1.Introduction to R

R

- R is an implementation of a functional programming language S.
- R has been developed and maintained by a core of statistical programmers, with the support of a large community.
- R is free.
- R is most widely used for statistical computing and graphics.
- R is also a fully programming language well suited to scientific programming in general.

R: pro and cons

- Pro: A picture says more than a thousands words
  - R and visualization are a perfect match. (ggplot2, ggvis, googleVis and rCharts.)
- Pro: R ecosystem
  - R has a rich ecosystem of cutting-edge packages and active community.
- Pro: R lingua franca of data science
  - R is developed by statisticians for statisticians who communicate ideas and concepts through R code and packages
- Pro/Con: R is slow
  - Although R can be experienced as slow due to poorly written code, there are multiple packages to improve R performance: pqR, renjin and FastR, Riposte and many more.
- Con: R has a steep learning curve

http://www.kdnuggets.com/2015/05/r-vs-python-data-science.html

R as a calculating environment

R can be used as a powerful calculator.
Arithmetic operations:

```r
(1+1/100)^100
```

```
## [1] 2.704814
```

```r
17%%5
```

```
## [1] 2
```

```r
17%/%5
```

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

```r
17^5
```

```
## [1] 1419857
```

symbol	meaning
+	addition
-	subraction
*	multiplication
/	division
^	exponential
%%	modulus
%/%	integer division

R has a number of built in functions

sin(x), cos(x), tan(x), exp(x), log(x), sqrt(x), floor(x), ceiling(x), round(x), …

```
exp(1)
```

```
## [1] 2.718282
```

```
options(digits = 16)
exp(1)
```

```
## [1] 2.718281828459045
```

```
pi
```

```
## [1] 3.141592653589793
```

```
sin(pi/6)
```

```
## [1] 0.4999999999999999
```

Variable

- We can assign a value to a variable and use the variable.
- For the assignment, we use command <-.
- Variable names made up of letters, numbers, . or _
    - provided it starts with a letter, or . then a letter.
    - names are case sensitive.
    - for example,
        - x, y, my_variable, a1, a2, .important_variable, x.input
    - wrong name:
        - 2016_income, .1grade, _x, y@gmail.com

Variable

To display the value of a variable `x`, we type `x` or `print(x)` or `show(x)`.

```
x <- 100
x
```

```
## [1] 100
```

We can show the outcome of assignment by parentheses.

```
(y <- (1+1/x)^x)
```

```
## [1] 2.70481
```

When assigning, the right-hand side is evaluated first, then that value is placed in the variable on the left-hand side.

```
n <- 1
n <- n+1
n
```

```
## [1] 2
```

R allows the use of = for variable assignment, in common with most programming languages.

Functions

Takes one or more argument (inputs) and produces one or more outputs (return values).

```
seq(from = 1, to = 9, by =2)
```

```
## [1] 1 3 5 7 9
```

```
seq(from = 1, to = 9)
```

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

You can access the built-in help by

```
help(function_name)
?function_name
```

Every function has a default order for arguments. If you provide arguments in this order, then they do not need to be named.

```
seq(1, 9, 2)
```

```
## [1] 1 3 5 7 9
```

```
seq(to = 9, from = 1)
```

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

```
seq(by=-2, 9, 1)
```

```
## [1] 9 7 5 3 1
```

Vectors

- Vector is an indexed list of variables
- three basic functions for constructing vectors

```
(x <- seq(1, 20, by = 2))
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19
```

```
(y <- rep(3, 4))
```

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

```
(z <- c(y, x))
```

```
##  [1]  3  3  3  3  1  3  5  7  9 11 13 15 17 19
```

- another method for sequence

```
(x <- 100:110)
```

```
##  [1] 100 101 102 103 104 105 106 107 108 109 110
```

```
(y <- 110:100)
```

```
##  [1] 110 109 108 107 106 105 104 103 102 101 100
```

Vectors(2)

- vector and index
- the name of the i-th element of vector `x` is `x[i]`

```
x<- 100:110
i <- c(1, 3, 2)
x[i]
```

```
## [1] 100 102 101
```

minus index:

```
j <- c(-1, -2, -3)
x[j]
```

```
## [1] 103 104 105 106 107 108 109 110
```

empty vetor:

```
x <- c()
```

Vector opreation

elementwise algebraic operation:

```
x <- c(1, 2, 3)
y <- c(4, 5, 6)
x*y
```

```
## [1]  4 10 18
```

```
y^x
```

```
## [1]   4  25 216
```

with unequal length of vectors:

```
c(1, 2, 3, 4) + c(1, 2)
```

```
## [1] 2 4 4 6
```

```
2 * c(1, 2, 3)
```

```
## [1] 2 4 6
```

```
(1:3)^2
```

```
## [1] 1 4 9
```

Vector opreation (2)

This works but with warning message:

```
c(1, 2, 3) + c(1, 2)
```

```
## Warning in c(1, 2, 3) + c(1, 2): 두 객체의 길이가 서로 배수관계에 있지 않습
## 니다
```

```
## [1] 2 4 4
```

functions taking vectors

```
sqrt(1:3)
```

```
## [1] 1.00000 1.41421 1.73205
```

```
mean(1:6)
```

```
## [1] 3.5
```

```
sort(c(5, 1, 3, 4, 2))
```

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

Examples : mean and variance

compare computed mean and variance with built-in functions

```
x <- c(1.2, 0.9, 0.8, 1, 1.2)
x.mean <- sum(x)/length(x)
x.mean - mean(x)
```

```
## [1] 0
```

```
x.var <- sum((x-x.mean)^2)/(length(x)-1)
x.var - var(x)
```

```
## [1] 0
```

Example : simple numerical integration

```
dt <- 0.005
t <- seq(0, pi/6, by = dt)
ft <- cos(t)
(I <- sum(ft) * dt)
```

```
## [1] 0.501549
```

`t` is a vector and `ft` is also a vector.

```
I - sin(pi/6)
```

```
## [1] 0.00154865
```

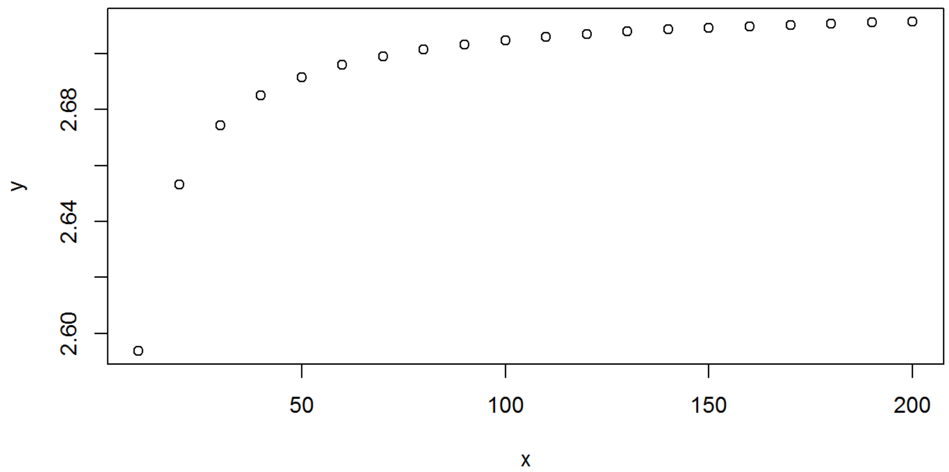
Note the difference between the numerical integration and theoretical value.

Example : exponential limit

```
x <- seq(10, 200, by = 10)
y <- (1 + 1/x)^x
exp(1) - y
```

```
##  [1] 0.12453937 0.06498412 0.04396305 0.03321799 0.02669380 0.02231169
##  [7] 0.01916546 0.01679689 0.01494937 0.01346800 0.01225375 0.01124034
## [13] 0.01038175 0.00964501 0.00900592 0.00844625 0.00795208 0.00751253
## [19] 0.00711903 0.00676471
```

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



Missing data

in R, missing data is represented by NA.

```
a <- NA    # assign NA to variable A
is.na(a)     # is it missing?
```

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

```
a <- c(11, NA, 13)
is.na(a)
```

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

```
mean(a)
```

```
## [1] NA
```

```
mean(a, na.rm = TRUE) #NAs can be removed
```

```
## [1] 12
```

Expression and assignment

Expression is a phrase of code that can be executed.

```
seq(10, 20, by=3)
```

```
## [1] 10 13 16 19
```

```
4
```

```
## [1] 4
```

```
mean(c(1,2,3))
```

```
## [1] 2
```

```
1 > 2
```

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

If the evaluation of the expression is saved using <-, then it called an assignment.

```
x1 <- seq(10, 20, by=3)
x2 <- 1>2
```

Logical expression

A logical expression is formed using

- the comparison operators
- `<`, `>`, `<=`, `>=`, `==`, and `!=` (not equal to)
- and the logical operators
- `&` (and), `|` (or), and `!` (not).

The value of a logical expression is either `TRUE` or `FALSE`.

- The integers 1 and 0 can also be used as `TRUE` or `FALSE`.

```
c(0, 0, 1, 1) | c(0, 1, 0, 1)
```

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

x[subset]

We can extract a subvector using a subset as a vector of TRUE/FALSE.

```
x <- 1:10
x%%4 == 0
```

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

```
( y <- x[ x%%4==0 ] )
```

```
## [1] 4 8
```

R also provide subset function, which ignore `NA`.

- whereas `x[subset]` preserve `NA`.

```
x <- c(1, NA, 3, 4)
x[x > 2]
```

```
## [1] NA  3  4
```

```
subset(x, subset = x>2 )
```

```
## [1] 3 4
```

For the index position of `TRUE` elements, use `which(x)`

```
x <- c(1, 1, 2, 3, 5, 8, 13)
which(x%%2 == 0)
```

```
## [1] 3 6
```

Example : rounding error

Many floating numbers are subject to rounding errors in digital computers.

```
2*2 == 4
```

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

```
sqrt(2)*sqrt(2) == 2
```

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

The solution is to use all.equal(x,y), which returns TRUE if the difference between x and y is smaller than some tolerance.

```
all.equal(sqrt(2)*sqrt(2), 2)
```

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

Matrix

Matrix is created from a vector using the function matrix:

- `matrix( data, nrow =1, ncol=1, byrow=TRUE )`
- data : vector of length at most nrow*ncol
    - if length of vector< nrow*ncol, then data is reused as many times as is needed
- nrow : number of rows
- ncol : number of columns
- byrow = TRUE : fill the matrix up row-by-row
- byrow = FALSE : fill the matrix up column-by-column, default

`diag(x)` : create diagonal matrix `rbind(...)` : join matrices with rows of the same length `cbind(...)` : join matrices with columns of the same length

Matrix example

```r
(A <- matrix(1:6, nrow=2, ncol=3, byrow=TRUE))
```

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

```r
A[1, 3] <- 0
A
```

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

```r
A[, 2:3]
```

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

```r
(B <- diag(c(1,2,3)))
```

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

Matrix operation

- Usual algebraic operations, including `*`, act elementwise.
- To perform matrix operation, we use `%*%`.
- `nrow(x)`, `ncol(x)`
- `det(x)` : determinant of `x`
- `t(x)` : transpose of `x`
- `solve(A, B)` : returns `x` such that `A %*% x = B`
- If `A` is invertable, the `solve(A)` is the inverse of `A`.

```
A <- matrix(c(3,5,2,3), nrow=2, ncol=2)
B <- matrix(c(1,1,0,1), nrow=2, ncol=2)
```

```
A %*% B
```

```
##      [,1] [,2]
## [1,]    5    2
## [2,]    8    3
```

```
A.inv <- solve(A)
A %*% A.inv  # we observe rounding error
```

```
##      [,1]         [,2]
## [1,]    1 -8.88178e-16
## [2,]    0  1.00000e+00
```

```
A^(-1) # not an inverse. ^(-1) applies elementwise.
```

```
##          [,1]     [,2]
## [1,] 0.333333 0.500000
## [2,] 0.200000 0.333333
```

Workspace

The objects that you create using R remain in existence until you explicitly delete them.

- `rm(x)` : remove object x
- `rm(list=ls())` : remove all objects

Working directory

When you run R, it uses one of the directories on your hard drive as a working directory,

- where it looks for user-written programs and data files.

Check the working directory.

```
getwd()
```

Change the working directory to "dir"

- "dir" should be an appropriate directory name
- / is for directory and file address, . refers current directory, .. refers parent directory

```
setwd("dir")
```

Writing script

We can type and evaluate all possible R expression at the prompt, it is much more convenient to write scripts,

- which simply comprise collections of R expression.
- We use the terms program and code synonymously with script. You can use built-in editor in Rgui or Rstudio.
- or text-editor like Tinn-R, emacs

Help

To find out more about an R command or function x, you can type `help(x)` or just `?x`.

If you cannot remember the exact name, then `help.search("x")`.

HTML help command : `help.start()`

package

R provides various useful packages to help you.
https://cran.r-project.org/web/packages/

To install a package:
`install.packages("packagename")`
To access the package:
`library("packagename")`

Or use package menu.