"))
library("Hmisc")
```

```
library("MASS")
help(package = "Hmisc")
help(package = "MASS")
```

2 R as a Calculator

2.1 Arithmetic

You can add and subtract using the obvious + and - symbols, while division is achieved with a forward slash / and multiplication is done by using an asterisk * like this:

```
7 + 3 - 5 * 2

## [1] 0

3^2 / 2

## [1] 4.5
```

Notice from this example that multiplication (5×2) is done before the additions and subtractions. Powers (like squared or cube root) use the caret symbol ^ and are done before multiplication or division. The trigonometric functions in R measure angles in radians. A circle is 2π radians, and this is 360, so a right angle (90) is $\pi/2$ radians. R knows the value of π as pi:

```
pi

## [1] 3.141593

sin(pi/2)

## [1] 1

cos(pi/2)

## [1] 6.123032e-17
```

Integer quotients and remainders are obtained using the notation %/% (percent, divide, percent) and %% (percent, percent) respectively.

```
119 %/% 13

## [1] 9

119 %% 13

## [1] 2

15421 %% 7==0

## [1] TRUE
```

The standard arithmetic operators +, -, *, /, and ^ are available, where x^y yields x^y. The common mathematical functions, such as log(), exp(), sin(), asin(), cos(), acos(), tan(), atan(), sign(), sqrt(), abs(), min(), and max(), are also available. Specifically, log(x, base = a) returns the logarithm of x to base a, where a defaults to exp(1).

Exercise

```
log(exp(sin(pi/4)^2) * exp(cos(pi/4)^2))

## [1] 1
```

which also shows that pi is a built-in constant. There are further convenience functions, such as log10() and log2(), but here we shall mainly use log(). A full list of all options and related functions is available upon typing ?log, ?sin, etc. Additional functions useful in statistics and econometrics are gamma(), beta(), and their logarithms and derivatives. See ?gamma for further information.

In R, the basic unit is a vector, and hence all these functions operate directly on vectors. A vector is generated using the function c(), where c stands for "combine" or "concatenate".

```
x <- c(1.8, 3.14, 4, 88.169, 13)
length(x)

## [1] 5

2 * x + 3

## [1]   6.600   9.280  11.000 179.338  29.000

5:1 * x + 1:5

## [1]  10.000  14.560  15.000 180.338  18.000

log(x)

## [1] 0.5877867 1.1442228 1.3862944 4.4792554 2.5649494
```

Subsetting

It is often necessary to access subsets of vectors. This requires the operator [, which can be used in several ways to extract elements of a vector.

For example

```
xx <- c("a", "b", "c", "c", "d", "a")
xx[1]

## [1] "a"

xx[1:4]

## [1] "a" "b" "c" "c"

xx[xx > "a"]

## [1] "b" "c" "c" "d"

u <- xx > "a"
u

## [1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE

xx[u]

## [1] "b" "c" "c" "d"
```

```
x[c(1, 4)]

## [1]  1.800 88.169

x[-c(2, 3, 5)]

## [1]  1.800 88.169
```

In statistics and econometrics, there are many instances where vectors with special patterns are needed. R provides a number of functions for creating such vectors,

```
ones <- rep(1, 10)
ones

##  [1] 1 1 1 1 1 1 1 1 1 1

even <- seq(from = 2, to = 20, by = 2)
even

##  [1]  2  4  6  8 10 12 14 16 18 20

trend <- 1981:2005
trend

##  [1] 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994
## [15] 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005
```

Here, ones is a vector of ones of length 10, even is a vector containing the even numbers from 2 to 20, and trend is a vector containing the integers from 1981 to 2005. Since the basic element is a vector, it is also possible to concatenate vectors. Thus

```
c(ones, even)

##  [1]  1  1  1  1  1  1  1  1  1  1  2  4  6  8 10 12 14 16 18 20
```

Cumulative Sums and Products

As mentioned, the functions cumsum() and cumprod() return cumulative sums and products.

```
x <- c(12,5,13)
cumsum(x)

## [1] 12 17 30

cumprod(x)

## [1]  12  60 780
```

In x, the sum of the first element is 12, the sum of the first two elements is 17, and the sum of the first three elements is 30. The function cumprod() works the same way as cumsum(), but with the product instead of the sum.

Minima and Maxima

There is quite a difference between min() and pmin(). The former simply combines all its arguments into one long vector and returns the minimum value in that vector. In contrast, if pmin() is applied to two or more vectors, it returns a vector of the pair-wise minima, hence the name pmin.

```
z<-matrix(c(1,5,6, 2,3,13), nrow = 3, ncol = 2, byrow = F)
min(z[,1],z[,2])

## [1] 1

pmin(z[,1],z[,2])

## [1] 1 3 6
```

In the first case, min() computed the smallest value in (1,5,6,2,3,2). But the call to pmin() computed the smaller of 1 and 2, yielding 1; then the smaller of 5 and 3, which is 3; then finally the minimum of 6 and 2, giving 2. Thus, the call returned the vector (1,3,2).

The max() and pmax() functions act analogously to min() and pmin().

Function minimization/maximization can be done via nlm() and optim(). For example, let's find the smallest value of

$$f(x) = x^2 - \sin(x)$$

```
nlm(function(x) return(x^2-sin(x)),8)

## $minimum
## [1] -0.2324656
##
## $estimate
## [1] 0.4501831
##
## $gradient
## [1] 4.024558e-09
##
## $code
## [1] 1
##
## $iterations
## [1] 5
```

Here, the minimum value was found to be approximately -0.23, occurring at $x = 0.45$. A Newton-Raphson method (a technique from numerical analysis for approximating roots) is used, running five iterations in this case. The second argument specifies the initial guess, which we set to be 8. (This second argument was picked pretty arbitrarily here, but in some problems, you may need to experiment to find a value that will lead to convergence.)

Calculus

R also has some calculus capabilities, including symbolic differentiation and numerical integration, as you can see in the following example.

```
D(expression(exp(x^2)),"x") # derivative

## exp(x^2) * (2 * x)

integrate(function(x) x^2,0,1)

## 0.3333333 with absolute error < 3.7e-15
```

Matrix Crossproduct

Given matrices x and y as arguments, return a matrix cross-product. This is formally equivalent to (but usually slightly faster than) the call t(x) %*% y (crossprod) or x %*% t(y) (tcrossprod)

```
x<-1:5
y<-2*1:5
x*y

## [1]  2  8 18 32 50

t(x) %*% y

##      [,1]
## [1,]  110

crossprod(x,y)

##      [,1]
## [1,]  110

x %*% t(y)

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2    4    6    8   10
## [2,]    4    8   12   16   20
## [3,]    6   12   18   24   30
## [4,]    8   16   24   32   40
## [5,]   10   20   30   40   50

tcrossprod(x,y)

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2    4    6    8   10
## [2,]    4    8   12   16   20
## [3,]    6   12   18   24   30
## [4,]    8   16   24   32   40
## [5,]   10   20   30   40   50

outer(x,y)

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2    4    6    8   10
## [2,]    4    8   12   16   20
## [3,]    6   12   18   24   30
## [4,]    8   16   24   32   40
## [5,]   10   20   30   40   50
```

A 2 × 3 matrix containing the elements 1:6, by column, is generated via

```
A <- matrix(1:6, nrow = 2)
A

##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6

#?matrix
t(A)

##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6

dim(A)

## [1] 2 3

nrow(A)

## [1] 2

ncol(A)

## [1] 3
```

Subsetting a Matrix

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1 × 1 matrix. This behavior can be turned off by setting drop = FALSE.

```
B <- A[1,]
C <- A[,1]
D <- A[2,2]
D

## [1] 4

class(D)

## [1] "integer"

E <- A[2,2,drop=F]
E

##      [,1]
## [1,]    4

class(E)

## [1] "matrix"
```

```
B

## [1] 1 3 5

F <- A[1, ,drop=F];F

##      [,1] [,2] [,3]
## [1,]    1    3    5

A1 <- A[, -2]
A1

##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6

det(A1)

## [1] -4

eigen(A1)

## $values
## [1]  7.5311289 -0.5311289
##
## $vectors
##            [,1]       [,2]
## [1,] -0.6078802 -0.9561723
## [2,] -0.7940288  0.2928046
```

A1 is a square matrix, and if it is nonsingular it has an inverse. One way to check for singularity is to compute the determinant using the R function det(). Here, det(A1) equals −4; hence A1 is nonsingular. Alternatively, its eigenvalues (and eigenvectors) are available using eigen(). Here, eigen(A1) yields the eigenvalues 7.531 and −0.531, again showing that A1 is nonsingular.

```
solve(A1)

##      [,1]  [,2]
## [1,] -1.5  1.25
## [2,]  0.5 -0.25

A1 %*% solve(A1)

##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1

diag(4)

##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

```
diag(2, 4, 4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    0    0    0
## [2,]    0    2    0    0
## [3,]    0    0    2    0
## [4,]    0    0    0    2
```

```
diag(rep(c(1,2), c(3, 3)))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    0    0    0    0    0
## [2,]    0    1    0    0    0    0
## [3,]    0    0    1    0    0    0
## [4,]    0    0    0    2    0    0
## [5,]    0    0    0    0    2    0
## [6,]    0    0    0    0    0    2
```

```
B=svd(A1) #singular-value decomposition #X = U D V'
B$u %*% diag(B$d) %*% t(B$v)
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
```

```
qr(A1) #QR decomposition
```

```
## $qr
##            [,1]      [,2]
## [1,] -2.2360680 -7.602631
## [2,]  0.8944272 -1.788854
##
## $rank
## [1] 2
##
## $qraux
## [1] 1.447214 1.788854
##
## $pivot
## [1] 1 2
##
## attr(,"class")
## [1] "qr"
```

It is also possible to form new matrices from existing ones.

```
cbind(1, A1)
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    5
## [2,]    1    2    6
```

```
rbind(A1, diag(4, 2))
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    4    0
## [4,]    0    4
```

```
x <- runif(20)
summary(x)
hist(x)
```

Exercise

Solve this system:

$$\begin{aligned} x_1 + x_2 &= 2 \\ -x_1 + x_2 &= 4 \end{aligned}$$

Its matrix form is as follows:

$$\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

```
## [1] -1  3
```

Set Operations

R includes some handy set operations, including these:

- union(x,y): Union of the sets x and y
- intersect(x,y): Intersection of the sets x and y
- setdiff(x,y): Set difference between x and y, consisting of all elements of x that are not in y
- setequal(x,y): Test for equality between x and y
- c %in% y: Membership, testing whether c is an element of the set y
- choose(n,k): Number of possible subsets of size k chosen from a set of size n

```
x <- c(1,2,5)
y <- c(5,1,8,9)
union(x,y)

## [1] 1 2 5 8 9

intersect(x,y)

## [1] 1 5

setdiff(x,y)
```

```
## [1] 2

setdiff(y,x)

## [1] 8 9

setequal(x,y)

## [1] FALSE

setequal(x,c(1,2,5))

## [1] TRUE

2 %in% x

## [1] TRUE

2 %in% y

## [1] FALSE

choose(5,2)

## [1] 10
```

2.2 Complex numbers in R

The following are the special R functions that you can use with complex numbers.

```
z <- 3.5-8i
Re(z)

## [1] 3.5

Im(z)

## [1] -8

Mod(z)

## [1] 8.732125

Conj(z)

## [1] 3.5+8i

is.complex(z)

## [1] TRUE
```

Calculate the modulus (the distance from z to 0 in the complex plane by Pythagoras; if x is the real part and y is the imaginary part, then the modulus is $\sqrt{x^2+y^2}$)

2.3 Testing for equality with real numbers

You need to be careful in programming when you want to test whether or not two computed numbers are equal. R will assume that you mean 'exactly equal', and what that means depends upon machine precision. Most numbers are rounded to an accuracy of 53 binary digits. Typically therefore, two floating point numbers will not reliably be equal unless they were computed by the same algorithm, and not always even then.

```
x <- sqrt(2)
x * x == 2

## [1] FALSE

x * x - 2

## [1] 4.440892e-16
```

So how do we test for equality of real numbers?

```
x <- 0.3 - 0.2
y <- 0.1
x == y

## [1] FALSE

identical(x,y)

## [1] FALSE

all.equal(x,y)

## [1] TRUE
```

2.4 Four important things

1. Variable names in R are case sensitive, so y is not the same as Y.
2. Variable names should not begin with numbers (e.g. 1x) or symbols (e.g. %x).
3. Variable names should not contain blank spaces (use back.pay not back pay).

For example: Objects obtain values in R by assignment ('x gets a value'). This is achieved by the gets arrow <- which is a composite symbol made up from 'less than' and 'minus' with no space between them. Thus, to create a scalar constant x with value 5 we type x<-5, and not x = 5. Notice that there is a potential ambiguity if you get the spacing wrong. Compare our x <- 5, 'x gets 5', withx < - 5where there is a space between the 'less than' and 'minus' symbol. In R, this is actually a question, asking 'is x less than minus 5?' and, depending on the current value of x, would evaluate to the answer either TRUE or FALSE.

4. You cannot write a function that directly changes its arguments.

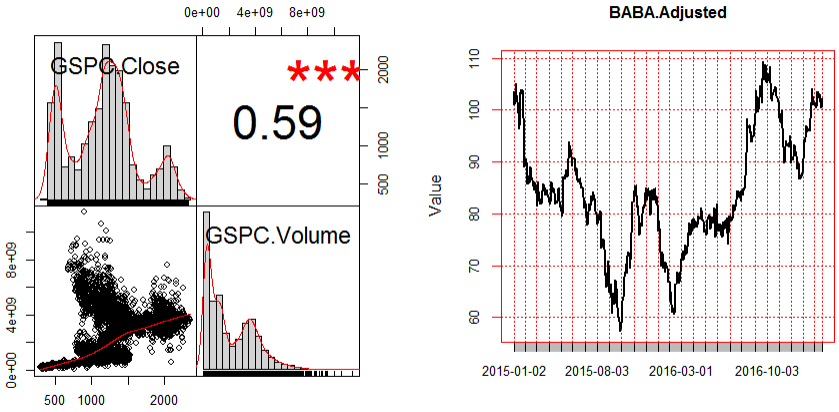
```
x <- c(13,5,12)
sort(x)

## [1]  5 12 13

x

## [1] 13  5 12
```

The argument to sort() does not change. If we do want x to change in this R code, the solution is to reassign the arguments:

```
x <- c(13,5,12)
x<-sort(x)
```

2.5 Homework

1. Read the manual of package "PerformanceAnalytics".
2. Draw the following graphic using "chart.Correlation". (left)
3. Draw the following graphic using "chart.TimeSeries". (right)



3 Data structures

The data types or modes that R can handle include numeric, character, logical (TRUE/FALSE), complex (imaginary numbers), and raw (bytes). R also has a wide variety of objects for holding data, including scalars, vectors, matrices, arrays, data frames, and lists.

```
attributes(mtcars)
?mtcars
```

3.1 Vectors

Vectors are one-dimensional arrays that can hold numeric data, character data, or logical data. The combine function c() is used to form the vector.

```
a <- c(1, 2, 5, 3, 6, -2, 4)
b <- c("one", "two", "three")
c <- c(TRUE, TRUE, TRUE, FALSE, TRUE, FALSE)
```

Here, a is numeric vector, b is a character vector, and c is a logical vector. Note that the data in a vector must only be one type or mode (numeric, character, or logical). You can't mix modes in the same vector. You can refer to elements of a vector using a numeric vector of positions within brackets.

```
a <- c(1, 2, 5, 3, 6, -2, 4)
a[c(1, 3, 5)]

## [1] 1 5 6

a[1:3]

## [1] 1 2 5
```

The colon operator used in the last statement is used to generate a sequence of numbers.

3.2 Matrices

A matrix is a two-dimensional array where each element has the same mode (numeric, character, or logical). Matrices are created with the matrix function . The general format is

```
myymatrix<-matrix(vector,nrow=number_of_rows,ncol=number_of_columns,
byrow=logical_value,dimnames=list(char_vector_rownames,char_vector_colnames))
```

where vector contains the elements for the matrix, nrow and ncol specify the row and column dimensions, and dimnames contains optional row and column labels stored in character vectors. The option byrow indicates whether the matrix should be filled in by row (byrow=TRUE) or by column (byrow=FALSE). The default is by column. The following listing demonstrates the matrix function.

```
y <- matrix(1:20, nrow = 5, ncol = 4)
y

##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20

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))
dim(mymatrix)

## [1] 2 2

mymatrix

##    C1 C2
## R1  1 26
## R2 24 68
```

You can identify rows, columns, or elements of a matrix by using subscripts and brackets. X[i,] refers to the ith row of matrix X, X[,j] refers to jth column, and X[i, j] refers to the ijth element, respectively.

```
y[2, ]

## [1]  2  7 12 17
```

```
y[, 2]

## [1]  6  7  8  9 10

y[1, 4]

## [1] 16

y[1 : 2, c(3, 4)]

##      [,1] [,2]
## [1,]   11   16
## [2,]   12   17

z <- as.vector(y)
dim(z) <- c(4, 5)
z

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
```

Avoiding Unintended Dimension Reduction

In the world of statistics, dimension reduction is a good thing, with many statistical procedures aimed to do it well. If we are working with, say, 10 variables and can reduce that number to 3 that still capture the essence of our data, we're happy. However, in R, something else might merit the name dimension reduction that we may sometimes wish to avoid. Say we have a four-row matrix and extract a row from it:

```
z

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20

r<-z[2,]
r

## [1]  2  6 10 14 18
```

This seems innocuous, but note the format in which R has displayed r. It's a vector format, not a matrix format. In other words, r is a vector of length 2, rather than a 1-by-2 matrix. We can confirm this in a couple of ways:

```
attributes(z)

## $dim
## [1] 4 5
```

```
attributes(r)

## NULL

str(z)

##  int [1:4, 1:5] 1 2 3 4 5 6 7 8 9 10 ...

str(r)

##  int [1:5] 2 6 10 14 18
```

Here, R informs us that z has row and column numbers, while r does not. Similarly, str() tells us that z has indices ranging in 1:4 and 1:2, for rows and columns, while r's indices simply range in 1:2. No doubt about it—r is a vector, not a matrix. This seems natural, but in many cases, it will cause trouble in programs that do a lot of matrix operations.

Fortunately, R has a way to suppress this dimension reduction: the drop argument. Here's an example, using the matrix z from above:

```
r <- z[2,, drop=FALSE]
r

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2    6   10   14   18

dim(r)

## [1] 1 5
```

Now r is a 1-by-2 matrix, not a two-element vector.

Extended Example: Image Manipulation

Image files are inherently matrices, since the pixels are arranged in rows and columns. If we have a grayscale image, for each pixel, we store the intensity— the brightness–of the image at that pixel. So, the intensity of a pixel in, say, row 28 and column 88 of the image is stored in row 28, column 88 of the matrix. For a color image, three matrices are stored, with intensities for red, green, and blue components, but we'll stick to grayscale here. For our example, let's consider an image of the Mount Rushmore National Memorial in the United States. Let's read it in, using the pixmap library.

```
library(pixmap)
mtrush1 <- read.pnm("mtrush1.pgm")
mtrush1
plot(mtrush1)
str(mtrush1)
```

The intensities in this class are stored as numbers ranging from 0.0 (black) to 1.0 (white), with intermediate values literally being shades of gray. For instance, the pixel at row 28, column 88 is pretty bright.

```
mtrush1@grey[28,88]
```

To determine the relevant rows and columns, you can use R's locator() function. In this manner, I found that Roosevelt's portion of the picture is in rows 84 through 163 and columns 135 through 177. (Note that row

numbers in pixmap objects increase from the top of the picture to the bottom, the opposite of the numbering used by locator().) So, to blot out that part of the image, we set all the pixels in that range to 1.0.

```
locator()
mtrush2 <- mtrush1
mtrush2@grey[84:163,135:177] <- 1
plot(mtrush2)
```

What if we merely wanted to disguise President Roosevelt's identity? We could do this by adding random noise to the picture. Here's code to do that:

```
blurpart <- function(img,rows,cols,q) {
     lrows <- length(rows)
     lcols <- length(cols)
     newimg <- img
     randomnoise <- matrix(nrow=lrows, ncol=ncols,runif(lrows*lcols))
     newimg@grey <- (1-q) * img@grey + q * randomnoise
     return(newimg)
}
```

As the comments indicate, we generate random noise and then take a weighted average of the target pixels and the noise. The parameter q controls the weight of the noise, with larger q values producing more blurring. The random noise itself is a sample from U(0,1), the uniform distribution on the interval (0,1). Note that the following is a matrix operation:

```
mtrush3 <- blurpart(mtrush1,84:163,135:177,0.65)
plot(mtrush3)
```

3.3 Arrays

Arrays are similar to matrices but can have more than two dimensions. They're created with an array function of the following form:

```
myarray <- array(vector, dimensions, dimnames)
```

where vector contains the data for the array, dimensions is a numeric vector giving the maximal index for each dimension, and dimnames is an optional list of dimension labels. The following listing gives an example of creating a three-dimensional (2x3x4) array of numbers.

```
z <- (1:24)
dim1 <- c("A1", "A2")
dim2 <- c("B1", "B2", "B3")
dim3 <- c("C1", "C2", "C3", "C4")
z <- array(1:24, c(2, 3, 4), dimnames = list(dim1, dim2, dim3))
z

## , , C1
##
##    B1 B2 B3
## A1  1  3  5
## A2  2  4  6
```

```
##
## , , C2
##
##    B1 B2 B3
## A1  7  9 11
## A2  8 10 12
##
## , , C3
##
##    B1 B2 B3
## A1 13 15 17
## A2 14 16 18
##
## , , C4
##
##    B1 B2 B3
## A1 19 21 23
## A2 20 22 24
```

Array Transposition

Transpose an array by permuting its dimensions and optionally resizing it.

```
A <- array(1:24,dim = c(2,3,4));A
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]   19   21   23
## [2,]   20   22   24
```

```
B <- aperm(A, c(2,3,1));B
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    7   13   19
## [2,]    3    9   15   21
## [3,]    5   11   17   23
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    2    8   14   20
## [2,]    4   10   16   22
## [3,]    6   12   18   24
```

```
C <- array(1:24,dim = c(3,4,2));C #diff
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
```

3.4 Data frames

A data frame is more general than a matrix in that different columns can contain different modes of data (numeric, character, etc.). It's similar to the datasets you'd typically see in SAS, SPSS, and Stata. Data frames are the most common data structure you'll deal with in R. A data frame is created with the data.frame() function :

```
mydata <- data.frame(col1, col2, col3)
```

where col1, col2, col3, ... are column vectors of any type.

```
patientID <- c(1, 2, 3, 4)
age <- c(25, 34, 28, 52)
diabetes <- c("Type1", "Type2", "Type1", "Type1")
status <- c("Poor", "Improved", "Excellent", "Poor")
patientdata1 <- cbind(patientID,age,diabetes,status)
patientdata2 <- data.frame(patientID, age, diabetes, status)
patientdata1
```

```
##      patientID age  diabetes status
## [1,] "1"       "25" "Type1"  "Poor"
## [2,] "2"       "34" "Type2"  "Improved"
## [3,] "3"       "28" "Type1"  "Excellent"
```

```
## [4,] "4"       "52" "Type1"  "Poor"

patientdata2

##   patientID age diabetes    status
## 1         1  25    Type1      Poor
## 2         2  34    Type2  Improved
## 3         3  28    Type1 Excellent
## 4         4  52    Type1      Poor

names(patientdata2)

## [1] "patientID" "age"       "diabetes"  "status"

is.matrix(patientdata1)

## [1] TRUE

is.matrix(patientdata2)

## [1] FALSE

patientdata2[1:2]

##   patientID age
## 1         1  25
## 2         2  34
## 3         3  28
## 4         4  52

patientdata2[c("diabetes", "status")]

##   diabetes    status
## 1    Type1      Poor
## 2    Type2  Improved
## 3    Type1 Excellent
## 4    Type1      Poor

patientdata2$age

## [1] 25 34 28 52
```

3.5 Factor

As you've seen, variables can be described as nominal, ordinal, or continuous. Nominal variables are categorical, without an implied order. Diabetes (Type1, Type2) is an example of a nominal variable. Even if Type1 is coded as a 1 and Type2 is coded as a 2 in the data, no order is implied. Ordinal variables imply order but not amount. Status (poor, improved, excellent) is a good example of an ordinal variable. You know that a patient with a poor status isn't doing as well as a patient with an improved status, but not by how much. Continuous variables can take on any value within some range, and both order and amount are implied. Age in years is a continuous variable and can take on values such as 14.5 or 22.8 and any value in between.

Categorical (nominal) and ordered categorical (ordinal) variables in R are called factors. Factors are crucial in R because they determine how data will be analyzed and presented visually. For vectors representing ordinal

variables, you add the parameter ordered=TRUE to the factor() function.

```
status <- factor(status, ordered=TRUE)
status

## [1] Poor      Improved  Excellent Poor
## Levels: Excellent < Improved < Poor

str(patientdata2)

## 'data.frame': 4 obs. of  4 variables:
##  $ patientID: num  1 2 3 4
##  $ age      : num  25 34 28 52
##  $ diabetes : Factor w/ 2 levels "Type1","Type2": 1 2 1 1
##  $ status   : Factor w/ 3 levels "Excellent","Improved",..: 3 2 1 3

summary(patientdata2)

##    patientID         age         diabetes       status
##  Min.   :1.00   Min.   :25.00   Type1:3   Excellent:1
##  1st Qu.:1.75   1st Qu.:27.25   Type2:1   Improved :1
##  Median :2.50   Median :31.00             Poor     :2
##  Mean   :2.50   Mean   :34.75
##  3rd Qu.:3.25   3rd Qu.:38.50
##  Max.   :4.00   Max.   :52.00
```

By default, factor levels for character vectors are created in alphabetical order. This worked for the status factor, because the order "Excellent," "Improved," "Poor" made sense. You can override the default by specifying a levels option.

```
status <- factor(status, order=TRUE, levels=c("Poor", "Improved", "Excellent"))
status

## [1] Poor      Improved  Excellent Poor
## Levels: Poor < Improved < Excellent

status2<-c("Poor", "Improved", "Excellent")
typeof(status2)

## [1] "character"

status3<-factor(status2,levels=c("Poor", "Improved", "Excellent"),labels=c("C","B","A"))
status3

## [1] C B A
## Levels: C B A
```

Be sure that the specified levels match your actual data values. Any data values not in the list will be set to missing.

Another Example

```
a<-c("A","C","B","C")
b<-as.factor(a)
b[5]<-"D"

## Warning in `[<-.factor`(`*tmp*`, 5, value = "D"):  invalid factor level, NA generated

c<-as.vector(b)
typeof(c)

## [1] "character"

c[5]<-"D"
b<-as.factor(c)
b

## [1] A C B C D
## Levels: A B C D
```

3.6 Logical operations

A crucial part of computing involves asking questions about things. Is one thing bigger than other? Are two things the same size? Questions can be joined together using words like 'and' 'or', 'not'. Questions in R typically evaluate to TRUE or FALSE but there is the option of a 'maybe' (when the answer is not available, NA). In R, < means 'less than', > means 'greater than', and ! means 'not'.

```
! #logical NOT
& #logical AND
| #logical OR
< #less than
<= #less than or equal to
> #greater than
= #greater than or equal to
== #logical equals (double =)
!= #not equal
#an abbreviation of identical(TRUE,x)
```

The function isTRUE() takes one argument. If that argument evaluates to TRUE, the function will return TRUE. Otherwise, the function will return FALSE.

```
isTRUE(6 > 4)

## [1] TRUE
```

The function identical() will return TRUE if the two R objects passed to it as arguments are identical.

```
identical('Twins', 'twins')

## [1] FALSE
```

You should also be aware of the xor() function, which takes two arguments. The xor() function stands for exclusive OR. If one argument evaluates to TRUE and one argument evaluates to FALSE, then this function will return TRUE, otherwise it will return FALSE.

```
xor(5 == 6, !FALSE)
```

```
## [1] TRUE
```

Use the which() function to find the indicies of ints that are greater than 7.

```
ints <- sample(10)
ints
```

```
##  [1]  1  2  6  7  9  8  3 10  5  4
```

```
ints>5
```

```
##  [1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE
```

```
which(ints > 7)
```

```
## [1] 5 6 8
```

Exercise:

```
(TRUE != FALSE) == !(6 == 7)
```

```
## [1] TRUE
```

```
TRUE & c(TRUE, FALSE, FALSE)
```

```
## [1]  TRUE FALSE FALSE
```

```
TRUE && c(TRUE, FALSE, FALSE)
```

```
## [1] TRUE
```

```
TRUE | c(TRUE, FALSE, FALSE)
```

```
## [1] TRUE TRUE TRUE
```

```
TRUE || c(TRUE, FALSE, FALSE)
```

```
## [1] TRUE
```

```
5 > 8 || 6 != 8 && 4 > 3.9
```

```
## [1] TRUE
```

Notice that AND operators are evaluated before OR operators.

3.7 Type conversions

R provides a set of functions to identify an object's data type and convert it to a different data type. Functions of the form is.datatype()return TRUE or FALSE, whereas as.datatype() converts the argument to that type.

```
ab <- c(1,2,3)
is.numeric(ab)
```

```
## [1] TRUE

is.vector(ab)

## [1] TRUE

ab <- as.character(ab)
is.numeric(ab)

## [1] FALSE

is.vector(ab)

## [1] TRUE

is.character(ab)

## [1] TRUE
```

You can use the functions listed in the following table to test for a data type and to convert it to a given type.

```
is.numeric()        as.numeric()
is.character()      as.character()
is.vector()         as.vector()
is.matrix()         as.matrix()
is.data.frame()     as.data.frame()
is.factor()         as.factor()
is.logical()        as.logical()
```

Different way to check Data type.

class(), mode() and typeof()

```
aaa<-gl(2,5)
class(aaa)     #Class of variable

## [1] "factor"

mode(aaa)      #type of data

## [1] "numeric"

typeof(aaa)    #one type of numeric

## [1] "integer"
```

3.8 List

Lists are the most complex of the R data types. A list allows you to gather a variety of (possibly unrelated) objects under one name. For example, a list may contain a combination of vectors, matrices, data frames, and even other lists. You create a list using the list() function.

```
mylist <- list(object1, object2)
```

[[is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame. $ is used to extract elements of a list or data frame by name; semantics are similar to that of [[.

```
Lst <- list(name="Fred", wife="Mary", child.ages=c(4,7,9))
Lst

## $name
## [1] "Fred"
##
## $wife
## [1] "Mary"
##
## $child.ages
## [1] 4 7 9

Lst[[2]]

## [1] "Mary"

Lst[[3]][2]

## [1] 7

#Lst[[1:2]] NG
Lst[["name"]]

## [1] "Fred"

Lst$name

## [1] "Fred"
```

Adding and Deleting List Elements

New components can be added after a list is created.

```
#add component
z <- list(a="abc",b=12)
z

## $a
## [1] "abc"
##
## $b
## [1] 12

z$c <- "sailing" # add a c component
# did c really get added?
z

## $a
## [1] "abc"
```

```
##
## $b
## [1] 12
##
## $c
## [1] "sailing"

#You can delete a list component by setting it to NULL.
z$b <- NULL
z

## $a
## [1] "abc"
##
## $c
## [1] "sailing"
```

Partial Matching

Partial matching of names is allowed with [[and $.

```
x <- list(aardvark = 1:5)
x$a

## [1] 1 2 3 4 5

x[["a"]]

## NULL

x[["a", exact = FALSE]]

## [1] 1 2 3 4 5
```

Extended Example: Text Concordance

Web search and other types of textual data mining are of great interest today. Let's use this area for an example of R list code. We'll write a function called findwords() that will determine which words are in a text file and compile a list of the locations of each word's occurrences in the text. This would be useful for contextual analysis.

Suppose our input file, testconcord.txt, has the following contents:

- The [1] here means that the first item in this line of output is item 1. In this case, our output consists of only one line (and one item), so this is redundant, but this notation helps to read voluminous output that consists of many items spread over many lines. For example, if there were two rows of output with six items per row, the second row would be labeled [7].

In order to identify words, we replace all nonletter characters with blanks and get rid of capitalization. The new file, testconcorda.txt, looks like this:

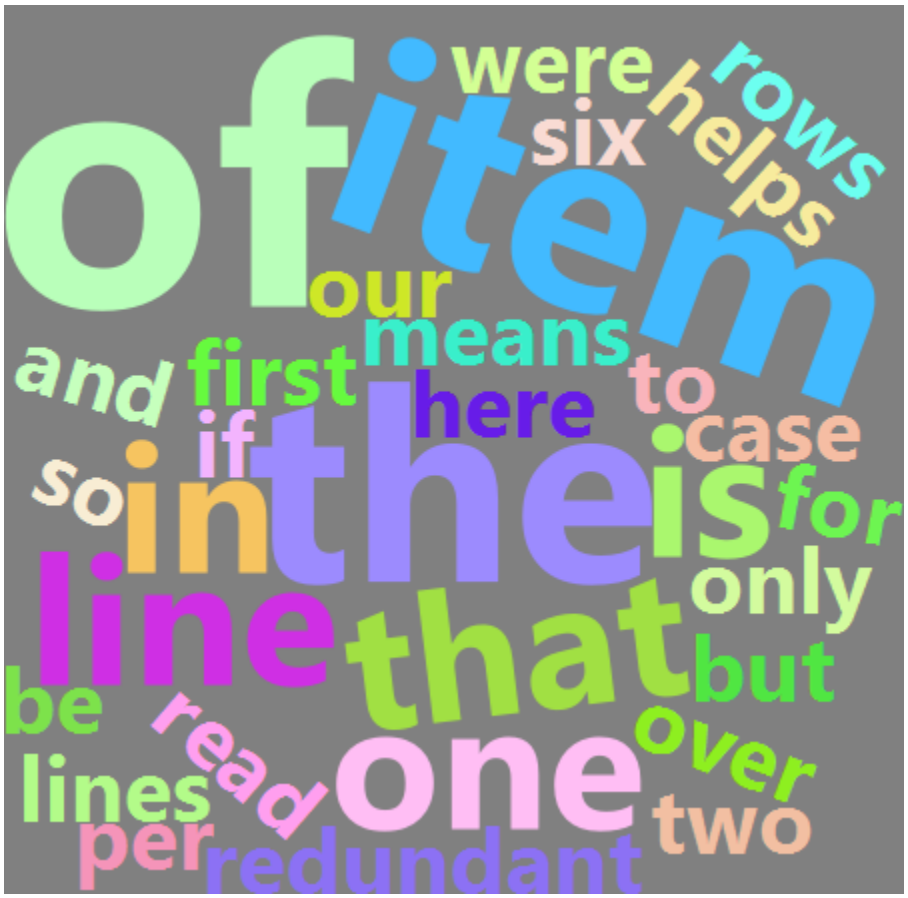
- the here means that the first item in this line of output is item in this case our output consists of only one line and one item so this is redundant but this notation helps to read voluminous output that consists of many items spread over many lines for example if there were two rows of output with six items per row the second row would be labeled

Then, for instance, the word item has locations 7, 14, and 27, which means that it occupies the seventh, fourteenth, and twenty-seventh word positions in the file.

```
findwords <- function(tf) {
   # read in the words from the file, into a vector of mode character
     txt <- scan(tf,"")
     wl <- list()
     for (i in 1:length(txt)) {
         wrd <- txt[i]  # i-th word in input file
         wl[[wrd]] <- c(wl[[wrd]],i)
     }
     return(wl)
}
txt <- scan("C:/Users/XXXHHF/Desktop/Lecture Note_ADvanced/2. Getting Started/Data/testconcorda.txt",
txt
class(txt)
words<-findwords("C:/Users/XXXHHF/Desktop/Lecture Note_ADvanced/2. Getting Started/Data/testconcorda.
```

Exercise

Using data "testconcorda.txt" and package "wordcloud2", draw the following plot.



3.9 Formulas

Formulas are constructs used in various statistical programs for specifying models. In R, formula objects can be used for storing symbolic descriptions of relationships among variables, such as the ~ operator in the formation of a formula:

```
f <- y ~ x
class(f)

## [1] "formula"
```

3.10 Missing Values

Missing values are denoted by NA or NaN for undefined mathematical operations.

- is.na() is used to test objects if they are NA
- is.nan() is used to test for NaN
- NA values have a class also, so there are integer NA, character NA, etc.
- A NaN value is also NA but the converse is not true

```
x <- 1/0
x

## [1] Inf

x <- 0/0
x

## [1] NaN
```

```
x <- c(1, 2, NaN, NA, 4)
is.na(x)

## [1] FALSE FALSE  TRUE  TRUE FALSE

is.nan(x)

## [1] FALSE FALSE  TRUE FALSE FALSE
```

3.11 attach(), detach(), with()

The attach() function adds the data frame to the R search path. When a variable name is encountered, data frames in the search path are checked in order to locate the variable.

```
summary(mtcars$mpg)
plot(mtcars$mpg, mtcars$disp)
```

This could also be written as

```
attach(mtcars)
summary(mpg)
plot(mpg, disp)
detach(mtcars)
```

The limitations with this approach are evident when more than one object can have the same name. An alternative approach is to use the with() function.

```
with(mtcars, {
summary(mpg, disp, wt)
plot(mpg, disp)
plot(mpg, wt)
})
```

The limitation of the with() function is that assignments will only exist within the function brackets. If you need to create objects that will exist outside of the with() construct, use the special assignment operator <<- instead of the standard one (<-). It will save the object to the global environment outside of the with() call.

```
with(mtcars, {
nokeepstats <- summary(mpg)
keepstats <<- summary(mpg)
})
nokeepstats
keepstats
```

3.12 Attach Misery

When we work with a single dataset, we use the attach command, as it is more convenient. However, there are rules that must be followed, and we see manyRnovices breaking these rules.

Entering the Same attach Command Twice

At this point it is useful to consult the help file of the attach function. It says that the function adds the data frame Parasite to its search path, and, as a result, the variables in the data frame can be accessed without using the $ notation. However, by attaching the data frame twice, we have made available two copies of each variable. If we make changes to, for example, Length and, subsequently, use Length in a linear regression analysis, we have no way of ensuring that the correct value is used.

Attaching a Data Frame and Demo Data

The typical use of such a package is that the reader accesses the help file of certain functions, goes to the examples at the end of a help file, and copies and runs the code to see what it does. Most often, the code from a help file loads a dataset from the package using the data function, or it creates variables using a random number generator. We have also seen help file examples that contain the attach and detach functions. When using these it is not uncommon to neglect to copy the entire code; the detach command may be omitted, leaving the attach function active. Once you understand the use of the demonstrated function, it is time to try it with your own data. If you have also applied the attach function to your data, you may end up in the scenario outlined in the previous section. Problems may also occur if demonstration data loaded with the data function contain some of the same variable names used as your own data files.

[See A Beginner's Guide to R for more details, P197]

4 Data input

4.1 Input and Output

Input

The source("filename") function submits a script to the current session. If the filename doesn't include a path, the file is assumed to be in the current working directory.

To save and use the .RData file, for example, you can use command as

```
auto<-read.xlsx("C:\\Users\\XXXHHF\\Documents\\R\\workfile\\auto.xlsx",
                1,header=TRUE,as.data.frame=TRUE)
View(auto)
save(auto,file="C:\\Users\\XXXHHF\\Documents\\R\\workfile\\auto.RData")
remove(auto)
load("C:\\Users\\XXXHHF\\Documents\\R\\workfile\\auto.RData")
```

Graphic Output

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

```
pdf("filename.pdf")           #PDF file
win.metafile("filename.wmf") #Windows metafile
png("filename.png")           #PBG file
jpeg("filename.jpg")          #JPEG file
bmp("filename.bmp")           #BMP file
postscript("filename.ps")     #PostScript file
```

4.2 Data input

R provides a wide range of tools for importing data. The definitive guide for importing data in R is the R Data Import/Export manual available at http://cran.r-project.org/doc/manuals/R-data.pdf. R can import data from the keyboard, from flat files, from Microsoft Excel and Access, from popular statistical packages, from specialty formats, and from a variety of relational database management systems.

Entering data from the keyboard

The edit() function in R will invoke a text editor that will allow you to enter your data manually.

```
mydata <- data.frame(age=numeric(0),
          gender=character(0), weight=numeric(0))
mydata <- edit(mydata)
```

Or

```
x=numeric(10)
data.entry(x)
```

Or

```
aa<-scan()
```

If you want to read in a single line from the keyboard, readline() is very handy.

```
inits <- readline("type your initials: ")
```

Scan

For dataframes, read.table is superb. But

```
read.table(file="C:\\Users\\XXXHHF\\Documents\\LYX\\2. Getting Started\\Data\\rt.txt")
```

It simply cannot cope with lines having different numbers of fields. However, scan and readLines come into their own with these complicated, non-standard files. It is much less friendly for reading dataframes than read.table, but it is substantially more flexible for tricky or non-standard files.

```
rt<-scan(file="C:\\Users\\XXXHHF\\Documents\\LYX\\2. Getting Started\\Data\\rt.txt")
rt<-scan(file="C:\\Users\\XXXHHF\\Documents\\LYX\\2. Getting Started\\Data\\rt.txt",sep="\n")
rt<-scan(file="C:\\Users\\XXXHHF\\Documents\\LYX\\2. Getting Started\\Data\\rt.txt",sep="\t")
rt<-scan(file=file.choose(),sep="\t")
```

The only difference between the three calls to scan is in the specification of the separator. The first uses the default which is blanks or tabs (the 10 items are the 10 numbers that we are interested in, but information about their grouping has been lost). The second uses the new line "\n" control character as the separator (the contents of each of the five lines have been stripped out and trimmed to create meaningless numbers, except for 138 and 59 which were the only numbers on their respective lines). The third uses tabs "\t" as separators (we have no information on lines, but at least the numbers have retained their integrity, and missing values (NA) have been used to pad out each line to the same length, 4.

```
sapply(1:5, function(i)
       as.numeric(na.omit(
          scan("C:\\Users\\XXXHHF\\Documents\\LYX\\2. Getting Started\\Data\\rt.txt",
          sep="\t",quiet=T)[(4*i-3):
              (4*i)])))
```

Read data into a vector or list from the console or file.

```
weight<-scan(file="C:\\Users\\XXXHHF\\Documents\\LYX\\2. Getting Started\\Data\\weight.data")
Forest<-scan(file="C:\\Users\\XXXHHF\\Documents\\
             LYX\\2. Getting Started\\Data\\ForestData.txt",what=double(),skip=1)
```

Importing data from a delimited text file

Read.table

You can import data from delimited text files using read.table() , a function that reads a file in table format and saves it as a data frame.

```
mydataframe <- read.table(file, header=logical_value,
                sep="delimiter", row.names="name")
```

where file is a delimited ASCII file , header is a logical value indicating whether the first row contains variable names (TRUE or FALSE), sep specifies the delimiter separating data values, and row.names is an optional parameter specifying one or more variables to represent row identifiers. Note that the sep parameter allows you

to import files that use a symbol other than a comma to delimit the data values. You could read tab-delimited files with sep="\t". The default is sep="", which denotes one or more spaces, tabs, new lines, or carriage returns.

The read.table function is one of the most commonly used functions for reading data. It has a few important arguments:

- file, the name of a file, or a connection
- header, logical indicating if the file has a header line
- sep, a string indicating how the columns are separated
- colClasses, a character vector indicating the class of each column in the dataset
- nrows, the number of rows in the dataset
- comment.char, a character string indicating the comment character
- skip, the number of lines to skip from the beginning
- stringsAsFactors, should character variables be coded as factors?

```
Forest<-read.table(file="ForestData.txt",header=TRUE)
str(Forest)
names(Forest)
Forest<-read.table(file="ForestData.txt",header=TRUE,stringsAsFactors=FALSE)
Forest<-read.table(file="ForestData.txt",header=TRUE,
                   colClass=c("integer","integer","character",
                   "character","double","integer","double","double","double"))
```

Option stringsAs Factors=FALSE will turn this behavior off for all character variables. Alternatively, you can use the colClasses option to specify a class (for example, logical, numeric, character, factor) for each column. See help(read.table) for details.
R will automatically

- skip lines that begin with a #
- figure out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table Telling R all these things directly makes R run faster and more efficiently.

There are three things to remember:

- The whole path and file name needs to be enclosed in double quotes: "c:\\abc.txt".
- header=T says that the first row contains the variable names (T stands for TRUE).
- Always use double \\ rather than \ in the file path definition.

Finally, there can be problems when you are trying to read variables that consist of character strings containing blank spaces (as in files containing place names). You can use read.table so long as you export the file from the spreadsheet using commas to separate the fields, and you tell read.table that the separators are commas using sep="," (to override the default blanks or tabs (\t):

```
map <- read.table("c:\\temp\\bowens.csv",header=T,sep=",")
```

However, it is quicker and easier to use read.csv in this case (see below).

Read.delim

You can save time by using read.delim, because then you can omit header=T:

```
Forest<-read.delim(file="ForestData.txt")
```

Separators and decimal points

The default field separator character in read.table is sep=" ". This separator is white space, which is produced by one or more spaces, one or more tabs \t, one or more newlines \n, or one or more carriage returns. If you do have a different separator between the variables sharing the same line (i.e. other than a tab within a.txt file) then there may well be a special read function for your case.

- for comma-separated fields use read.csv("c:\\temp\\file.csv");
- for semicolon-separated fields read.csv2("c:\\temp\\file.csv");
- for tab-delimited fields with decimal points as a commas, use read.delim2("c:\\temp\\file.txt").

More details, see ?read.table.

Importing data from a URL

```
data2 <- read.table ("http://www.bio.ic.ac.uk/research/mjcraw/therbook/data/cancer.txt", header=T)
head(data2)
```

Or

```
urlhandle=url("http://www.math.smith.edu/r/testdata")
ds=readLines(urlhandle)
```

Or

```
ds=read.table("http//www.math.smith.edu/r/testdata")
```

Importing data from Excel

The best way to read an Excel file is to export it to a comma-delimited file from within Excel and import it to R using the method described earlier.

```
install.packages("xlsx")
library("xlsx")
Forest<-read.xlsx("ForestData.xlsx",1,header=TRUE,as.data.frame=TRUE)
str(Forest)
levels(Forest$month)
Forest$month<-factor(Forest$month,order=TRUE,
              levels=c("jan","feb","mar","apr","may","jun",
              "jul","aug","sep","oct","nov","dec"))
levels(Forest$month)
```

Importing data from stata

Using foreign package, you can import file from stata.

```
library("foreign")
auto<-read.dta("C:\\Users\\XXXHHF\\Documents\\LYX\\1.1. Getting Started\\Data\\auto.dta")
head(auto)
```

Importing data from clipboard

Using the following command, you can import data from clipboard. (using auto.xlsx)

```
TRData <- read.table("clipboard", header = T, sep = "\t")
```

Using the Built-in data from R

There are many built-in data sets within the datasets package of R.

```
data()
```

To see the data sets in extra installed packages as well

```
data(package = .packages(all.available = TRUE))
```

Many of the contributed packages contain data sets, and you can viewtheir names using the try function.

```
try(data(package = "MASS") )
```

Other Commands:

```
#help(dataname)
help(CO2)
#data(dataname,package="rpart")
help(mtcars)
data(mtcars)
help(mtcars)
```

For further information of import data, also see http://www.biosino.org/R/R-doc/onepage/R-data_cn.html for details.

Example

Task 1: Download single file "http://www.stateair.net/web/assets/historical/1/Beijing_2017_HourlyPM25_created20170301.csv" and read the csv file into R environment.

```
u <- "http://www.stateair.net/web/historical/1/1.html"
page_parse <- htmlParse(u, encoding = "utf-8")
links <- getHTMLLinks(u)
links
#http://www.stateair.net/web/assets/historical/1/Beijing_2017_HourlyPM25_created20170301.csv
#http://www.stateair.net/web/assets/historical/1/Beijing_2016_HourlyPM25_created20170201.csv
```

```
#......
filename<-"Beijing_2017_HourlyPM25_created20170301.csv"
u1 <- "http://www.stateair.net/web/assets/historical/1/"
u.full <- paste(u1,filename,sep = "")
#Beijing.2017<-read.csv(u.full)
Beijing.2017<-read.csv(u.full,skip = 3,encoding = "utf-8")
class(Beijing.2017)
View(Beijing.2017)
head(Beijing.2017$Unit)
#Changing encoding
?iconv
Beijing.2017$Unit <- iconv(Beijing.2017$Unit,from="windows-1252",to="GB2312")
head(Beijing.2017$Unit)
```

Task 2: Download all files from URL "http://www.stateair.net/web/historical/1/1.html" #and read all csvs file into R environment as one data. (save as data.frame)

```
#download multi-files
#1 Create CSV file URL
linkcsv<-links[str_detect(string=links,".csv")]
linkcsv
class(linkcsv)
linkcsv_list <- as.list(linkcsv)
linkcsv_list
# execute download
all.data <- data.frame()
for (i in 1:length(linkcsv_list)) {
all.data <- rbind(all.data,read.csv(linkcsv_list[[i]],skip = 3,encoding = "utf-8"))
Sys.sleep(3)
}
class(all.data)
head(all.data);tail(all.data)
```

4.3 Data output

Saving history

It is very useful to be able to see all of the lines of R code that one has typed during a particular session. You may want to copy the lines into a text editor to make minor alterations, or you may simply want to paste multiple lines back into R to repeat certain operations. To see all of your lines of input code just type:

```
history(Inf)
```

This opens a window called R History through which you can scroll, highlight and copy using Ctrl+C. You could then open a new Untitled R Editor window (File > New Script) and paste the selected lines of code using Ctrl+V. Alternatively, you might want to save the entire history to file, for use on a subsequent occasion:

```
savehistory(file = "history.txt")
```

To retrieve the history for use on another occasion use

```
loadhistory(file = "history.txt")
```

Then you can access it by history(Inf)in the new session.

Saving graphics

For speed and simplicity, you can click on a graph (the bar on top of the R Graphics Device goes darker blue) then press Ctrl+C (to copy the graph), then switch to a word processor and paste using Ctrl+V. For publication-quality graphics, however, you will want to save each figure in a separate file as a PDF or PostScript file. There are a great many options (see ?pdf and ?postscript for details) but the basics are very simple. Here we set the graphics device to produce a PDF:

```
pdf("c:\\temp\\fig1.pdf")
data(mtcars)
plot(mtcars$wt~mtcars$mpg)
```

Now, any plot directives are sent to this file. To switch off writing graphics to file, type:

```
dev.off()
```

Exporting data to ascii file

In addition to the read.table command, R also has a write.table command. With this function, you can export numerical information to an ascii file.

```
For.sep <- Forest[Forest$month == "sep", ]
write.table(For.sep, file = "For_sep.txt", sep = " ", quote = FALSE, append = FALSE, na = "NA")
```

The first argument in the write.table function is the variable that you want to export, and, obviously, you also need a file name. The sep = " " ensures that the data are separated by a space, the quote = FALSE avoids quotation marks around character strings (headings), na="NA" allows you to specify how missing values are represented, and append=FALSE opens a new file.

Save as a R data file

```
save(For.sep,file="For_sep.Rdata")
load("For_sep.Rdata")
save(list=ls(all=TRUE),file="all.Rdata")
```

Pasting into an Excel spreadsheet

Writing a vector from R to the Windows Clipboard uses the function writeClipboard(x) where x is a single vector of characters, so you need to build up a spreadsheet in Excel by pasting (Ctrl+V) one column at a time.

```
writeClipboard(as.character(factor.name))
```

Go into Excel and press Ctrl+V.

Checking files from the command line

It can be useful to check whether a given filename exists in the path where you think it should be.

```
file.exists("c:\\temp\\Decay.txt")
```

For more on file handling, see ?files.

4.4 Annotating datasets

Data analysts typically annotate datasets to make the results easier to interpret. Typically annotation includes adding descriptive labels to variable names and value labels to the codes used for categorical variables.

Variables labels

Unfortunately, R's ability to handle variable labels is limited. One approach is to use the variable label as the variable's name and then refer to the variable by its position index. The following code renames age to "Age at hospitalization (in years)".

```
names(patientdata2)[2] <- "Age at hospitalization (in years)"
patientdata2[2]

##   Age at hospitalization (in years)
## 1                                25
## 2                                34
## 3                                28
## 4                                52
```

Clearly this new name is too long to type repeatedly. You may be better off trying to come up with better names (for example, admissionAge).

Value labels

```
gender <- c(1,2,1,2)
patientdata2 <- data.frame(patientID, gender, age, diabetes, status)
patientdata2$gender <- factor(patientdata2$gender, levels = c(1,2), labels = c("male", "female"))
patientdata2$gender

## [1] male   female male   female
## Levels: male female
```

Extended Example: Reading PUMS Census Files

The U.S. Census Bureau makes census data available in the form of Public Use Microdata Samples (PUMS). The term microdata here means that we are dealing with raw data and each record is for a real person, as opposed to statistical summaries. Data on many, many variables are included. The data is organized by household. For each unit, there is first a Household record, describing the various characteristics of that household, followed by one Person record for each person in the household. Character positions 106 and 107 (with numbering starting at 1) in the Household record state the number of Person records for that household. (The number can be very large, since some institutions count as households.) To enhance the integrity of the data, character position 1 contains H or P to confirm that this is a Household or Person record. So, if you read an H record, and it tells you there are three people in the household, then the following three records should be P records, followed by another H record; if not, you've encountered an error.

```
# reads in PUMS file pf, extracting the Person records, returning a data
# frame; each row of the output will consist of the Household serial
# number and the fields specified in the list flds; the columns of
# the data frame will have the names of the indices in flds
extractpums <- function(pf,flds) {
   dtf <- data.frame()  # data frame to be built
   con <- file(pf,"r")  # connection
   # process the input file
   repeat {
        hrec <- readLines(con,1)  # read Household record
        if (length(hrec) == 0) break  # end of file, leave loop
        # get household serial number
        serno <- intextract(hrec,c(2,8)) # how many Person records?
        npr <- intextract(hrec,c(106,107))
        if (npr > 0)
           for (i in 1:npr) {
              prec <- readLines(con,1)  # get Person record
              # make this person's row for the data frame
              person <- makerow(serno,prec,flds)
              # add it to the data frame
              dtf <- rbind(dtf,person)
           }
   }
   return(dtf)
}
# set up this person's row for the data frame
makerow <- function(srn,pr,fl) {
   l <- list()
   l[["serno"]] <- srn
   for (nm in names(fl)) {
      l[[nm]] <- intextract(pr,fl[[nm]])
   }
   return(l)
}
# extracts an integer field in the string s, in character positions # rng[1] through rng[2]
intextract <- function(s,rng) {
     fld <- substr(s,rng[1],rng[2])
     return(as.integer(fld))
}
```

At the beginning of extractpums(), we create an empty data frame and set up the connection for the PUMS file read. The main body of the code then consists of a repeat loop. Within the repeat loop, we alternate reading a Household record and reading the associated Person records. The number of Person records for the current Household record is extracted from columns 106 and 107 of that record, storing this number in npr. That extraction is done by a call to our function intextract(). The for loop then reads in the Person records one by one, in each case forming the desired row for the output data frame and then attaching it to the latter via rbind().

Note how makerow() creates the row to be added for a given person. Here the formal arguments are srn for the household serial number, pr for the given Person record, and fl for the list of variable names and column fields. When makerow() executes, fl will be a list with two elements, named Gender and Age. The string pr, the current Person record, will have Gender in column 23 and Age in columns 25 and 26. We call intextract() to pull out the desired numbers. The intextract() function itself is a straightforward conversion of characters to numbers, such as converting the string "12" to the number 12.

In this data set, gender is in column 23 and age in columns 25 and 26. In the example, our filename is pumsa. The following call creates a data frame consisting of those two variables.

```
pumsdf <- extractpums("pumsa",list(Gender=c(23,23),Age=c(25,26)))
```

Appendix A: Some commom mistake in R programming

There are some common mistakes made frequently by both beginning and experienced R programmers. If your program generates an error, be sure the check for the following:

1. Using the wrong case —help(), Help(), and HELP() are three different functions (only the first will work).
2. Forgetting to use quote marks when they're needed —install.packages- ("gclus") works, whereas install.packages(gclus) generates an error.
3. Forgetting to include the parentheses in a function call —for example, help() rather than help. Even if there are no options, you still need the ().
4. Using the \ in a pathname on Windows —R sees the backslash character as an escape character. setwd ("c:\mydata") generates an error. Use setwd("c:/mydata") or setwd("c:\\mydata") instead.
5. Using a function from a package that's not loaded —The function order. clusters() is contained in the gclus package. If you try to use it before loading the package, you'll get an error.

The error messages in R can be cryptic, but if you're careful to follow these points, you should avoid seeing many of them.

Appendix B: Some points

There are some features of the language you should be aware of:

1. The period (.) has no special significance in object names. But the dollar sign ($) has a somewhat analogous meaning, identifying the parts of an object. For example, A$x refers to variable x in data frame A.
2. R doesn't provide multiline or block comments . You must start each line of a multiline comment with #. For debugging purposes, you can also surround code that you want the interpreter to ignore with the statement if(FALSE){. . . }. Changing the FALSE to TRUE allows the code to be executed.
3. Indices in R start at 1, not at 0.

4. Assigning a value to a nonexistent element of a vector, matrix, array, or list will expand that structure to accommodate the new value. For example, consider the following:

```
x <- c(8, 6, 4)
x[7] <- 10
x

## [1]  8  6  4 NA NA NA 10
```

Appendix C: Some useful functions for working with data object

```
length(object)    #Number of elements/components.
dim(object)       #Dimensions of an object.
str(object)       #Structure of an object.
class(object)     #Class or type of an object.
mode(object)      #How an object is stored.
names(object)     #Names of components in an object.
c(object, object,...) #Combines objects into a vector.
cbind(object, object, ...) #Combines objects as columns.
rbind(object, object, ...) #Combines objects as rows.
object Prints the object. head(object)     #Lists the first part of the object.
tail(object)     #Lists the last part of the object.
ls()             #Lists current objects.
rm(object, object, ...) #Deletes one or more objects.
newobject <- edit(object) #Edits object and saves as newobject.
fix(object)     #Edits in place.
```

Appendix D: Escape sequences

Formatting is controlled using escape sequences, typically within double quotes:

```
\n newline
\r carriage return
\t tab character
\b backspace
\a bell
\f form feed
\v vertical tab
```

9. 矩阵的奇异值分解

函数 `svd(A)` 是对矩阵 A 作奇异值分解，即 $A = UDV^T$, 其中 U, V 是正交阵， D 为对角阵，也就是矩阵 A 的奇异值． `svd(A)` 的返回值也是列表，`svd(A)$d` 表示矩阵 A 的奇异值，即矩阵 D 的对角线上的元素． `svd(A)$u` 对应的是正交阵 U, `svd(A)$v` 对应的是正交阵 V. 例如，

```
> svdA<-svd(A); svdA
$d
[1] 17.4125052  0.8751614  0.1968665
$u
           [,1]        [,2]       [,3]
[1,] -0.2093373  0.96438514  0.1616762
[2,] -0.5038485  0.03532145 -0.8630696
[3,] -0.8380421 -0.26213299  0.4785099
$v
           [,1]         [,2]       [,3]
[1,] -0.4646675 -0.833286355  0.2995295
[2,] -0.5537546  0.009499485 -0.8326258
[3,] -0.6909703  0.552759994  0.4658502
> attach(svdA)
> u %*% diag(d) %*% t(v)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8   10
```

在上面的语句中， `attach(svdA)` 是说明下面的变量 `u`, `v`, `d` 是附属于 `svdA` 的，关于 `attach()` 函数的使用方法将在 2.6.2 节作详细介绍.