Lecture 1: R 基础(一)

张伟平

Thursday 27$^{\text{th}}$ August, 2009

Contents

Chapter 1

Introduction

R 是一个免费开源的用于统计计算和作图的语言和软件环境. 支持的操作系统包括各种UNIX平台, Windows 和MacOS 等. R 提供了广泛的统计工具(线性和非线性建模, 经典的统计检验, 时间序列, 生存分析, 分类, 聚类... 等等)和灵活高质量的图形工具. R 的功能可以通过添加 **package** 来扩充.

1.1 R website

R 的官方网站为R Project Website. 该网站包含 R 在各种操作系统下的安装文件, R 的帮助文档包括一些免费的书籍等等. 目前 R 软件的最新版本为2.9.1. R 软件在Windows 下的安装是非常简单的. 在其它的操作系统下的安装也很容易, 请参考R FAQ: How can R be installed.

着重于R 在特殊的应用领域以及统计模型等方面的项目包括

- Bioconductor: Bioinformatics with R

- Rgeo: Spatial Statistics with R
- gR: gRaphical models in R
- Robust: Robust Statistics with R
- Rmetrics: Financial Market Analysis with R

1.2 Differences between R and S

1. R 由Ross Ihaka 和Robert Gentleman 共同创立, 可以视为是由AT&T 贝尔实验室所创的S 语言的另外一种实施. S 语言现在主要内含在Insightful 公司的S-PLUS 软件中.

2. R 和 S 在设计理念上存在有着许多不同. 我们可以视S 为一种目前有着三种引擎或者内核的语言: “old S engine” (*S version 3; S-Plus 3.x and 4.x*), “new S engine” (*S version 4; S-Plus 5.x and above*), 以及R. 因此R 和S 的差异主要是“engines”.

以下S 指S 引擎, R 指R 引擎.

1. 词法作用域(lexical scope): 考虑如下函数

↑Code

```
cube <- function(n) {
  sq <- function() n * n
  n * sq()
}
```

↓Code

在S 下, sq() 不知道变量n, 除非它全局指定.

↑Example

```
S> cube(2)
Error in sq():  Object "n" not found
Dumped
S> n <- 3
S> cube(2)
[1] 18
```

↓Example

在R 中, cube() 被调用时其产生的“environment”也被调用:

↑Example

```
R> cube(2)
[1] 8
```

↓Example

2. 模型

- 在S 中, 表示响应变量y 对x^3 的回归用`lm(y~ x^3)`, 而在R 中, 应该用`lm(y ~ I(x^3))`.
- **glm** 族对象在R 和S 中尽管功能相同, 但是其成分名称不同.
- 选项*na.action* 在R 中默认为“na.omit”, 而S 中没有.
- R 中`y ~ x+0` 和`y~x-1` 等价, 没有参数的模型可以通过`y~0` 指定.

3. 详细的或者其他的R 与S 之间的差别请参看: What are the differences between R and S?

1.3 Start with R

R 是一种解释性程序语言, 因此不必像 C 或者 Fortran 之类的编译语言首先要构成一个完整的程序形式. 当 R 运行时, 所有变量, 数据, 函数及结果都以**对象** (objects) 的形式存在计算机的活动内存中, 并冠有相应的名字代号. 我们可以通过用一些运算符(如算术，逻辑，比较等) 和一些函数(其本身也是对象) 来对这些对象进行操作.

这是R(S)与其他主要的统计系统之间的重要差异. 在R中,一个统计分析一般分几步完成, 中间的结果都是存储在对象里. 因此在回归或者判别分析时, SAS和SPSS会给出非常多的输出结果, 但R 将给出最少的输出, 而将结果存储在一个拟合对象中, 可以使用R函数进行后续的分析.

R 语言中最简单的命令莫过于通过输入一个对象的名字来显示其内容了. 例如, 一个名为 n 的对象, 其内容是数值10:

↑Example

```
> n
[1] 10
```

↓Example

方括号中的数字1表示从 n 的第一个元素开始显示. 其实该命令的功能在这里于函数 print 相似, 输出结果与 print(n) 相同(但有些情况下, 例如内嵌在一个函数或循环中时,就必须得用print函数).

↑Example

```
> n<-1:30
> n
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
[21] 21 22 23 24 25 26 27 28 29 30
```

↓Example

对象的名字必须是以一个字母开头(A-Z 或 a-z), 中间可以包含字母, 数字(0-9), 点(.)及下划线(_). 因为 R 对对象的名字区分大小写, 所以 x 和 X 就可以代表两个完全不同的对象.

R有一个内建的帮助工具以获得指定名称函数的更多信息. 例如, 函数solve, 使用

↑Code

```
> help(solve)
```

↓Code

或者

```
> ?solve
```

↑Code

↓Code

对一些特殊的字符或者有着语法意义的词汇(如if,for,function等), 则必须将其放在双引号或者单引号中:

```
> help("[[") ; help("if")
```

↑Code

↓Code

默认状态下，函数help只会在被载入内存中的包中搜索。选项try.all.package在缺省值是FALSE, 但如果把它设为TRUE，则可在所有包中进行搜索:

↑Example

```
> help("bs")
No documentation for 'bs' in specified packages and libraries:
you could try 'help.search("bs")'
```

```
> help("bs", try.all.packages = TRUE)
Help for topic 'bs' is not in any loaded package
but can be found in the following packages:
Package     Library
splines    /usr/lib/R/library
```

↓Example

也可以启动R的HTML格式帮助文档

```
> help.start()
```

↑Code
↓Code

help.search()(等价的??)命令给出更多查找帮助的方式.另外, 也可以使用example()命令查看某指定的主题. 例如

```
> help.search("tree") 等价地, ??mean
```

↑Code
↓Code

会列出所有在帮助页面含有"tree"的函数. 注意如果有一些包是最近才安装的, 应该首先使用函数help.search中的rebuild选项来刷新数据库(e.g.,

help.search("tree", rebuild = TRUE)). 而

```
> example(mean); example(InsectSprays)等
```

↑Code ↓Code

会启动相应主题的范例.

使用函数 apropos 能在所有被载入内存的包中找出所有名字含有指定字符串的函数, 例如

↑Example

```
> apropos(help)
[1] "help" ".helpForCall" "help.search"
[4] "help.start"
```

↓Example

如果R命令是放在一个文件(例如commands.R)里,则可以使用

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

↑Code ↓Code

在Windows下也可以使用文件菜单运行此文件. 函数sink把控制台上下续的所有输出结果定向输出到指定文件或者重新将下续命令输出到控制台上,例如:

```
sink("record.lis") 和  sink()
```

↑Code ↓Code

分别表示将命令结果输出到当前工作目录下的文件"record.lis", 或者重新将命令输出到控制台上.

命令可以使用分号(';')来隔开, 或者使用一个新行. 基本命令可以成组的放在花括号('{' 和 '}')之间. 注释符号('#')可以从任何地方开始, 表示其后一直到该行结束部分被注释掉. 如果一个命令没有完成, R将会在下续的第二行开始处给出一个提示符

```
+
```

表示继续读入命令, 直到该命令语法完整为止.

命令 objects 与 ls 列出当前 R 进程(内存)中的所有对象名称例如

↑Example

```
> name <- "Carmen"; n1 <- 10; n2 <- 100; m <- 0.5
```

```
> ls()
[1] "m" "n1" "n2" "name"
```

↓Example

如果只要显示出在名称中带有某个指定字符的对象，则通过设定选项 pattern 来实现(可简写为 pat):

↑Example

```
> ls(pat = "m")
[1] "m" "name"
```

↓Example

如果进一步限为显示在名称中以某个字母开头的对象，则可:

↑Example

```
> ls(pat = "^m")
[1] "m"
```

↓Example

rm 命令可以将指定的对象从内存中删除. rm(x)删除名为x的对象, rm(x,y)删除名为x和y的对象,而rm(list=ls(all=TRUE))将删除当前内存中的所有对象.

Chapter 2

Data with R

2.1 Objects

R中常用的对象(objects)包括向量(vector), 因子(factor), 数组(array), 矩阵(matrix), 数据框(data frame), 时间序列(ts), 列表(list)等等. 所有的对象都有两个内在属性: 类型(mode)和长度(length). 类型是对象元素的基本种类, 常用的有四种: 数值型(numeric), 字符型(character), 复数型(complex)和逻辑型(logical)(FALSE或TRUE). 虽然也存在其它的类型(raw), 但是并不能用来表示数据, 例如函数或表达式; 长度是对象中元素的数目. 对象的类型和长度可以分别通过函数mode和length得到. 例如

↑Example

```
> x <- 1
> mode(x)
[1] "numeric"
```

```
> length(x)
[1] 1
> A <- "Gomphotherium"; compar <- TRUE; z <- 1i
> mode(A); mode(compar); mode(z)
[1] "character"
[1] "logical"
[1] "complex"
```

↓Example

无论什么类型的数据，缺失数据总是用NA(不可用)来表示; 对很大的数值则可用指数形式表示:

↑Example

```
> N <- 2.1e23
> N
[1] 2.1e+23
```

↓Example

当前工作空间中的所有对象名称可以用函数objects()来查看.

R可以正确地表示无穷的数值, 如用Inf和-Inf表示$\pm\infty$, 或者用NaN(非数字)表示不是数字的值. 例如

↑Example

```
> x <- 5/0
> x
[1] Inf
> exp(x)
[1] Inf
> exp(-x)
[1] 0
> x - x
[1] NaN
```

↓Example

下表给出了表示数据的对象的类别概览

对象	类型	是否允许同一个对象中有多种类型?
向量	数值型, 字符型, 复数型, 或逻辑型	否
因子	数值型或字符型	否
数组	数值型, 字符型, 复数型, 或逻辑型	否
矩阵	数值型, 字符型, 复数型, 或逻辑型	否
数据框	数值型, 字符型, 复数型, 或逻辑型	是
时间序列(ts)	数值型, 字符型, 复数型, 或逻辑型	否
列表	数值型，字符型, 复数型, 或逻辑型 函数, 表达式,...	是

向量是一个变量, 其意思也即人们通常认为的那样; 因子是一个分类变量; 数组是一个k维的数据表; 矩阵是数组的一个特例, 其维数k = 2. 注意, 数组或者矩阵中的所有元素都必须是同一种类型的; 数据框是由一个或几个向量和(或)因子构成, 它们必须是等长的, 但可以是不同的数据类型; “ts”表示时间序列数据, 它包含一些额外的属性, 例如频率和时间; 列表可以包含任何类型的对象, 包括列表! 对于一个向量, 用它的类型和长度足够描述数据; 而对其它的对象则另需一些额外信息, 这些信息由外在的属性(attribute)给出. 例如我们可以引用这些属性中的*dim*属性, 其是用来表示对象的维数, 比如一个2行2列的的矩阵, 它的dim属性是一对数值[2,2], 但是其长度是4.例如:

↑Example

```
> x
   C1 C2
R1  1  3
R2  2  4
> attributes(x)
$dim
[1] 2 2
$dimnames
$dimnames[[1]]
[1] "R1" "R2"
$dimnames[[2]]
[1] "C1" "C2"
> attributes(x)$dim
[1] 2 2
```

↓Example

类型之间可以通过as.*something*()形式的命令来转换. 例如:

↑Example

```
> z<-0:9
> digits<-as.character(z)
> digits
 [1] "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
> as.numeric(digits)->x
> x
 [1] 0 1 2 3 4 5 6 7 8 9
> mode(x)
[1] "numeric"
```

↓Example

所有对象都有一个属性: 类*class*. 可以通过函数class来得到其类型. 这个特殊的属性被用来在R中进行面向对象的程序设计.

2.2 Reading data in a file

R使用工作目录(working directory)来完成读写文件. 可以使用命令 getwd()和setwd()来获得和设定工作目录, 例如

```
setwd("C:/data") 或者setwd("/home/paradis/R")
```

↑Code ↓Code

R 可以使用函数read.table(有好几种变体形式), scan and read.fwf 来进行读取txt (ASCII)文件中的数据. 命令

```
> mydata <- read.table("data.dat")
```

↑Code ↓Code

从当前工作目录中的文件data.dat中读取数据, 创建一个名为mydata的类型为数据框的对象. 函数read.table有很多参数:

[使用?read.table来了解各个参数的含义]

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
dec = ".",row.names, col.names, as.is = FALSE,
```

↑Code

```
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 = "#")
```

↓Code

函数read.table有几种变体形式

[各个变体表示什么?]

↑Code

```
read.csv(file, header = TRUE, sep = ",", quote="\"", dec=".",
fill = TRUE, ...)
read.csv2(file, header = TRUE, sep = ";", quote="\"", dec=",",
fill = TRUE, ...)
read.delim(file, header = TRUE, sep = "\t", quote="\"", dec=".",
fill = TRUE, ...)
read.delim2(file, header = TRUE, sep = "\t", quote="\"", dec=",",
fill = TRUE, ...)
```

↓Code

函数scan比read.table更加灵活. 它们之间的一个区别在于scan可以指定读入变量的类型,例如

```
> mydata <- scan("data.dat", what = list("", 0, 0))
```

↑Code

↓Code

读取了文件data.dat中三个变量, 第一个是字符型变量, 后两个是数值型变量. 另一个重要的区别在于scan()可以用来创建不同的对象, 向量, 矩阵, 数据框, 列表... 缺省情况下, scan创建一个向量类型的对象.

[使用?scan来了解各个参数]

↑Code

```
scan(file = "", what = double(0), nmax = -1, n = -1, sep = "",
quote = if (sep=="\n") "" else "'\"", dec = ".",
skip = 0, nlines = 0, na.strings = "NA",
flush = FALSE, fill = FALSE, strip.white = FALSE, quiet = FALSE,
blank.lines.skip = TRUE, multi.line = TRUE, comment.char = "")
```

↓Code

函数read.fwf可以用来读取文件中一些固定宽度格式的数据

[使用?read.fwf来了解各个参数]

```
read.fwf(file, widths, sep="\t", as.is = FALSE,
skip = 0, row.names, col.names, n = -1, ...)
```

↑Code ↓Code

除选项widths外,其余选项基本和read.table相同. 例如

↑Example

数据

```
A1.501.2
A1.551.3
B1.601.4
B1.651.5
C1.701.6
C1.751.7
```

```
> mydata <- read.fwf("data.txt", widths=c(1, 4, 3))
> mydata
  V1 V2 V3
1  A 1.50 1.2
2  A 1.55 1.3
3  B 1.60 1.4
4  B 1.65 1.5
5  C 1.70 1.6
6  C 1.75 1.7
```

↓Example

2.3 Saving data

函数write.table可以在文件中写入一个对象, 一般是写一个数据框, 也可以是其它类型的对象(向量, 矩阵...). 参数和选项:

↑Code

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

↓Code

若想用更简单的方法将一个对象(例如x, 可以是向量, 矩阵,或者数组)写入文件, 可以使用命令`write(x,file ="data.txt")`. 这里有两个选项:

- nc(或者ncol), 用来定义文件中的列数(在缺省情况下, 如果x是字符型数据, 则nc=1; 对于其它数据类型nc=5);
- append,用来决定是附加到文件data.txt内容后(TRUE), 还是替换掉原来已有的内容(FALSE).

要记录一组任意数据类型的对象, 我们可以使用命令`save(x, y, z, file= "xyz.RData")`. 可以使用选项ASCII=TRUE使得数据在不同的机器

之间更简易转移. 数据(用R的术语来说叫做工作空间) 可以在使用`load("xyz.RData")`之后被加载到内存中. 函数save.image()是`save(list =ls(all=TRUE),file=".RData")`的一个简捷方式。

2.4 generating data

生成一个向量(函数`c()`,算子`":"`,函数`seq()`等等:

↑Example

```
> x<-c(1,2,3,4,5)      > ch<-c("sa","ba")      > 1:5-1
> x                    > ch                     [1] 0 1 2 3 4
[1] 1 2 3 4 5           [1] "sa" "ba"          > 1:(5-1)
> y<-1:5               > z<-seq(1,5)            [1] 1 2 3 4
> y                    > z
[1] 1 2 3 4 5          [1] 1 2 3 4 5
```

↓Example

函数seq()功能强大:

[seq(length.out =0)的结果是什么?]

```
seq(from, to)                  #生成从from到to间隔为1的序列
seq(from, to, by= )            #生成从from到to间隔为by的序列
seq(from, to, length.out= )    #生成从from到to的长度为length.out的
序列
seq(along.with= )              #生成序列1:length(along.with)
seq(from)                      #生成序列1:length(from)
seq(length.out= )              #生成序列1:length(length.out)
```

↓Code

如果想用键盘输入一些数据也是可以的，只需要直接使用默认选项的scan函数, 以Ctrl+Z结束.

↑Example

```
> z <- scan()                  > z
1: 1.0 1.5 2.0 2.5             [1] 1.0 1.5 2.0 2.5
5:
Read 4 items
```

↓Example

重复某个(些)值可以使用函数rep():

↑Example

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

```
> rep(1:2,each=2)
 [1] 1 1 2 2
```

↓Example

生成规则的因子可以使用函数gl():

↑Example

```
> gl(2,3)
[1] 1 1 1 2 2 2
Levels: 1 2
> gl(2,2,label=c("F","M"))
[1] F F M M
```

```
> gl(2,3,length=4)
[1] 1 1 1 2
Levels: 1 2
```

```
Levels: F M
```

↓Example

最后, 函数expand.grid()创建一个数据框，结果是把各参数的各水平完全搭配:

↑Example

```
> expand.grid(h= c(60, 80), w=c(100),s= c("M","F"))
   h     w  s
1 60   100 M
2 80   100 M
3 60   100 F
4 80   100 F
```

↓Example

产生随机序列可以使用R内建的各种函数:

分布名称	函数
Gaussian (normal)	rnorm(n, mean=0, sd=1)
exponential	rexp(n, rate=1)
gamma	rgamma(n, shape, scale=1)
Poisson	rpois(n, lambda)
Weibull	rweibull(n, shape, scale=1)
Cauchy	rcauchy(n, location=0, scale=1)
beta	rbeta(n, shape1, shape2)
'Student' (t)	rt(n, df)
Fisher-Snedecor(F)	rf(n, df1, df2)
Pearson (χ^2)	rchisq(n, df)
binomial	rbinom(n, size, prob)
multinomial	rmultinom(n, size, prob)
geometric	rgeom(n, prob)
hypergeometric	rhyper(nn, m, n, k)
logistic	rlogis(n, location=0, scale=1)
lognormal	rlnorm(n, meanlog=0, sdlog=1)
negative binomial	rnbinom(n, size, prob)
uniform	runif(n, min=0, max=1)
Wilcoxon's statistics	rwilcox(nn, m, n), rsignrank(nn, n)

大多数这种统计函数都有相似的形式, 只需用d、p或者q去替代r, 比如密度函数(dfunc(x, ...)), 累计概率分布函数(也即分布函数)(pfunc(x, ...))和分位

数函数(qfunc(p, ...), $0 < p < 1$).

↑Example

```
> pnorm(0)                    > qnorm(0.05)
[1] 0.5                       [1] -1.644854
> dnorm(0)                    > 1/sqrt(2*pi)
[1] 0.3989423                 [1] 0.3989423
```

↓Example

2.5 Manipulating objects

2.5.1 Creating objects

▷ *vector* 向量是R中的基本对象,前面我们已经介绍了使用函数c(),seq()等等创建向量. 向量可以是数值型,字符型和逻辑性.

```
vector(mode = "logical", length = 0) as.vector(x, mode = "any")
```

↑Code

```
is.vector(x, mode = "any")
```
↓Code

▷ *factor* 一个因子不仅包括分类变量本身还包括变量不同的可能水平(即使它们在数据中不出现). 因子函数factor用下面的选项创建一个因子:

```
factor(x, levels = sort(unique(x), na.last = TRUE),
   labels = levels, exclude = NA, ordered = is.ordered(x))
```
↑Code ↓Code

于此函数相关的函数包括ordered(同样功能, 为和S语言兼容), is.factor, is.ordered, as.factor 和as.ordered等. 函数levels可以查看一个因子对象的水平.

▷ *matrix* 一个矩阵实际上是有一个附加属性（维数dim）的向量

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

例如

↑Example

```
> matrix(data=5, nr=2, nc=2)
     [,1] [,2]
[1,] 5    5
[2,] 5    5
> matrix(1:6, 2, 3)
     [,1] [,2] [,3]
[1,] 1    3    5
[2,] 2    4    6
> matrix(1:6, 2, 3, byrow=TRUE)
     [,1] [,2] [,3]
[1,] 1    2    3
[2,] 4    5    6
```

```
> x <- 1:6
> x
[1] 1 2 3 4 5 6
> dim(x)
NULL
> dim(x) <- c(2, 3)
> x
     [,1] [,2] [,3]
[1,] 1    3    5
[2,] 2    4    6
```

↓Example

▷ *data.fame* 前面我们已经看到一个数据框可以由函数read.table 间接创建; 这里也可以用函数data.frame 来创建. 数据框中的向量必须有相同的长

度, 如果其中有一个比其它的短, 它将“循环”整数次(以使得其长度与其它向量相同):

↑Example

```
> x <- 1:2; n <- 10; M <- c(10, 35); y <- 2:4
> data.frame(x, n)
  x n
1 1 10
2 2 10
> data.frame(A=x, B=M)
  A B
1 1 10
2 2 35
> data.frame(x, y)
Error in data.frame(x, y) :
arguments imply differing number of rows: 2, 3
```

↓Example

▷ *array* 数据(array)是矩阵的多维推广.

↑Code

```
array(data = NA, dim = length(data), dimnames = NULL)
as.array(x)
is.array(x)
```

↓Code

例如

↑Example

```
> x<-array(1:2,dim=c(2,2,2))
> x
, , 1
     [,1] [,2]
[1,]    1    1
[2,]    2    2
, , 2
     [,1] [,2]
[1,]    1    1
```

```
> x[1,,]
     [,1] [,2]
[1,]    1    1
[2,]    1    1
> x[,2,]
     [,1] [,2]
[1,]    1    1
[2,]    2    2
```

```
[2,]    2    2
```

↓Example

▷ *list* 列表可以用list函数创建, 方法与创建数据框类似. 它对其中包含的对象没有什么限制. 和data.frame()比较, 缺省值没有给出对象的名称; 用前面的向量x和y举例:

↑Example

```
> L2<-list(A=x,B=as.character(y))
> L2
$A
[1] 1 2

$B
[1] "2" "3" "4"
```

```
> names(L2)
[1] "A" "B"

> L2$A
[1] 1 2
```

↓Example

▷ *expression* 表达式(Expression)类型的对象在R中有着很基础的地位, 是R能够解释的字符序列. 所有有效的命令都是表达式. 一个命令被直接从键盘输入后, 它将被R求值, 如果是有效的则会被执行. 在很多情况下, 构造一个不被求值的表达式是很有用的: 这就是函数expresssion要做的. 当然也可以随后用eval()对创建的表达式进行求值.

↑Example

```
> x <- 3; y <- 2.5; z <- 1
> exp1 <- expression(x / (y + exp(z)))
> exp1
expression(x/(y + exp(z)))
> eval(exp1)
[1] 0.5749019
```

↓Example

表达式也可以在其它地方用来在图表中添加公式(见后); 表达式可以由字符型变量创建; 一些函数把表达式当作参数, 例如可以求偏导数的函数D().

↑Example

```
> D(exp1, "x")
1/(y + exp(z))
> D(exp1, "y")
-x/(y + exp(z))^2
> D(exp1, "z")
-x * exp(z)/(y + exp(z))^2
```

↓Example

▷ *ts* 函数ts可以由向量(一元时间序列)或者矩阵(多元时间序列)创建一个时间序列类型对象, 并且有一些表明序列特征的选项(带有缺省值):

```
ts(data = NA, start = 1, end = numeric(0), frequency = 1,
deltat = 1, ts.eps = getOption("ts.eps"), class, names)
```

↑Code ↓Code

try ?ts for details.

2.5.2 Operators

R 中的赋值运算符号有“<-”,“<<-”和“=”. 例如赋值(value)到对象 x:

[三种赋值有什么区别?]

↑Code

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

↓Code

R 中的运算符如下:

运算符					
数学运算		比较运算		逻辑运算	
+	加法	<	小于	!x	逻辑非
-	减法	>	大于	x&y	逻辑与
*	乘法	<=	小于	x&&y	同上
/	除法	>=	大于	x|y	逻辑或
^	乘方	==	等于	x||y	同上
%%	模	!=	不等于	xor(x, y)	异或
%/%	整除				

数学运算符和比较运算符作用于两个元素上(x + y, a < b); 数学运算符不只是作用于数值型或复数型变量, 也可以作用在逻辑型变量上; 在后一种情况中, 逻辑型变量被强制转换为数值型. 比较运算符可以适用于任何类型: 结果是返回一个或几个逻辑型变量.

逻辑型运算符适用于一个(对“!”运算符)或两个逻辑型对象(其它运算符), 并且返回一个(或几个)逻辑性变量. 运算符“逻辑与”和“逻辑或”存在两种形式: “&”和“|”作用在对象中的每一个元素上并且返回和比较次数相等长度的逻辑值: “&&”和“||”只作用在对象的第一个元素上.

R 中的数学运算中“^”和“<-,-,=,<<-”为从右向左, 而其他运算都是从左到右. 例如

[R Language Definition]

↑Code

↓Code

```
2^2^3 = 2^8 ≠ 4^3;  1 - 1 - 1 = -1 ≠ 1
```

比较运算符作用在两个被比较对象的每个元素上(如果需要, 将循环使用最短的变量), 从而返回一个同样大小的对象. 为了“整体”比较两个对象, 可以使用两个函数: identical和all.equal

↑Example

```
> x <- 1:3; y <- 1:3
> x == y
[1] TRUE TRUE TRUE
> identical(x, y)
[1] TRUE
> all.equal(x, y)
[1] TRUE
```

```
> x=1:2;y=1:3
> x==y
[1]  TRUE  TRUE FALSE
Warning message:
In x == y : longer object length is not
a multiple of shorter object length
```

↓Example

identical比较数据的内在关系, 如果对象是严格相同的返回TRUE, 否则返回FALSE. all.equal用来判断两个对象是否“近似相等”, 返回结果为TRUE或者对二者差异的描述. 后一个函数在比较数值型变量时考虑到了计算过程中的近似. 在计算机中数值型变量的比较有时很是令人惊奇.

↑Example

```
> 0.9 == (1 - 0.1)
[1] TRUE
> identical(0.9, 1 - 0.1)
```

```
> 0.9 == (1.1 - 0.2)
[1] FALSE
> identical(0.9, 1.1 - 0.2)
```

```
[1] TRUE                                [1] FALSE
> all.equal(0.9, 1 - 0.1)               > all.equal(0.9, 1.1 - 0.2)
[1] TRUE                                [1] TRUE
> all.equal(0.9, 1.1 - 0.2, tolerance = 1e-16)
[1] "Mean relative difference: 1.233581e-16"
```

↓Example

对字符型对象的运算常用的几个函数有substr()和paste()等. 例如

↑Example

```
> colors <- c("red", "yellow", "blue")
> more.colors <- c(colors, "green", "magenta", "cyan")
> substr(colors, 1, 2)
[1] "re" "ye" "bl"
> paste(colors, "flowers")
[1] "red flowers"    "yellow flowers" "blue flowers"
> paste("several ", colors, "s", sep="")
[1] "several reds"    "several yellows" "several blues"
```

```
> paste("I like", colors, collapse = ", ")
[1] "I like red, I like yellow, I like blue"
```

↓Example

2.5.3 Accessing the values of an object: the indexing system

下标系统可以用来有效、灵活且有选择性地访问一个对象中的元素; 下标可以是数值型的或逻辑型的. 举例来说, 对向量

↑Example

```
> x <- 1:5
> x[3]
[1] 3
> x[c(F,T)]
[1] 2 4
```

```
> x[c(1,3)]
[1] 1 3
> x[x>2]
[1] 3 4 5
> x[3] <- 20
```

```
> x[-2]
 [1] 1 3 4 5
> x[-c(1,3)]
 [1] 3 5
```

↓Example

当用一个非整数的数字访问向量某个元素是,小数部分就会被略去. 例如

↑Example

```
> x
[1] 1 2
> x[0.5]
numeric(0)
> x[1.5]
[1] 1
```

↓Example

对矩阵

↑Example

```
> x <- matrix(1:6, 2, 3)
> x
     [,1] [,2] [,3]
[1,]    1    3    5
```

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

```
[2,]  2    4    6
> x[, 3] <- 21:22
> x
     [,1] [,2] [,3]
[1,]  1    3   21
[2,]  2    4   22
> x[, 3]
[1] 21 22
```

```
[1,]    3    5
[2,]    4    6
> x[, 3, drop = FALSE]
     [,1]
[1,] 21
[2,] 22
> x[-1,]
 [1] 2 4 22
```

↓Example

下标系统也普遍适用于数组和数据框, 使用和数组维数同样多的下标(例如, 一个三维数组：x[i, j, k], x[, , 3], x[, , 3, drop = FALSE], 等等). 记住下标必须使用方括号, 而圆括号是用来指定函数的参数的.

对于列表, 访问不同的元素可以通过单一的或者双重的方括号来实现; 它们的区别是: 单个括号返回一个列表, 而双重括号将提取列表中的对象. 在得到的结果中也可以使用下标, 就像之前在向量、矩阵等情况中看到的那样. 例如, 一个列表中的第3个对象是一个向量, 它的取值可以使用my.list[[3]][i]来访问, 如

果是一个三维数组则使用my.list[[3]][i, j, k]等等; 另一个区别是my.list[1:2]将返回一个列表, 包含原始列表的第1个和第2个元素, 而my.list[[1:2]]不会给出期望的结果.

2.5.4 Accessing the values of an object with names

对象的元素也可以通过名字访问. 例如对向量

↑Example

```
> x<-1:3                          > x
> names(x)                         a b c
NULL                               1 2 3
> names(x)<-c("a","b","c")        > x["b"]
                                   b
                                   2
```

↓Example

对于矩阵和数据框, colnames和rownames分别是列和行的标签. 它们可以通过各自的函数来访问, 或者通过dimnames返回包含两个名称向量的列表.

↑Example

```
> x<-matrix(1:4,2)
> x
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> dimnames(x)
NULL
> dimnames(x)<-list(c("R1","R2"),c("C1","C2"))
> x
   C1 C2
R1  1  3
R2  2  4
```

```
> rownames(x)<-c("r1","r2")
> colnames(x)<-c("c1","c2")
> x
   c1 c2
r1  1  3
r2  2  4
> x["R2",]
 [1] 2 4
```

↓Example

对于list对象, 我们可以使用*mylist$name*的形式访问其名为name的列表. 例如

↑Example

```
> x<-matrix(1:4,2); y<-1:3; x<-list(A=x,B=y)
> x$B                    > x$A[1,2]
[1] 1 2 3                 [1] 3
> x$B[2]
[1] 2
```

↓Example

2.5.5 Arithmetics and simple functions

向量是可以进行算术运算的,例如

↑Example

```
> x <- 1:4              > y <- 1:2
> y <- rep(1, 4)        > z <- x + y
> z <- x + y            > z
> z                     [1] 2 4 4 6
```

```
[1] 2 3 4 5
> a <- 10
> z <- a * x
> z
[1] 10 20 30 40
```

```
> y <- 1:3
> z <- x + y
Warning message: In x + y : longer object length
 is not a multiple of shorter object length
> z
[1] 2 4 6 5
```

↓Example

R中用来处理数据的函数太多了而不能全部列在这里。读者可以找到所有的基本数学函数(log, exp, log10, log2, sin, cos, tan, asin, acos, atan, abs, sqrt, . . .), 专业函数(gamma, digamma, beta, besselI, . . .), 同样包括各种统计学中有用的函数. 下表中列出了一部分函数:

函数	意义
`sum(x)`	对x中的元素求和
`prod(x)`	对x中的元素求连乘积
`max(x)`	x中元素的最大值
`min(x)`	x中元素的最小值

`which.max(x)`	返回x中最大元素的下标
`which.min(x)`	返回x中最小元素的下标
`range(x)`	与c(min(x), max(x))作用相同
`length(x)`	x中元素的数目
`mean(x)`	x中元素的均值
`median(x)`	x中元素的中位数
`var(x) or`	x中元素的的样本方差；如果x是一个矩阵
`cov(x)`	或者一个数据框，将计算协方差阵
`cor(x)`	如果x是一个矩阵或者一个数据框则计算
	相关系数矩阵(如果x是一个向量则结果是1)
`var(x, y) or`	x和y的协方差，如果是矩阵或数据框则计算x和y
`cov(x, y)`	对应列的协方差
`cor(x, y)`	x和y的线性相关系数，如果是矩阵或者数据框则
	计算相关系数矩阵

下面的函数返回更复杂的结果

`round(x, n)`	将x中的元素四舍五入到小数点后n位
`rev(x)`	对x中的元素取逆序

sort(x)	将x中的元素按升序排列；要按降序排列可以用命令rev(sort(x))
rank(x)	返回x中元素的秩
log(x, base)	计算以base为底的x的对数值
scale(x)	如果x是一个矩阵， 则中心化和标准化数据；若只进行中心化则使用选项scale=FALSE，只进行标准化则center=FALSE(缺省值是center=TRUE, scale=TRUE）
pmin(x,y,...)	逐对比较x和y的分量，返回一个向量，它的第i个元素是x[i], y[i],...中最小值
pmax(x,y,...)	同上，取最大值
cumsum(x)	返回一个向量，它的第i个元素是从x[1]到x[i]的和
cumprod(x)	同上，取乘积
cummin(x)	同上，取最小值
cummax(x)	同上，取最大值
match(x, y)	返回一个和x的长度相同的向量，表示x中与y中元素相同的元素在y中的位置(没有则返回NA)
which(x == a)	返回一个包含x符合条件(当比较运算结果为真的下标

	的向量，在这个结果向量中数值i说明x[i] == a
choose(n, k)	计算从n个样本中选取k个的组合数
na.omit(x)	忽略有缺失值(NA)的观察数据(如果x是矩阵或数据框则忽略相应的行)
na.fail(x)	如果x包含至少一个NA则返回一个错误消息
unique(x)	如果x是一个向量或者数据框，则返回一个类似的对象但是去掉所有重复的元素(对于重复的元素只取一个)
table(x)	返回一个表格,给出x中重复元素的个数列表(尤其对于整数型或者因子型变量)
table(x, y)	x与y的列联表
subset(x, ...)	返回x中的一个满足特定条件...的子集，该条件通常是进行比较运算：x$V1 < 10；如果x是数据框，选项select给出要保留的变量（或者用负号表示去掉）
sample(x, size)	从x中无放回抽取size个样本，选项replace = TRUE表示有放回的抽样

2.5.6 Matrix Computation

在矩阵运算中, 有如下几个常用的运算(函数):

Definition

%*% 矩阵相乘 t(A) 矩阵转置 eigen(A) 矩阵A的特征值(向量)
solve(A)矩阵求逆(solve(A,b)用于解线性方程组AX=b)

例如

[A*B,A/B表示什么意思呢?]

↑Example

```
> A<-matrix(1:4,2,2)
> B<-diag(2)
 >A%*%B
      [,1] [,2]
 [1,]    1    3
 [2,]    2    4
> solve(A)
     [,1] [,2]
```

```
> eigen(A)
$values
[1]  5.3722813 -0.3722813

$vectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

```
[1,]   -2  1.5
[2,]    1 -0.5
> dim(A)
[1] 2 2
```

↓Example

在执行形如t(A)%*%B的运算时,更有效率的方式是使用函数crossprod(A,B) [为什么?]
其他的矩阵运算函数包括%x% Kronecker乘积, cbind(), rbind(), dim(), diag(), nrow(), ncol(), lower.tri(), upper.tri(), qr(), svd().... 例如

↑Example

```
> A<-cbind(1:2,3:4)
> A
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
> A<-rbind(c(1,3),c(2,4))
> A
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

↓Example

函数lower.tri和upper.tri用来访问矩阵的下三角元素以及上三角元素,例如

↑Example

```
> A
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> B<-A
> B[lower.tri(A,diag=T)]<-0
> B
     [,1] [,2] [,3]
[1,]    0    4    7
[2,]    0    0    8
[3,]    0    0    0
```

```
> lower.tri(A,diag=T)
      [,1]  [,2]  [,3]
[1,] TRUE FALSE FALSE
[2,] TRUE  TRUE FALSE
[3,] TRUE  TRUE  TRUE
> B[lower.tri(A)]<-0
> B
     [,1] [,2] [,3]
[1,]    0    4    7
[2,]    0    0    8
[3,]    0    0    0
```

↓Example

奇异值分解: $A = UDV^T$, 其中矩阵U和T为正交矩阵, D为对角阵且D^2的对角元为矩阵A^TA的特征根. R 中使用函数svd()进行奇异值分解:

↑Example

```
> svd(A)->A.svd
> A.svd
$d
[1] 1.684810e+01 1.068370e+00 3.069525e-16

$u
           [,1]        [,2]       [,3]
[1,] -0.4796712  0.77669099  0.4082483
[2,] -0.5723678  0.07568647 -0.8164966
[3,] -0.6650644 -0.62531805  0.4082483

$v
           [,1]       [,2]       [,3]
[1,] -0.2148372 -0.8872307 -0.4082483
```

```
[2,] -0.5205874 -0.2496440  0.8164966
[3,] -0.8263375  0.3879428 -0.4082483
```

```
> A.svd$u%*%diag(A.svd$d)%*%t(A.svd$v)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

↓Example

由奇异值分解的性质, 当矩阵A为可逆方阵时, 很容易计算它的逆矩阵: $A^{-1} = VD^{-1}U^T$.

Choleski 分解: 若方阵$A > 0$, 则存在一个上三角阵U,使得$A = U^TU$. R中使用函数chol() 对方阵A作Choleski分解:

↑Example

```
> A                    > chol(A)
     [,1] [,2] [,3]         [,1] [,2] [,3]
[1,]    1    2    3    [1,]    1    2    3
```

```
[2,]    2    5    6   [2,]    0    1    0
[3,]    3    6   10   [3,]    0    0    1
```

↓Example

有了Choleski分解后, 我们就可以很方便的计算正定阵A的逆: $A^{-1} = U^{-1}(U^{-1})^T$. 这种计算比直接使用Gauss消元法计算$A^{-1}$要稳定的多. 也可以用于线性方程组求解.

QR分解: $A = QR$, 其中Q为一正交阵, R为一上三角阵. 例如

↑Example

```
> qr.A<-qr(A)
> qr.A
$qr
           [,1]       [,2]        [,3]
[1,] -3.7416574 -8.0178373 -12.0267559
[2,]  0.5345225 -0.8451543   0.5070926
[3,]  0.8017837 -0.4001484   0.3162278
$rank
[1] 3
```

```
$qraux
[1] 1.2672612 1.9164504 0.3162278
$pivot
[1] 1 2 3
attr(,"class")
[1] "qr"
```

↓Example

此输出为一个qr类. 使用函数qr.Q() 和qr.R() 对此qr类以得到矩阵Q和R:

↑Example

```
> Q<-qr.Q(qr.A)
> R<-qr.R(qr.A)
> Q%*%R
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    5    6
[3,]    3    6   10
```

↓Example

矩阵的条件数: 函数kappa() 可以用来计算矩阵的条件数.

↑Example

```
> kappa(A)
[1] 213.1021
```

↓Example

外积: 函数outer()在计算两个向量的外积时很有用.

↑Example

```
> x<-1:5
> outer(x,x,"*")
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
```

```
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

↓Example

在对矩阵进行行或者列操作中, 函数apply()也是很常用. 比如计算对矩阵A按列计算均值:

↑Example

```
> apply(A,2,mean)
[1] 2.000000 4.333333 6.333333
```

↓Example