Introduction to R

STAT2005

Chapter 1

What is statistical programming?

- Computer programming involves controlling computers, telling them what calculations to do, what to display, etc.
- Statistical programming involves doing computations to aid in statistical analysis.
  - Data must be summarized and displayed.
  - Models must be fit to data, and the results displayed.
- We aim to provide a foundation for an understanding of how those applications work: what are the calculations they do, and how could you do them yourself

A brief history of R and S-PLUS

- Due to the large demand of software for statistical analysis, numerous statistical packages are developed in the market. Among them, three most popular commercial products are
  - —SAS: Statistical Analysis System: This package is very popular in banks and business sectors.
  - —SPSS: Statistical Package for Social Science: This package is widely used by social scientist and researchers.
  - —MINITAB: This package is very popular in the university community. It is very user-friendly and is perfect for teaching statistics.

- R is based on the computer language S, developed by John Chambers and others at Bell Laboratories in 1976.
- S is a computational language developed for data analysis and graphics
- Due to its success, S-PLUS, an extension of S, with extensive programming ability was developed. However, S-PLUS is a commercial package just as SAS, SPSS and MINITAB
- They made it open source in 1995, and thousands of people around the world have contributed to its development.
- Being open source implies R is freely available, and its users could see how it is written, and to improve it.

The R package

- It is important to note that R is a computer language as well as a statistical package.
- R is an interpreted language, in which individual language expressions are read and then executed immediately as soon as the command is entered.
- C and C++, by contrast, are complied languages, in which complete programs are translated by a compiler into the appropriate executable programs.

Why use a command line?

- Menu-based interfaces are very convenient when applied to a limited set of commands.
- However, a command line interface is open ended.
- If you want to program a computer to do something that no one has done before, you can easily do it by breaking down the task into the parts that make it up, and then building up a program to carry it out.
- This may be possible in some menu-driven interfaces, but it is much easier in a command-driven interface.

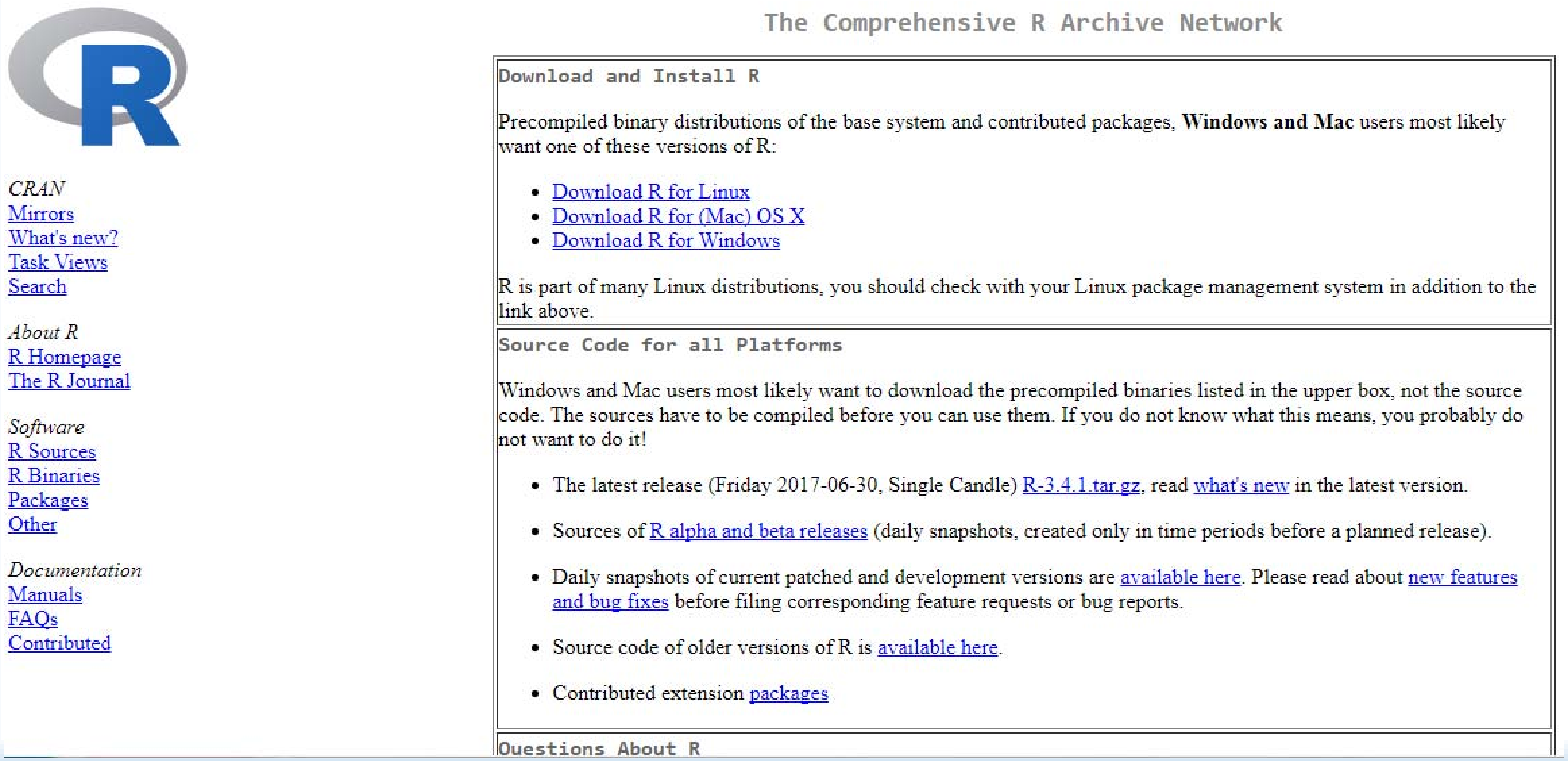
Getting help

- The R package comes with a complete on-line help manual in html format.
- It is also useful for beginner to learn the R command by typing `help()` within the R package.
- There are many good tutorial notes on R language. The following are three good references.
    - An introduction to R (from the Help manuals of the R package)
    - https://cran.r-project.org/manuals.html
    - https://stackoverflow.com/documentation/r/topics

Installation of R

Download from the official website:

https://cran.r-project.org/



The Comprehensive R Archive Network

CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

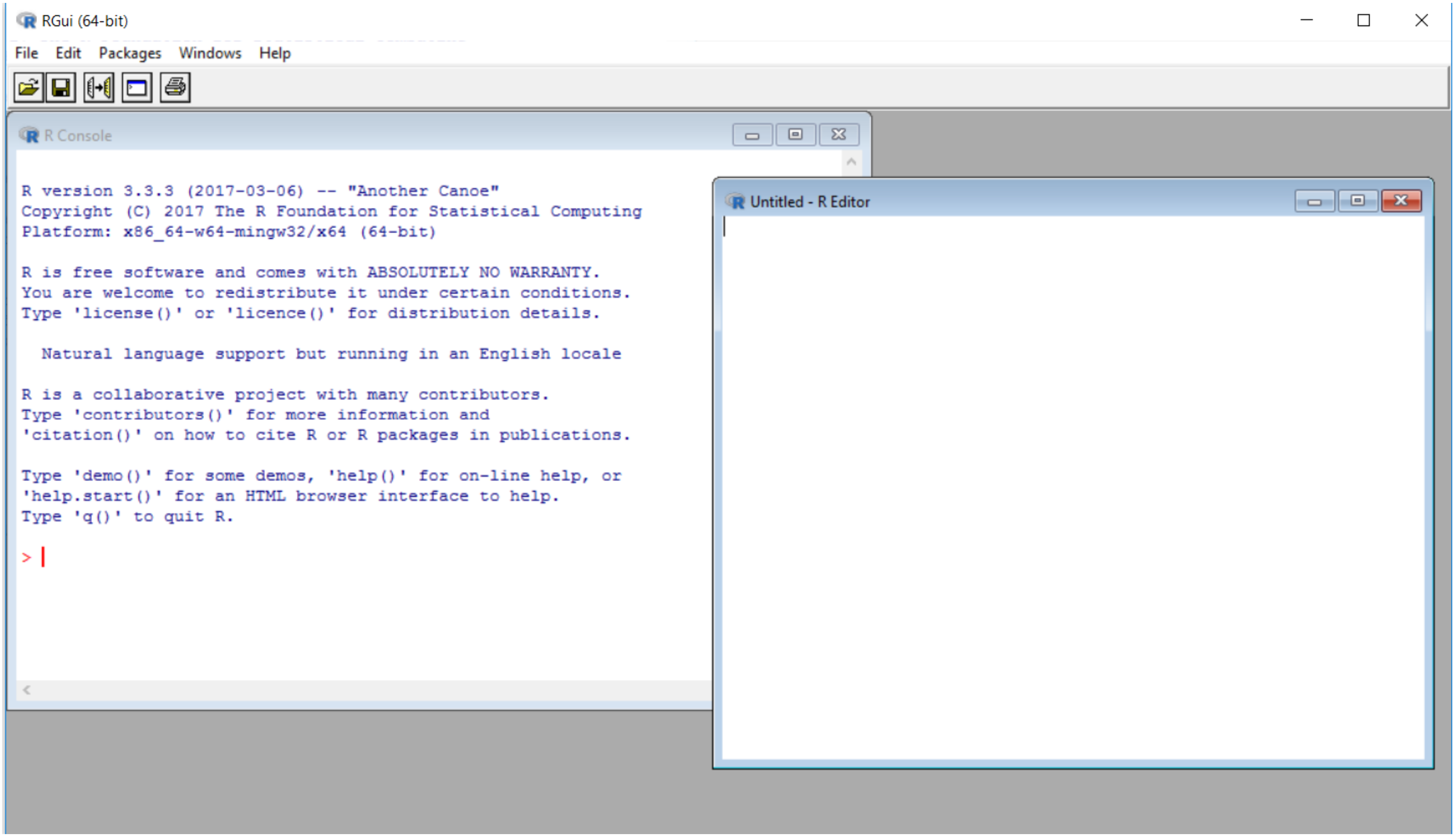
Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (Friday 2017-06-30, Single Candle) R-3.4.1.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

R running on Windows

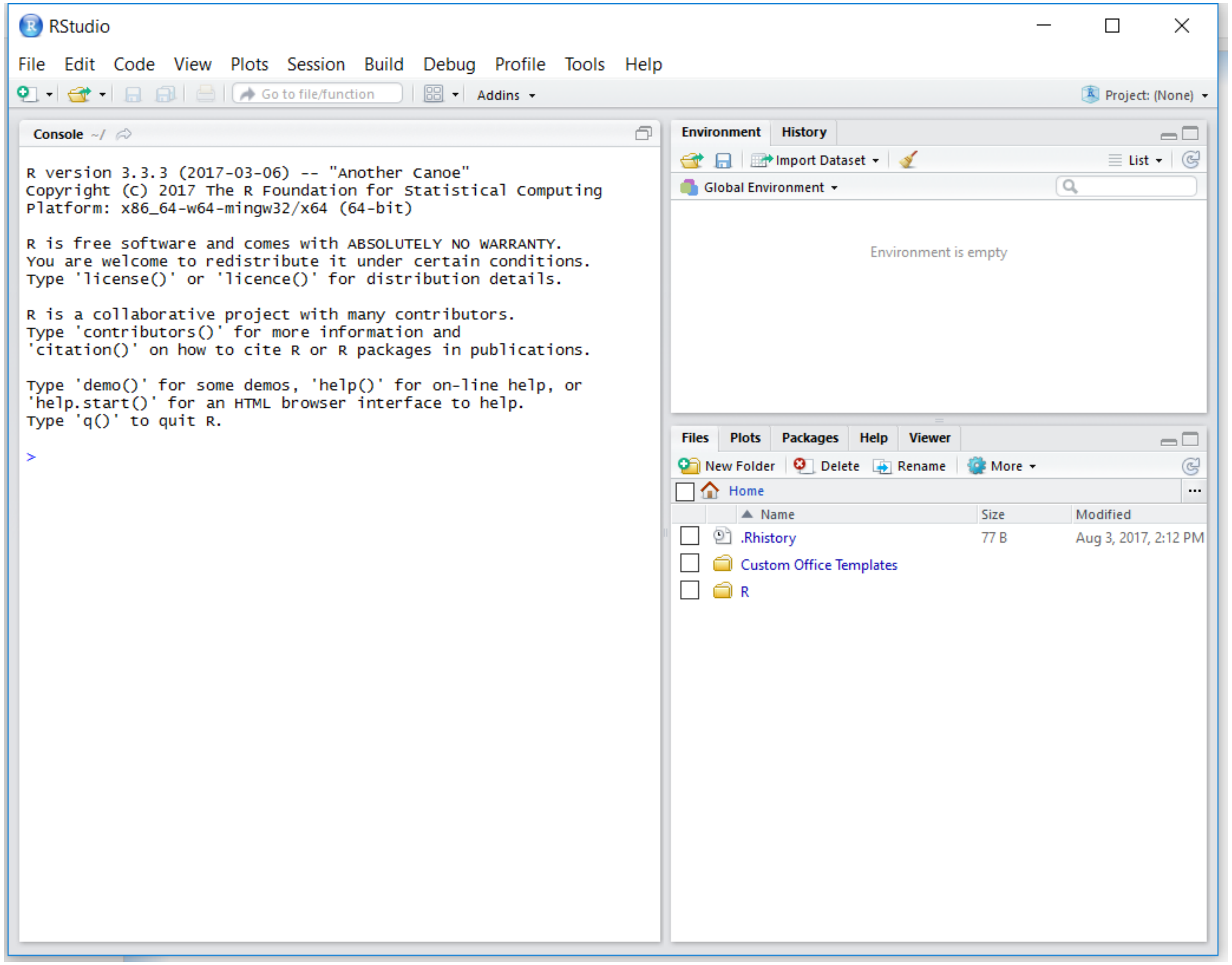


```
R version 3.3.3 (2017-03-06) -- "Another Canoe"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

RStudio

- RStudio provides an integrated development environment (IDE).
- IDEs were first developed in the 1970s to help programmers: they allow you to edit your program, to search for help, and to run it.
- When your first attempt doesn’t work, they offer support for diagnosing and fixing errors.
- RStudio is an IDE for R programming, first released in 2011. It is produced by a Boston company named RStudio, and is available for free use.

You can download the "Open Source Edition" of "RStudio Desktop" from www.rstudio.com/ .



```
R version 3.3.3 (2017-03-06) -- "Another Canoe"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Getting started

R can be used as a calculator

```
> 3+7
# can be used as a simple calculator
[1] 10
> 3*21
[1] 63
> 1/0
[1] Inf
# Inf is reserve word for infinity in R
> 0/0
[1] NaN
# NaN is reserve word for Not a Number in R
```

Generate Sequence

```
> 1:7       # generate sequence from 1 to 7
[1] 1  2  3  4  5  6  7
```

Function for combining

```
> c(3,4,1,6)
# combining numbers into a vector
[1] 3 4 1 6
```

Assignment

```
> newvec <- c(3,4,1,6)
# assign the vector to newvec
> newvec
[1] 3 4 1 6
```

- Note that the hashtag (#) is used to denote comments. R ignores all text from the # character to the end of the line.
- R is case sensitive. This means `newvec` is different from `Newvec`.
- R has excellent and powerful graphical procedures. Input `demo(graphics)` for a demo.
- Whenever quitting R, you will be asked if you want to save the workspace image.
- If yes, all the variables or objects created will be saved and will be restored next time when R is invoked. `ls()`will display all the objects exist in this R session.
- We can remove all the objects by using `rm(list=ls())`

The assignment operator

In R, both of the following statements have the effect of assigning the value 3 to the variable x.

```
x=3
x<-3
```

when the R language (and S before it) was first created, `<-` was the only choice of assignment operator.

However many modern languages (such as C, for example) use `=` for assignment, so beginners using R often found the arrow notation cumbersome, and were prone to use `=` by mistake.

But R uses `=` for yet another purpose: associating function arguments with values. E.g., `f(x=3)`.

To make things easier for new users familiar with languages like C, R added the capability in 2001 to also allow `=` be used as an assignment operator, on the basis that the intent is usually clear by context.

So, `x=3` means "assign 3 to x", whereas `f(x=3)` means "call function f, setting the argument x to 3".

Some R traditionalists prefer `<-` for clarity

Technically, the difference between the two operators is that **<-** is a global assignment operator, while **=** is a local assignment operator. For example,

```
> sin(x=1)
[1] 0.841471
> x
Error: object 'x' not found
> sin(x<-1)
[1] 0.841471
> x
[1] 1
```

Rules for R variable names

- A variable name must start with a letter and can be a combination of letters, digits, period (`.`) and underscore (_). If it starts with period (`.`), it cannot be followed by a digit.
- A variable name cannot start with a number or underscore (_)
- Variable names are case-sensitive
- Reserved words cannot be used as variable name (`TRUE`, `FALSE`, `NULL`, `if`…)

The help function

- A useful feature to learn R programming is the help documentation.
- It provides an easy and convenient way to understand the built-in functions. It comes with the syntax, explanation, reference, related functions and examples on the usage of that function.
- For example, try the help on these functions, `help()` `help(sort)`, `help(matrix)`, `help(rep)`, `help(seq)`, `help(cumsum)`.
- Later on, when you encounter syntax errors and need helps, on-line help always come in handy.

Saving R commands and outputs

- We can save the R commands and outputs by pressing `Save History ...` and `Save to File ...` in the File menu respectively.
- For example, we can save the R commands and outputs to a file named `new.r`. We can read and edit these files using any text editor.
- Some recommended text editors for programmers are
  - Notepad++ (Windows only), see https://notepad-plus-plus.org/
  - Sublime Text (Windows/macOS/Linux), see https://www.sublimetext.com/

Data objects

Data are stored as objects in R. This is an abstract term that can be assigned to a variable.

There are many types of objects, e.g.

- Vector
- Matrix
- List
- Data frame

We will discuss these objects one by one.

Vector

In R, a vector is represented as an one-dimensional array, which is a systematic arrangement of similar objects. A scalar is considered as a vector with single element. Vector is the simplest data type.

```
> x<-5:1                      # numeric
> x
[1] 5 4 3 2 1
> mode(x)                     # display the mode of x
[1] numeric
> length(x)                   # display the length of x
[1] 5
```

```
> x<2                      # logical
[1] FALSE FALSE FALSE FALSE  TRUE
> y<-x<2
# we can save the result to y
> y
[1] FALSE FALSE FALSE FALSE  TRUE
mode(y)
[1] logical

> z<-c("a","b","c")    # character
> mode(z)
[1] character
> length(z)
[1] 3
```

We can combine vectors of different data types.

```
> c(x,y)
# combining numeric with logical vectors
becomes numeric
 [1] 5 4 3 2 1 0 0 0 0 1


> c(x,y,z)
# combining numeric, logical and character
vectors
 [1] "5"     "4"     "3"     "2"     "1"
"FALSE"
 [7] "FALSE" "FALSE" "FALSE" "TRUE"  "a"
"b"
[13] "c"
```

Simple patterned vectors

We have seen the use of the `:` operator for producing simple sequences of integers.

Patterned vectors can also be produced using the `seq()` function as well as the `rep()` function.

For example, the sequence of odd numbers less than or equal to 21 can be obtained using

```
> seq(1, 21, by = 2)
 [1]  1  3  5  7  9 11 13 15 17 19 21
```

Repeated patterns are obtained using `rep()`. Consider the following examples.

```
> rep(3, 12)       # repeat the value 3, 12 times
 [1] 3 3 3 3 3 3 3 3 3 3 3 3
> rep(seq(2, 20, by = 2), 2)
# repeat the pattern 2 4 ... 20, twice
 [1]  2  4  6  8 10 12 14 16 18 20  2  4  6  8 10
12 14 16 18 20
> rep(c(1, 4), c(3, 2)) # repeat 1, 3 times and 4,
twice
[1] 1 1 1 4 4
> rep(c(1, 4), each = 3) # repeat each value 3
times
[1] 1 1 1 4 4 4
> rep(1:10, rep(2, 10)) # repeat each value twice
 [1]  1  1  2  2  3  3  4  4  5  5  6  6  7  7  8
8  9  9 10 10
```

Extracting elements from vectors

Square brackets **[]** are used to index and extract vector elements. For example

```
> x<-1:5
> x[3]
[1] 3
> x[3:5]
[1] 3 4 5
> x[c(1,3,5)]
[1] 1 3 5
> x[x>3]
[1] 4 5
```

Exercise

Using `seq()` and `rep()` as needed, create the vectors.

(a) 0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4

(b) 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5

Matrix

- Matrix is a collection of numbers in a rectangular form.
- A matrix with dimension n by m means that the matrix has n rows and m columns.
- Matrix is very important in statistics and mathematics, we will discuss it in details later.

```
> m<-matrix(1:12,nrow=3,ncol=4)
# 3 by 4 matrix filled in columnwise
> m
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> dim(m)                # dimension of m
[1] 3 4
> nrow(m)               # number of row in m
[1] 3
> ncol(m)               # number of column in m
[1] 4
> matrix(1:12,nrow=3,byrow=T)  # by row
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

```
> m[2,]                       # select 2nd row
[1]  2  5  8 11
> m[,3]                       # select 3rd column
[1] 7 8 9
> m[2,3]                      # select an element
[1] 8
> m[1:2,2:4]                  # select submatrix
     [,1] [,2] [,3] # equivalent to m[c(1,2),c(2,3,4)]
[1,]    4    7   10
[2,]    5    8   11

> m[-2,]                      # exclude the second row
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    3    6    9   12
```

Exercise

1. What would you expect from the output of `m[c(1,2),c(3,2,4)]`?
2. What would you expect from the output of `matrix(1:10, nrow=3)`?
3. What would you expect from the output of

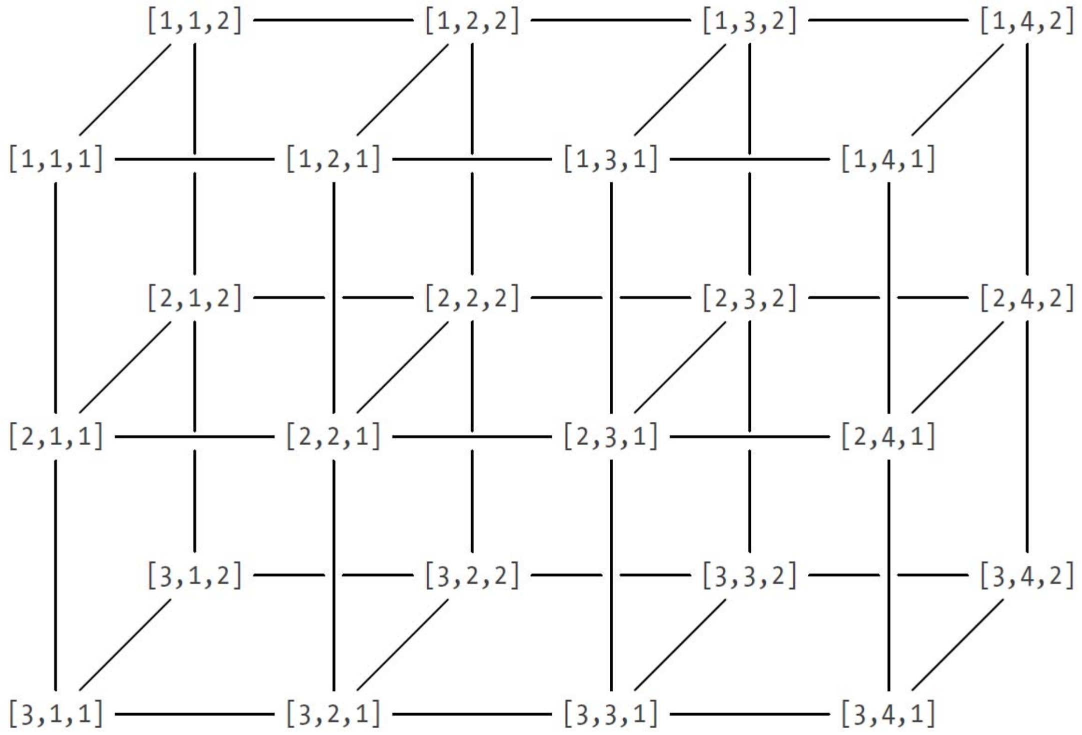
   `m[,-3]`?

We may combine two matrices column-wise (`cbind`) or row-wise (`rbind`). Let us illustrate this by the following commands:

```
> m1<-matrix(1:8,nrow=2)# create a 2x4 matrix m1
> m2<-matrix(1:6,nrow=3)# create a 3x2 matrix m2
> rbind(m,m1)           # combine m and m1 row-wise
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
[4,]    1    3    5    7
[5,]    2    4    6    8
> cbind(m,m2)       # combine m and m2 column-wise
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    4    7   10    1    4
[2,]    2    5    8   11    2    5
[3,]    3    6    9   12    3    6
```

Multidimensional Arrays

Just as a matrix is the result of increasing the dimension of a vector, the dimension of a matrix can also be increased to get more complex data structures.



```
> X <- array(data=1:24,dim=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
```

List

List is the most general object in R and is more powerful than vectors and matrices since it can hold multiple types of objects

```
> x<-1:3                       # numeric vector
> y<-c("a","b")                # character vector
> z<-matrix(1:6,nrow=2)        # numeric matrix
> w<-list(x,y,z)               # define a list
> w                            # display the list w
[[1]]:                         # 1st component of w
[1] 1 2 3

[[2]]:                         # 2nd component of w
[1] "a" "b"

[[3]]:                         # 3rd component of w
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

- Note that list has two subscripts. The first subscript denotes the component of the list, and the second subscript denotes the elements within that component.
- Selecting elements from a list requires two separate subscripts. The first subscript tells which component of the list to select, and the second tells which elements of that component to select.

```
> w[[1]]          # first component of w
[1] 1 2 3
> w[[2]][2]
# second elements of the second component of w
[1] "b"
> w[[3]][1:2,2]
# element [1:2,2] of the third component of w
[1] 3 4
```

List can be assigned a names attribute as follow.

```
> names(w)<-c("x","y","z")     # assign name to w
> w
$x:
[1] 1 2 3
$y:
[1] "a" "b"
$z:
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

To display the first component of `w`, type

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

- The function `unlist()` will change a list object to a long vector.

```
> u<-unlist(w)# change w into a character vector u
> u
 [1] “1" "2" "3" "a" "b" "1" "2" "3" "4" "5" "6“
```

- The function `as.numeric()` will change the vector to numeric.

```
> as.numeric(u)
 [1]  1  2  3 NA NA  1  2  3  4  5  6
# change w back to numeric, NA = missing value
```

Data frame

- Data frames are actually a special kind of list, or structure.
- The most important distinction is that in a data frame the members must all be vectors of equal length.
- The data frame is one of the most important and frequently used tools in R for statistical data analysis.
- An example is `women` (comes with R) which contains the average weights (in pounds) of American women aged 30 to 39 of particular heights (in inches)
- A summary of the data frame can be obtained with the command `summary(women)`.

Extracting data frame elements and subsets

- Columns in a data frame can also be addressed using their names with the `$` operator. For example, the `weight` column can be extracted as `women$weight`
- Thus, we can extract all heights for which the weights exceed 140 using

```
women$height[women$weight > 140]
```

- The `with()` function allows us to access columns of a data frame directly without using the `$` operator.
- For example, we can divide the weights by the heights in the women data frame using

```
with(women, weight/height)
```

Exercise

Try the following operations with a data frame:

(a) `women[[1]]`

(b) `women[1]`

(c) `women["weight"]`

(d) `women["wei"]`

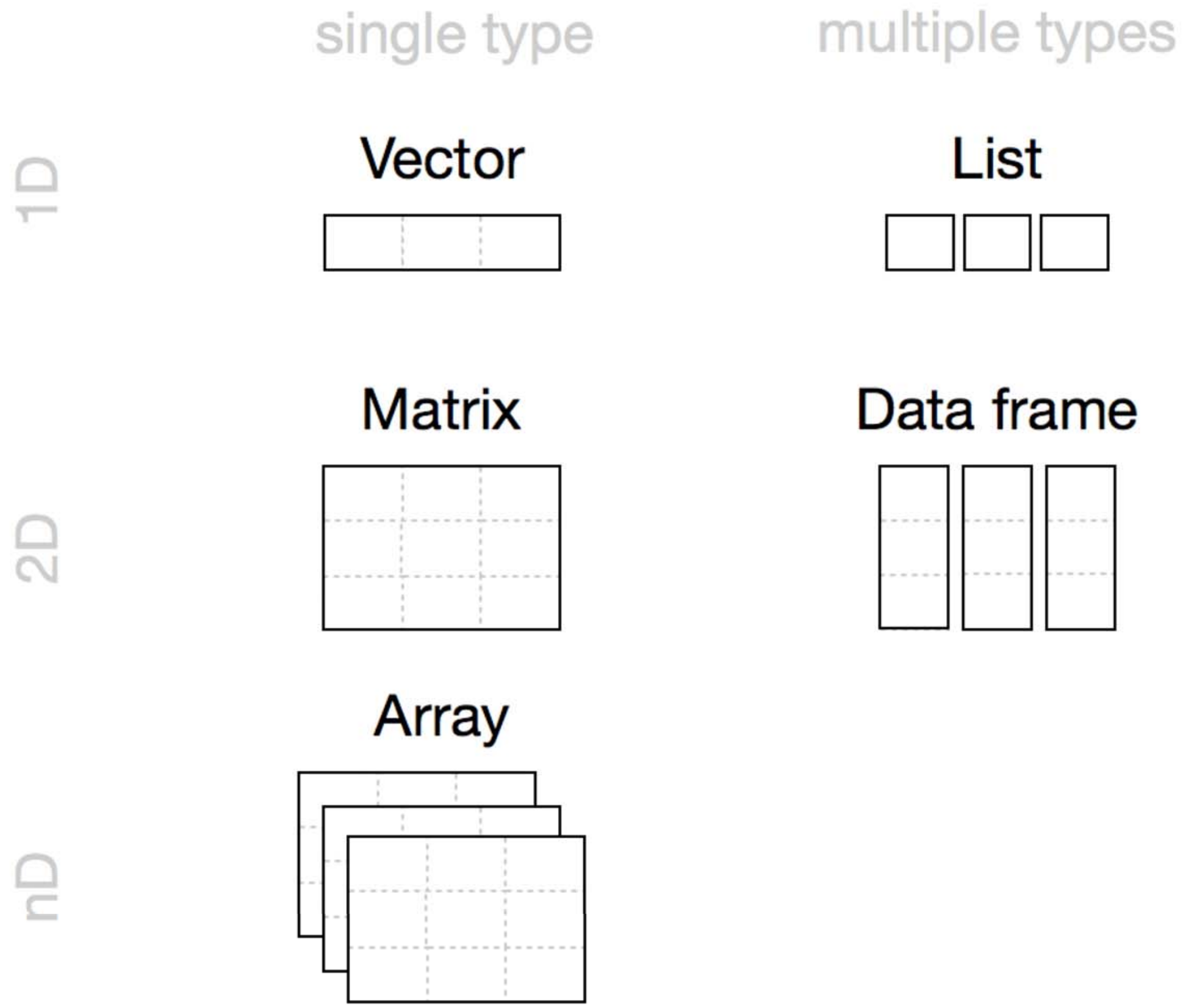
(e) `women$wei`

Creating data frame

Data frames can be created using the `data.frame()` function.

```
member <- data.frame(
    name = c("Tom", "May"),
    age = c(22, 20)
)
member
```

```
  name age
1  Tom  22
2  May  20
```


single type
multiple types
1D
2D
nD
Vector
List
Matrix
Data frame
Array

Factor

Factors offer an alternative way to store character vector. For example, a factor with four elements and two levels `control` and `treatment` can be created using

```
> grp <- c("control", "treatment", "control",
"treatment")
> grp
[1] "control" "treatment" "control" "treatment"
> grp <- factor(grp)
> grp
[1] control treatment control treatment
Levels: control treatment
```

Factors can be an efficient way of storing character data when there are repeated elements.

This is because the levels of a factor are internally coded as integers. To see what the codes are for our factors, we can type

```
> as.integer(grp)
[1] 1 2 1 2
```

The labels for the levels are stored just once each, rather than being repeated. The codes are indices of the vector of levels:

```
> levels(grp)
[1] "control" "treatment"
> levels(grp)[as.integer(grp)]
[1] "control" "treatment" "control"
"treatment"
```

Relabelling a factor

We can redefine the labels of a factor object as follows

```
> levels(grp) <- c("1","2")
> grp
[1] 1 2 1 2
Levels: 1 2
```

Note that that the levels of a factor are stored as character strings, even if the original data vector was numeric. So, logical expressions should all be expressed in character form.

```
> grp == "1"
[1]  TRUE FALSE  TRUE FALSE
```

mode() vs. class()

mode() and class() are two functions to describe a data object in R.

- mode() describe the **data type** used for storage, e.g., numeric, logical, character, etc.
- class() describe the **object class** of the input variable, e.g., numeric, integer, list, matrix, factor, etc.

```
> mode(1L);class(1L)
[1] "numeric"
[1] "integer"
> mode(factor(c("a","b")));class(factor(c("a","b")))
[1] "numeric"
[1] "factor"
> mode(matrix(0,2,2));class(matrix(0,2,2))
[1] "numeric"
[1] "matrix"
```

Approximate storage of numbers

- One important distinction in computing is between exact and approximate results.
- It is possible in a computer to represent any rational number exactly, but it is more common to use approximate representations: usually floating point representations.
- It is a binary (base-two) variation on scientific notation. For example, we might write a number to four significant digits in scientific notation as 6.926×10^{-4}.
- The number above would be written as $1.011_2 \times 2^{-11}$ if four binary digit precision was used.

- In base 2, 4/5 is 0.110011001100… .
- Since R stores only 53 bits, some rounding error will occur in the storage.
- We can observe the rounding error with the following experiment.

```
> n <- 1:10
> 1.25 * (n * 0.8) - n
 [1] 0.000000e+00 0.000000e+00 4.440892e-16
0.000000e+00 0.000000e+00
 [6] 8.881784e-16 8.881784e-16 0.000000e+00
0.000000e+00 0.000000e+00
```

- Some operations are particularly prone to rounding error: for example, subtraction of two nearly equal numbers, or (equivalently) addition of two numbers with nearly the same magnitude but opposite signs.

Missing values and other special values

The missing value symbol is **NA**. Missing values often arise in real data problems, but they can also arise because of the way calculations are performed.

```
> some.evens <- NULL
# creates a vector with no elements
> some.evens[seq(2, 20, 2)] <- seq(2, 20, 2)
> some.evens
 [1] NA  2 NA  4 NA  6 NA  8 NA 10 NA 12 NA 14 NA 16 NA 18
NA 20
```

When there exist missing values, the `is.na()` function could be used to detect them. For instance,

```
> is.na(some.evens)
 [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
FALSE  TRUE FALSE
[13]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

The `!` symbol means "not," so we can locate the non-missing values in `some.evens` as follows.

```
> !is.na(some.evens)
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
TRUE FALSE  TRUE FALSE  TRUE
[13] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
TRUE
```

We could then display only the even numbers

```
> some.evens[!is.na(some.evens)]
 [1]  2  4  6  8 10 12 14 16 18 20
```

Expressions and operators

In R, expressions and operators on scalers are easy to follow. However, operations can be performed on each component of a vector or matrix. This is called the vector-based operation.

```
> x<-4:1    # define a numeric vector
> 2*(x-1)   # operate on each element of x
[1] 6 4 2 0
> x^2
[1] 16  9  4  1
```

Priority	Operator	Meaning
1	$	component selection
2	[] [[]]	subscripts, elements
3	^	exponentiation
4	-	unary minus
5	:	sequence operator
6	%% %/% %*%	modulus, integer division, matrix multiplication
7	* /	multiplication, division
8	+ -	addition, subtraction
9	< > <= >= == !=	comparison
10	!	not
11	& | && ||	vectorized *and or*, control *and or*
12	<- -> =	assignments

Example

`2 - -1` should be interpreted as `2 - (-1)` or equivalently `2 + 1`.

```
> 2 - -1
[1] 3
```

`1+5^3*4` should be interpreted as `1+((5^3)*4)`.

```
> 1+5^3*4
[1] 501
```

Exercise

1. In the expression `48:(14*3)`, are the brackets really necessary? What happens when you type `48:14*3`?

2. Using one line of code, how would you obtain the squares of the numbers 48, 47, . . . , 14?

There are many commonly used operations. Here are some examples:

```
> x<-4:1
> x%%3                   # modulus
[1] 1 0 2 1
> x%/%3                  # integer divide
[1] 1 1 0 0
> x==1                   # equality (note the double equal sign)
[1] FALSE FALSE FALSE  TRUE
> x!=2                   # negation
[1]  TRUE  TRUE FALSE  TRUE
> (x>1)&(x<4)            # logical and
[1] FALSE  TRUE  TRUE FALSE
> (x<=1)|(x>3)           # logical or
[1]  TRUE FALSE FALSE  TRUE
```

Note the difference between `!(x>1)&(x<4)` and `!((x>1)&(x<4))`. That is why is it is advisable to use parenthesis whenever possible.

Operator associativity

It is possible to have multiple operators of same priority in an expression. In such case the order of execution is determined through associativity.

Most operators in R have associativity **from left to right**. Some exceptions are exponentiation (^) and leftward assignment (<-, =) and they are **from right to left**.

Examples

`5%%3%/%2` means `(5%%3)%/%2` (left to right)

`2^3^2` means `2^(3^2)` (right to left)

Built-in functions

There are many built-in functions in R. Besides the common mathematical functions like `sqrt`, `abs`, `sin`, `cos`, `log`, `exp`, … etc. there are functions worth mentioning:

Name	Operations
ceiling	smallest integer greater than or equal to element
floor	largest integer less than or equal to element
trunc	ignore the decimal part
round	round up for positive and round down for negative
sort	sort the vector in ascending or descending order
sum, prod	sum and produce of a vector
cumsum, cumprod	cumulative sum and product
min, max	return the smallest and largest values
range	return a vector of length 2 containing the min and max
mean	return the sample mean of a vector
var	return the sample variance of a vector
sd	return the sample standard deviation of a vector
which	return the indices of TRUE elements of a logical object

Example: Using built-in functions

```
> x<-sqrt(1:5)# square root of 1 to 5
> y<-c(-x,x)       # combine –x to x
> y                # display y
 [1] -1.000000 -1.414214 -1.732051 -2.000000
 [5] -2.236068  1.000000  1.414214  1.732051
 [9]  2.000000  2.236068

# we can simply use (y<-c(-x,x)) to create y and
display y using one command
> options(digits=3) # controls the number display
> y                  # display y
 [1] -1.00 -1.41 -1.73 -2.00 -2.24  1.00  1.41  1.73
2.00  2.24
```

```
> (y<-sort(y)) # sort y in ascending order and display
[1] -2.24 -2.00 -1.73 -1.41 -1.00  1.00  1.41  1.73  2.00
2.24
> trunc(y)     # truncated y
 [1] -2 -2 -1 -1 -1  1  1  1  2  2
> round(y)     # round
 [1] -2 -2 -2 -1 -1  1  1  2  2  2
> ceiling(y)        # ceiling
 [1] -2 -2 -1 -1 -1  1  2  2  2  3
> floor(y)     # floor
 [1] -3 -2 -2 -2 -1  1  1  1  2  2
> (z<-1:5)
# create z and display, a useful short-cut
 [1] 1 2 3 4 5
> cumsum(z)         # cumulative sum
[1]  1  3  6 10 15
> cumprod(z)        # cumulative product
[1]   1   2   6  24 120
```

```
mean(sqrt(y))      # NaN means Not a Number
```

The above expression does not work because some elements in y are negative. We can use `which()` to filter elements that are non-negative.

```
which(y>=0)        # return index in y which >=0
y[which(y>=0)]     # select elements in y whcih >=0
mean(sqrt(y[which(y>=0)]))
# compute the mean of square root of elements in y
which >=0
```