BioHTA2012

R Crash Course

Xiaobei Zhao

Bioinformatics Centre, University of Copenhagen

April 23, 2012

About the lecture

It is a three-hour lecture made exclusively for you, who have a math-heavy background. We are getting familiar with the language (quickly!) but will not go into details.

- ❍ Introduction to R
- ❍ R basics
- ❍ Case studies using biological data
  - Data handling
  - Visualization
  - Statistics
- ❍ Appedix
  - A. Exercise on R basics and solutions

    If time permitted we will do the exercise in class; otherwise you may take it home and finish for fun.
  - B. Advanced topics in R (Not required for the course)

    Provided only for your reference, we will not discuss it in the lecture.
- ❍ Literature for further reading

One practical issue

The slides and the R code used here can be downloaded at,

```
Absalon (Virtuelt læringsmiljø)
  5550-B4-4F12;Bioinformatics of high throughput analyses
    Course material
      Pre-lecture slides
        R_crash_pre_lecture.pdf
        R_crash_pre_lecture.R
```

The final slides with soluaiotns will be uploaded to the directory `Post-lecture slides` after the lecture.

Part I
Introduction to R

- History of R
- Why R?
- Running R
- Getting help!
- Installing package
- Loading package

History of R

- An implementation of the S language;
  (S-PLUS is another such an implementation but is commercial.)
- R was created by Ross Ihaka and Robert Gentleman in 1993 and is developed by the R Development Core Team;
- R is named partly after the first names of the first two R authors, and partly as a play on the name of S :)

Why R?

- An open source programming language (*i.e.* free for download)
- A software environment for statistical computing and graphics
- A large amount of add-on packages available
- High flexibility in syntax, *e.g.* no need to define an object's type in advance or upon declaration, though you may sacrifice a little speed.
- Source code is written primarily in `C`, `Fortran`, and `R`, therefore with a simple and efficient interface to `C` and `Fortran`.

[1, 2, 3]

Running R

Assuming that you have R installed.

Run R interactively by clicking R GUI or at a terminal prompt ($) typing R,

```
$ R
R version 2.14.2 (2012-02-29)
...
Type 'q()' to quit R.

>
```

and you will be put into an R *command line prompt* (>), where you can issue commands.

Run R by reading codes from a file in a terminal (semi-interactively),

```
> source("somefile.R")
```

Run R from commandline (non-interactively),

```
$ R --vanilla --silent --slave --file=somefile.R
```

Getting help!

Access R help system,

```
> help(mean)
> help("mean")
> ?mean                             # an alternative
```

Note: The hash symbol (#) is to comment until the end of the line.

Search by fuzzy matching using a keyword,

```
> help.search("heatmap")
> ??heatmap                         # an alternative

Help files with alias or concept or title matching 'heatmap' using fuzzy matching:

GMD::get.sep                  Get row or column lines of separation for heatmap.3
GMD::heatmap.3                Enhanced Heatmap Representation with Dendrogram and
                              Partition
gplots::heatmap.2             Enhanced Heat Map
stats::heatmap                Draw a Heat Map

...
```

Note: Your screen might be different from mine - it depends on the packages you have installed!

Search online,

```
> RSiteSearch("heatmap")
A search query has been submitted to http://search.r-project.org
The results page should open in your browser shortly
```

Installing package

Install from CRAN by `install.packages`,

```
> install.packages("gplots")
```

Then you are asked to select a `CRAN` mirror for use.

Install through `Bioconductor`,

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("ChromHeatMap") # DO NOT RUN; it might take some time for downloading
```

Get more information from `http://www.bioconductor.org/install/`

Loading package

Load package by `library` and `require`,

```
library(gplots)
require(gplots)
```

... if the package does not exist,

```
> library(gplots)
Error in library(gplots) : there is no package called 'gplots'
> require(gplots)
Loading required package: gplots
Warning message:
In library(package, lib.loc = lib.loc, character.only = TRUE, logical.return = TRUE,  :
  there is no package called 'gplots'
```

Note: Please refer to the *help documentation* for the (small) differences between the `library` and `require`.

Part II
R basics

- ❍ Working directory
- ❍ Expressions and assignments
- ❍ Assignment operators
- ❍ Naming convention
- ❍ Types of objects
- ❍ Constants
- ❍ Special values
- ❍ Data structures
- ❍ Oprations
- ❍ Control flows
- ❍ Function

Working directory

Get the working directory (WD) by `getwd`,

```
> getwd()
[1] "/Users/xiaobei/Project/RCrash"
> ## list directory contents
> dir()
[1] "TMP"
```

Set the working directory by `setwd`,

```
> ## store current directory
> oldDir <- getwd()
> ## set wd
> setwd("TMP")
> ## check whether the specified wd is set
> getwd()
[1] "/Users/xiaobei/Project/RCrash/TMP"
> ## switch back
> setwd(oldDir)
> getwd()
[1] "/Users/xiaobei/Project/RCrash"
```

Note:

- `setwd` takes either an absolute or a relative path.
- These actions can also be done by selecting options from R GUI Menu.

Expressions and assignments

Elementary R commands consist of either expressions or assignments.

Evaluate expressions,

```
> 5+6
[1] 11
```

Assign values to variables,

```
> x <- 5+6
> x
[1] 11
> y <- x+7
> y
[1] 18
```

Assignment operators

Two ways to assign a value to a name,

```
x <- value
x = value
```

Get more options for `assignment operators`,

```
> help(assignOps,package="base")
```

Naming convention

Legal names in R fowllows these rules,

- must start with a letter (A-Z or a-z)
- can contain letters, digits (0-9), periods (".") or underscore ("_")
- *case-sensitive*

Note:

- Good names are *self-explanatory*;
- Avoid assigning names of predefined R objects (*e.g.* constants, functions, etc.);
- The underscore had a different meaning in very old versions of R and was not allowed in variable names, though it is commonly used now.

Types of objects

R supports a few basic types of objects: `integer`, `numeric`, `logical`, `character` (string), `factor`, etc. - they can be mixed to build more complex objects or data structures.

```
integer
numeric
logical
character
factor
```

Types - Integer and Numeric

The `numeric` in R means "double", i.e. internally stored as a double precision floating point number.

```
> typeof(1)
[1] "double"
> is.numeric(1)
[1] TRUE
>
> typeof(as.integer(1))
[1] "integer"
> is.numeric(as.integer(1))
[1] TRUE
>
> typeof(1:3)
[1] "integer"
> typeof(matrix(1:12,ncol=4))
[1] "integer"
```

Note:

- In R, a single integer number is stored as a double precision float by default;
- An `ingeter` is always a `numeric`. But `numeric` also includes the fraction ($2/3$) and the irrational numbers (π).

Types - Integer and Numeric (Cont'd)

However, you can force an object to be, *e.g.* an integer or a double precision float.

```
> as.integer(12)
[1] 12
> as.double(12)
[1] 12
```

Types - logical

The `logical` (or boolean) data type is used to store `TRUE`/`FALSE` data,

```
> ?logical

...
Description:

     Create or test for objects of type "logical", and the basic
     logical constants.

Usage:

     TRUE
     FALSE
     T; F

     logical(length = 0)
     as.logical(x, ...)
     is.logical(x)
...
```

Note: The two logical values are in capitalized letters. And the shorter formats (T/F) are not recommended.

Types - logical (Cont’d)

Convert other types into `logical`,

```
> as.logical("TRUE")
[1] TRUE
> as.logical("True")
[1] TRUE
> as.logical("true")
[1] TRUE
> as.logical("T")
[1] TRUE
> as.logical("t")                     # wont't work
[1] NA
> as.logical("r")                     # wont't work
[1] NA
>
> as.logical(1)
[1] TRUE
> as.logical(10)
[1] TRUE
> as.logical(0)
[1] FALSE
> as.logical(-1)
[1] TRUE
> as.logical(-1.5)
[1] TRUE
```

Note:

- ❍ It is recommanded to use capitalized characters to convert, though other variants might also work.
- ❍ Any numeric other than zero is `TRUE` after conversion.

Types - character

A `character` variable (or a string) is any number of characters enclosed within a pair of double (") or single (') quotes. It can contain any combination of letters, numbers, symbols and spaces.

Create a vector of strings,

```
> c("abc",'123', "Hello, R!", "")
[1] "abc"       "123"       "Hello, R!" ""
```

Note:

- the last one is an *empty* string, which contains no characters.
- The quotes should be in straight vertical, or "typewriter" style. "Smart" curly quotes (“, ”, ‘ or ’) won’t be recognized by R.

Types - character (Cont'd)

An example of an invalid string would be,

```
> ""Thank you", she said."
Error: unexpected symbol in """Thank"
```

Note: When R hits the second quote, it assumes the string ends there; the continuing text (*Thank...*) causes an error.

We can circumvent this problem by using backslash (\) to escape,

```
> "\"Thank you\", she said."
[1] "\"Thank you\", she said."
```

Or by using different quote types,

```
> '"Thank you", she said.'
[1] "\"Thank you\", she said."
> "Breakfast at Tiffany's"
[1] "Breakfast at Tiffany's"
```

Convert other types into `character`,

```
> as.character(TRUE)      # convert from a logical
[1] "TRUE"
> as.character(12)        # convert from a numeric
[1] "12"
```

Types - factor

`factor` variables are categorical with discrete `levels`. The `levels` can be represented by integers (not recommended) or characters.

Create a `factor` variable without specified levels

```
> x <- c("low","high","medium","high","high","low")
> x
[1] "low"    "high"   "medium" "high"   "high"   "low"
>
> y <- factor(x)
> y
[1] low    high   medium high   high   low
Levels: high low medium
> sort(y)
[1] high   high   high   low    low    medium
Levels: high low medium
```

Note: the `factor` variable with unspecified levels follows alphabetical order.

Types - factor (Cont’d)

Create a `factor` with specified levels

```
> y <- factor(x, levels=c("low", "medium", "high"))
> y
[1] low    high   medium high   high   low
Levels: low medium high
> sort(y)
[1] low    low    medium high   high   high
Levels: low medium high
```

Note: Now the `factor` variable follows the order of specified levels - this is one of the advantages of the `factor` over `character`.

Constants

Build-in constants in R,

```
LETTERS
letters
month.abb
month.name
pi
```

Examples,

```
> help(Constants,package="base")
> pi
[1] 3.141593
> print(pi, digits=16)
[1] 3.141592653589793
>
> letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
[16] "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

Special values

We have met two special values: `TRUE` and `FALSE` that are words reserved in R.

There are more special values:

- `NA`: the missing value indicator
- `NULL`: the null object
- More: `Inf`, `NaN`, ...

You can get more information from the help system! See `R_crash.R`.

Special values (cont'd)

Examples,

```
> x <- NA
> is.na(x)
[1] TRUE
> length(x)
[1] 1
>
> y <- NULL
> is.null(y)
[1] TRUE
> length(y)
[1] 0
>
> 1/0
[1] Inf
> -1/0
[1] -Inf
> 0/0
[1] NaN
```

Note: `NA` has a length of 1 and `NULL` has a length of 0.

Part II (Cont’d)
R basics - Data structures

- vector
- list
- matrix
- data.frame

vector

`vector` is a collection of elements.

```
> letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
[15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
> is.vector(letters)
[1] TRUE
> length(letters)
[1] 26
>
> ## a vector of length one
> pi
[1] 3.141593
> is.vector(pi)
[1] TRUE
> length(pi)
[1] 1
```

Operations on vector

Create a vector by generating a sequence using the colon mark (`:`) or `seq`
The syntax,

```
from:to

seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)
```

Examples,

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> 10:1
 [1] 10  9  8  7  6  5  4  3  2  1
> seq(1,10)
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(1,10,2)
[1] 1 3 5 7 9
> seq(1,10,length.out=3)
[1]  1.0  5.5 10.0
```

Operations on vector (Cont’d)

Create a vector by combining elements using c,

```
> c(2,4,10,"z","a")
[1] "2"  "4"  "10" "z"  "a"
> x <- c(21:25,letters[1:5],NA)
> x
 [1] "21" "22" "23" "24" "25" "a"  "b"  "c"  "d"  "e"  NA
> typeof(x)
[1] "character"
> mode(x)
[1] "character"
```

Note: `c` produced a "simple" vector that consists of objects of same `type` (or `mode`), by converting `numeric` into `character`.

Operations on vector (Cont'd)

Indexing (also called subsetting),

```
> x <- seq(21,30)
> x
 [1] 21 22 23 24 25 26 27 28 29 30
> x[1]                   # the first element
[1] 21
> x[length(x)]           # the last element
[1] 30
> x[2:4]                 # the 2nd, 3rd and 4th elements
[1] 22 23 24
> x[c(1,3,5)]            # the 1st, 3rd and 5th elements
[1] 21 23 25
```

Note: R indices start at 1, not 0!

Operations on vector (Cont'd)

Replacement,

```
> x
 [1] 21 22 23 24 25 26 27 28 29 30
> x[2] <- 99
> x
 [1] 21 99 23 24 25 26 27 28 29 30
> x[3] <- NA
> x
 [1] 21 99 NA 24 25 26 27 28 29 30
```

Operations on vector (Cont'd)

Add element(s) by `[]`,

```
> x <- 1:5
> x[10] <- 99
> x
 [1]  1  2  3  4  5 NA NA NA NA 99
> x <- letters[1:3]
> x[10] <- "hi!"
> x
 [1] "a"   "b"   "c"   NA    NA    NA    NA    NA    NA    "hi!"
```

Note: A new element is added to the end of the 'vector' by a new index. If the index is not continuous with current indices, NA values are inserted.

list

`list` is a special kind of `vector`, usually consisting of elements of different types.

Create a list by `list`,

```
> x <- list(21:25, letters[1:6])                # unnamed list
> x
[[1]]
[1] 21 22 23 24 25

[[2]]
[1] "a" "b" "c" "d" "e" "f"

> is.list(x)                                    # check if it is a `list'
[1] TRUE
> is.vector(x)                                  # check if it is a `vector'
[1] TRUE
> length(x)                                     # get the length
[1] 2
> str(x)                                        # display the structure
List of 2
 $ : int [1:5] 21 22 23 24 25
 $ : chr [1:6] "a" "b" "c" "d" ...
```

Note:

- The element indices are indicated by `[[` and `]]`;
- `str` is a useful function to display the *structure* of an R object.

list (Cont'd)

Create a list by `list` with named elements,

```
> x <- list(number=21:25, letter=letters[1:6])  # named list
> x
$number
[1] 21 22 23 24 25

$letter
[1] "a" "b" "c" "d" "e" "f"

> names(x)                                      # get the names
[1] "number" "letter"
```

Note: The element names are indicated by `$`.

Operations on list

Create a new list by combining elements using c,

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

[[2]]
[1] "a" "b" "c" "d" "e" "f"

[[3]]
[1] 3.141593

> str(y)
List of 3
 $ : int [1:5] 21 22 23 24 25
 $ : chr [1:6] "a" "b" "c" "d" ...
 $ : num 3.14
>
> ## try this?
> y2 <- c(x,pi,1:3)
```

Note: c produced a ‘list’ (a *complex* vector), when at least one of the elements is a ‘list’.

Operations on list (Cont'd)

Convert to a list,

```
> x3
  id label measure
1  1     a     1.0
2  2     b     5.5
3  3     c    10.0
> as.list(x3)
$id
[1] 1 2 3

$label
[1] "a" "b" "c"

$measure
[1]  1.0  5.5 10.0
```

Note: It is now converted to a list with elements in equal length - every element has a length of 3!

Operations on list (Cont'd)

Let's look at a list

```
> x <- list(number=21:25, letter=letters[1:6], unit=c("C","F"))
> x
$number
[1] 21 22 23 24 25

$letter
[1] "a" "b" "c" "d" "e" "f"

$unit
[1] "C" "F"
```

Indexing: to access a single element by `[[ ]]` or `$somename`,

```
> x[[1]]
[1] 21 22 23 24 25
> x$unit
[1] "C" "F"
```

Indexing: to obtain a (sub)list of element(s) by `[ ]`,

```
> x[c(1,3)]
$number
[1] 21 22 23 24 25

$unit
[1] "C" "F"
```

Operations on list (Cont’d)

Replacement

Replace an entire element,

```
> x <- list(letter=letters[1:6], unit=c("C","F"))
> x
$letter
[1] "a" "b" "c" "d" "e" "f"

$unit
[1] "C" "F"

> x$letter <- letters[7:1]
> x
$letter
[1] "g" "f" "e" "d" "c" "b" "a"

$unit
[1] "C" "F"
```

Replace a sub-element inside an element,

```
> x$letter[1] <- "z"
> x
$letter
[1] "z" "f" "e" "d" "c" "b" "a"

$unit
[1] "C" "F"
```

Operations on list (Cont'd)

Add element(s) by `[[]]`,

```
> x <- list(letter=letters[1:6], unit=c("C","F"))
> x
$letter
[1] "a" "b" "c" "d" "e" "f"

$unit
[1] "C" "F"

> length(x)
[1] 2
> x[[5]] <- 21:25
> x
$letter
[1] "a" "b" "c" "d" "e" "f"

$unit
[1] "C" "F"

[[3]]
NULL

[[4]]
NULL

[[5]]
[1] 21 22 23 24 25
```

Note: A new element is added to the end of the 'list' by a new index. If the index is not continuous with current indices, `NULL` values are inserted.

Operations on list (Cont'd)

Add element(s) by $,

```
> x <- list(letter=letters[1:6], unit=c("C","F"))
> x
$letter
[1] "a" "b" "c" "d" "e" "f"

$unit
[1] "C" "F"

> x$id <- 1:3
> x
$letter
[1] "a" "b" "c" "d" "e" "f"

$unit
[1] "C" "F"

$id
[1] 1 2 3

> x[[3]]
[1] 1 2 3
```

Note: A new element is added to the end of the 'list' by a new name. Its index is continuous with current indices.

matrix

`matrix` is a rectangular array of element of same type.

Create a 2D matrix by `matrix`
The syntax,

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

```
> matrix(1:12,ncol=4)                  # fill in elements by column
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> matrix(1:12,ncol=4,byrow=TRUE)    # fill in elements by row
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

Note: To create multi-dimentional matrix, please check `array`. (Not required for the course)

Operations on matrix

Creat a matrix and save to a variable,

```
> x <- matrix(1:12,ncol=4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Get size,

```
> dim(x)                              # retrieve the dimension
[1] 3 4
> nrow(x)                             # return the number of rows
[1] 3
> ncol(x)                             # return the number of columns
[1] 4
> length(x)                           # total number of elements
[1] 12
```

Note: `length` returns the total number of elements in `matrix`, compared with that of `data.frame` on Page 58.

Set size,

```
> dim(x) <- c(4,3)
> x
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Operations on matrix (Cont'd)

Get names,

```
> rownames(x)                             # retrieve row names
NULL
> colnames(x)                             # retrieve column names
NULL
```

Set names,

```
> rownames(x) <- 1:nrow(x)                # set the row names
> colnames(x) <- letters[1:ncol(x)] # set the column names
> rownames(x)
[1] "1" "2" "3" "4"
> colnames(x)
[1] "a" "b" "c"
> x
  a b  c
1 1 5  9
2 2 6 10
3 3 7 11
4 4 8 12
```

Operations on matrix (Cont'd)

Indexing by row/column numbers using [and],

```
> x[c(1,3),]                         # get rows by row numbers
  a b  c
1 1 5  9
3 3 7 11
> x[,2:3]                            # get columns by column numbers
  b  c
1 5  9
2 6 10
3 7 11
4 8 12
```

Indexing by row/column names,

```
> x["2",]                            # by row names
 a  b  c
 2  6 10
> x[,c("a","c")]                     # by column names
  a  c
1 1  9
2 2 10
3 3 11
4 4 12
```

Indexing by both row and column,

```
> x[1:2,2:3]
  b  c
1 5  9
2 6 10
```

Operations on matrix (Cont'd)

Indexing by conditions.

Select rows by values in the 1st ("a") column,

```
> x
  a b  c
1 1 5  9
2 2 6 10
3 3 7 11
4 4 8 12
> x[x[,"a"]>=2,]
  a b  c
2 2 6 10
3 3 7 11
4 4 8 12
```

Select columns by values in the 2nd row,

```
> x[,x[2,]>5]
  b  c
1 5  9
2 6 10
3 7 11
4 8 12
```

Operations on matrix (Cont'd)

Select given combined conditions on both a row and a column,

```
> x[x[,"a"]>2,x[1,]>=5]
  b  c
3 7 11
4 8 12
```

Operations on matrix (Cont'd)

Replacement - similar as vectors but involving two dimensions.

Creat a matrix with missing values,

```
> y <- matrix(ncol=4,nrow=4)
> y
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
[4,]   NA   NA   NA   NA
```

Replace the diagonal with zeros by indices,

```
> y[1,1] <- y[2,2] <- y[3,3] <- y[4,4] <- 0
> y
     [,1] [,2] [,3] [,4]
[1,]    0   NA   NA   NA
[2,]   NA    0   NA   NA
[3,]   NA   NA    0   NA
[4,]   NA   NA   NA    0
```

Operations on matrix (Cont’d)

Replace the lower triangle with numbers by conditions,

```
> row(y)>col(y)
      [,1]  [,2]  [,3]  [,4]
[1,] FALSE FALSE FALSE FALSE
[2,]  TRUE FALSE FALSE FALSE
[3,]  TRUE  TRUE FALSE FALSE
[4,]  TRUE  TRUE  TRUE FALSE
> y[row(y)>col(y)] <- 1:sum(row(y)>col(y))
> y
     [,1] [,2] [,3] [,4]
[1,]    0   NA   NA   NA
[2,]    1    0   NA   NA
[3,]    2    4    0   NA
[4,]    3    5    6    0
```

Operations on matrix (Cont'd)

Add row(s) using `rbind`,

```
> x <- matrix(1:8,ncol=2)
> x
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
>
> rbind(x,11:12)
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
[5,]   11   12
> rbind(x,11:12,21:22)              # add multiple rows at once
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
[5,]   11   12
[6,]   21   22
```

Operations on matrix (Cont’d)

Add column(s) using `cbind`,

```
> x <- matrix(1:8,ncol=2)
> x
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
>
> cbind(x,11:14)
     [,1] [,2] [,3]
[1,]    1    5   11
[2,]    2    6   12
[3,]    3    7   13
[4,]    4    8   14
> cbind(x,11:14,21:24)              # add multiple columns at once
     [,1] [,2] [,3] [,4]
[1,]    1    5   11   21
[2,]    2    6   12   22
[3,]    3    7   13   23
[4,]    4    8   14   24
```

Operations on matrix (Cont'd)

Add a column when the vector is shorter,

```
> x <- matrix(1:8,ncol=2)
> x
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
>
> cbind(x,11)
     [,1] [,2] [,3]
[1,]    1    5   11
[2,]    2    6   11
[3,]    3    7   11
[4,]    4    8   11
> cbind(x,c(21,22))
     [,1] [,2] [,3]
[1,]    1    5   21
[2,]    2    6   22
[3,]    3    7   21
[4,]    4    8   22
> cbind(x,c(31,32,33))
     [,1] [,2] [,3]
[1,]    1    5   31
[2,]    2    6   32
[3,]    3    7   33
[4,]    4    8   31
Warning message:
In cbind(x, c(31, 32, 33)) :
  number of rows of result is not a multiple of vector length (arg 2)
```

Note: The shorter vector is recycled, with a warning when elements in the vector were not equally reused. Similar scenario when adding a row.

Operations on matrix (Cont'd)

Combine two matrices,

```
> x <- matrix(1:12,ncol=3)
> x
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> y4 <- matrix(16:21,ncol=3)
> y4
     [,1] [,2] [,3]
[1,]   16   18   20
[2,]   17   19   21
> rbind(x,y4)
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
[5,]   16   18   20
[6,]   17   19   21
```

... if corresponding dimensions do not match?

```
> cbind(x,y4)
Error in cbind(x, y4) : number of rows of matrices must match (see arg 2)
```

Operations on matrix (Cont'd)

Add cell(s) by new indices?

```
> x <- matrix(1:12,ncol=3)
> x[5,6] <- 99
Error in x[5, 6] <- 99 : subscript out of bounds
> x[5,] <- 99
Error in x[5, ] <- 99 : subscript out of bounds
> x[,6] <- 99
Error in x[, 6] <- 99 : subscript out of bounds
```

Note: It is not doable for specified cell(s)/row(s)/column(s) - out of bounds!

However, if we treat the matrix as a vector it is doable. But the dimensions are flattened.

```
> x <- matrix(1:12,ncol=3)
> x[20] <- 99
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 NA NA NA NA NA NA NA 99
```

data.frame

`data.frame` is a rectangular array of elements, usually of different types of columns. It is a special form of list, *i.e.* a list with equal-length elements.

Create a data.frame by `data.frame`
The syntax,

```
data.frame(..., row.names = NULL, check.rows = FALSE,
           check.names = TRUE,
           stringsAsFactors = default.stringsAsFactors())
```

Create a data.frame without specifying the names,

```
> x <- data.frame(1:3,letters[1:3],seq(1,10,length.out=3))
> x
  X1.3 letters.1.3. seq.1..10..length.out...3.
1    1            a                        1.0
2    2            b                        5.5
3    3            c                       10.0
```

Create a data.frame with specified names,

```
> x2 <- data.frame(id=1:3,label=letters[1:3],measure=seq(1,10,length.out=3))
> x2
  id label measure
1  1     a     1.0
2  2     b     5.5
3  3     c    10.0
```

Operations on data.frame

Display the structure,

```
> str(x2)
'data.frame':   3 obs. of  3 variables:
 $ id     : int  1 2 3
 $ label: Factor w/ 3 levels "a","b","c": 1 2 3
 $ measure: num  1 5.5 10
```

Note: `data.frame` convertes character vectors to factors by default.

But this can be switched by setting `stringsAsFactors` to `FALSE`,

```
> x3 <- data.frame(id=1:3,label=letters[1:3],measure=seq(1,10,length.out=3),
+                  stringsAsFactors=FALSE)
> x3
  id label measure
1  1     a    1.0
2  2     b    5.5
3  3     c   10.0
> str(x3)
'data.frame':   3 obs. of  3 variables:
 $ id     : int  1 2 3
 $ label: chr  "a" "b" "c"
 $ measure: num  1 5.5 10
```

Operations on data.frame (Cont’d)

Size and names (similar as in matrix),

```
> x <- data.frame(id=1:3,label=letters[1:3],measure=seq(1,10,length.out=3),
+                 stringsAsFactors=FALSE)
> x
  id label measure
1  1     a     1.0
2  2     b     5.5
3  3     c    10.0
> dim(x)                            # retrieve the dimension
[1] 3 3
> length(x)                         # number of columns
[1] 3
> rownames(x)
[1] "1" "2" "3"
> colnames(x)
[1] "id"      "label"   "measure"
```

Note: `length` returns the number of columns in `data.frame`; while the total number of elements in `matrix` (See Page 44) because the former is a *special* `list` and the latter a *special* `vector`.

Operations on data.frame (Cont’d)

Indexing (similar as in matrix),

```
> x <- data.frame(id=1:3,label=letters[1:3],measure=seq(1,10,length.out=3),
+                   stringsAsFactors=FALSE)
> x
  id label measure
1  1     a     1.0
2  2     b     5.5
3  3     c    10.0
> x[1:2,]
  id label measure
1  1     a     1.0
2  2     b     5.5
> x[,3]
[1]  1.0  5.5 10.0
> x[,c("id","measure")]
  id measure
1  1     1.0
2  2     5.5
3  3    10.0
```

Columns can also be accessed via $

```
> x$measure
[1]  1.0  5.5 10.0
```

Operations on data.frame (Cont'd)

Replacement (similar as in matrix),

```
> x <- data.frame(id=1:3,label=letters[1:3],measure=seq(1,10,length.out=3),
+                  stringsAsFactors=FALSE)
> x
  id label measure
1  1     a     1.0
2  2     b     5.5
3  3     c    10.0
> x$measure <- x$measure*2
> x
  id label measure
1  1     a       2
2  2     b      11
3  3     c      20
> x[2,2:3] <- c("f",50)
> x
  id label measure
1  1     a       2
2  2     f      50
3  3     c      20
```

Operations on data.frame (Cont'd)

Add row(s) by `rbind`

```
> x <- data.frame(id=1:3,label=letters[1:3],measure=seq(1,10,length.out=3),
+                 stringsAsFactors=FALSE)
> x
  id label measure
1  1     a     1.0
2  2     b     5.5
3  3     c    10.0
> x0 <- c(0,"control",0)
> y <- rbind(x0, x)
> y
  id   label measure
1  0 control       0
2  1       a       1
3  2       b     5.5
4  3       c      10
> class(y)
[1] "data.frame"
```

Operations on data.frame (Cont'd)

Add column(s) by `cbind`

```
> x <- data.frame(id=1:3,label=letters[1:3],measure=seq(1,10,length.out=3),
+                  stringsAsFactors=FALSE)
> x
  id label measure
1  1     a     1.0
2  2     b     5.5
3  3     c    10.0
> x0 <- c("BRIC","BINF","BINF")
> y <- cbind(x,where=x0)
> y
  id label measure where
1  1     a     1.0  BRIC
2  2     b     5.5  BINF
3  3     c    10.0  BINF
> str(y)
'data.frame':   3 obs. of  4 variables:
 $ id     : int  1 2 3
 $ label  : chr  "a" "b" "c"
 $ measure: num  1 5.5 10
 $ where  : Factor w/ 2 levels "BINF","BRIC": 2 1 1
```

Add column(s) by $

```
> y$who <- c("xb","xb","as")
> y
  id label measure where who
1  1     a     1.0  BRIC  xb
2  2     b     5.5  BINF  xb
3  3     c    10.0  BINF  as
```

Operations on data.frame (Cont’d)

Add row(s) by new indices?

```
> x <- data.frame(id=1:3,label=letters[1:3],measure=seq(1,10,length.out=3),
+                  stringsAsFactors=FALSE)
>
> x[4,] <- c(4,NA,NA)
> x[6,] <- c(6,NA,NA)
> x
  id label measure
1  1     a     1.0
2  2     b     5.5
3  3     c    10.0
4  4  <NA>      NA
5 NA  <NA>      NA
6  6  <NA>      NA
```

Note: It is doable with a new row index, either continuous or discontinuous with current indices. NA values are inserted for leaved "holes".

Operations on data.frame (Cont'd)

Add column(s) by new indices?

```
> x <- data.frame(id=1:3,label=letters[1:3],measure=seq(1,10,length.out=3),
+                 stringsAsFactors=FALSE)
> x
  id label measure
1  1     a     1.0
2  2     b     5.5
3  3     c    10.0
> x[,4] <- NA
> x
  id label measure V4
1  1     a     1.0 NA
2  2     b     5.5 NA
3  3     c    10.0 NA
> x[,6] <- NA
Error in `[<-.data.frame`(`*tmp*`, , 6, value = NA) :
  new columns would leave holes after existing columns
```

Note: It is doable only with a new column index that is continuous with current indices.

Operations on data.frame (Cont'd)

Add cell(s) by new indices?

```
x <- data.frame(id=1:3,label=letters[1:3],measure=seq(1,10,length.out=3),
+                  stringsAsFactors=FALSE)
> x
  id label measure
1  1     a     1.0
2  2     b     5.5
3  3     c    10.0
> x[4,1] <- 4
> x[6,1] <- 6
> x[1,4] <- NA
> x[1,6] <- NA
Error in `[<-.data.frame`(`*tmp*`, 1, 6, value = NA) :
  new columns would leave holes after existing columns
```

Note: Similar as above, it is doable for all new row indices but only for the new column index that is continuous.

Operations on data.frame (Cont'd)

Attach to access columns directly!

To show how `attach` works, let's first clean the workspace by removing all objects.
(You may save current workspace by `save.image` before cleaning; you can reload it by `load` afterwards!)

```
> save.image(file="workspace01.RData") # save current workspace
> rm(list=ls())                        # clean the workspace
> ls()                                 # list names of the objects in current workspace
character(0)
```

Note: `character(0)` indicates there's no object available in current workspace.

Operations on data.frame (Cont’d)

Now we create a data.frame and attach it,

```
> x <- data.frame(id=1:3,label=letters[1:3],measure=seq(1,10,length.out=3))
> x
  id label measure
1  1     a     1.0
2  2     b     5.5
3  3     c    10.0
> head(ls(pos=2))
[1] "acf"        "acf2AR"     "add.scope"  "add1"
[5] "addmargins" "aggregate"
> label                               # columns are not accessible directly.
Error: object 'label' not found
> attach(x)
> label                               # columns are accessible now!
[1] a b c
Levels: a b c
> id
[1] 1 2 3
> head(ls(pos=2))
[1] "id"      "label"   "measure"
```

Note:

❍ All columns are accessible by the names now;

❍ `ls(pos=2)` retrieves the names in *secondary* environment that has been changed after `attach`!

❍ Naming conflicts and unintended data overwrites can occur when attaching multiple data.frames that have names in common;

❍ The names can also conflict with those in the global environment.

Operations on data.frame (Cont'd)

It is a good habit to `detach` the data.frame that has been attached after use.

```
> detach(x)
```

Summary of R data structures

	vector	list	matrix	data.frame
homogeneous*	yes	no	yes	no
association		a vector with possible heterogeneous elements	a two dimens- ional vector	a special list of ele- ments in equal length
access an element	$[i]$	$[[i]]$ or $\$name$	$[i, j]$	$[i, j]$
access a column	n/a	n/a	$[, j]$	$[, j]$ or $\$name$
access a row	n/a	n/a	$[i,]$	$[i,]$

Note: *Here "homogeneous" means elements of the same type.

You would definitely add more to the table along the way learning R :)

Part II (Cont'd) R basics - Oprations

- Arithmetic
- Conditions
- Brackets
- Vector operation
- A series of apply's
- Indexing
- Sorting
- Tabulation

Arithmetic

The arithmetic operators,

```
x + y     addition
x - y     subtraction
x * y     multiplication
x / y     division
x ^ n     exponentiation
x %% y    mod
```

Get help!

```
> help(Arithmetic,package="base")
```

Arithmetic (Cont’d)

The arithmetic functions for vector,

`abs`	`absolute value`
`max`	`maximum`
`mean`	`arithmetic mean`
`min`	`minimum`
`range`	`returns a vector containing the minimum and maximum`
`sqrt`	`square root`
`sum`	`sum`

The arithmetic functions for matrix,

```
colSums
rowSums
colMeans
rowMeans
```

Conditions

`a == b`	equality
`a > b ( a >= b )`	greater than (or equal to)
`a < b ( a <= b )`	less than (or equal to)
`a != b`	inequality
`a & b; a && b`	logical AND
`a | b; a || b`	logical OR
`! a`	logical NOT

Get help!

```
> help(Logic,package="base")
```

Note: For logical AND and OR - “The shorter form performs elementwise comparisons in much the same way as arithmetic operators. The longer form evaluates left to right examining only the first element of each vector. Evaluation proceeds only until the result is determined.”

Brackets

Usage summary of the ,

Type	Symbol	Applications	Example
round brackets	( )	syntactic structure of expressions	
or parentheses		overriding operator precedence	`x*(y+z)`
		passing arguments to functions	`function(n){...}`
		invoking a function	`Fibonacci(10)`
		passing condition(s) to "if"	`if(a==b){...}`
		passing a "traversal" to "for"	`for(i in 1:10){...}`
square brackets	[ ]	indexing (subsetting)	`v[5]; m[1:3,]`
double square brackets	[[ ]]	access one element of a list	`x[[3]]`
curly brackets	{ }	define the beginning and ending	`function(n){...}`
or braces		of blocks of code	`if(a==b){...}`
			`for(i in 1:10){...}`

Note: the constructs of `function`, the `if` statement and the `for` statement will be talked in the coming slides.

Vector operation

One of the main advantages of R is vector operation - an operation performed in a vectorized way.

Let's first have vectorized arithmetic

```
> v <- 1:5
> v
[1] 1 2 3 4 5
> v**2
[1]  1  4  9 16 25
> v%%2
[1] 1 0 1 0 1
> v + c(1,3)                             # c(1,3) is recycled
[1] 2 5 4 7 6
Warning message:
In v + c(1, 3) :
  longer object length is not a multiple of shorter object length
```

Note: we got a similar "warning" due to incomplete recycling of the shorter vector.

Most R functions work in a vectorized way,

```
> toupper(c("a","b","c"))
[1] "A" "B" "C"
> sqrt(c(1,2,3))
[1] 1.000000 1.414214 1.732051
> paste("chr",c(1:22,"X","Y"),sep="")    # Concatenate Strings
 [1] "chr1"  "chr2"  "chr3"  "chr4"  "chr5"  "chr6"  "chr7"  "chr8"
 [9] "chr9"  "chr10" "chr11" "chr12" "chr13" "chr14" "chr15" "chr16"
[17] "chr17" "chr18" "chr19" "chr20" "chr21" "chr22" "chrX"  "chrY"
```

Apply’s

Some R functions (mostly user-defined) do not support vectorized operation - *e.g.* `Fibonacci` function we will create on Page 95.

There are a series of "apply" functions would apply a function over vector, or even list and matrix.

- ❍ `sapply` over `vector`;
- ❍ `lapply` over `list`;
- ❍ `apply` over `matrix` and `data.frame`.

Note: Here we listed the data structures that are commonly used with the apply’s but are not limited to.

Apply’s (Cont’d)

Apply `Fibonacci` (Page 95) over vector by `sapply`,

```
> x <- 1:10
> sapply(x,Fibonacci)
[[1]]
[1] 0

[[2]]
[1] 0 1

[[3]]
[1] 0 1 1

[[4]]
[1] 0 1 1 2

[[5]]
[1] 0 1 1 2 3

[[6]]
[1] 0 1 1 2 3 5

[[7]]
[1] 0 1 1 2 3 5 8

[[8]]
[1]  0  1  1  2  3  5  8 13

[[9]]
[1]  0  1  1  2  3  5  8 13 21

[[10]]
 [1]  0  1  1  2  3  5  8 13 21 34
```

Apply’s (Cont’d)

Apply over list by `lapply`,

```
> x <- list(a=1:5,b=letters[1:10])
> x
$a
[1] 1 2 3 4 5

$b
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

> lapply(x,length)
$a
[1] 5

$b
[1] 10

> length(x)
[1] 2
```

Apply’s (Cont’d)

Apply over matrix/data.frame by `apply`
Syntax,

```
apply(X, MARGIN, FUN, ...)
## MARGIN: 1 indicates rows,
##         2 indicates columns,
##         c(1, 2) indicates both rows and columns (i.e. all cells).
```

```
> x <- matrix(1:12,ncol=4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> apply(x,1,max)
[1] 10 11 12
> apply(x,2,max)
[1]  3  6  9 12
> apply(x,c(1,2),max)
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Indexing

We have used indexing in different data structures. Here we present two common indexing strategies.

Indexing by indices/positions,

```
> x <- data.frame(a=letters[1:6],b=7:12)
> x
  a  b
1 a  7
2 b  8
3 c  9
4 d 10
5 e 11
6 f 12
> x[c(1,3,5),]                          # select 1st, 3rd and 5th rows
  a  b
1 a  7
3 c  9
5 e 11
```

Indexing by conditions,

```
> x[(1:nrow(x))%%2==0,]                 # select even rows
  a  b
2 b  8
4 d 10
6 f 12
> x[x[,2]%%3 == 0,]                     # select rows if the elements in the 2nd column
                                        # is a multiple of 3.
  a  b
3 c  9
6 f 12
```

Sorting

Sort a vector by `sort`,

```
> mydata <- data.frame(label=letters[1:6],score=c(7, 13, 19, 11, 17, 5))
> mydata
  label score
1     a     7
2     b    13
3     c    19
4     d    11
5     e    17
6     f     5
> attach(mydata)
> sort(score)
[1]  5  7 11 13 17 19
> sort(score, decreasing=TRUE)          # sort in decreasing order
[1] 19 17 13 11  7  5
> detach(mydata)
```

Sorting (Cont'd)

Sort one vector by the other vector using `sort.list`,

```
> mydata
  label score
1     a     7
2     b    13
3     c    19
4     d    11
5     e    17
6     f     5
> attach(mydata)
> sort.list(score)                      # original indices of rearranged "score"
[1] 6 1 4 2 5 3
> label[sort.list(score)]               # rearrange "label" by "score"
[1] f a d b e c
Levels: a b c d e f
```

Similarly we can rearrange the whole data.frame,

```
> mydata[sort.list(score),]
  label score
6     f     5
1     a     7
4     d    11
2     b    13
5     e    17
3     c    19
> detach(mydata)
```

Sorting (Cont'd)

A graphical representation of sorting-related functions,

	R objects							Obtained in R
Original	**v.ori**	**7**	**13**	**19**	**11**	**17**	**5**	`v.ori`
	i.ori1	1	2	3	4	5	6	`1:length(v.ori)`
	i.new1	2	4	6	3	5	1	`rank(v.ori)`
New	**v.new**	**5**	**7**	**11**	**13**	**17**	**19**	`sort(v.ori) or v.ori[i.ori2]`
	i.ori2	6	1	4	2	5	3	`sort.list(v.ori)`
	i.new2	1	2	3	4	5	6	`1:length(v.new)`

v.ori: original vector - the input
v.new: new vector after rearrangement - the output
i.ori: index of elements in original vector, before(1) and after (2) rearrangement
i.new: index of elements in new vector, before(1) and after (2) rearrangement

Note:

- `sort` returns rearranged elements of a vector;
- `sort.list` returns rearranged indices (a permutation), which can order the original vector into a new vector that is the same as using `sort` directly, or order a third vector accordingly;
- `order` does almost the same action as does `sort.list`.

Sorting (Cont'd)

A graphical representation of sorting-related functions,

	R objects							Obtained in R
Original	**v.ori**	**7**	**13**	**19**	**11**	**17**	**5**	`v.ori`
	i.ori1	1	2	3	4	5	6	`1:length(v.ori)`
	i.new1	2	4	6	3	5	1	`rank(v.ori)`
New	**v.new**	**5**	**7**	**11**	**13**	**17**	**19**	`sort(v.ori) or v.ori[i.ori2]`
	i.ori2	6	1	4	2	5	3	`sort.list(v.ori)`
	i.new2	1	2	3	4	5	6	`1:length(v.new)`

v.ori: original vector - the input
v.new: new vector after rearrangement - the output
i.ori: index of elements in original vector, before(1) and after (2) rearrangement
i.new: index of elements in new vector, before(1) and after (2) rearrangement

Note (Cont'd):

- ❍ `rank` returns the ranks, *i.e.* indices of `v.new` before rearrangement.
- ❍ `sort.list` returns the indices of `v.ori` after rearrangement, therefore sometimes also called anti-ranks.

Tabulation

Let’s start with a data.frame

```
> x <- data.frame(id=1:8,
+                 gender=c("F", "M", "F", "M", "M", "M", "M", "M"),
+                 grade=c(1, 2, 2, 3, 2, 1, 2, 1)
+                 )
> x
  id gender grade
1  1      F     1
2  2      M     2
3  3      F     2
4  4      M     3
5  5      M     2
6  6      M     1
7  7      M     2
8  8      M     1
```

Build a contingency table by `table` over one vector,

```
> table(x$gender)
F M
2 6
```

Build a contingency table over multiple vectors,

```
> table(x$grade,x$gender)
    F M
  1 1 2
  2 1 3
  3 0 1
```

Tabulation (Cont'd)

Access an element in a 2D table by `dimnames`,

```
> mytable <- table(x$grade,x$gender)
> mytable
    F M
  1 1 2
  2 1 3
  3 0 1
> dimnames(mytable)         # a 2D table!
[[1]]
[1] "1" "2" "3"

[[2]]
[1] "F" "M"

> mytable["2","M"]          # retrieve the male at grade 2
[1] 3
```

Convert to `matrix` or `data.frame`,

```
> as.matrix(mytable)
    F M
  1 1 2
  2 1 3
  3 0 1
> as.data.frame(mytable)
  Var1 Var2 Freq
1    1    F    1
2    2    F    1
3    3    F    0
4    1    M    2
5    2    M    3
6    3    M    1
```

Tabulation (Cont'd)

Access an element in a 3D table by `dimnames`,

```
> x$specialty <- c("math", "bio", "bio", "math", "math", "math", "math", "bio")
> mytable2 <- table(x$grade,x$gender,x$specialty)
> mytable2
, ,  = bio
   F M
  1 0 1
  2 1 1
  3 0 0

, ,  = math
   F M
  1 1 1
  2 0 2
  3 0 1

> dimnames(mytable2)          # a 3D table!
[[1]]
[1] "1" "2" "3"

[[2]]
[1] "F" "M"

[[3]]
[1] "bio"  "math"

> mytable2[,,"math"]          # retrieve "math" specialists
   F M
  1 1 1
  2 0 2
  3 0 1

> mytable2[,"F","math"]       # retrieve female "math" specialists
1 2 3
1 0 0
> mytable2["3","M","math"]    # retrieve male "math" specialists at grade 3
[1] 1
```

Part II (Cont'd)
R basics - Control flows

There are two basic control-flow constructs of the R language,

- ❍ Logic-control flow by `if`/`else`
- ❍ Loop-control flow by `for`

Logic-control flow

Construct a logic-control flow by `if/else`
The syntax,

```
## variant 1
if ( condition ) {
command(s)
}

## variant 2
if ( condition ) {
command(s)
} else {
command(s)
}

## variant 3
if ( condition ) {
command(s)
} else if ( condition ) {
command(s)
} else {
command(s)
}
```

Logic-control flow (Cont'd)

Make a logic flow,

```
> mycoin <- function(x){
+   if (x<0 | x>1) {
+     print("Not a probability!")
+   } else if (x>=0 & x<=0.5) {
+     print("A head!")
+   } else {
+     print("A tail!")
+   }
+ }
>
> mycoin(-3)
[1] "Not a probability!"
> mycoin(0)
[1] "A head!"
> mycoin(0.2)
[1] "A head!"
> mycoin(0.5)
[1] "A head!"
> mycoin(0.7)
[1] "A tail!"
> mycoin(1)
[1] "A tail!"
> mycoin(1.5)
[1] "Not a probability!"
```

Loop-control flow

Construct a loop-control flow by `for`
The syntax,

```
for (var in seq) {
command1
command2
...
}
```

Make a loop flow to generate a sequence and let each element be "its left neighbour" doubled!

```
> n <- 10                           # set the number of elements required
> v <- numeric(n)                   # initialize a vector with length "n"
> v
 [1] 0 0 0 0 0 0 0 0 0 0
> v[1] <- 1                         # initialize the 1st element in "v"
> for (i in 2:length(v)){
+   v[i] <- v[i-1]*2
+ }
> print(v)
 [1]   1   2   4   8  16  32  64 128 256 512
```

Loop-control flow (Cont'd)

There are at least two ways to loop over a sequence,

Loop over the elements in a vector,

```
> v <- 1:3
> for (e in v){
+   print(e*2)
+ }
[1] 2
[1] 4
[1] 6
```

Loop over the indices of the elements in a vector,

```
> for (i in 1:length(v)){
+   print(v[i]*2)
+ }
[1] 2
[1] 4
[1] 6
```

Part II (Cont'd)
R basics - Function

Function

A function is a collection of R commands that performs a specific task, enabling the reuse of code within a program or across multiple programs.

Create a function by `function`
The syntax,

```
function (arglist) {
command1
command2
...
return(an.R.object)
}
```

Note: "an.R.object" is the name of the R object that is returned.

Function (Cont'd)

Let's make a function to generate `Fibonacci` numbers,

```
> Fibonacci <- function(n,x1=0,x2=1){
+   if(n<=0){
+     stop("`n' must be positive.")
+   }
+
+   v <- numeric(n)                 # initialize a vector with length "n"
+   v[1] <- x1                      # set the initial seeds
+   v[2] <- x2
+
+   if(n<=2){
+     return(v[1:n])
+   } else {
+     for (i in 3:n){
+       v[i] <- v[i-1]+v[i-2]
+     }
+   }
+   return(v)
+ }
>
> Fibonacci(10)                                 # using default
 [1]  0  1  1  2  3  5  8 13 21 34
> Fibonacci(10,5,6)                             # by positional arguments
 [1]   5   6  11  17  28  45  73 118 191 309
> Fibonacci(x1=5,x2=6,n=10)                     # by named arguments
 [1]   5   6  11  17  28  45  73 118 191 309
```

Note:

- The parameters 'x1' and 'x2' have default arguments (0 and 1);
- The order of unnamed arguments should follow the function definition; the order of named arguments is arbitrary;
- See also Page 77 for application with `apply`.

Part III
Case studies using biological data

Let’s look at case studies using

1. the famous Fisher’s iris data set
2. the Hair and Eye Color data set

, including applications in

- ❍ Data handling
- ❍ Visualization
- ❍ Statistics

Part III (Cont'd)
Case studies - Data handling

- Load data by `read.table`
- Save data by `write.table`

Data handling

This `iris` data set is similar as the build-in `iris` data in R - it is saved in a file "iris.txt" with column names in small letters! The file is in `Data sets for lectures > Datasets for R lectures`.
Load data as a data.frame by `read.table`
The syntax,

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
           dec = ".", row.names, col.names,
           as.is = !stringsAsFactors,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#",
           allowEscapes = FALSE, flush = FALSE,
           stringsAsFactors = default.stringsAsFactors(),
           fileEncoding = "", encoding = "unknown", text)
```

Note:

- "file" is the *path* of the data file.
- Since most of the parameters have default arguments, only arguments different from default need be specified.

Data handling (Cont'd)

Example,

```
> x <- read.table("iris.txt",header=TRUE,sep="\t")
> str(x)
'data.frame':   150 obs. of  5 variables:
 $ sepal.length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ sepal.width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ petal.length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ petal.width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
> head(x)
  sepal.length sepal.width petal.length petal.width species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
>
```

Note: Similar as `data.frame`, `read.table` convertes character vectors to factors by default.

Data handling (Cont'd)

Save data to a file by `write.table`

The syntax,

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"),
            fileEncoding = "")
```

Note: Similar as `read.table`, only arguments different from default need be specified.
Example,

```
> write.table(x,file="iris.txt",sep="\t",
+             row.names=FALSE,col.names=TRUE,
+             quote=FALSE)
```

Part III (Cont'd) Case studies - Visualization

- Making a scatter plot by `plot`
- Overlaying plots - add features to an existing plot: `points`, `lines`, `abline`, `legend`
- Making and arranging subplots
- Other plot functions: `qqnorm`, `barplot`, `hist`, `boxplot`
- lattice
- ggplot2
  (Details on the advanced methods - like `lattice` and `ggplot2` - will be covered in later lectures!)
- Saving plots

Visualization - Making a scatter plot

The syntax,

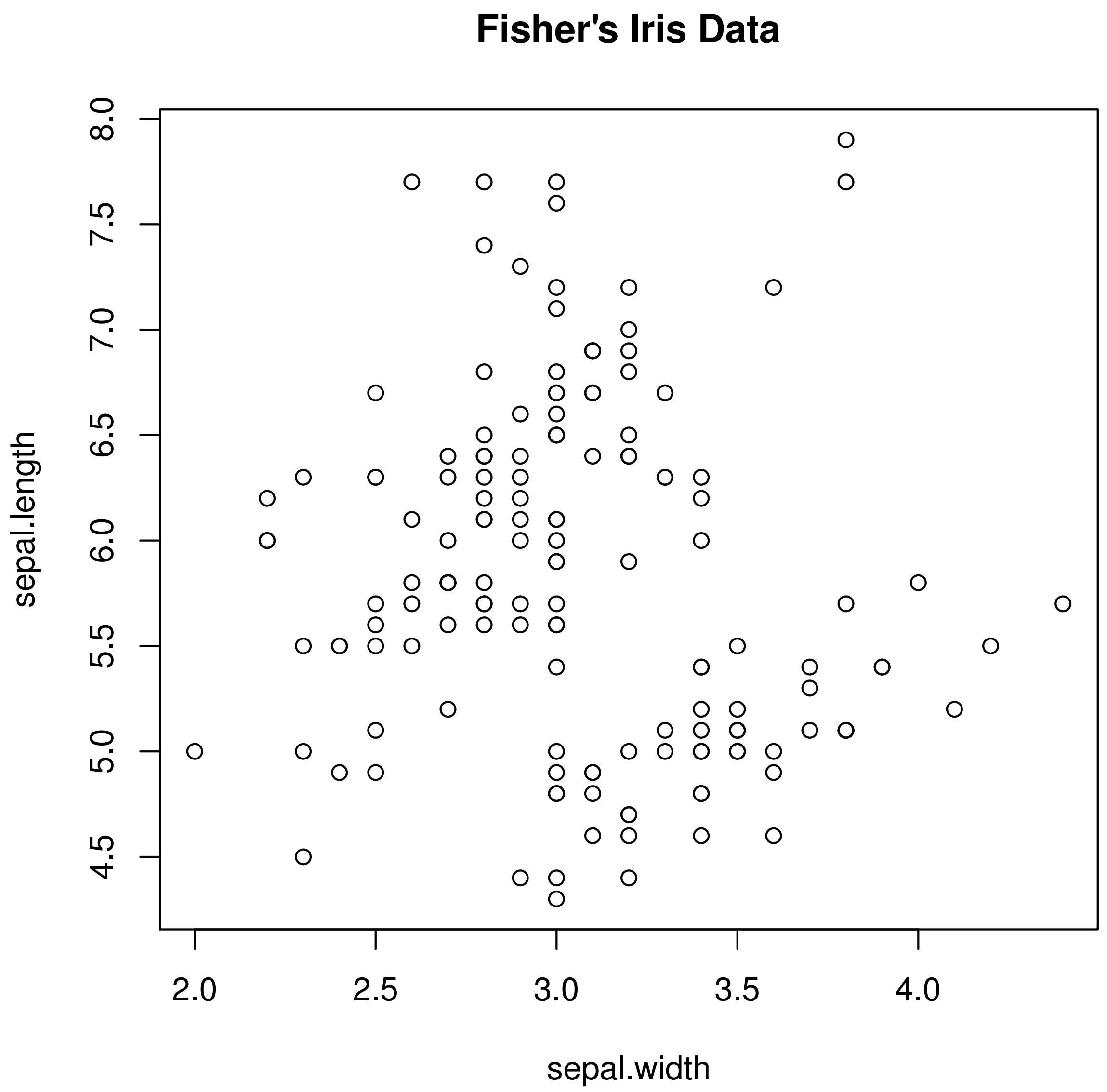
```
plot(x, y, ...)
```

Example,

```
> attach(x)
> plot(x=sepal.width,y=sepal.length,main="Fisher's Iris Data")
```

Visualization - Making a scatter plot

Visualize the plot,



Visualization - Making a scatter plot

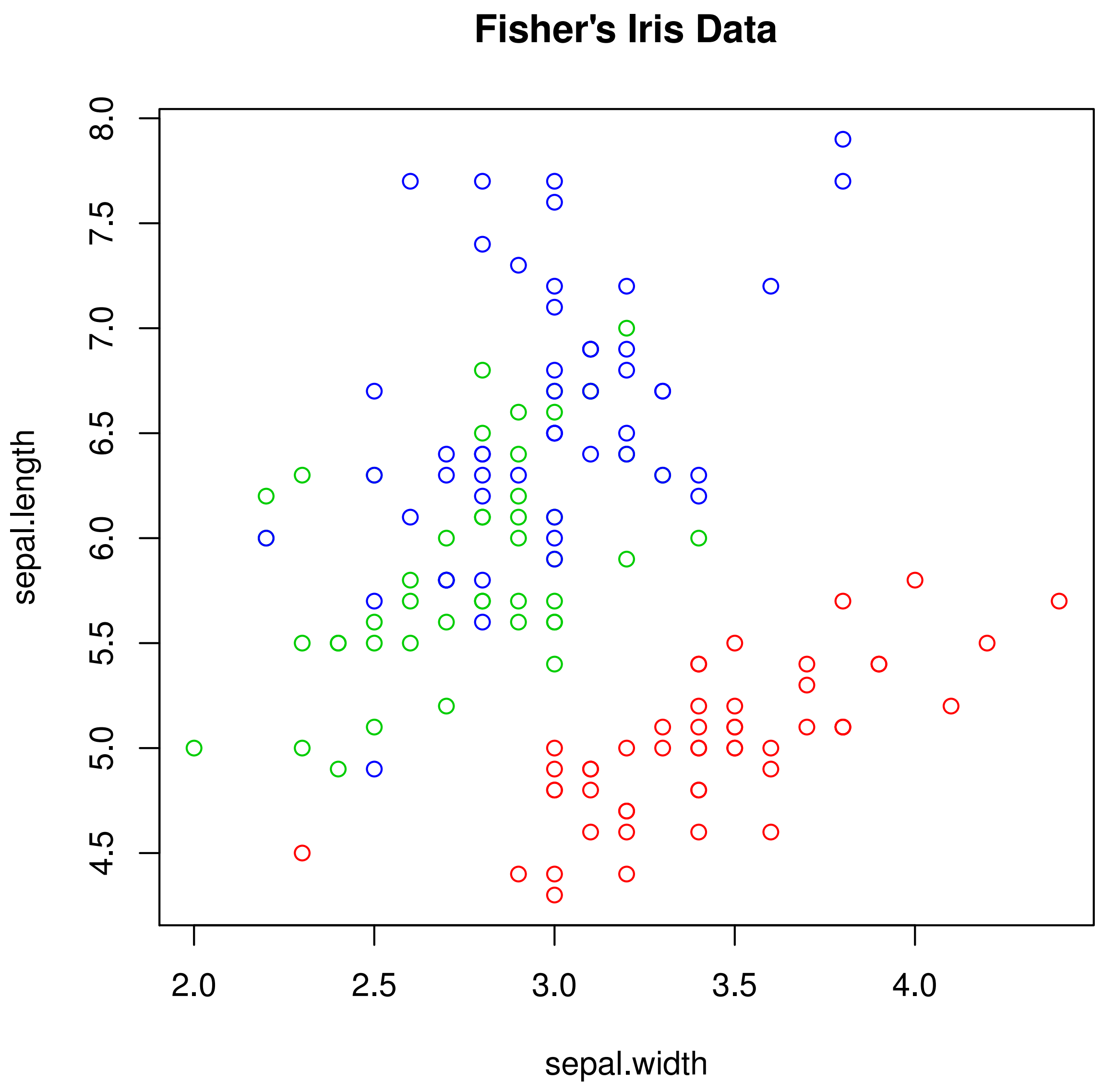
Color by species,

```
> mycolor <- c("red","green3","blue")
> plot(sepal.width,sepal.length,main="Fisher's Iris Data",
+      col=mycolor[species])
```

Note: "species" is a factor variable for indexing "mycolor".

Visualization - Making a scatter plot

Visualize the coloring,



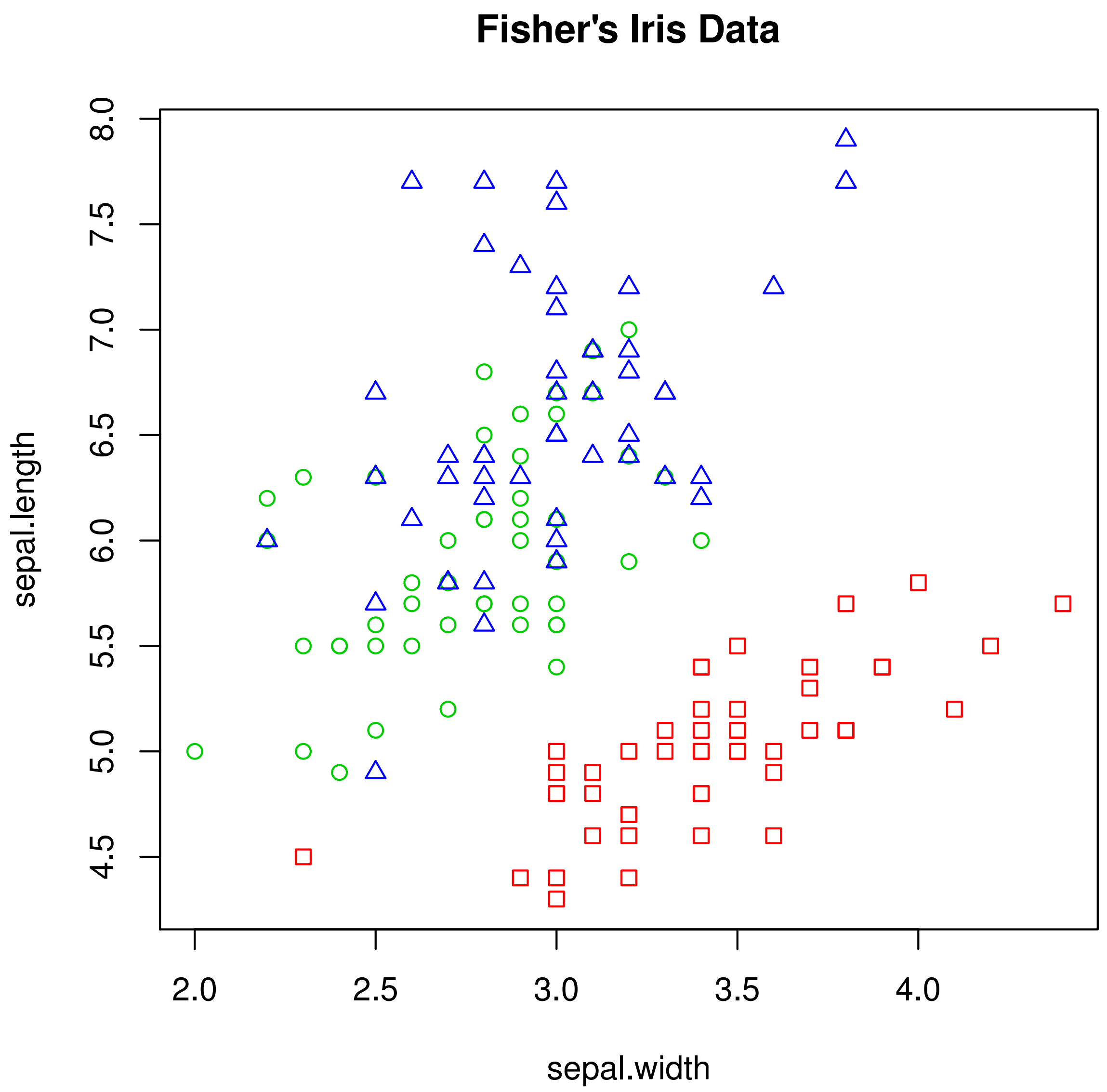
Visualization - Making a scatter plot

Point type by species,

```
> mypch <- 0:2
> plot(sepal.width,sepal.length,main="Fisher's Iris Data",
+      col=mycolor[species],
+      pch=mypch[species])
```

Visualization - Making a scatter plot

Visualize the point types,



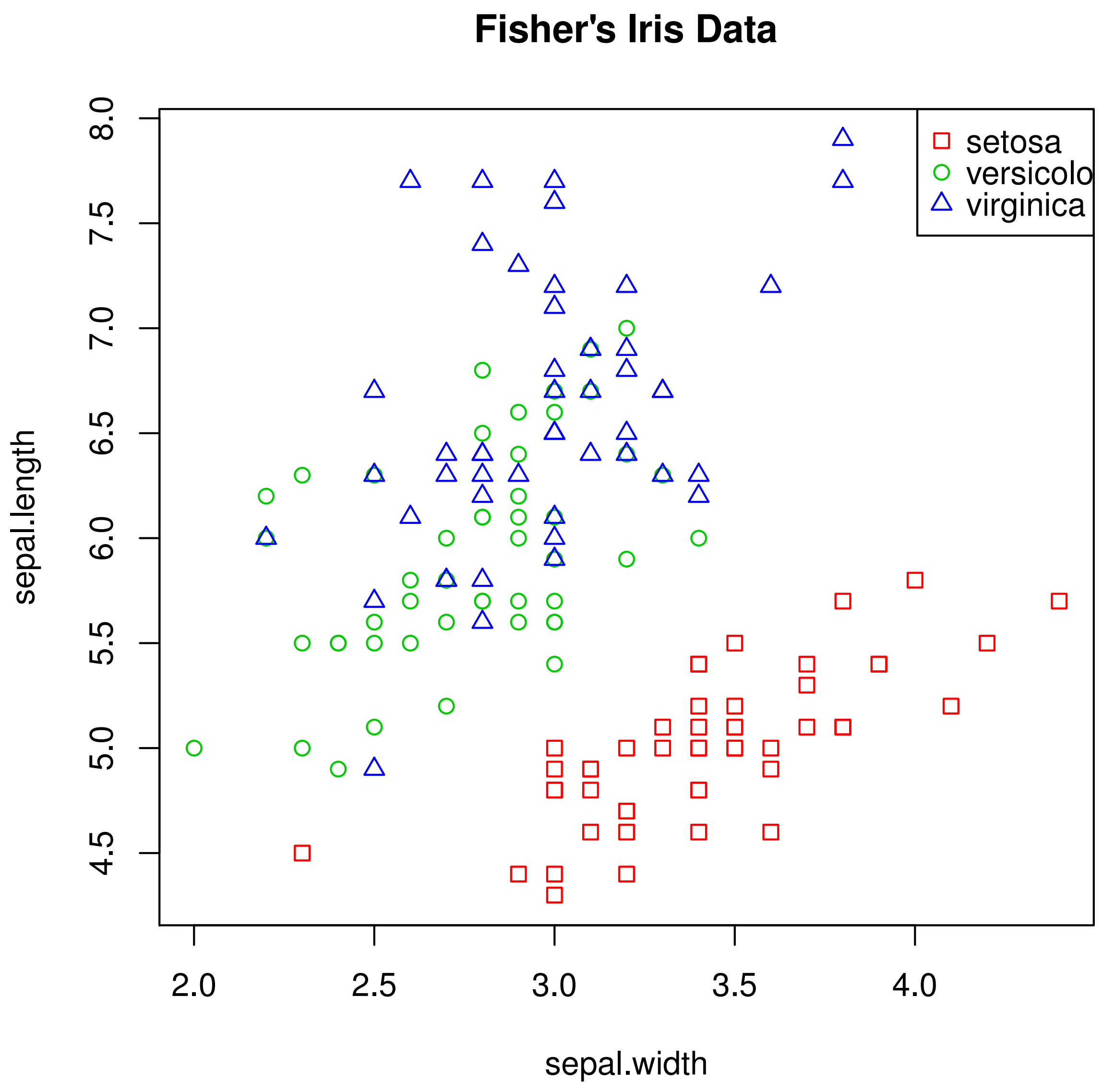
Visualization - Making a scatter plot

Add legends - to indicate species,

```
> legend("topright",legend=levels(species),
+        col=mycolor,pch=mypch)
```

Visualization - Making a scatter plot

Visualize the legend,



Visualization - Making a scatter plot

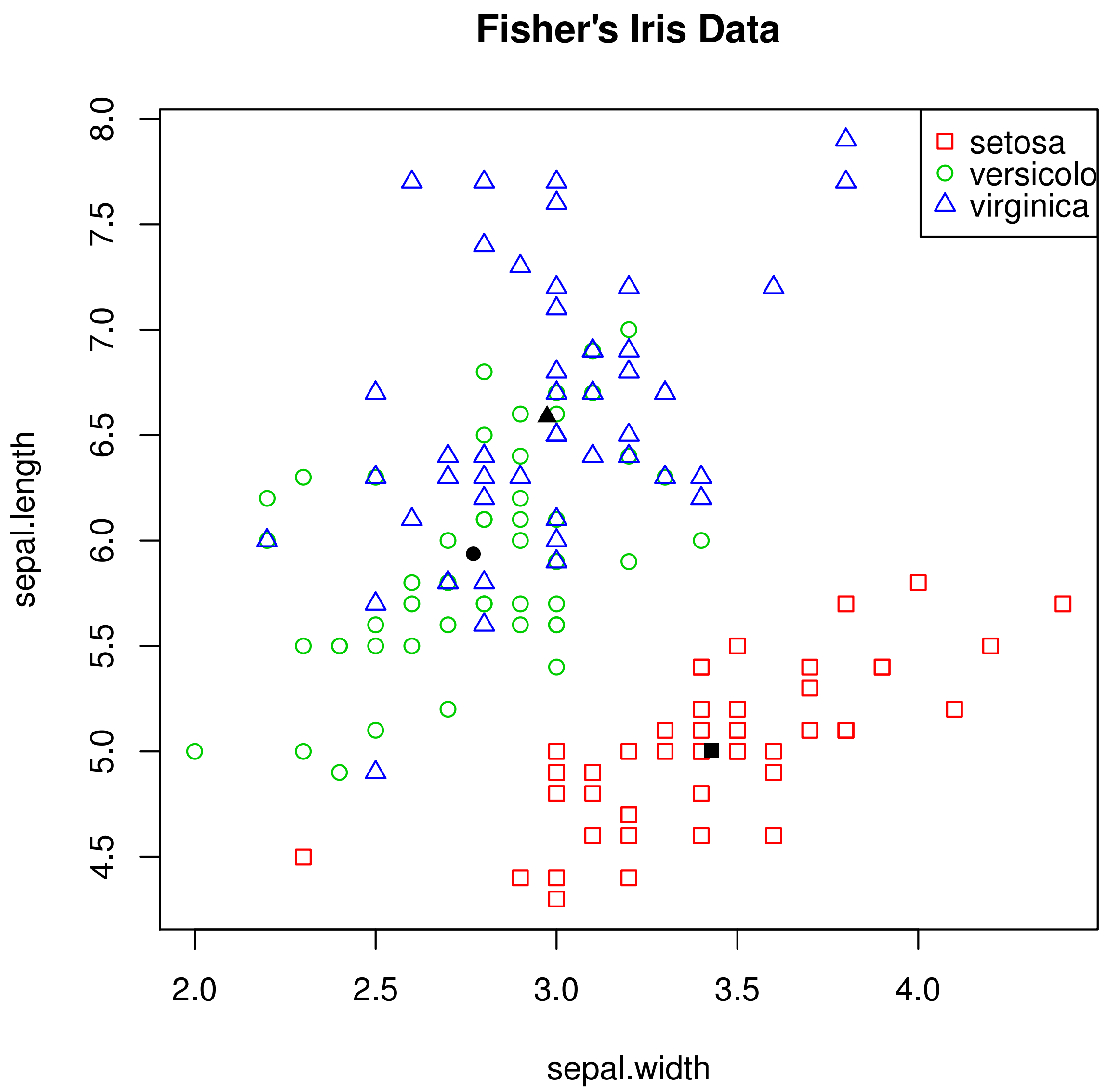
Add points - to show the data points with the mean sepal.width and sepal.length in each species,

```
> require(plyr)
> mymeans <- ddply(.data=x,
+                  .variables=c("species"),
+                  .fun=summarise,
+                  sepal.width=mean(sepal.width),
+                  sepal.length=mean(sepal.length)
+                  )
> mymeans
     species sepal.width sepal.length
1     setosa       3.428        5.006
2 versicolor       2.770        5.936
3  virginica       2.974        6.588
> mypch2 <- 15:17
> points(x=mymeans$sepal.width,y=mymeans$sepal.length,
+        col="black",pch=mypch2)
```

Note: `ddply` is an advanced "apply" function that apply over a data.frame in a block-wise manner; it applied over a set of sub-data.frames that has been splitted by `.variables`.

Visualization - Making a scatter plot

Visualize the added points,



Visualization - Subplots

Put and arange subplots within a single figure by `par`,
The syntax,

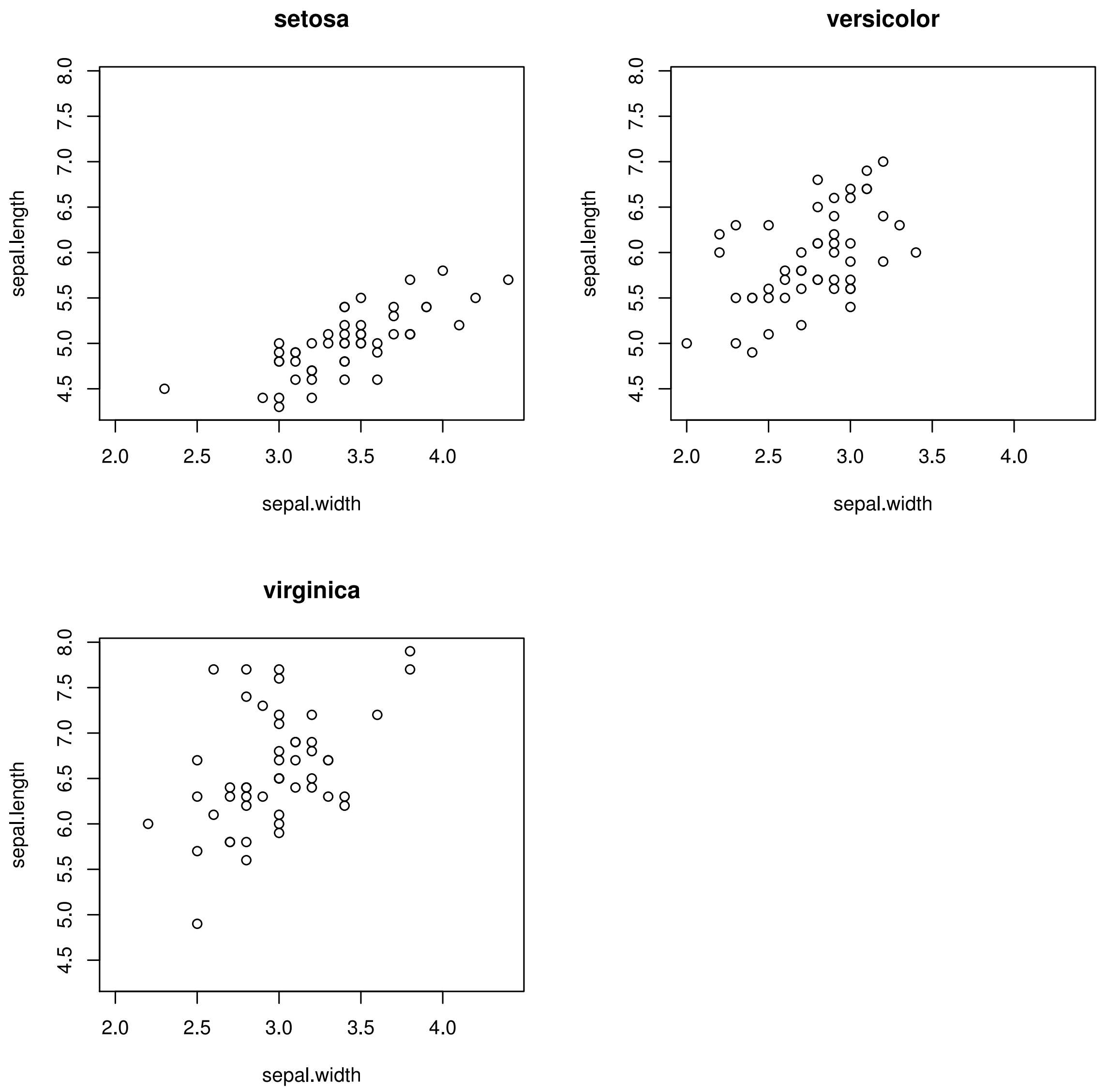
```
par(mfrow=c(nr,nc))          #fill in by row
par(mfcol=c(nr,nc))          #fill in by column
```

Example,

```
> par(mfrow=c(2,2))
> for (e in levels(species)){
+   plot(sepal.width[species==e],sepal.length[species==e],main=e,
+        xlab="sepal.width",ylab="sepal.length",
+        xlim=range(sepal.width),ylim=range(sepal.length))
+ }
```

Visualization - Subplots

Visualize the subplots,



Visualization - QQ plot

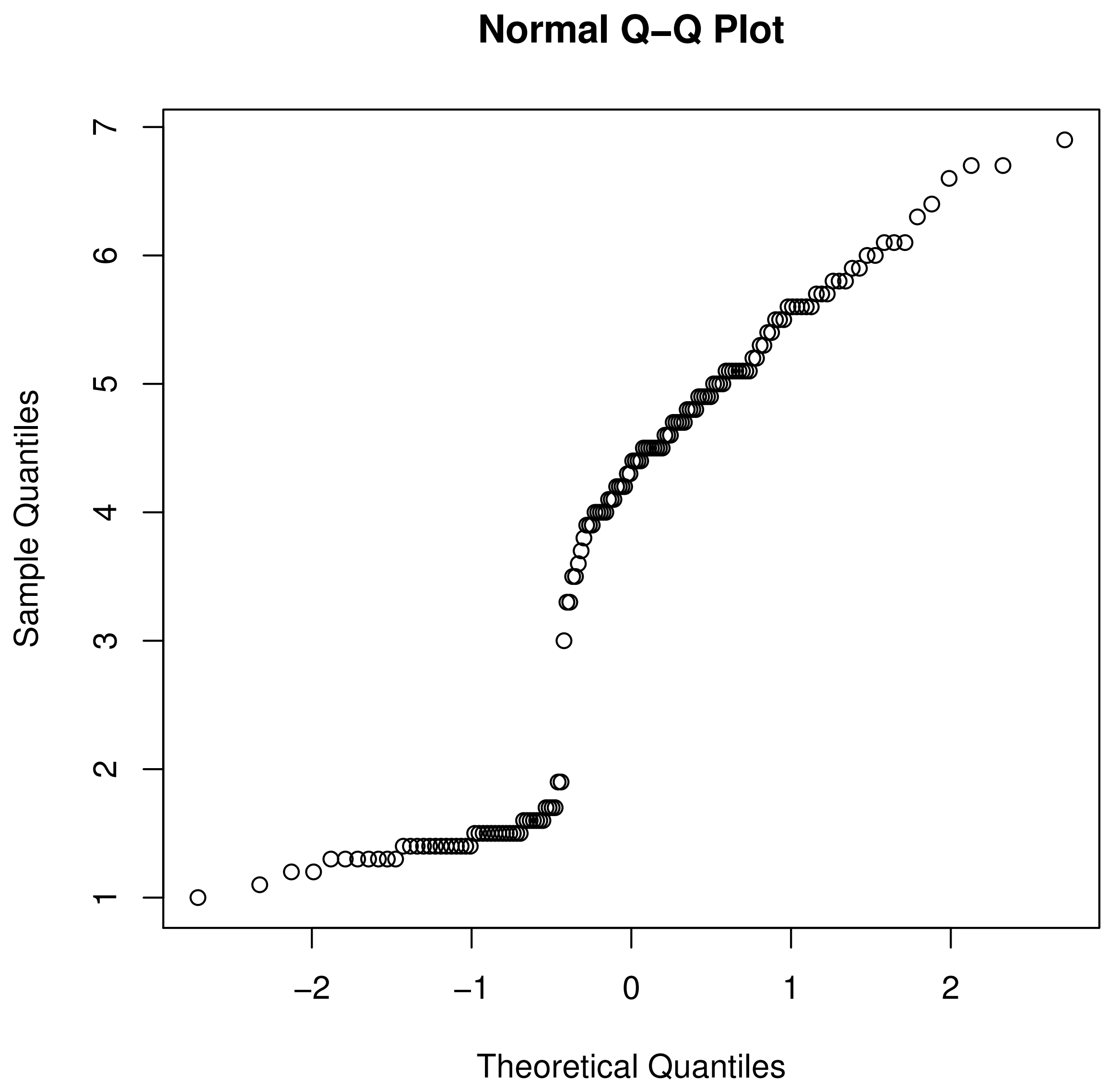
Does petal length have a normal distribution? Let’s see ...

```
> dev.new()
> qqnorm(petal.length)
```

Note: `dev.new` created a new window for plotting.

Visualization - QQ plot

Visualize the QQ-plot,



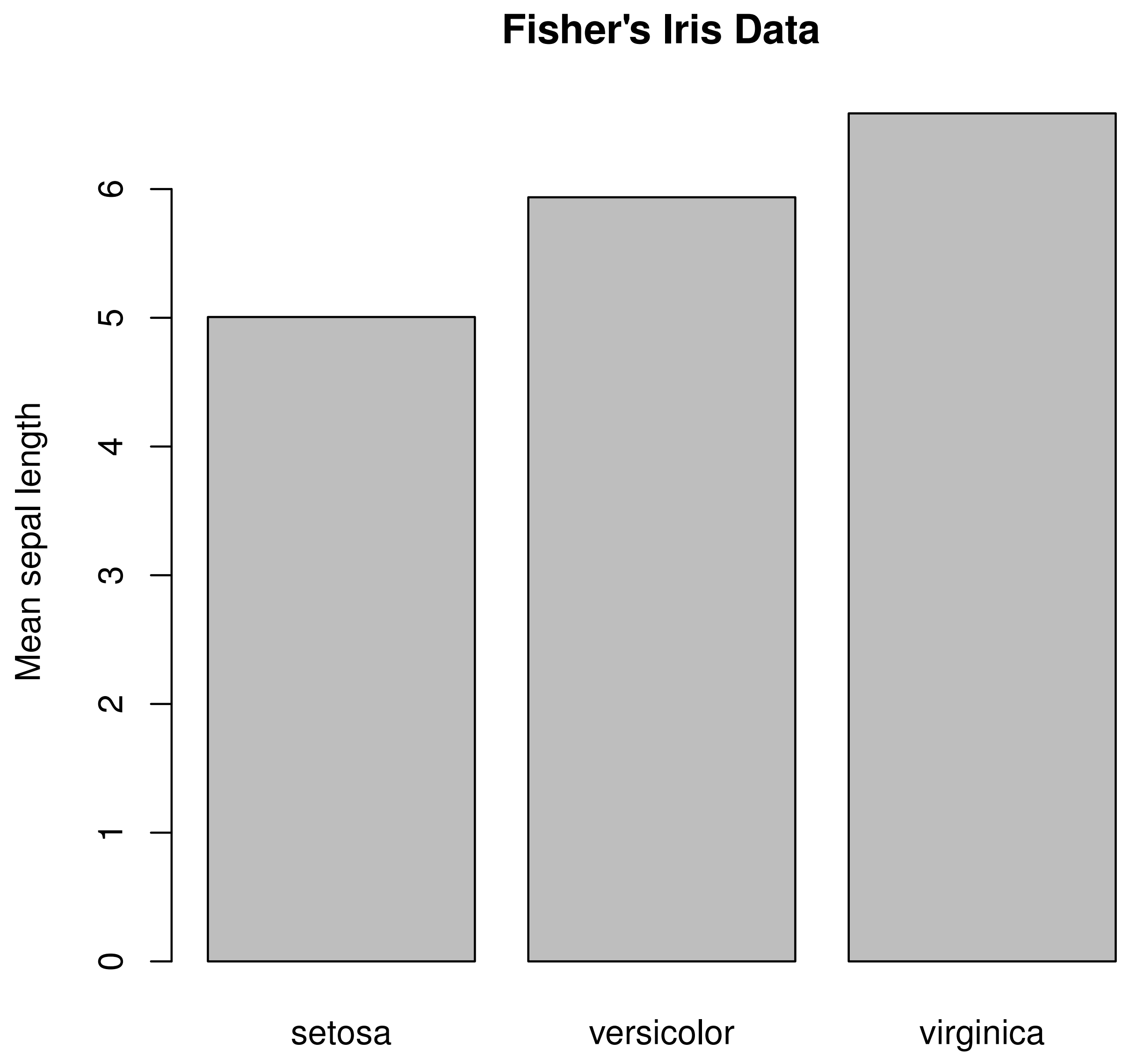
Visualization - Barplot

A `barplot` of mean sepal length by species,

```
> barplot(mymeans$sepal.length,names=mymeans$species,
+         ylab="Mean sepal length", main="Fisher's Iris Data")
```

Visualization - Barplot

Visualize the barplot,



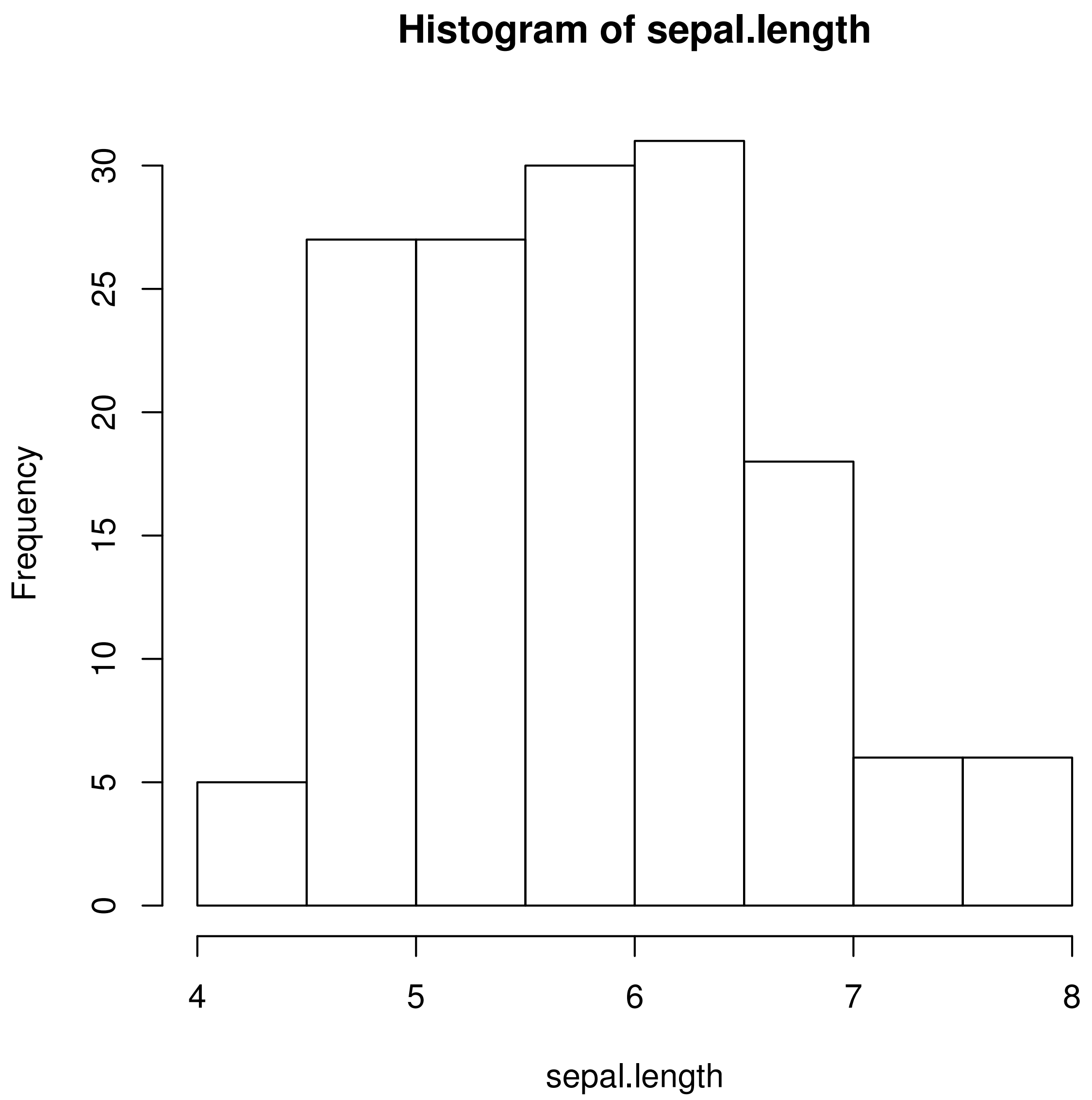
Visualization - Histogram

Make a histogram,

```
hist(sepal.length)
```

Visualization - Histogram

Visualize the histogram,



Visualization - Histogram (Cont'd)

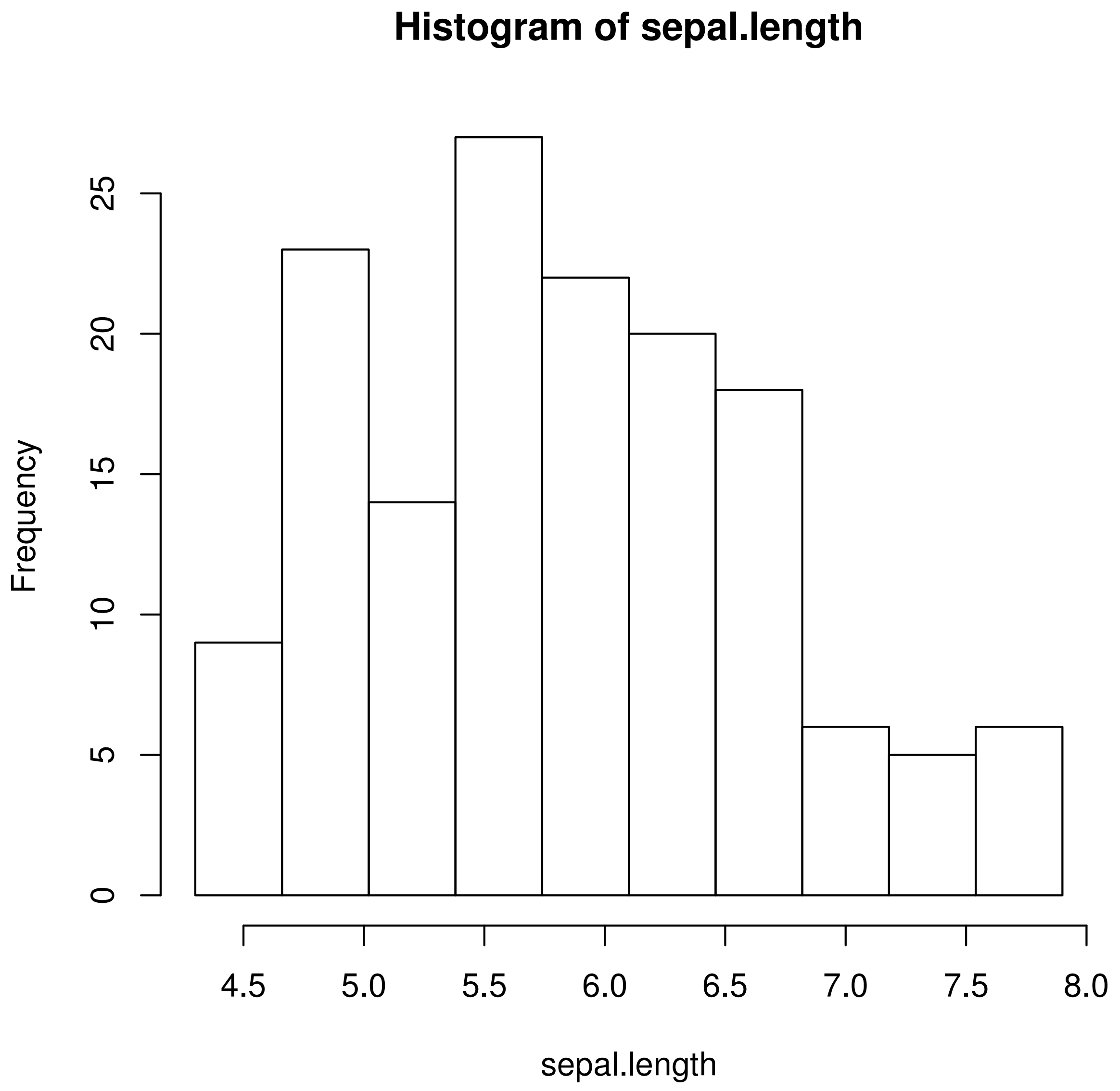
Make a histogram with specified bins,

```
> binCount <- 10
> mybreaks <- seq(min(sepal.length),max(sepal.length),length.out=binCount+1)
> hist(sepal.length,breaks=mybreaks)
```

Note: The `breaks` - *i.e.* the bin boundaries - is one longer than the number of the 'bins'.

Visualization - Histogram (Cont'd)

Visualize the histogram with 10 bins,



Visualization - Histogram (Cont'd)

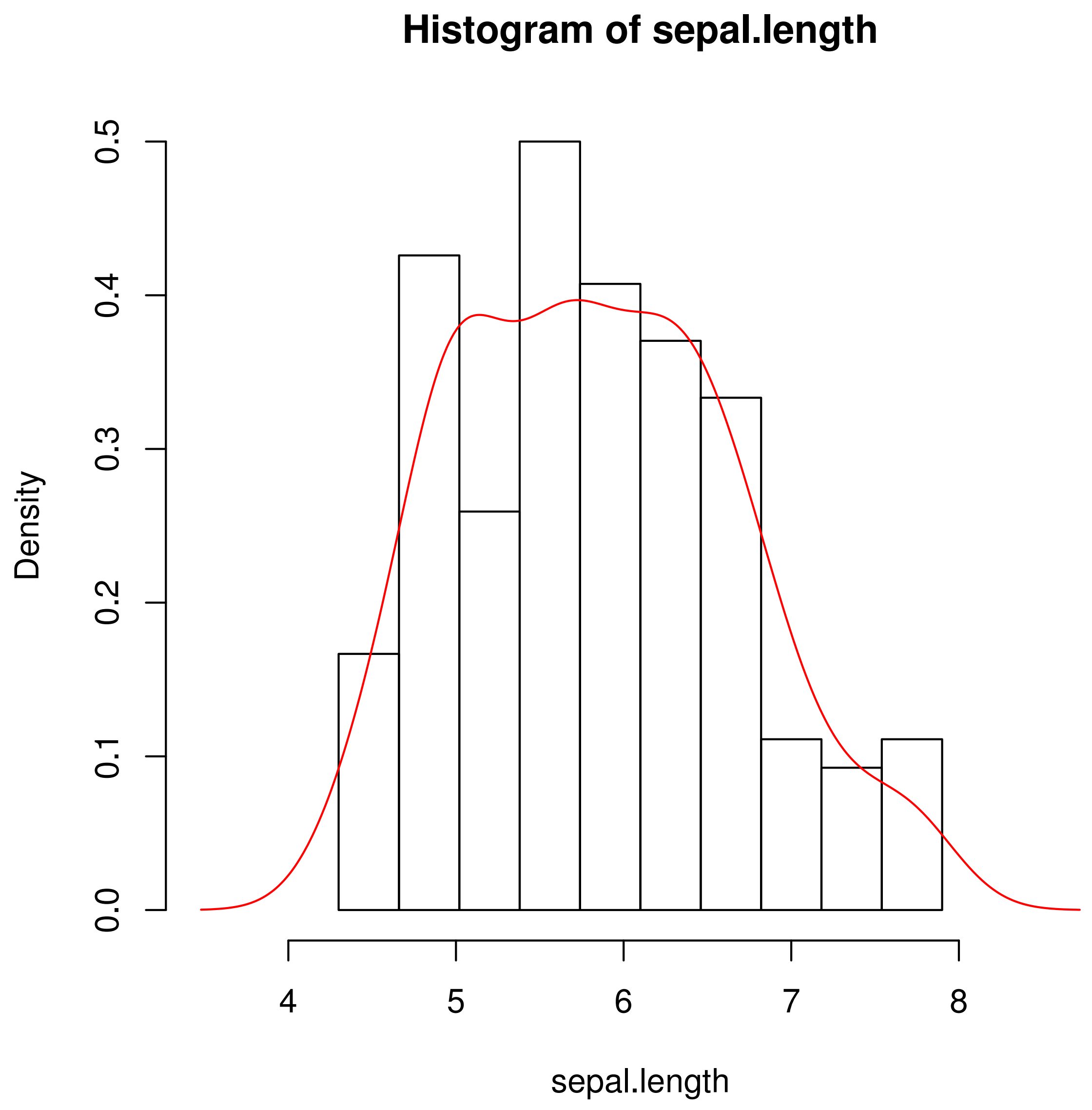
Make a histogram and add a density line,

```
> d <- density(sepal.length)        # determine densities
> myhist <- hist(sepal.length,breaks=mybreaks)
> hist(sepal.length,breaks=mybreaks,
+      xlim=range(d$x),ylim=range(myhist$density,range(d$y)),
+      probability=TRUE)
> lines(d, col="red")
```

Note: Here we used `probability` instead of `frequency`; and we specified the range according to both the histogram and the density line!

Visualization - Histogram (Cont'd)

Visualize the histogram with density line,



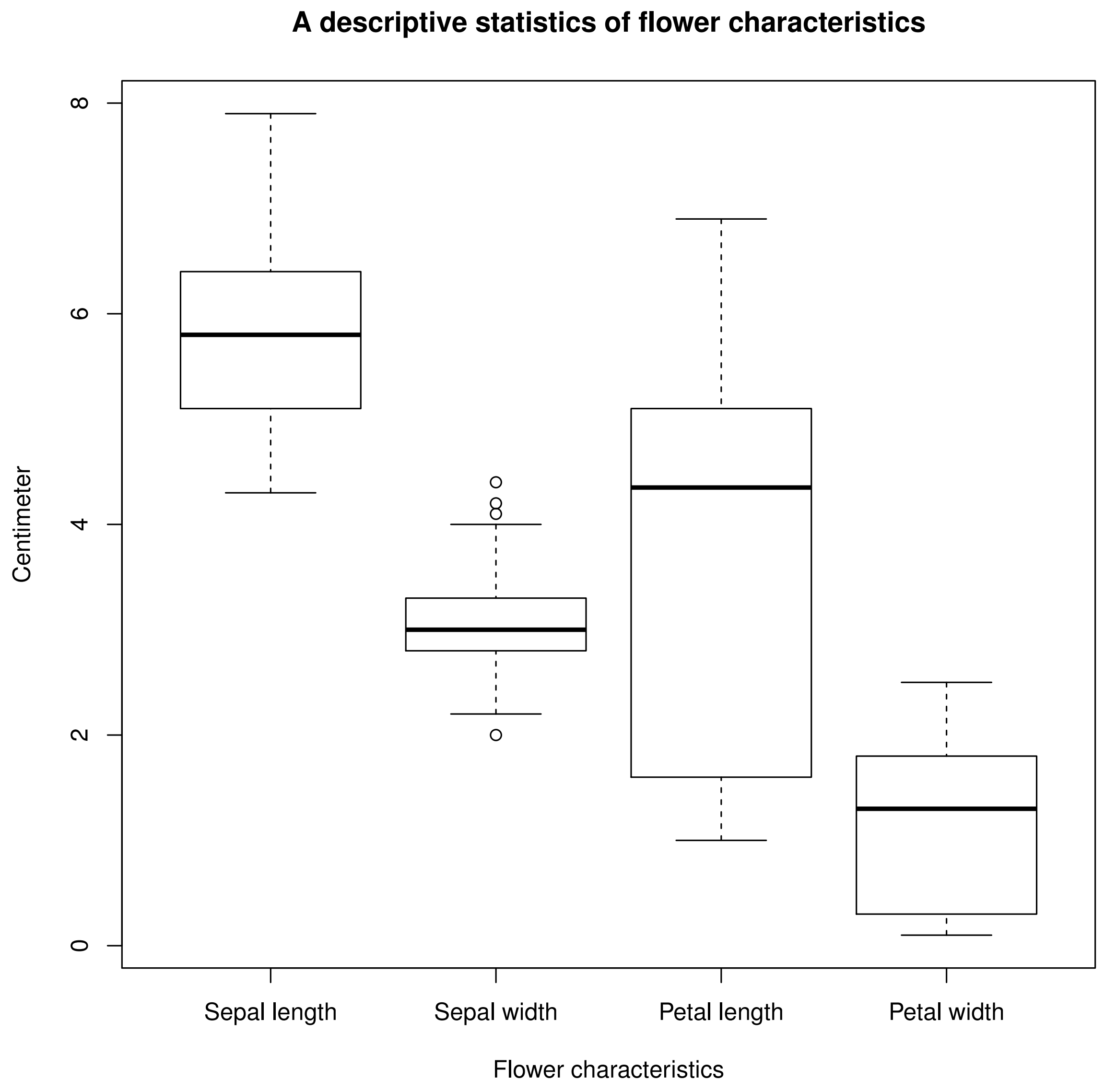
Visualization - Boxplot

A descriptive statistics of flower characteristics by `boxplot`,

```
> boxplot(sepal.length,sepal.width,petal.length,petal.width,
+         xlab="Flower characteristics",ylab="Centimeter",
+         names = c("Sepal length","Sepal width","Petal length","Petal width"),
+         main="A descriptive statistics of flower characteristics")
>
> detach(x)
```

Visualization - Boxplot

Visualize the boxplot,



Visualization - lattice

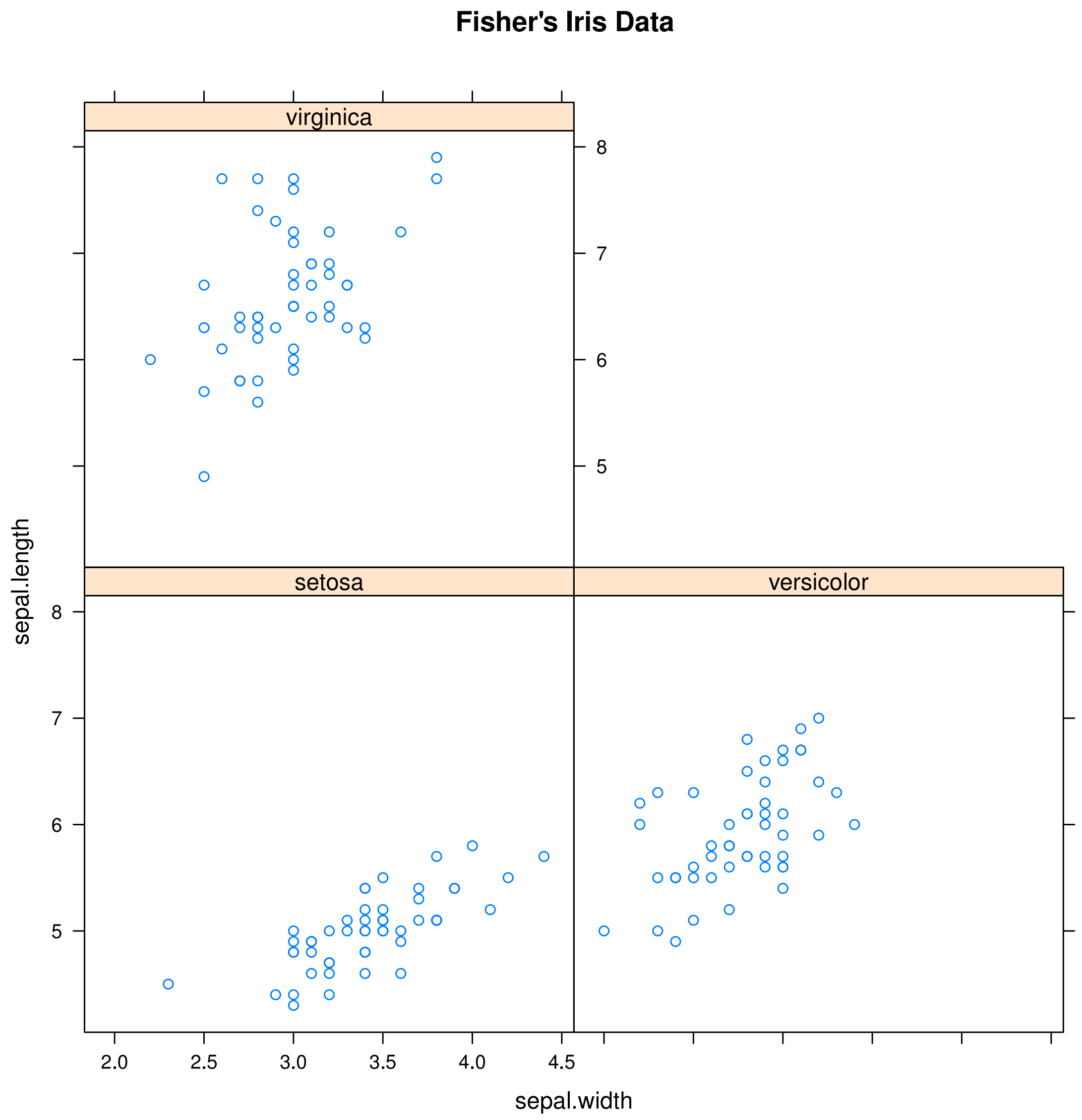
An enhanced version of `plot` in the `lattice` package is `xyplot`,

```
> library(lattice)
> xyplot(sepal.length~sepal.width|species,main="Fisher's Iris Data")
```

Note: `lattice` provides a simple ways to produce multi-panel (sub)plots. Here we used a `formula` object that is generally of the form $y \sim x|g1 * g2 * ...$, where x and y are the primary variables (like `sepal.width` and `sepal.length` in the above example), and $g1, g2, ...$ are the conditioning variables (like `species`). It produces plots of y (on the y-axis) versus x (on the x-axis) conditional on the variables $g1, g2, ...$.

Visualization - lattice

Visualize the plot by `lattice`,



Visualization - ggplot2

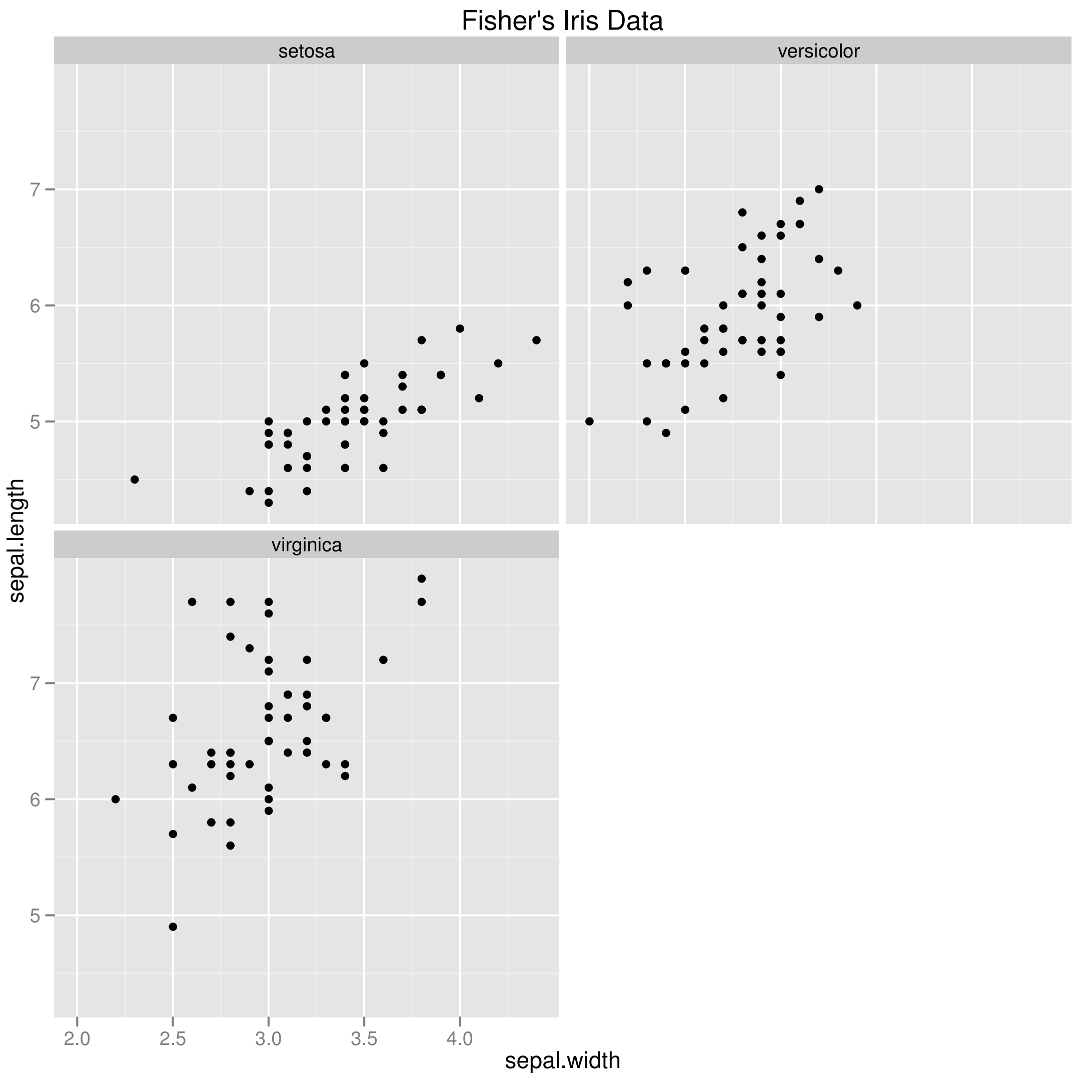
An enhanced version of `plot` in the `ggplot2` package is `ggplot`,

```
library(ggplot2)
ggplot(data=x,aes(x=sepal.width,y=sepal.length)) +
  geom_point() +
  facet_wrap(~species,ncol=2) +
  opts(title="Fisher's Iris Data")
```

Note: `ggplot2` is designed to work in a *layered fashion*, starting with a layer showing the raw data then adding layers of annotation and statistical summaries.
Here the primary variables (`sepal.width` and `sepal.length`) are specified by `aes` and the the conditioning variable `species` is added by `facet_wrap`.

Visualization - ggplot2

Visualize the plot by `ggplot2`,



Visualization - Saving plots

Save a plot when the plotting window is open,

```
dev.copy2pdf(file="plotname.pdf",width=8,height=8)
```

Note: "width" and "height" are in inches.

Save a plot by specifying a saving device in advance,

```
pdf(file="plotname.pdf",width=8,height=8)
plot-command(s)
dev.off()
```

Note: `dev.off`() shuts down the device to make sure the plot is saved properly.

Part III (Cont'd)
Case studies - Statistics

- Correlation by `cor`
- Association and correlation test by `cor.test`
- Model fitting by `lm`
- Student's t-test by `t.test`
- Wilcoxon/Mann-Whitney rank test by `wilcox.test`
- One-way test for equal means by `oneway.test`
- Kruskal-Wallis rank sum test by `kruskal.test`
- Pearson's chi-squared test for count data by `chisq.test`
- Fisher's exact test by `fisher.test`
- Summary of useful functions and tests for statistics in R

Statistics - Correlation

Let's calculate the Pearson's correlation coefficient (r; $-1 \leq r \leq +1$) between sepal.length and sepal.width by `cor`,

```
> cor(x$petal.length,x$petal.width)
[1] 0.9628654
> cor(x$sepal.length,x$sepal.width)
[1] -0.1175698
```

Statistics - Association and correlation test

Now let's test whether Pearson's r is significantly different than zero by `cor.test`,

```
> cor.test(x$petal.length,x$petal.width)

        Pearson's product-moment correlation

data:  x$petal.length and x$petal.width
t = 43.3872, df = 148, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.9490525 0.9729853
sample estimates:
      cor
0.9628654
```

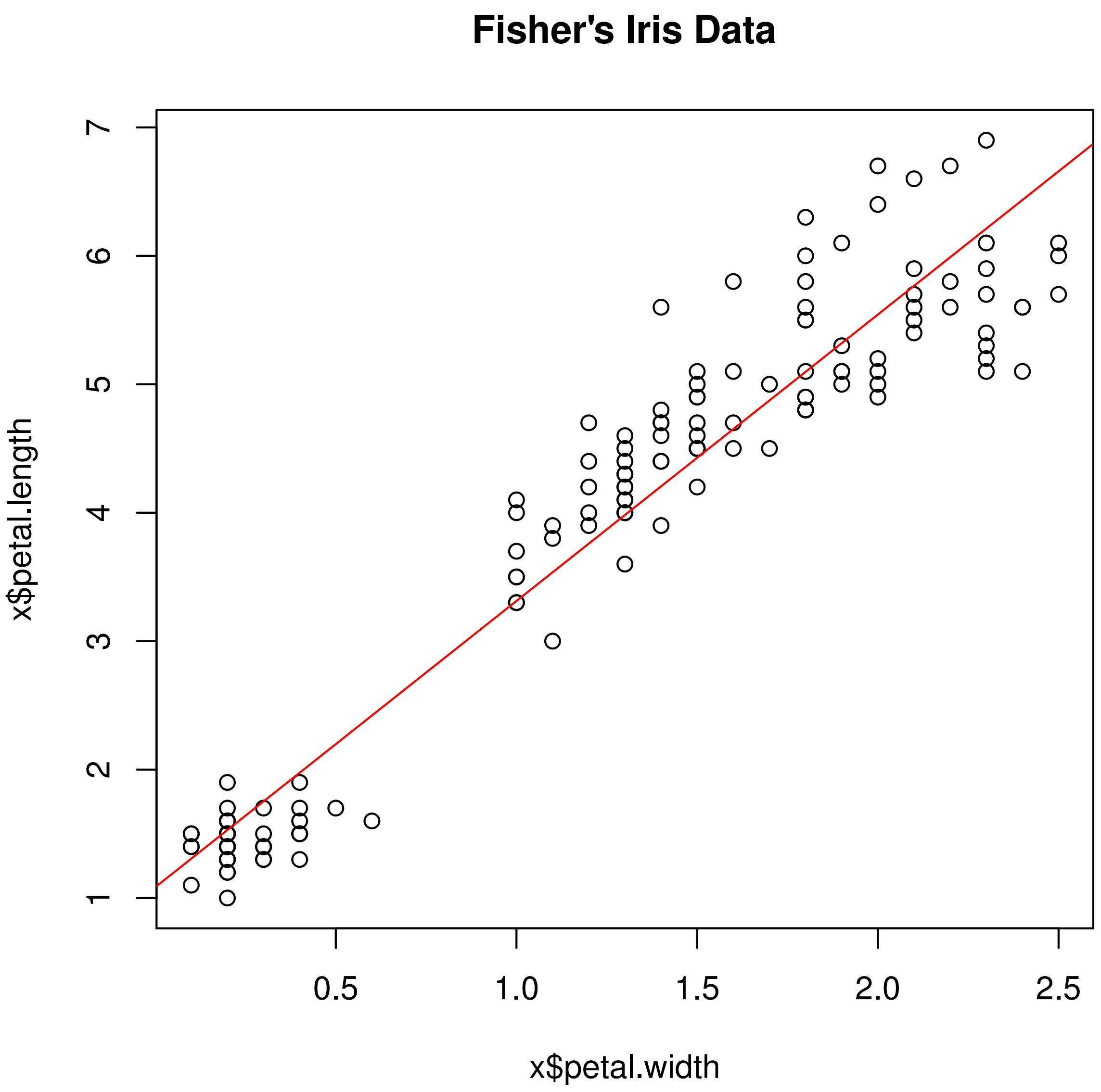
Statistics - Model fitting

Fit a model between sepal.length and sepal.width,

```
> mylm <- lm(petal.length~petal.width,data=x)
> mylm

Call:
lm(formula = petal.length ~ petal.width, data = x)

Coefficients:
(Intercept)  petal.width
      1.084        2.230

> plot(x$petal.width,x$petal.length,main="Fisher's Iris Data")
> abline(mylm,col="red")
```

Statistics - Model fitting

Visualize the fitting,



Statistics - Student’s t-test

Compare mean petal length between species:

```
> t.test(petal.length~species, data=x)
Error in t.test.formula(petal.length ~ species, data = x) :
  grouping factor must have exactly 2 levels
```

So, subset ...

```
> t.test(petal.length~species, data=x[x$species%in%c("virginica","versicolor"),])

        Welch Two Sample t-test

data:  petal.length by species
t = -12.6038, df = 95.57, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -1.49549 -1.08851
sample estimates:
mean in group versicolor  mean in group virginica
                   4.260                    5.552
```

Statistics - Wilcoxon/Mann-Whitney rank test

Using only ranks and therefore a good choice when data is far from normally distributed.

```
> wilcox.test(petal.length~species,
+             data=x[x$species%in%c("virginica","versicolor"),])

        Wilcoxon rank sum test with continuity correction

data:  petal.length by species
W = 44.5, p-value < 2.2e-16
alternative hypothesis: true location shift is not equal to 0
```

Statistics - One-way test for equal means

Useful for data with more than two levels.

```
> oneway.test(petal.length~species,data=x)

        One-way analysis of means (not assuming equal variances)

data:  petal.length and species
F = 1828.092, num df = 2.000, denom df = 78.073, p-value < 2.2e-16
```

Statistics - Kruskal-Wallis rank sum test

Useful for data with more than two levels and a good choice when data is far from normally distributed

```
> kruskal.test(petal.length~species,data=x)

        Kruskal-Wallis rank sum test

data:  petal.length by species
Kruskal-Wallis chi-squared = 130.411, df = 2, p-value < 2.2e-16
```

- Statistics (Cont'd)

The next few tests require tabled count data, for which we will use the `HairEyeColor` data from the `datasets` package.

```
> require(datasets)
> HairEyeColor
, , Sex = Male

       Eye
Hair    Brown Blue Hazel Green
  Black    32   11    10     3
  Brown    53   50    25    15
  Red      10   10     7     7
  Blond     3   30     5     8

, , Sex = Female

       Eye
Hair    Brown Blue Hazel Green
  Black    36    9     5     2
  Brown    66   34    29    14
  Red      16    7     7     7
  Blond     4   64     5     8
```

Statistics - Pearson’s chi-squared test

Get the data for "Male",

```
> maleHairEyeColor <- HairEyeColor[,,"Male"]
> maleHairEyeColor
       Eye
Hair    Brown Blue Hazel Green
  Black    32   11    10     3
  Brown    53   50    25    15
  Red      10   10     7     7
  Blond     3   30     5     8
```

Test whether there is an association between hair color (brown and blond only) and eye color (brown and blue only),

```
> maleHairEyeBrBlBu <- maleHairEyeColor[c("Brown","Blond"),c("Brown","Blue")]
> maleHairEyeBrBlBu
       Eye
Hair    Brown Blue
  Brown    53   50
  Blond     3   30
>
> chisq.test(maleHairEyeBrBlBu)

        Pearson's Chi-squared test with Yates' continuity correction

data:  maleHairEyeBrBlBu
X-squared = 16.8119, df = 1, p-value = 4.127e-05
```

Statistics - Fisher’s exact test

Test whether there is an association between hair color and eye color,

```
> maleHairEyeBrBlBu
       Eye
Hair    Brown Blue
  Brown    53   50
  Blond     3   30
>
fisher.test(maleHairEyeBrBlBu)

        Fisher's Exact Test for Count Data

data:  maleHairEyeBrBlBu
p-value = 1.068e-05
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
  2.964561 56.807444
sample estimates:
odds ratio
  10.44231
```

Summary of statistical tests

Tests on location and correlation,

	Location test		Correlation test cor.test()
	Two data sets	More than two data sets	
Normal distribution (Parametric test)	Paired-sample t-test t.test()	One-way ANOVA oneway.test()	Pearson’s Correlation method="pearson"
Non-normal distribution (Non-parametric test)	Wilcoxon test wilcox.test()	Kruskal-Wallis test kruskal.test()	Spearman or Kendall method="spearman" method="kendall"

Tests on odds ratios in 2×2 contingency table,

	Small data sets	Large data sets
Exact test	Fisher’s exact test fisher.test()	
Approximation test		Pearson’s chi-square test chisq.test()

Summary of statistical functions

Useful functions for statistics,

`cor`	`correlation(s)`
`cov`	`covariance(s)`
`ecdf`	`empirical cumulative distribution, an inverse of "quantile"`
`glm`	`Fitting Generalized Linear Models`
`lm`	`Fitting Linear Models`
`mean`	`arithmetic mean`
`quantile`	`sample quantiles`
`sd`	`standard deviation`
`summary`	`result summaries`
`var`	`variance`

Summary of statistical functions (Cont’d)

Useful tests for statistics,

`anova`	Analysis of variance (or deviance) tables
`binom.test`	Exact Binomial Test
`chisq.test`	Pearson's Chi-squared Test for Count Data
`cor.test`	Test for Association/Correlation Between Paired Samples
`fisher.test`	Fisher's Exact Test for Count Data
`kruskal.test`	Kruskal-Wallis Rank Sum Test
`oneway.test`	Test for Equal Means in a One-Way Layout
`poisson.test`	Exact Poisson tests
`t.test`	Student's t-Test
`var.test`	F Test to Compare Two Variances
`wilcox.test`	Wilcoxon Rank Sum and Signed Rank Tests

Appendix

Appendix A.
Exercise on R basics and solutions

A1. Exercise on R basics

```
1. Create the following vcector: 7 13 19 11 17 5

2. Write a for loop to print the elements that are larger
than 10 in the vector

3. Write a function that reads a vector (remember to send it
along as an argument to the function) and returns a vector of
all elements that are larger than their left neighbour (for the
above example: 13, 19, 17).

Think of at least two ways to do this and test your function(s)
on the vector above.
```

A2. Solutions to the exercise on R basics

1. Create the following vcector: 7 13 19 11 17 5

```
> v <- c(7, 13, 19, 11, 17, 5)
> v
[1]  7 13 19 11 17  5
```

2. Write a for loop to print the elements that are larger than 10 in the vector

```
> for (element in v){
+   if (element > 10){
+     print(element)
+   }
+ }
[1] 13
[1] 19
[1] 11
[1] 17
```

A2. Solutions to the exercise on R basics (Cont'd)

```
3. Write a function that reads a vector (remember to send it
along as an argument to the function) and returns a vector of
all elements that are larger than their left neighbour
```

Way 1 - by loop and logic controls,

```
> myfun1 <- function(x){
+   res <- c()
+   for (i in 2:length(x)){
+     if (x[i]>x[i-1]){
+       res <- c(res,x[i])
+     }
+   }
+   return(res)
+ }
> myfun1(v)
[1] 13 19 17
```

A2. Solutions to the exercise on R basics (Cont'd)

```
3. Write a function that reads a vector (remember to send it
along as an argument to the function) and returns a vector of
all elements that are larger than their left neighbour
```

Way 2 - by vector indexing,

```
> myfun2 <- function(x){
+   the.element <- x[2:length(x)]
+   the.neighbour <- x[1:(length(x)-1)]
+   res <- the.element[the.element>the.neighbour]
+   return(res)
+ }
> myfun2(v)
[1] 13 19 17
```

Take home message: In R, vector operations in general are quite efficient!!

Appendix B. Advanced topics in R

These topics are beyond the course's scope and will not be covered during the lecture but are useful.

Random Number Generation

This is useful to generate simulated data, upon which you would like to test your R code.

To specify seeds,

```
set.seed(seed, kind = NULL, normal.kind = NULL)
```

Random samples and permutations,

```
sample(x, size, replace = FALSE, prob = NULL)
```

Generate random deviates from a normal distribution,

```
rnorm(n, mean = 0, sd = 1)
```

Note: Similarly, we have `rbeta`, `rbinom`, `rchisq`, `rexp`, `rgamma`, `rgeom`, `rhyper`, `rpois`, `runif`, etc.

Environments & namespaces

List names of the objects in the specified environment.

```
ls(name, pos = -1, envir = as.environment(pos),
   all.names = FALSE, pattern)
```

Assign a value to a name in an environment.

```
assign(x, value, pos = -1, envir = as.environment(pos),
       inherits = FALSE, immediate = TRUE)
```

Get an R object with a given name in an environment.

```
get(x, pos = -1, envir = as.environment(pos), mode = "any",
    inherits = TRUE)
```

OOP in R

❍ A very brief introduction to OOP in R (Chap 2.1.3 in [3])

❍ Classes, methods and generic functions (Chap 9 and 10 in [4])

Writing R Extensions

A manual by R Development Core Team ([5])

- ❍ Create your own packages
- ❍ Write R help files
- ❍ Interfaces to foreign languages (e.g. C and Fortran)

Literature

[1] William N. Venables, David M. Smith, and R Development Core Team. *An Introduction to R*. Network Theory Limited, January 2009.

[2] Peter Dalgaard. *Introductory Statistics with R*. Springer, August 2008.

[3] Robert Gentleman. *R Programming for Bioinformatics*. Chapman and Hall/CRC, July 2008.

[4] John Chambers. *Software for Data Analysis: Programming with R*. Springer, July 2008.

[5] R Development Core Team. *Writing R Extensions*. Vienna, Austria, 2012.

Index

I think before I print ...