KAAU
جامعة الملك عبد العزيز

كلية العلوم

قسم الإحصاء

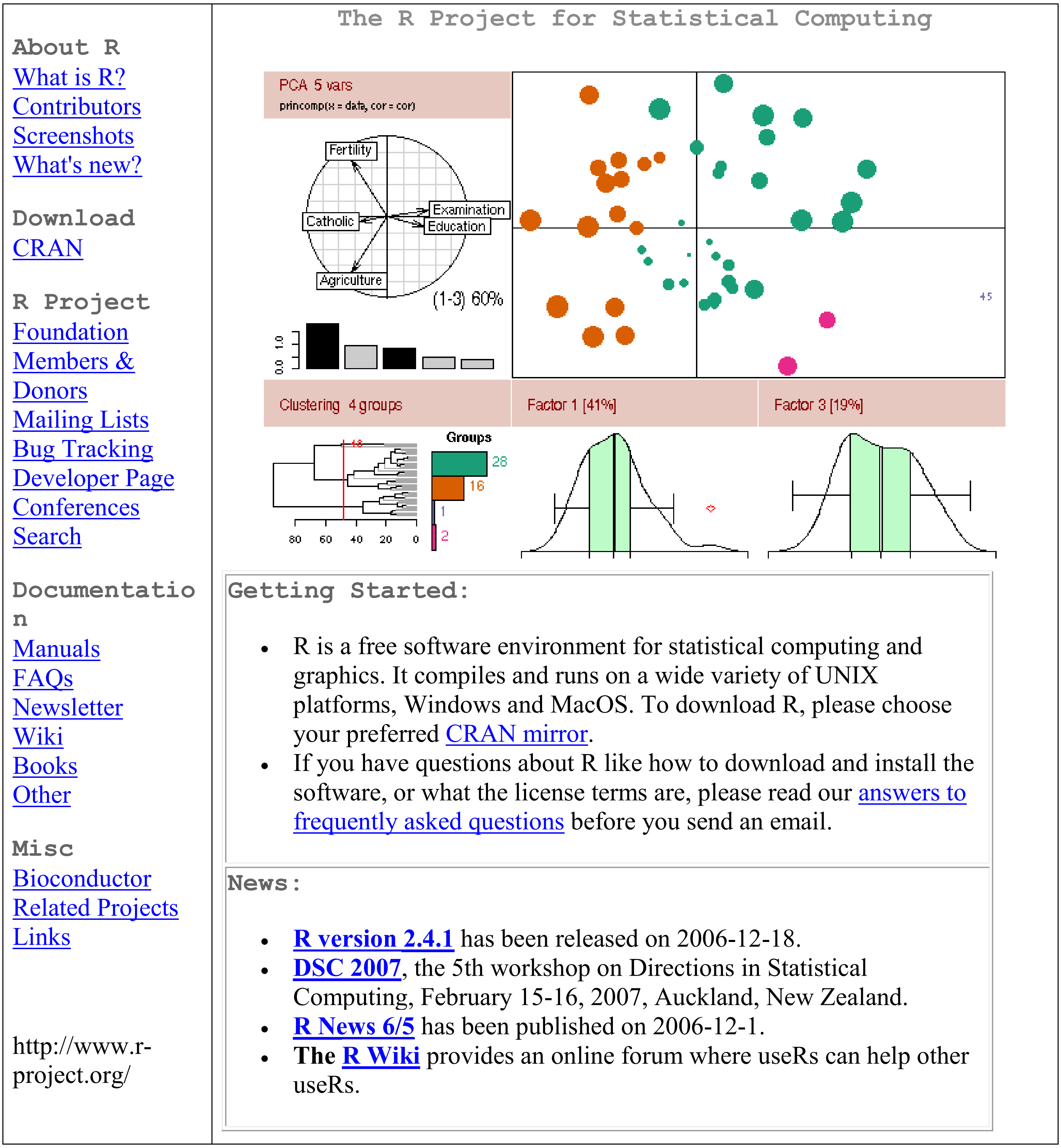
معمل إحصائـــي 2
STAT 442

الفصل الدراسي الثاني /1429-1430

Introduction

What is R?

R is a very powerful programming environment for statistical research and data analysis, including the ability to easily generate numbers, manipulate arrays of various dimensions, and to produce very quality graphics.



The R Project for Statistical Computing

About R
What is R?
Contributors
Screenshots
What's new?

Download
CRAN

R Project
Foundation
Members & Donors
Mailing Lists
Bug Tracking
Developer Page
Conferences
Search

Documentation
Manuals
FAQs
Newsletter
Wiki
Books
Other

Misc
Bioconductor
Related Projects
Links

http://www.r-project.org/

Getting Started:

- R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To download R, please choose your preferred CRAN mirror.
- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

News:

- **R version 2.4.1** has been released on 2006-12-18.
- **DSC 2007**, the 5th workshop on Directions in Statistical Computing, February 15-16, 2007, Auckland, New Zealand.
- **R News 6/5** has been published on 2006-12-1.
- **The R Wiki** provides an online forum where useRs can help other useRs.

Features:-

- An interactive programmed, an effective data handling and storage facility
- An array oriented. Can generate, manipulate, and operate on large array using simple commands.
- It is very graphical. A large number of high level graphics commands are available to produce publication quality graphics both on your screen and on a printer.
- An interpreted language, in which individual language expressions are read and then immediately executed.
- Well developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.

General instruction:-

- A name is any combination of letters, numbers, and periods (.), and if it starts with '.' the second character must not be a digit, and can not start with number.
- File name and variables can be more than 8 characters in length.
- It is case sensitive: the object X is not the same as object x.
- Any comment after # on a given line not execute.
- Commands are separated either by semi-colon(';'), or by a new line.
- If a command is not complete at the end line, R will give a different prompt, by default + .
- Data are stored in _data. Subdirectory.

Help Facilities

R has online help system. To start the help system you have many choices:

- For general help:
  1) **>help()**
  2) Click on [Help]

- For a specific command or function:
  1) **>help** (command name), for example, **>help(mean)** or
  2) **>? Function name**, for example, **>? mean**

- For help on characters: the argument must be enclosed in double quotes,
  >help("[[")

- For searching for entries
  The **help.search** command, for example, **>help.search("linear models")**

- The examples on a help topic can normally be run by
  > example(topic),

Data objects

Data modes:

In R, data object is a collection of values. The modes of values are as follows:

- Logical: the values T(or TRUE) and F(FALSE).
- Numeric: real numbers, integers, decimal or scientific notation.
- Complex: complex numbers of the form a+bi (3+1.23 i), (a and b) are numeric.
- Character: enclosed by double quotes (") or apostrophes ('), such a "Sara" or 'Sara'.

❖ If you want to know the mode of any object use **mode ( )** function

Types of data objects:

There are seven basic types of data objects in R:

1) Vector (an ordered set of values) – one way array of ordered data.
2) Matrix (two dimensions).
3) Array (a matrix with more than two dimensions)
4) Data frame (generalized matrices that allow a mix of columns with different data modes).
5) List (a list of components, where each component can be a data object of different data types).
6) Factor (categorical data)
7) Time series.

Operators in R

I. Names and Assignment:

The assignment operator (<- or =) used to associate names and values.
For example

x <- 7 or **x =7** # stores the value 7 in an object named x

You can check of the object x either by typing x or **print (x)**.

Note:

All assignments in R remain until removed or overwritten. The **rm()** command used to remove a variable.
Example:

```
>Print(x)
[1] 7
>rm(x)        # remove x
>x
Error: object "x" not found.
```

To display the names of the objects which are currently stored within R,

```
> objects()
```

Missing values

When an element or value is "not available" or a "missing value " the data values are represented by such special symbols NA. when a value (missing data, square root or logarithm of negative number). For these cases, any operation on NA becomes NA.

The function **is.na(x)** gives a logical vector of the same size as x with value TRUE if and only if the corresponding element in x is NA.

```
>x<-c(1:3,NA) ; x
 >is.na(x)     # is TRUE both for NA and NAN values.
[1] FALSE FALSE FALSE  TRUE
>x= =NA
[1] NA NA NA NA
>sum(x)
NA
```

There is a second kind of "missing" values which are produced by numerical computation; it is called Not a Number, NAN, values. Examples are

```
>0/0           # give NAN
>Inf – Inf     # give NAN
>xx=Inf/Inf
> is.nan(xx)     # is TRUE  only for NAN values.
> x<-c(1,2,3,NAN,4,5,NAN,7)
> sum(x)
[1] NaN
> log(-2)
[1] NaN
Warning message:
NaNs produced in: log(x)

> x<-c(1,2,3,NaN,4,5,NaN,7)
> is.na(x)
[1] FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
```

To remove missing values from x:

```
>x= x[!is.na(x)]
[1] 1 2 3 4 5 7
```

II. Arithmetic operators :

Operator	Description	Priority
( )	parentheses	1
** or ^	Exponentiation	2
:	Sequences of numbers	3
* /	Multiply, divide	4
+ -	Add, subtract	5

III. Logical and comparison operators:

Operator	Description	Operator	Description
<	Smaller than	&	Factorized And
>	Larger than	|	Factorized Or
= =	Equal to	!	Not
<=	Smaller than or equal to	! =	Not equal to
>=	Larger than or equal to		

Use of Brackets

Name of bracket	bracket	Function
Round brackets	( )	For function calls like in **mean(x)**, and to set priorities
Square brackets	[ ]	Index brackets in **x[3]** used to access or extracts data
Curly brackets	{ }	Block delimiter for grouping sequences of commands as in functions or if statements

DATA STRUCTURE

I. Vector

A vector is an ordered collection of values.

A) creating a vector

The following table has useful functions for creating vector

function	Symbol	description	example
Concatenate command	c( )	Combines values with any mode	X<–c(2,3,8,0,-7)
Sequence command	seq(from= ,to= ,by=)	Regular sequences of numbers	X<– seq(1,10,1)
	from : to		X<–1:10
Replicate command	rep(x, times=)	Takes a pattern and replicates it	X<–rep(1, 5)

c:
Combine values with any modes

Examples:

```
 > c(1,7:9)
 > c(1:5, 10.5, "next")
 > c("This", "is","Stat","442")
> z<- 0:9
[1] 0 1 2 3 4 5 6 7 8 9
> digits<- as.character(z)
[1] "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
> d<- as.integer(digits)
[1] 0 1 2 3 4 5 6 7 8 9
```

seq
Sequence Generation

```
from:to
     a:b
seq(from, to)
seq(to)

   seq(from, to, by=)
   seq(from, to, length=)
```

Arguments:

from: starting value of sequence.

to: (maximal) end value of the sequence.

by: increment of the sequence.

length: desired length of the sequence.

Examples:

```
 > 1:4
 >  pi:6 # float
 >  6:pi # integer
 > seq(0,1, length=11)
 > seq(1,9, by = 2)   # match
 > seq(1,9, by = pi)  # stay below
 > seq(1,6, by = 3)
 > seq(1.575, 5.125, by=0.05)
 > seq(17)  # same as 1:17
> seq(-pi,pi,0.5)
> seq(-pi,pi,length=10)
> seq(1,by=0.05,length=10)
> seq(10,2,-2)

> 1:4
[1] 1 2 3 4
>  pi:6   # float
[1] 3.141593 4.141593 5.141593
>  6:pi   # integer
[1] 6 5 4
>  seq(0,1, length=11)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
>  seq(1,9, by = 2)  # match
[1] 1 3 5 7 9
> seq(1,9, by = pi)  # stay below
[1] 1.000000 4.141593 7.283185
> seq(1,6, by = 3)
[1] 1 4
> seq(1.575, 5.125, by=0.05)
 [1] 1.575 1.625 1.675 1.725 1.775 1.825 1.875 1.925 1.975 2.025 2.075
2.125
[13] 2.175 2.225 2.275 2.325 2.375 2.425 2.475 2.525 2.575 2.625 2.675
2.725
```

```
[25] 2.775 2.825 2.875 2.925 2.975 3.025 3.075 3.125 3.175 3.225 3.275
3.325
[37] 3.375 3.425 3.475 3.525 3.575 3.625 3.675 3.725 3.775 3.825 3.875
3.925
[49] 3.975 4.025 4.075 4.125 4.175 4.225 4.275 4.325 4.375 4.425 4.475
4.525
[61] 4.575 4.625 4.675 4.725 4.775 4.825 4.875 4.925 4.975 5.025 5.075
5.125
>  seq(17)   # same as 1:17
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
> seq(-pi,pi,0.5)  #can not get to pi
 [1] -3.1415927 -2.6415927 -2.1415927 -1.6415927 -1.1415927 -0.6415927
 [7] -0.1415927  0.3584073  0.8584073  1.3584073  1.8584073  2.3584073
[13]  2.8584073
> seq(-pi,pi,length=10)
 [1] -3.1415927 -2.4434610 -1.7453293 -1.0471976 -0.3490659  0.3490659
 [7]  1.0471976  1.7453293  2.4434610  3.1415927
> seq(1,by=0.05,length=10)
 [1] 1.00 1.05 1.10 1.15 1.20 1.25 1.30 1.35 1.40 1.45
> seq(10,2,-2)
[1] 10  8  6  4  2
```

Rep:

Replicate Elements of Vectors and Lists

```
rep(x, times, ...)
rep(x, times, length.out, each, ...)
rep.int(x, times)
```

Arguments:

x: a vector (of any mode including a list)

times: optional non-negative integer. A vector giving the number of times to repeat each element if of length 'length(x)', or to repeat the whole vector if of length 1.

length.out: optional integer. The desired length of the output vector.

each: optional integer. Each element of 'x' is repeated 'each' times.

...: further arguments to be passed to or from other methods.

Examples:

```
> rep(1:4, 2)
> rep(1:4, each = 2)      # not the same.
> rep(1:4, c(2,2,2,2))    # same as second.
> rep(1:4, c(2,1,2,1))
> rep(1:4, each = 2, len = 4)    # first 4 only.
>  rep(1:4, each = 2, len = 10)   # 8 integers plus two recycled 1's.
>  rep(1:4, each = 2, times = 3)  # length 24, 3 complete replications
>  rep(1, 40*(1-0.8)+1e-7)
>  rep(c(“yes”,”no”), c(4,2))
>  rep(1:3,1:3)
>  rep(c(1,3,2),length=10)
>  rep(c(T,T,F),2)

>  rep(1:4, 2)
[1] 1 2 3 4 1 2 3 4
> rep(1:4, each = 2)      # not the same.
[1] 1 1 2 2 3 3 4 4
>  rep(1:4, c(2,2,2,2))    # same as second.
[1] 1 1 2 2 3 3 4 4
> rep(1:4, c(2,1,2,1))
[1] 1 1 2 3 3 4
> rep(1:4, each = 2, len = 4)    # first 4 only.
[1] 1 1 2 2
> rep(1:4, each = 2, len = 10)   # 8 integers plus two recycled 1's.
 [1] 1 1 2 2 3 3 4 4 1 1
> rep(1:4, each = 2, times = 3)  # length 24, 3 complete replications
 [1] 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4
> rep(1, 40*(1-.8)+1e-7)
[1] 1 1 1 1 1 1 1 1
> rep(c("yes","no"), c(4,2))
[1] "yes" "yes" "yes" "yes" "no"  "no"
> rep(1:3,1:3)
[1] 1 2 2 3 3 3
> rep(c(T,T,F),2)
[1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE
```

Vector Arithmetic

Vector arithmetic is element-wise (element by element). Vectors must be of same length or a warning message is issued.

Examples:

```
> x<-1:10
> x*2
> x^2
> y<-6:2
> y+x
> w<-1:4
> x+w
```

```
> x<-1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> x*2
 [1]  2  4  6  8 10 12 14 16 18 20
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> x^2
 [1]   1   4   9  16  25  36  49  64  81 100
> y<-6:2
> y
[1] 6 5 4 3 2
> y+x
 [1]  7  7  7  7  7 12 12 12 12 12
> w<-1:4
> w
[1] 1 2 3 4
> x+w
 [1]  2  4  6  8  6  8 10 12 10 12
Warning message:
longer object length
        is not a multiple of shorter object length in: x + w
```

B) Accessing elements in a vector.

- R uses brackets, [indices], to select elements of a vector.
- To delete elements from a vector, use the minus sign. [- indices].
- Extracting elements using logical values, [logical condition]

Such index vector can be any of four distinct types:

1. Indices of positive integral quantities:

Examples:

> X<-seq(2,10,2)
> X[2] # list the second element in X
> X[3:5] # list the 3rd, 4th and 5th element in X same as **X[c(3,4,5)]**
> X[c(1,3,5)] # list the 1st, 3rd, and 5th element in X (order not required)
> X[6]

You also could use rep( ) or seq() inside []
> X[seq( )];X[rep( )]

```
> X<-seq(2,10,2)
> X[2]
[1] 4
> X[3:5]
[1]  6  8 10
> X[c(1,3,5)]
[1]  2  6 10
> X[8]
[1] NA
>letters
>LETTERS
>LETTERS[1:3]
>letters[2:4]
```

2. Indices of negative integral quantities:

```
> y<-2:6; y[-4]
> y[-c(1:3)]
 [1]  5  6
```

3. Indices of character strings:

```
> fruit<-c(5,10,1,20)
> names(fruit)<- c("orange"," banana", "apple"," peach")
> lunch <- fruit[c("apple", "orange")]
apple orange
    1      5
```

4. Logical Indices:

```
 > y<- -2:3
 >  log(y)
 > y>0
 > y[y>0]
 > log(y[y>0])
 > log(y>0)
 > y[y<3]
 > y[y<0|y>2]
 > y[ (y<3&y>2)]

 > y<- -2:3    # -2 -1 0 1 2 3
 > log(y)
[1]      NaN       NaN      -Inf 0.0000000 0.6931472 1.0986123
Warning message:
NaNs produced in: log(x)
> y>0
[1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
> y[y>0]   # list element in y such that they are more than 0
[1] 1 2 3
> log(y[y>0])
[1] 0.0000000 0.6931472 1.0986123
> log(y>0)
[1] -Inf -Inf -Inf    0    0    0
> y[y<3]      # list element in y such that they are less than 3
[1] -2 -1  0  1  2
> y[y<0|y>2]   # list element in y such that they are less than 0 or more than 2
[1] -2 -1  3
> y[ (y<3&y>2)] # list element in y such that they are less than 0 and more than 2

numeric(0)

> z=c(2,5,4,NA,3,-2)
> z[!is.na(z)]
[1] 2,5,4,3,-2

> w = z[!is.na(z)]; w
> (z+1)[(!is.na(z)) & z>0] -> S ; S
> z
> z[is.na(z)] <- 0
> z
2  5  4  0  3 -2
```

Example:

Suppose we have height (in inches) and weight (in pounds) of 9 people

Weight	60	61	62	63	64	65	66	67	68
Height	120	125	130	135	140	145	150	155	135

```
> height<- c(seq(120,155,5),135)

> weight<- 60:68
# use logical expression to see how many less than 140
> height<140
> sum(height <40)
> height [height <140]
> height[height >150]
> weight[weight<65]
> weight[weight<60]  #  no height less than 60
> height [height <140& height!=120]
#combine weight and height
> height[weight>65]
```

```
> height<- c(seq(120,155,5),135)
> weight<- 60:68
> # use logical expression to see how many less than 140
> height<140
[1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE
> sum(height <40)
[1] 0
> height [height <140]
[1] 120 125 130 135 135
> height[height >150]
[1] 155
> weight[weight<65]
[1] 60 61 62 63 64
> weight[weight<60]  #no height less than 60
numeric(0)
> height [height <140& height!=120]
[1] 125 130 135 135
> #combine weight and height
> height[weight>65]
[1] 150 155 135
```

Printing in R:

> cat( ) # Concatenate and Print (for general printing)
> Print( ) #print numeric or character data
> Paste( ) # Concatenate vectors after converting to character.

Examples

```
> x=1:10
> print(x)
 [1]  1  2  3  4  5  6  7  8  9 10
> cat("class = ", x, "\n")
class =  1 2 3 4 5 6 7 8 9 10
> cat("class = ", x)
class =  1 2 3 4 5 6 7 8 9 10>
>  paste(1:12)
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12"
>     paste("A", 1:6, sep = "")
[1] "A1" "A2" "A3" "A4" "A5" "A6"
>     paste("Today is", date())
[1] "Today is Sun Sep 24 01:33:04 2006"
```

Same as

```
> cat("Today is", date())
Today is Sun Sep 24 01:35:19 2006>    cat("Today is", date(), "\n")
Today is Sun Sep 24 01:35:43 2006
```

Some Arithmetic and Statistical R Functions (built-in)

R Function	Notes
log(x), log10(x), exp(x), sqrt(x)	ln(x), $\log_{10}$(x), e^x, $\sqrt{x}$
Sin(x), cos(x), tan(x)	Trigonometric function
max(x), min(x), length(x), range(x)	Maximum, minimum, number of elements, and range of a vector
sign(x), abs(x), sort(x), sum(x), prod(x)	Sign, absolute value, sort in ascending order, summation, product of elements in a vector x
ceiling(x)	Rounds to the next higher integer
floor(x)	Rounds to the next lower integer
trunc(x)	Cuts off all digits after the decimal point
round(x), round(x, 3), round(x, -1)	Rounds to the nearest integer. The second argument is the number of significant number of digits desired, negative value to round large number to nearest 10 or 100, etc.
cor(x,y), mean(x), var(x), quantile(x), median(x), summary( ), stem(x), hist(x)	Statistical function
% / % **% %**	Quotient(integer division), modulo function (remainder) % / % and % % always satisfy e1 = = (e1 % / % e2))e2+e1 % % e2
cumsum(x), cumprod(x)	Returns an object which, for each element, is the sum (product) of all of the elements to that point.
gamma	Gamma function

Examples

```
> ceiling(2.4)
[1] 3
> floor(2.4)
[1] 2
> trunc(2.4)
[1] 2
> w=2.346789
> round(w)
[1] 2
> round(w,3)
[1] 2.347
> round(w,-1)
[1] 0
> w=50.34
> round(w,-1)
[1] 50

> x<-1:5
> x%%2    # calculate the reminder from the division
[1] 1 0 1 0 1
> x%/%2      # calculate the integer of the division
[1] 0 1 1 2 2
> x/2     # give exact value of division
[1] 0.5 1.0 1.5 2.0 2.5

> cumsum(x)
[1]  1  3  6 10 15
cumprod(x)
[1]   1   2   6  24 120

> Betafun<-function(a,b)
+ {
+ x<-gamma(a)
+ y<- gamma(b)
+ z<- gamma(a+b)
+ beta<-x*y/z }
> Betafun(1,2)
> x<-Betafun(1,2)
> x
[1] 0.5
```

Experiment with the following R functions to predict what they do and give the answer

Example 1:

```
ceiling(c(-1.9,-1.1,1.1,1.9))
floor(c(-1.9,-1.1,1.1,1.9))
trunc(c(-1.9,-1.1,1.1,1.9))

x<-c(123456,.123456,.000123456)
round(x)

> ceiling(c(-1.9,-1.1,1.1,1.9))
[1] -1 -1  2  2
> floor(c(-1.9,-1.1,1.1,1.9))
[1] -2 -2  1  1
> trunc(c(-1.9,-1.1,1.1,1.9))
[1] -1 -1  1  1>
> x<-c(123456,.123456,.000123456)
> round(x)
[1] 123456      0      0
> round(x,3)
[1] 123456.000      0.123      0.000
> round(x[1],-1)
[1] 123460
> round(x[1],-2)
[1] 123500
```

Example 2:

Calculate the sin, cosine and tanget for numbers ranging from 0 to 2*pi with distance 0.1 between them. Note that tan(x)=sin(x)/cos(x)
Calculate difference between tan(x) and sin(x)/cos(x) for the values above. Which values are exactly equal? What is the maximum differences.

Example 3:

Calculate the first 50 powers of 2, (i.e. 2*1, 2*2, 2*2*2)
Calculate the squares of integer numbers from 1 to 50.
Which pairs are equal?(i.e 2^n=n^2). How many are there?

Sol. of Example 2:

```
> x<-seq(0,2*pi,0.1)
> a<-sin(x)
> b<-cos(x)
> c<-tan(x)
>
> diff<-c-a/b
>
> diff==0
 [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE
FALSE FALSE FALSE
[13] FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
FALSE  TRUE
[25]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
TRUE FALSE
[37]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE
TRUE FALSE
[49]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
TRUE FALSE
[61]  TRUE  TRUE  TRUE

> length(diff)
[1] 63
> sum(diff==0)
[1] 43
> max(abs(diff))
[1] 1.421085e-14
```

Sol. of Example 3:

```
> int<-1:50
> int
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
49 50
> x<-2^int  # first 50 power of 2
> y<-int^2 # square 1:50
> eq<-x==y  # examine which pairs are equal
> eq
 [1] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
FALSE FALSE FALSE
```

```
[25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
FALSE FALSE FALSE
[37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
FALSE FALSE FALSE
[49] FALSE FALSE
> sum(eq) # sum of equal pairs
[1] 2
> int[eq]  # list all equal pairs
[1] 2 4
> length(x[eq])  # how many such that x=y
[1]  2
```

II.Matrix

A matrix is a two dimensional array, it consists of elements of the same type and displayed in rectangular form. The first index denotes the row; the second index denotes the column of the specified elements, i.e. X_{ij} represents the element in the i^{th} row and j^{th} column.

A) Creating matrix

The following table has some related R functions

Function	Description	Example

matrix( )	Creates matrix, takes a vector argument and turns it into a matrix **matrix(data, nrow, ncol, byrow = F)**	**matrix(1:12,3,4)**
		matrix(1:12,3)
		matrix(1:12,ncol=4)
		matrix(1:12,3,4,byrow=T)

cbind( )	Combines vectors column by column	**x <– cbind(c(1:4),c(5:8))**
rbind( )	Combines vectors row by row	**x <– rbind(c(1:4),c(5:8))**
dim( )	Returns or changes the dim attribute, which describes the dimensions of a matrix, data frame or array	**x <– 1:8** **dim(x) <– c(2,4)**

matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)

Arguments:

data: an optional data vector.

nrow: the desired number of rows

ncol: the desired number of columns

byrow: logical. If 'FALSE' (the default) the matrix is filled by columns, otherwise the matrix is filled by rows.

dimnames: A 'dimnames' attribute for the matrix: a 'list' of length 2 giving the row and column names respectively.

Examples:

```
matrix(1:12)
matrix(1:12,nrow=3,ncol=4)
matrix(1:12,nrow=3,ncol=4, byrow=TRUE)
matrix(1:12, ncol=4, byrow=TRUE)  #  same as above – simple division
matrix(1:12, ,4, byrow=TRUE)         #  same as above
matrix(1:13, 3, 4)
matrix(1:13, 3)
```

Use of dimnames argument

```
mdat <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol=3, byrow=TRUE,
+                dimnames = list(c("row1", "row2"), c("C.1", "C.2", "C.3")))
>    mdat
   C.1 C.2 C.3
row1   1   2   3
row2  11  12  13
```

Or write after you defined the matrix mdat

```
dimnames(mdat) = list(c("row1", "row2"), c("C.1", "C.2", "C.3"))
dimnames(mdat) = list(paste("row", letters[1:3]), paste("Col", letters[1:3]))
> dimnames(mdat) = list(paste("row", letters[1:2]), paste("Col", 1:3))
>    mdat
   C.1 C.2 C.3
row1   1   2   3
row2  11  12  13
```

Note :

To suppress either row or column labels use the NULL

```
> dimnames(mdat) = list(NULL, c("C.1", "C.2", "C.3"))
> mdat
   C.1 C.2 C.3
[1,]   1   2   3
[2,]  11  12  13
```

```
*** Try this ***
mat1=rep(1:4,rep(3,4))
matrix(mat1,3,4)
matrix(mat1,3)
matrix(mat1,4,byrow=T)

is.matrix(mat1)
```

```
x<-rbind(c(1:4),c(5,8))
x<-cbind(c(1:4),c(5,8))

y=1:9
w=2:10
z=3:5
```

see what you get from

```
rbind(y,w)
cbind(y,w)
rbind(y,z)
rbind(y,w,z)
```

cbind(0, rbind(1, 1:3))

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

```
x <– 1:8
dim(x) <– c(2,4)
dim(x) <– c(4,2)
```

B) Matrix arithmetic

The arithmetic operation (+, - , *, /) are applied in an element wise manner (element by element), the matrices should be the same dimension.
% * % operator performs matrix multiplication on two conformable matrices.

Example :

```
x <– matrix(1:4,2)
    1   3
    2   4

y<– matrix(c(1:2, 1:2),2)
    1   1
    2   2
```

Calculate: **x+y; x-y; x*y; x/y**

The following table has some related R functions

Function	**Description**
nrow( ), ncol ( )	Returns the number of row or the column of the matrices
dimnames( )	Returns or changes the dimnames attribute of a matrix or array
diag( )	Either creates a diagonal matrix or extracts the diagonal elements of a matrix
Solve( )	Calculate the inverse
var( )	Covariance matrix of the columns
t(x)	Transpose of x
eigen(x)	Eigenvalues and eigenvectors of x

Solve: Solve a System of Equations

Description:

This generic function solves the equation 'a %*% x = b' for 'x', where 'b' can be either a vector or a matrix.

Arguments:

- a: a square numeric or complex matrix containing the coefficients of the linear system.
- b: a numeric or complex vector or matrix giving the right-hand side(s) of the linear system. If missing, 'b' is taken to be an identity matrix and 'solve' will return the inverse of 'a'.

The function solve is used to solve a system of equations

Z1+2z2+3z3=3

2z1+3z2+2z3=0

3z1+2z2+z3=1

```
mat=rbind(c(1,2,3),c(2,3,2),c(3,2,1))
y=c(3,0,1)
z=solve(mat,y)
```

To make a check do this

```
mat%*% z    # you should get y
```

C) Matrix indexing:

R uses square brackets, [row num, col num], to select elements of a matrix, also you can use the labels of row and column to access the element.
Access the elements in the x matrix

```
x[a,]
x[,b]
x[-a,]
x[,-b]
x[a:b,c:d]
##########################################################
x[1,2]                  #  returns the element x12
x[,2]                   #  returns all elements in the second column
x[1,]                   #  returns all elements in the first row
x[-3,]                  # remove third row
x[1:3,2:3]              # select sub matrix consists of row 1-3 and column 2-3
x[-(1:2),2:3]
x[“rowa”,”colb”]         # it is equivalent x[1,2]
x[“rowc”]               #  it is equivalent x[3,]

m=matrix(1:36,9,4)
m[2,3]
m[,3]
m[2,]
cbind(m[,3])
m[,-3]
m[-(3:8),2:4]
diag(m)
m[1:4,1:4]
diag(m[1:4,1:4])
diag(4)
```

Example:

The following table contains information of cars

	price	mileage	weight
USA	8895	33	2560
Korea	7402	33	2345
Japan	6319	37	1845

Answer the following:

1. Create a matrix.
2. Find the price of Korea car.
3. Give the mileage of all cars
4. Find information of USA car
5. Remove the information of Japan car
6. Select sub matrix consists of rows1-3 and column 2-3
7. Remove rows 1-2 and select column 2-3

Apply() Function:

Apply Functions Over Array Margins

Description:

Returns a vector or array or list of values obtained by applying a function to margins of an array.

Usage:

```
apply(X, MARGIN, FUN, ...)
```

Arguments:

X: the array to be used.

MARGIN: a vector giving the subscripts which the function will be applied over. '1' indicates rows, '2' indicates columns, 'c(1,2)' indicates rows and columns.

FUN: the function to be applied: see Details. In the case of functions like '+', '%*%', etc., the function name must be backquoted or quoted.

...: optional arguments to 'FUN'.

Examples:

```
## Compute row and column sums for a matrix:
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
dimnames(x)[[1]] <- letters[1:8]
apply(x, 2, mean, trim = .2)
col.sums <- apply(x, 2, sum)
row.sums <- apply(x, 1, sum)
rbind(cbind(x, Rtot = row.sums), Ctot = c(col.sums, sum(col.sums)))

## Sort the columns of a matrix
apply(x, 2, sort)

ma <- matrix(c(1:4, 1, 6:8), nr = 2)
ma
apply(ma, 1, table)          #  a list of length 2
apply(ma, 1, quantile)        # 5 x n matrix with rownames
```

R actually has built-in functions to compute column or row sums, means, variances and standard deviations which are faster than using apply:

```
> colMeans(x)
> colSums(x)
> colVars(x)
> colStdevs(x)
> rowMeans(x)
> rowSums(x)
> rowVars(x)
> rowStdevs(x)
```

Outer() Function:

```
outer(X, Y, FUN="*", ...)
X %o% Y
```

Arguments: X, Y: First and second arguments for function 'FUN'. Typically a vector or array.
FUN: a function to use on the outer products,

Examples:

```
> x=1:3
> y=rep(3,5)
> outer(x,y,FUN="+")
     [,1] [,2] [,3] [,4] [,5]
[1,]    4    4    4    4    4
[2,]    5    5    5    5    5
[3,]    6    6    6    6    6
 > outer(x,y)
     [,1] [,2] [,3] [,4] [,5]
[1,]    3    3    3    3    3
[2,]    6    6    6    6    6
[3,]    9    9    9    9    9
> outer(x,y,FUN="^")
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1
[2,]    8    8    8    8    8
[3,]   27   27   27   27   27
```

III. Array

The syntax for creating an array is:

Array (data, dim)

Example

```
> x=array (1:24, c (3, 4, 2))
> x
, , 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

> dimnames(x)<- list (c("A","B","C"),c("a","b","c","d"),c("first", " second"))
> x
, , first

  a b c  d
A 1 4 7 10
B 2 5 8 11
C 3 6 9 12

, ,  second

   a  b  c  d
A 13 16 19 22
B 14 17 20 23
C 15 18 21 24
```

Exercise:

Use of apply function on array

```
> array(1:4, c(2,2,3))
, , 1
   [,1] [,2]
[1,]   1    3
[2,]   2    4
, , 2
   [,1] [,2]
[1,]   1    3
[2,]   2    4
, , 3
   [,1] [,2]
[1,]   1    3
[2,]   2    4

> apply( array(1:4, c(2,2,3)),2,sum)
[1]  9 21
> apply( array(1:4, c(2,2,3)),1,sum)
[1] 12 18
> apply( array(1:4, c(2,2,3)),3,sum)
[1] 10 10 10
```

The following table contains information of 3 countries:

	GDP	Pop	Inflation
Austria	**197**	**8**	**1.8**
France	**1355**	**58**	**1.7**
Germany	**2075**	**81**	**1.8**

Answer the following:

1. Create a matrix name it country.data
2. Find the total population for all three countries.
3. Add variable area (84,544,358)
4. Add country Switzerland (265, 7, 1.8,41)
5. Find the maximum value for all variables.
6. Find the mean for all variables.
7. Change the population of Austria to 10 millions
8. Give data for France.
9. Find population for Germany.
10. Define variable EU=c("EU","EU","EU","non-EU") and add it to the variables
11. Find the maximum value for all variables and see what the entries look like (Error)????.

VI. Data frame

Are very similar to matrices except that they allow the columns to contain different types of data of the same length, whereas a matrix is restricted to one type of data only. Data frames still have to be in rectangular form as matrices.

A) Creating Data frame:

- **read.table** read data from external files.
- **Data.frame** binds together objects of different kinds.

The syntax for creating a data frame is:

data.frame(data1,data2,…)

B) Data frame Arithmetic:

You can only apply numeric computations to numeric variables in data frame

C) Data frame Indexing:

Same tools used with matrix indexing can be used and also we can use $ to extract vector, part of list, or an array.

Example:

```
country.frame<-data.frame(country.data,EU)
apply(country.frame[,1:4],2,max)

#to access each variable in data frame
country.frame$pop
country.frame[,"pop"]
country.frame[,2]
```

Car information Example

```
car.inf=matrix(c(8895,33,2560,7402,33,2345,6319,37,1895),3,3,byrow=T)
country=c("USA","Korea","Japan")
inf=c("Price","Mileage","Weight")
dimnames(car.inf)=list(country,inf)
dimnames(car.inf)
```

Compute the price of cars after 25% discount, then add this variable to car.inf data frame

```
dis=car.inf[,1]-car.inf[,1]*0.25
car.inf=cbind(car.inf,dis)

type=c("sporty","compact","van")
car =cbind(car.inf,type)
car
car.frame =data.frame(car.inf,type)
car.frame

car.frame$millage
car.frame["USA",]                                    #Select info. of USA cars
```

To add a row to data frame

```
SA=data.frame(7000,40,2000,4000,"van")
inf=c("Price","Mileage","Weight","dis","type")
dimnames(SA)=list("SA",inf)
d1=rbind(SA,car.frame)
d1
      Price Mileage Weight    dis     type
SA     7000    40   2000   4000.00    van
USA    8895    33   2560   6671.25  sporty
Korea  7402    33   2345   5551.50 compact
Japan  6319    37   1895   4739.25    van
```

What happen if you use this

```
d2=rbind(SA=c(7000,40,2000,4000,"van"),car.frame)?????
```

see the output of this

```
d2$dis
d1$dis
```

Access data.frame with logical vector

```
car.frame$Price[dis>5000]
car.frame$Price[Weight>2000]              # Problem!!! Weight was not defined
car.frame$Price["Weight">2000]            # another problem try "Weight">2000
car.frame["USA",]
```

can write

```
car.frame$Price[car.fram$Weight>2000]
```

# dis was defined separately so did not make a problem

```
car.frame$Mileage[car.frame[,1]>7000]
car.frame$Mileage[car.frame$Price>7000]
```

what is car.frame?????

```
car.frame[dis>4000,1:4]
car.frame[dis>5000,1:4]
```

subset:

returns subsets of vectors or data frames that meet specific requirements

```
new.frame=subset(car.frame,subset=dis>4000,select=1:4)
```

V.List

A list allows a programmer to tie together related data that do not have the same structure (different lengths or modes).

A) Creating List:

Used (list) function.

B) List Indexing:

To access the elements in a list, used a double square brackets [[   ]] then the sub elements by using a single square brackets.

Example:

```
g=list(1:10,c(T,F),c("Hey","You"))
#to name all elements in a list
names(g)=c("number","bool","message")
g=list(number=1:10,bool=c(T,F),message=c("Hey","You"))
#add another element
g$comment="we assign a new element"
```

Example:

Create list contains four components, first component: car.inf data frame, second component: vector of the names of the companies that made the cars, third component: vector of model numbers of the cars.

```
company=c("Toyota","Kia","Mersedis")
modl=c(1990,2005,2006,2004)
#carlist=list(car.inf,company,modl)
carlist=list(car.frame,company,modl)

#to name component in a list
names(carlist)=c("car.information","company","model")
names(carlist)

#Acces the elements in carlist
carlist[[2]] # gives company vector
carlist $car.information
carlist $car.information$type
carlist $company[2]
carlist[[3]][2]
apply(carlist[[1]][,1:4],2,max)

#make a list  inside a list
m=matrix(1:4,2)
carlist2=list(carlist,m)
names(carlist2)=c("e","p")
```

carlist2$e$car.information$dis
carlist2[[1]][[1]][,"dis"]
carlist2[[1]][[2]][3]

The `lapply` and `sapply` Functions

`lapply` and sapply operate on components of a list or vector

- `lapply` will always return a list
- `sapply` is a more user friendly version of lappy and will attempt to simplify the result into a vector or array

```
# Example 1: lapply                    # Example 2: sapply
> l <- list(Sex=Sex,Eth=Eth)         > l <-
list(Sex=Sex,Eth=Eth)
> lapply(l,table)                      > sapply(l,table)
$Sex                                Sex Eth
F  M                                F 80  69
80  66                              M 66  77
$Eth
A  N
69  77
```

<u>**Example:**</u> **(lapply,sapply)**

```
x <- list(a = 1:10, beta = exp(-3:3), logic =c(TRUE,FALSE,FALSE,TRUE))
    # compute the list mean for each list element
    lapply(x,mean)
    # median and quartiles for each list element
    lapply(x, quantile, probs = 1:3/4)
lapply(x,"-",1)

#better look
sapply(x, mean)
sapply(x, quantile)
```

- **`lapply`: takes any structure, gives a list of results**
- **`sapply`: like lapply, but simplifies the result if possible**
- **`apply`: only used for arrays**
- **`tapply`: used for ragged arrays: vectors with an indexing specified by one or more factors.**

Adding elements to a list can be achieved by

- adding a new component name:

```
> L1=list(Item1=c(7,2,5,8),Item2=c(T,T,F,T),Item3=c("a","h","m","t"))
> L1$Item4=c("apple","orange","melon","grapes")
# alternative ways
> L1[["Item4"]]=c("apple","orange","melon","grapes")
# OR
> L1[[4]]=c("apple","orange","melon","grapes")   #without name
> names(L1)[4]="Item4"
```

What is names(L1)???

```
> names(L1)
[1] "Item1" "Item2" "Item3" "Item4"
```

VI.Factor:

A factor is a vector object used to specify a discrete classification (grouping) of the components of other vectors of the same length.

EXAMPLE:

Suppose we have a sample of 30 tax accountants from all the states and territories of Australia and their individual state of origin is specified by a character vector of state :

```
> state<- c("tas", "sa", "qld", "nsw" ,"nsw", "nt", "wa", "wa", "qld", "vic", "nsw",
          "vic"," qld", "qld", "sa", "tas", "sa", "nt", "wa", "vic", "qld", "nsw",
           "nsw", "wa", "sa", "act", "nsw", "vic", "vic", "act")

> statef <- factor(state)
>statef
```

To find out the levels of a factor the function level() can be used.

```
> levels( statef)
[1] "act" "nsw" "nt" "qld" "sa" "tas" "vic" "wa"
>table(statef)
```

ANOVA

To encode a vector as a factor use

```
factor(rep(c("tr1","tr2"),c(10,10)))
 [1] tr1 tr1 tr1 tr1 tr1 tr1 tr1 tr1 tr1 tr1 tr2 tr2 tr2 tr2 tr2 tr2 tr2 tr2
[19] tr2 tr2
```

The function tapply():

To continue the previous example, suppose we have the incomes of the same tax accountants in another vector (in suitably large units of money)

```
> incomes <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70, 42, 56, 61, 61, 61, 58, 51, 48,
                 65, 49, 49, 41, 48, 52, 46, 59, 46, 58, 43)
```

To calculate the sample mean income for each state we can now used the special function tapply():

```
> inc.means <- tapply(incomes, statef, mean)
> inc.means

   qld      act     nsw        nt    qld     sa    tas    vic     wa
56.00    44.500  57.33333   55.50  53.00   55.00  60.50  56.00   52.250
```