

# R 软件入门

李东风

2016 年 12 月 17 日



# 目录

<b>第一章 R 语言介绍</b>	<b>1</b>
1.1 R 的历史和特点	1
1.2 R 的下载与安装	3
1.3 R 的使用样例	5
1.3.1 四则运算	5
1.3.2 数学函数	6
1.3.3 输出	8
1.3.4 向量计算与变量赋值	10
1.3.5 绘图示例	11
1.3.6 汇总统计示例	12
1.3.7 运行一个源程序文件	14
1.4 帮助	15
<b>第二章 R 编程基础</b>	<b>17</b>
2.1 数据类型和数据结构	17
2.1.1 常量与变量	18
2.1.2 数值型向量	18
2.1.3 逻辑型向量	24
2.1.4 字符型向量	25
2.1.5 向量的下标	29
2.1.6 日期和时间的处理	34
2.1.7 因子	40
2.1.8 R 矩阵和多维数组	43
2.1.9 数据框	49
2.1.10 列表	52
2.1.11 R 工作空间	56
2.2 R 输入输出	58

2.2.1	输入输出的简单方法	58
2.2.2	文本格式数据访问	60
2.2.3	Excel 表访问	60
2.2.4	Access 数据库访问	63
2.2.5	Oracle 数据库访问	63
2.2.6	MySQL 数据库访问	64
2.2.7	格式化输出	66
2.2.8	文件输入输出	67
2.2.9	目录和文件管理	71
2.3	程序控制结构	71
2.3.1	表达式	71
2.3.2	分支结构	72
2.3.3	循环结构	73
2.3.4	函数	74
2.3.5	程序调试	76
2.4	R 程序效率	81
2.4.1	向量化编程	81
2.4.2	减少显式循环	85
2.4.3	R 的计算函数	94
<b>第三章</b>	<b>R 数据整理与概括</b>	<b>101</b>
3.1	数据整理	101
3.1.1	数据读取技巧	101
3.1.2	数据框子集与合并	106
3.1.3	排序与标准化	111
3.1.4	长表与宽表的变形与整理	111
3.2	数据概括	119
3.2.1	数据框变量简单概括	119
3.2.2	分类变量概括	122
3.2.3	数据框概括	128
3.2.4	分类概括	129
<b>第四章</b>	<b>R 绘图</b>	<b>133</b>
4.1	常用图形	133
4.1.1	条形图	133
4.1.2	直方图和密度估计图	138
4.1.3	盒形图	140

4.1.4	QQ 图	143
4.1.5	散点图	144
4.1.6	曲线图	148
4.1.7	三维曲面图	150
4.2	图形定制	152
4.2.1	低级图形函数	152
4.2.2	图形参数	159
4.2.3	图形设备	163
4.2.4	更多中文字体	164
<b>第五章</b>	<b>用 R 作随机模拟</b>	<b>167</b>
5.1	随机模拟	167
5.1.1	介绍	167
5.1.2	随机模拟示例	169
<b>第六章</b>	<b>用 Rcpp 连接 C++ 代码</b>	<b>173</b>
6.1	Rcpp 介绍	173
6.1.1	简单样例	174
6.1.2	R 与 C++ 的类型转换	179
6.2	Rcpp 属性	180
6.2.1	介绍	180
6.2.2	R 中编译链接 C++ 代码	183
6.2.3	Rcpp 属性的其它功能	184
6.3	Rcpp 提供的向量类的 C++ 数据类型	187
6.3.1	RObject 类	187
6.3.2	IntegerVector 类	188
6.3.3	NumericVector 类	191
6.3.4	Rcpp 的其它向量类	196
6.4	Rcpp 提供的其它数据类型	197
6.4.1	Named 类型	197
6.4.2	List 类型	199
6.4.3	Rcpp 的 DataFrame 类	200
6.4.4	Rcpp 的 Function 类	201
6.4.5	Rcpp 的 Environment 类	203
6.4.6	Rcpp 的 S4 类和 ReferenceClass 类	204
6.4.7	R 的数学函数库	205
6.5	Rcpp 糖	206

6.5.1	向量化的运算符 . . . . .	208
6.5.2	Rcpp 糖对调用 R 函数的改进 . . . . .	209
6.5.3	返回糖表达式的函数 . . . . .	211
6.5.4	数学函数 . . . . .	215
6.6	用 Rcpp 帮助制作 R 扩展包 . . . . .	217
6.6.1	介绍 . . . . .	217
6.6.2	生成扩展包 . . . . .	219

# 第一章 R 语言介绍

## R 软件介绍

- 历史；特点；用途。
- R 网站，下载安装，扩展包安装。
- 运行样例：计算器；作图；自定义函数。
- 帮助系统使用。

## 1.1 R 的历史和特点

### R 的历史

- S 语言: Rick Becker, John Chambers 等人在贝尔实验室开发<sup>1</sup>，著名的 C 语言、Unix 系统也是贝尔实验室开发的。

---

<sup>1</sup> 第一个版本开发于 1976-1980，基于 Fortran；于 1980 年移植到 Unix，并对外发布源代码。1984 年出版的“棕皮书”

Becker, Richard A. and John M. Chambers (1984), “S: An Interactive Environment for Data Analysis and Graphics”, Wadsworth Advanced Books Program, Belmont CA

总结了 1984 年为止的版本，并开始发布授权的源代码。这个版本叫做旧 S。与我们现在用的 S 语言有较大差别。

1989-1988 对 S 进行了较大更新，变成了我们现在使用的 S 语言，称为第二版。1988 年出版的“蓝皮书”做了总结：

Becker, Richard A., John M. Chambers and Allan R. Wilks (1988), “The New S Language”, Chapman and Hall, New York.

1992 年出版的“白皮书”描述了在 S 语言中实现的统计建模功能，增强了面向对象的特性。软件称为第三版，这是我们现在用的多数版本。

Chambers, John M. and Trevor Hastie, eds. (1992), “Statistical Models in S”, Chapman and Hall, New York.

1998 年出版的“绿皮书”描述了第四版 S 语言，主要是编程功能的深层次改进。现行的 S 系统并没有都采用第四版，S-PLUS 的第 5 版才采用了 S 语言第四版。

John M. Chambers (1998), “Programming with Data,” New York: Springer

- 商业版本为 S-PLUS, 1988 年发布, 现在为 Tibco Software 拥有。命运多舛, 多次易主。
- R 是一个自由软件, GPL 授权, 最初由新西兰 Auckland 大学的 Ross Ihaka 和 Robert Gentleman 于 1997 年发布, 实现了与 S 语言基本相同的功能和统计功能。现在由 R 核心团队开发, 但全世界的用户都可以贡献软件包。
- R 的网站: <http://www.r-project.org/>。

### R 的特点

- 自由软件, 免费;
- 完整的程序设计语言, 基于函数和对象, 可以自定义函数, 调入 C、C++、Fortran 编译的代码;
- 具有完善的数据类型, 如向量、矩阵、因子、数据集、一般对象等, 代码像伪代码一样简洁、可读。
- 强调交互式数据分析, 支持复杂算法描述, 图形功能强。
- 实现了经典的、现代的统计方法, 如参数和非参数假设检验、线性回归、广义线性回归、非线性回归、可加模型、树回归、混合模型、方差分析、判别、聚类、时间序列分析等。
- 统计科研工作者广泛使用 R 进行计算和发表算法。R 有数以千计的软件包 (截止 2015 年 11 月有七千多个)。

### R 语言和 R 软件的技术特点

- 函数编程 (functional programming)。R 语言虽然不是严格的 functional programming 语言, 但可以遵照其原则编程, 得到可验证的可靠程序。
- 支持对象类和类方法。基于对象的程序设计。
- 数据框是基本的观测数据类型, 类似于数据库的表。
- 开源软件 (Open source software)。可深入探查, 开发者和用户交互。
- 算法与接口。
- 主要数值算法采用已广泛测试和采纳的算法实现, 如排序、随机数生成、线性代数 (LAPACK 软件包)。



### 参考书

- R.L. Kabacoff(2012)《R 语言实战》，人民邮电出版社。
- R 网站上的初学者手册“An Introduction to R”和其它技术手册。
- John M. Chambers(2008), “Software for Data Analysis—Programming with R”, Springer.
- 薛毅、陈立萍（2007）《统计建模与 R 软件》，清华大学出版社。
- 汤银才（2008），《R 语言与统计分析》，高等教育出版社。
- 李东风（2006）《统计软件教程》，人民邮电出版社。

## 1.2 R 的下载与安装

### R 的下载

- 以 MS Windows 操作系统为例。
- R 的主网站在: <https://www.r-project.org/>。
- 下载从 CRAN 的镜像, 如 <http://mirror.bjtu.edu.cn/cran/>。选“Download R for Windows—base—Download R 3.2.2 for windows”链接进行下载。
- 在“Download R for Windows”链接的页面, 除了 base 为 R 的安装程序, 还有 contrib 为 R 附加的扩展软件包下载链接（一般不需要从这里下载）, 以及 Rtools 链接, 是在 R 中调用 C、C++ 和 Fortran 程序代码时需要用的编译工具。

### 安装

- 按提示安装, 建议选择 32 位版本。
- 安装后获得一个桌面快捷方式, 如“R i386 3.2.2”(这是 32 位版本)。
- 重要步骤: 在 C 盘或 D 盘建立一个文件夹(也叫做子目录), 如 C:\work。把 R 的快捷方式拷贝入此文件夹, 在 Windows 资源管理器中, 右键单击此快捷方式, 在弹出菜单中选“属性”, 把“快捷方式”页面的“起始位置”的内容变成空白。
- 启动在 work 文件夹中的 R 快捷方式, 出现命令行界面。
- R 主要依靠命令行执行功能。

### 辅助软件

- R 可以把一段程序写在一个以.r 为扩展名的文本文件中，如 “date.r”，称为一个**源程序文件**，然后在 R 命令行用

```
source('date.r')
```

运行源程序。

- 这样的文件可以用记事本生成和编辑。
- 建议使用 notepad++ 软件，为 MS Windows 下记事本程序的增强型软件。安装后，在 MS Windows 资源管理器中右键弹出菜单会有 “edit with notepadpp” 选项。
- R 还有其它的图形界面如 RStudio。可以以后自己尝试比较。

### R 扩展软件包的安装

- R 扩展软件包提供了特殊功能。
- 以安装 sos 包为例。sos 包用来搜索某些函数的帮助文档。
- 在 R 图形界面选菜单 “程序包—安装程序包”，在弹出的 “CRAN mirror” 选择窗口中选择一个中国的镜像如 “China (Beijing 2)”，然后再弹出的 “Packages” 选择窗口中选择要安装的扩展软件包名称，即可完成下载和安装。
- 还可以用如下程序制定镜像网站并安装指定的扩展包：

```
options(repos=c(  
  CRAN='http://mirror.bjtu.edu.cn/cran/')  
  print(.libPaths())  
  install.packages('sos', lib=.libPaths()[2])
```

这里指定了扩展包安装位置是 R 的主目录，不指定就可能会安装在用户的个人文档目录中。

### 练习

- 下载 R 安装程序，安装 R，建立 work 文件夹并在其中建立 R 的快捷方式。
- 下载安装 notepad++ 软件。
- 在 R 图形界面中下载安装 sos 扩展软件包。

## 1.3 R 的使用样例

### 命令行界面

- 命令行界面：输入一行命令，就在后面显示计算结果。
- 可以用向上和向下箭头访问历史命令；
- 可以从上面的命令中用鼠标拖选加亮后，用 Ctrl+C 复制后用 Ctrl+V 粘贴，或用 Ctrl+X 一步完成复制粘贴，粘贴的目标都是当前命令行。

#### 1.3.1 四则运算

##### 四则运算

- 四则运算如：

```
5 + (2.3 - 1.125)*3.2/1.1 + 1.23E3
```

结果为 1238.418。

- 这里 1.23E3 是科学记数法，表示  $1.23 \times 10^3$ 。
- 用星号表示乘法，用正斜杠/表示除法。
- 用 ^ 表示乘方运算，如

```
2^10
```

结果为 1024。

- 重要提示：关闭中文输入法，否则输入一些中文标点导致程序错误。

- 例：从 52 张扑克牌中任取 3 张，有多少种不同的组合可能？

- 解答：有

$$C_{52}^3 = \frac{52!}{3!(52-3)!} = \frac{52 \times 51 \times 50}{3 \times 2 \times 1}$$

种，在 R 中计算如：

```
52*51*50/(3*2)
```

- 结果为 22100。

## 练习

- 某人存入 10000 元 1 年期定期存款，年利率 3%，约定到期自动转存（包括利息）。
- 问：（1）10 年后本息共多少元？
- （2）需要存多少年这 10000 元才能增值到 20000 元？

### 1.3.2 数学函数

#### 数学函数——平方根、指数、对数

- 例：

```
sqrt(6.25)
exp(1)
log10(10000)
```

- `sqrt(6.25)` 表示  $\sqrt{6.25}$ ，结果为 2.5。
- `exp(1)` 表示  $e^1$ ，结果为  $e = 2.718282$ 。
- `log10(10000)` 表示  $\lg 10000$ ，结果为 4。`log` 为自然对数。

### 数学函数—取整

- 例:

```
round(1.1234, 2)
round(-1.9876, 2)
floor(1.1234)
floor(-1.1234)
ceiling(1.1234)
ceiling(-1.1234)
```

- `round(1.1234, 2)` 表示把 1.1234 四舍五入到两位小数。
- `floor(1.1234)` 表示把 1.1234 向下取整, 结果为 1。
- `ceiling(1.1234)` 表示把 1.1234 向上取整, 结果为 2。

### 数学函数—三角函数

- 例:

```
pi
sin(pi/6)
cos(pi/6)
sqrt(3)/2
tan(pi/6)
```

- `pi` 表示圆周率  $\pi$ 。
- `sin` 正弦, `cos` 余弦, `tan` 正切, 自变量以弧度为单位。
- `pi/6` 是  $30^\circ$ 。

### 数学函数—反三角函数

- 例:

```
pi/6
asin(0.5)
acos(sqrt(3)/2)
atan(sqrt(3)/3)
```

- `asin` 反正弦, `acos` 反余弦, `atan` 反正切, 结果以弧度为单位。

### 分布函数和分位数函数

- 例:

```
dnorm(1.98)
pnorm(1.98)
qnorm(0.975)
```

- `dnorm(x)` 表示标准正态分布密度  $\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$ 。
- `pnorm(x)` 表示标准正态分布函数  $\Phi(x) = \int_{-\infty}^x \phi(t) dt$ 。
- `qnorm(y)` 表示标准正态分布分位数函数  $\Phi^{-1}(y)$ 。
- 还有其它许多分布的密度函数、分布函数和分位数函数。

- 例:

```
qt(1 - 0.05/2, 10)
```

- `qt(y)` 表示  $t(10)$  分布的分位数函数。比如, 为了求自由度为 10 的  $t$  检验的双侧临界值, 就可以用上面程序。

### 1.3.3 输出

### 简单输出

- 命令行的计算结果直接显示。
- 在程序文件运行时，需要用 `print` 函数显示一个表达式的结果：

```
print(sin(pi/2))
```

- `cat` 函数显示多项内容，包括数值和文本，文本包在两个单撇号或两个双撇号中：

```
cat('sin(pi/2)=', sin(pi/2), '\n')
```

- `cat` 函数最后一项一般是 `'\n'`，表示换行。忘记此项将不换行。
- 注意：要避免打开中文输入法导致误使用中文标点。

### `sink` 运行记录

- R 使用经常是在命令行逐行输入命令（程序），结果紧接着显示在命令后面。
- 如何保存这些命令和显示结果？
- 用 `sink` 函数打开一个文本文件开始记录。
- 结束记录时用空的 `sink()` 即可关闭文件不再记录。

### 练习

- 用 `cat` 语句显示

```
log10(2)=*** log10(5)=***
```

其中 `***` 应该代以实际函数值。

- 用 `sink` 函数开始把运行过程记录到文件 “log001.txt” 中，在命令行试验几个命令，然后关闭运行记录，查看生成的 “log001.txt” 的内容。

### 1.3.4 向量计算与变量赋值

#### R 向量例子

- R 语言以向量为最小单位。用 `<-` 赋值。

```
x1 <- 1:10  
x1
```

- 变量: 在程序语言中, 变量用来保存输入的值或计算的结果。
- 变量可以存放各种不同类型的值, 如单个数值、多个数值 (称为向量)、单个字符串、多个字符串 (称为字符型向量), 等等。
- 单个数值称为**标量**。
- 向量可以和一个标量作四则运算, 结果是每个元素都和这个标量作四则运算, 如:

```
x1 + 200  
2*x1  
2520/x1
```

- 两个等长的向量可以进行四则运算, 相当于对应元素进行四则运算, 如

```
x2 <- x1 * 3  
x2  
x2 - x1
```

#### 工作空间介绍

- 在命令行中定义的变量, 在退出 R 时, 会提问是否保存工作空间, 一般选择保存。
- 再次启动 R 后, 能够看到以前定义的所有变量的值。



### 练习

- 某人存入 10000 元 1 年期定期存款，年利率 3%，约定到期自动转存（包括利息）。列出 1、2、……、10 年后的本息金额。
- 显示 2 的 1,2,..., 20 次方。
- 定义 x1 为 1 到 10 的向量，定义 x2 为 x1 的 3 倍，然后退出 R，再次启动 R，查看 x1 和 x2 的值。

### 1.3.5 绘图示例

#### 函数曲线示例

- $x^2$  函数曲线图：

```
curve(x^2, -2, 2)
```

第二、第三自变量是绘图区间。

- $\sin(x)$  函数曲线图：

```
curve(sin(x), 0, 2*pi)
```

- 添加参考线：

```
abline(h=0)
```

#### 条形图示例

- 绘制男生、女生人数的条形图：

```
barplot(c(' 男生 '=10, ' 女生 '=7),  
        main=' 男女生人数 ')
```

- 可以人为定制颜色。
- 个数一般是从数据中统计得到的。

### 散点图示例

- 散点图:

```
plot(1:10, 2*(1:10))
```

- 第一个自变量是各个点的横坐标值，第二个自变量是对应的纵坐标值。

### R 软件自带的图形示例

- 示例演示:

```
demo("graphics")  
demo("image")
```

### 练习

- 画  $\exp(x)$  在  $(-2, 2)$  区间的函数图形。
- 画  $\ln(x)$  在  $(0.01, 10)$  区间的函数图形。

### 1.3.6 汇总统计示例

#### 表格数据

- 统计用的输入数据典型样式是 Excel 表那样的表格数据。
- 表格数据特点：每一列应该是相同的类型（或者都是数值，或者都是文字，或者都是日期），每一列应该有一个名字。
- 这样的表格数据，一般可以保存为.csv 格式：数据项之间用逗号分开，文件本身是文本型的，可以用普通记事本程序查看和编辑。
- Excel 表可以用“另存为”命令保存为.csv 格式。

### 读入表格数据

- 例: “patients.csv” 是这样一个 csv 格式表格数据文件, 可以用 Excel 打开, 也可以用记事本程序或 notepad++ 打开。
- 读入到 R 中:

```
pa.tab <- read.csv("patients.csv", header=TRUE)
print(pa.tab)
head(pa.tab)
```

这里 `header=TRUE` 指明第一行作为变量名行。

- 读入的变量 `pa.tab` 称为一个数据框 (data.frame)。
- `head` 函数返回数据框或向量的前几项。比较大的表最好不要显示整个表, 会使得前面的运行过程难以查看。

### 练习

- 用 MS Excel 查看 “class.csv” 的内容 (双击即可)。
- 用记事本程序或 notepad++ 软件查看 “class.csv” 的内容。
- 读入 “class.csv” 到 R 数据框 `cl` 中, 查看 `cl` 内容。

### R 基本统计功能演示

- 性别是一个分类变量。用 `table` 函数计算每个不同值的个数:

```
table(pa.tab[, '性别'])
```

- “病人属于” 的取值情况:

```
table(pa.tab[, '病人属于'])
```

- 性别和 “病人属于” 的交叉分类计数表格计算:

```
table(pa.tab[, '性别'], pa.tab[, '病人属于'])
```

- 数值型变量可以计算各种不同的统计量。`summary` 给出最小值、最大值、中位数、四分之一分位数、四分之三分位数和平均值。如

```
x3 <- 1:100
summary(x3)
```

- 中位数是从小到大排序后排在中间的值。四分之一和四分之三分位数类似。
- 统计函数: 以一个数值型向量为自变量, 包括 `sum`(求和), `mean`(平均值), `var`(样本方差), `sd`(样本标准差), `min`(最小值), `max`(最大值), `range`(最小值和最大值) 等。如

```
min(x3)
sd(x3)
```

## 练习

- 用如下程序定义一个变量 `x4`:

```
x4 <- c(62, 80, 54, 91, 75)
```

求 `x4` 的平均值和最小值、最大值。

### 1.3.7 运行一个源程序文件

#### 源程序文件

- 可以把 R 程序保存在一个扩展名为 “.r” 的源程序文件（也叫做脚本文件）中，然后在 R 中用 `source("文件名")` 命令运行。
- 如文件 `date.r` 中包含如下内容：

```
## 把如 FEB07 这样的日期转换成 R 的日期型数据。
## 00-19 为 2000+, 20-99 为 1900+。
monyy2date <- function(x){
  year.cutoff <- 20
```

```
mon.map <- c('JAN'='01', 'FEB'='02',
             'MAR'='03', 'APR'='04',
             'MAY'='05', 'JUN'='06',
             'JUL'='07', 'AUG'='08',
             'SEP'='09', 'OCT'='10',
             'NOV'='11', 'DEC'='12'
            )

mon <- mon.map[toupper(substr(x, 1, 3))]
year2 <- as.numeric(substr(x, 4, 5))
sele <- year2 <= year.cutoff
year2[sele] <- year2[sele] + 2000
year2[!sele] <- year2[!sele] + 1900
date <- as.POSIXct(paste(year2, '-', mon, '-01', sep=''))
date
}
```

- 如下 `source` 函数可以运行这个程序文件：

```
source("date.r")
```

## 练习

- 运行 `date.r` 源程序文件。用 `monyy2date` 转换 FEB07 和 OCT66 为 R 日期格式。

## 1.4 帮助

### 在线帮助

- 用帮助菜单中“html 帮助”到互联网浏览器中查看帮助文档。“Search engine and keywords”项下面有分类的帮助。有软件包列表。
- 在命令行，用问号后面跟随函数名查询某函数的帮助。
- 安装 `sos` 附加包 (package)，用 `findfn(函数名)` 查询。

**example 函数**

- 为了查询某个函数的用法，在命令行用问号后面紧随函数名的命令查询；用 `example(函数名)` 的格式可以运行此函数的样例，如：

```
> example(mean)

mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.10))
[1] 8.75 5.50
```

## 第二章 R 编程基础

### R 语言基础编程

- R 的各种数据类型：数值型、字符型、逻辑型、日期，向量、矩阵、数据框、列表等。
- R 工作空间管理。
- R 程序控制结构: if else, for, repeat, break.
- R 函数：函数调用规则，自定义函数。

### 2.1 数据类型和数据结构

#### R 语言数据类型

- 数值；
- 逻辑型：TRUE, FALSE;
- 文本（字符串）；
- 支持缺失值；
- 支持复数等。

#### R 语言数据结构

- 向量；
- 矩阵和数据框；
- 多维数组；
- 列表；
- 对象；
- 带名字的元素、行名、列名等。

### 2.1.1 常量与变量

#### 常量

- 数值型：包括整型、单精度、双精度等，一般不需要区分。写法如 123, 123.45, -123.45, -0.012,  $1.23E2$ ,  $-1.2E-2$  等。
- 字符型。用两个单撇号或两个双撇号包围，如 'Li Ming' 或 "Li Ming"。
- 字符型支持中文，如 '李明' 或 "李明"。国内的中文编码主要有 GB 编码和 UTF8 编码，有时会遇到编码错误造成乱码的问题，MS Windows 下 R 程序一般用 GB 编码。
- 逻辑性：TRUE 和 FALSE。
- 缺失值：用 NA 表示。字符型的空白值不会自动识别为缺失值，需要自己规定。
- 复数：如  $2.2 + 3.5i$ ,  $1i$  等。

#### 变量

- 程序语言中的变量用来保存输入的值或者计算得到的值。
- 变量都有变量名，R 变量名必须以字母、数字和句点组成，变量名的第一个字符不能取为数字。R 现在也允许变量名中有下划线。
- 变量名大小写区分：y 和 Y 是两个不同的变量名。
- 变量名举例：x, x1, X, disease.tab, diseaseData。

#### 定义变量

- 用<-把右边的常量或者计算结果保存到左边的变量名对应存储空间中。
- 例：

```
x5 <- 6.25
x6 <- sqrt(x1)
```

### 2.1.2 数值型向量



### c() 函数

- 把若干个值保存在一起，称为一个**向量**。
- 用 `c()` 函数把多个元素或向量组合成一个向量。如

```
marks <- c(10, 6, 4, 7, 8)
x <- c(1:3, 10:13)
x1 <- c(1, 2)
x2 <- c(3, 4)
x <- c(x1, x2)
x
```

- 10:13 这样的写法是从 10 到 13 的整数组成的向量。
- 显示向量时在左边方括号中提示显示行的第一个元素的下标:

```
> 1234501:1234520
[1] 1234501 1234502 1234503 1234504 1234505 1234506
[7] 1234507 1234508 1234509 1234510 1234511 1234512
[13] 1234513 1234514 1234515 1234516 1234517 1234518
[19] 1234519 1234520
```

比如，第 13 号元素 1234513，前面有一个 “[13]”。

- 例子中，程序行开头的大于号表示这是在命令行中运行的，并不是用户自己输入的大于号。讲义中有时省略这个提示符。
- `length(x)` 可以求 `x` 的长度。
- 长度为零的向量：`numeric(0)`。
- 长度为 10 的数值型向量，元素都初始化为零：`numeric(10)`。

### 向量与标量运算

- 单个数值称为**标量**。
- 向量与标量的运算为每个元素与标量的运算。
- 四则运算：`+` `-` `*` `/` `^`(加、减、乘、除、乘方)。

- 如:

```
> x <- c(1, 10)
> x + 2
[1] 3 12
> x - 2
[1] -1 8
> x * 2
[1] 2 20
> x / 2
[1] 0.5 5.0
```

```
> x ^ 2
[1] 1 100
> 2 / x
[1] 2.0 0.2
> 2 ^ x
[1] 2 1024
```

### 有缺失值的四则运算

- 有缺失值的运算: 缺失元素参加的运算相应结果元素仍缺失。
- 如:

```
> c(1, NA, 3) + 10
[1] 11 NA 13
```

### 等长向量运算

- 为对应元素两两运算。
- 如:

```
> x1 <- c(1, 10)
> x2 <- c(4, 2)
> x1 + x2
[1] 5 12
> x1 - x2
[1] -3 8
> x1 * x2
[1] 4 20
> x1 / x2
[1] 0.25 5.00
```

### 不等长向量的运算

- 如果长度为倍数关系，每次从头重复利用短的一个。如

```
> x1 <- c(10, 20)
> x2 <- c(1, 3, 5, 7)
> x1 + x2
[1] 11 23 15 27
> x1 * x2
[1] 10 60 50 140
```

- 如果长度不是倍数关系，会给出警告信息。如

```
> c(1,2) + c(1,2,3)
[1] 2 4 4
警告信息:
In c(1, 2) + c(1, 2, 3) :
  长的对象长度不是短的对象长度的整倍数
```

### 向量函数

- 一元函数以向量为自变量，对每个元素计算。函数有 `sqrt`, `log10`, `log`, `exp`, `sin`, `cos`, `tan` 等。如：

```
> sqrt(c(1, 4, 6.25))  
[1] 1.0 2.0 2.5
```

### 排序

- `sort(x)` 返回排序结果；`rev(x)` 返回把各元素排列次序反转后的结果；`order(x)` 返回排序用的下标。如

```
> x <- c(33, 55, 11)  
> sort(x)  
[1] 11 33 55  
> rev(sort(x))  
[1] 55 33 11  
> order(x)  
[1] 3 1 2  
> x[order(x)]  
[1] 11 33 55
```

- 例子中，`order(x)` 结果中 3 是 `x` 的最小元素 11 所在的位置下标，1 是 `x` 的第二小元素 33 所在的位置下标，2 是 `x` 的最大元素 55 所在的位置下标。

### 统计函数

- 统计函数: `sum`, `mean`, `var`, `sd`, `min`, `max`, `range` 等。
- `prod` 求所有元素的成绩。
- `cumsum` 和 `cumprod` 计算累加和累乘积。

```
> cumsum(1:5)
[1]  1  3  6 10 15
> cumprod(1:5)
[1]  1  2  6 24 120
```

### 练习

- 显示 1 到 100 的整数的平方根和立方根（提示：立方根就是三分之一次方）。
- 设有 10 个人的小测验成绩为：

77 60 91 73 85 82 35 100 66 75

- (1) 把这 10 个成绩存入变量 x;
- (2) 从小到大排序;
- (3) 计算 `order(x)`，解释 `order(x)` 结果中第 3 项代表的意义。
- (4) 计算这些成绩的平均值、标准差、最小值、最大值、中位数。

### seq 函数

- `seq` 函数是冒号运算符的推广。
- 如: `seq(5)` 等同于 `1:5`。
- `seq(2,5)` 等同于 `2:5`。
- `seq(11, 15, by=2)` 产生 11,13,15。
- `seq(0, 2*pi, length=100)` 产生从 0 到  $2\pi$  的等间隔序列，序列长度指定为 100。
- S 函数可以带自变量名调用。
- `seq(to=5, from=2)` 等同于 `2:5`。

### rep 函数

- 产生一个初值为零的长度为 n 的向量: `x <- rep(0, n)`。
- `rep(c(1,3), 2)` 相当于 `c(1,3,1,3)`。
- `rep(c(1,3), c(2,4))` 相当于 `c(1,1,3,3,3,3)`。
- `rep(c(1,3), each=2)` 相当于 `c(1,1,3,3)`。

### 2.1.3 逻辑型向量

#### 比较和逻辑值

- 逻辑值: TRUE, FALSE, 缺失时为 NA。一般产生自比较, 如:

```
> sele <- (log10(15) < 2); print(sele)
[1] TRUE
```

- 向量比较结果为逻辑型向量。如:

```
> c(1, 3, 5) > 2
[1] FALSE TRUE TRUE
> (1:4) >= (4:1)
[1] FALSE FALSE TRUE TRUE
```

- 与 NA 比较产生 NA, 如

```
> c(1, NA, 3) > 2
[1] FALSE NA TRUE
```

- 为了判断每个元素是否 NA, 用 `is.na` 函数, 如
- 比较运算符为 `<`, `<=`, `>`, `>=`, `==`, `!=`。

## 逻辑运算

- 为了表达如“ $x > 0$  而且  $x < 1$ ”；“ $x \leq 0$  或者  $x \geq 1$ ”等复合的比较，使用逻辑运算把两个比较连接起来。
- 逻辑运算符为`&`，`|`，`!`，分别表示“同时成立”、“两者至少其一成立”、“条件的反面”。
- 比如，设`age<=3`表示婴儿，`sex=='女'`表示女性，则 `age<=3 & sex=='女'` 表示女婴；`age<=3 | sex=='女'` 表示婴儿或妇女；`!(age<=3 | sex=='女')` 表示既非婴儿也非妇女。
- `&&`，`||` 称为短路的标量逻辑与和逻辑或，仅用在 `if`, `while` 语句中，只要第一个比较已经决定最终结果就不计算第二个比较。
- `all(cond)` 测试 `cond` 的所有元素为真；`any(cond)` 测试 `cond` 至少一个元素为真。
- 如

```
> is.na(c(1, NA, 3) > 2)
[1] FALSE TRUE FALSE
> any(is.na(c(1, NA, 3) > 2))
[1] TRUE
```

### 2.1.4 字符型向量

#### 字符型向量与字符型函数

- 字符型向量是元素为字符串的向量。
- `paste` 函数：连接两个字符型向量，元素一一对应连接。缺省用空格连接。如 `paste(c("ab", "cd"), c("ef", "gh"))` 相当于 `c("ab ef", "cd gh")`。
- 可以一对多连接，可以在连接时把数值型转换为字符型，如 `paste("x", 1:3)` 相当于 `c("x 1", "x 2", "x 3")`。

- 用 `sep=` 指定分隔符, 如 `paste("x", 1:3, sep="")` 相当于 `c("x1", "x2", "x3")`。
- 使用 `collapse=` 参数连接字符型向量的各个元素为一个字符串。如 `paste(c("a", "b"), collapse="")` 相等于是 `"ab"`。

### 转换大小写

- `toupper` 转为大写, `tolower` 转为小写。
- 如 `toupper('aB cd')` 结果为 `"AB CD"`, `tolower(c('aB', 'cd'))` 结果为 `"ab" "cd"`。
- 用于不区分大小写的比较, 比如, 不论 `x` 的值是 `'JAN'`, `'Jan'` 还是 `'jan'`, `toupper(x)=='JAN'` 的结果都为 `TRUE`。

### 取子串

- `substr(x, start, stop)` 从字符串 `x` 中取处从第 `start` 个到第 `stop` 个的子串, 如:

```
> substr('JAN07', 1, 3)
[1] "JAN"
```

- 如果 `x` 是一个字符型向量, 将对每个元素取子串。如

```
> substr(c('JAN07', 'MAR66'), 1, 3)
[1] "JAN" "MAR"
```

### 类型转换

- 用 `as.numeric` 把内容是数字的字符型值转换为数值, 如:

```
> substr('JAN07', 4, 5)
[1] "07"
> substr('JAN07', 4, 5) + 2000
Error in substr("JAN07", 4, 5) + 2000 :
  二进制运算符中有非数值参数
```



```
> as.numeric(substr('JAN07', 4, 5)) + 2000
[1] 2007
> as.numeric(substr(c('JAN07', 'MAR66'), 4, 5))
[1] 7 66
```

- 可以转换一个向量的每个元素为数值型。
- `as.character` 把数值型转换为字符型，如

```
> as.character((1:5)*5)
[1] "5" "10" "15" "20" "25"
```

- 如果本来已经是字符型则结果保持原样。

### **strsplit**

- 经常会有输入表中的一栏包含多个值的情况，比如，某个表的一列中是三次小测验的成绩，三个成绩用逗号分开，如

```
x <- '10, 8, 7'
```

- 为了把这样逗号分开的内容拆分出来，用 `strsplit` 函数。如

```
> strsplit(x, ',', fixed=TRUE)[[1]]
[1] "10" " 8" " 7"
> as.numeric(strsplit(x, ',')[[1]])
[1] 10 8 7
```

可以把三个成绩转化为有 3 个元素的数值型向量。

- `strsplit` 的第二个元素用来指定分隔符，可以指定一个其它的分隔符，如

```
> x2 <- '10 8 7'
> as.numeric(strsplit(x2, ' ', fixed=TRUE)[[1]])
[1] 10 8 7
```

- 但是注意空格作分隔符，多个连续空格会被认作多个分隔符。
- 一到多个空格作分隔符，可以用如

```
> x2 <- '10 8 7'
> as.numeric(strsplit(x2, '[ ]+')[[1]])
[1] 10 8 7
```

这里分隔符用了正则表达式的方法。

### 字符型数据获取

- 利用字符串处理的方法可以从网页文件或不规则的 Excel 文件读取数据。
- `grep`, `grep1` 函数从字符串中查询某个模式，模式用 perl 格式的**正则表达式** (regular expression) 定义。
- `sub`, `gsub` 替换某模式。
- 正则表达式功能强大但也不容易掌握。
- 详细介绍略。

### 字符串替换功能

- 用 `gsub` 可以替换字符串中的子串。比如:

```
> x <- '1, 3; 5'
```

这样的数值串分隔符即有逗号由有分号，可以把分号都换成逗号, 如:

```
> gsub(';', ' ', x, fixed=TRUE)
[1] "1, 3, 5"
> as.numeric(strsplit(
  gsub(';', ' ', x, fixed=TRUE), ',')[[1]])
[1] 1 3 5
```

- 用这样的方法，可以把字符串中非标准的内容，如中文的标点，重复的空格等替换成一致的表示。

## 练习

- 用 paste 函数生成如下的一系列文件名:

res1.txt, res2.txt, ..., res100.txt。

- 取出如下变量 x 中的年份，转换为数值型。

```
x <- c('1995-03-11', '2008-10-31', '2014-12-15')
```

## 2.1.5 向量的下标

## 向量下标——整数下标

- 设 `x <- c(1, 4, 6.25)`。
- `x[2]` 取出第二个元素；`x[2] <- 99` 修改第二个元素。
- `x[c(1,3)]` 取出第 1、3 号元素；`x[c(1,3)] <- c(11, 13)` 修改第 1、3 号元素。
- 下标可重复，如 `x[c(1,3,1)]` 为 1, 6.25, 1。

## 向量下标——负整数下标

- 负下标表示“扣除”，如 `x[-c(1,3)]` 为第二个元素 4。

## 逻辑下标

- 下标可以是一个条件，如 `x[x>3]` 取出 4, 6.25。
- 例：示性函数。设输入向量 x, 求每个元素的示性函数值 (元素非负时取 1, 否则取 0)。

```
y <- numeric(length(x))
y[x >= 0] <- 1
y[x < 0] <- 0 # 此语句多余
```

- 逻辑下标功能在数据框数据筛选时很有用。

### `which()` 和 `which.min()`

- 函数 `which()` 可以用来找到满足条件的下标，比如：

```
> x <- c(3, 4, 3, 5, 7, 5, 9)
> which(x > 5)
[1] 5 7
> seq(along=x)[x > 5]
[1] 5 7
```

- 用 `which.min()`, `which.max` 求最小值下标和最大值下标。最值不唯一时只取第一个。如

```
> which.min(x)
[1] 1
> which.max(x)
[1] 7
```

### 集合运算

- 可以把向量 `x` 看成一个集合，但是其中的元素允许有重复。
- 用 `unique(x)` 可以获得 `x` 的所有不同值。如

```
> unique(c(1, 5, 2, 5))
[1] 1 5 2
```

- 用 `a %in% x` 判断 `a` 是否属于向量 `x`，如

```
> 5 %in% c(1,5,2)
[1] TRUE
```

- 当 `a` 是向量时，对 `a` 的每个元素判断，如

```
> c(5, 7) %in% c(1, 5, 2)
[1] TRUE FALSE
```

- 用 `intersect(x,y)` 求交集，不含重复元素，如：

```
> intersect(c(5, 7), c(1, 5, 2, 5))  
[1] 5
```

- 用 `union(x,y)` 求并集，不含重复元素，如：

```
> union(c(5, 7), c(1, 5, 2, 5))  
[1] 5 7 1 2
```

- 用 `setdiff(x,y)` 求差集，不含重复元素，如：

```
> setdiff(c(5, 7), c(1, 5, 2, 5))  
[1] 7
```

- `setequal(x,y)` 判断两个集合是否相等，不受次序与重复元素的影响，如：

```
> setequal(c(1,5,2), c(2,5,1))  
[1] TRUE  
> setequal(c(1,5,2), c(2,5,1,5))  
[1] TRUE
```

## 元素名

- 向量可以为每个元素命名。如

```
ages <- c(" 李明"=30, " 张聪"=25, " 刘颖"=28)
```

或

```
ages <- c(30, 25, 28)
names(ages) <- c(" 李明", " 张聪", " 刘颖")
```

- 这时可以用 `ages[" 张聪"]`, `ages[c(" 李明", " 刘颖")]` 这样的方法访问。
- 这样建立了字符串到数值的映射表。
- 在矩阵和数据框中尤其有用。

### 整个数组

- 用 `x[] <- 0` 把 `x` 的所有元素赋 0，但长度不变。
- 这与 `x <- 0` 不同，后者把 `x` 变成标量 0。

### 用 R 向量作映射

- R 在使用整数作为向量下标时，允许使用重复下标，这样可以把数组 `x` 看成一个  $1:n$  的整数到  $x[1], x[2], \dots, x[n]$  的一个映射表。
- 比如，某商店有三种礼品，编号为 1,2,3，价格为 68, 88 和 168。令

```
> price.map <- c(68, 88, 168)
```

- 设某个收银员在一天内分别售出礼品编号为 3,2,1,1,2,2,3，可以用如下的映射方式获得售出的这些礼品对应的价格：

```
> items <- c(3,2,1,1,2,2,3)
> y <- price.map[items]
> print(y)
[1] 168 88 68 68 88 88 168
```

- R 向量可以用字符串作下标，字符串下标也允许重复，所以可以把带有元素名的 R 向量看成是元素名到元素值的映射表。
- 比如，设 `sex` 为 10 个学生的性别（男、女）：

```
> set.seed(1)
> sex <- sample(c('男', '女'), size=10,
+             replace=TRUE)
> print(sex)
[1] "男" "男" "女" "女" "男" "女" "女"
[8] "女" "女" "男"
```

- 希望把每个学生按照性别分别对应到蓝色和红色。

- 首先建立一个 R 向量当作映射：

```
> sex.color <- c('男'='blue', '女'='red')
```

- 用 R 向量 sex.color 当作映射，可以获得每个学生需要的颜色：

```
> cols <- sex.color[sex]
> print(cols)
      男      男      女      女      男
"blue" "blue" "red"  "red" "blue"
      女      女      女      女      男
"red"  "red"  "red"  "red" "blue"
```

- 这样的映射结果中带有不必要的元素名，用 `unname()` 函数可以去掉元素名，如

```
> print(unname(cols))
[1] "blue" "blue" "red"  "red"  "blue"
[6] "red"  "red"  "red"  "red"  "blue"
```

## 练习

- 把 “class.csv” 读入为 R 数据框 d.class, 取出其中的 name 和 age 列到变量 name 和 age 中, 用如下程序:

```
d.class <- read.csv('class.csv', header=TRUE)
name <- d.class[, 'name']
age <- d.class[, 'age']
```

- (1) 求出 age 中第 3, 5, 7 号的值;
- (2) 用变量 age, 求出达到 15 岁及以上的那些值;
- (3) 用变量 name 和 age, 求出 Mary 与 James 的年龄。
- (4) 求 age 中除 Mary 与 James 这两人之外的那些人的年龄值, 保存到变量 age1 中。
- (5) 函数 `order(x)` 给出了  $1:n$  ( $n$  是  $x$  的元素个数) 的排列, 可以看成  $1:n$  到  $x[1], x[2], \dots, x[n]$  的一个映射。求这个映射的逆映射, 并解释其含义。

### 2.1.6 日期和时间的处理

#### R 日期时间

- R 中用一种叫做 POSIXct 和 POSIXlt 的特殊数据类型保存日期和时间, 可以仅包含日期部分, 也可以同时有日期和时间。
- 技术上, POSIXct 把日期时间保存为从 1970 年 1 月 1 日零时到该日期时间的时间间隔秒数, 所以数据框中需要保存日期时用 POSIXct 比较合适, 需要显示时再转换成字符串形式; POSIXlt 把日期时间保存为一个包含年、月、日、星期、时、分、秒等成分列表, 所以求这些成分可以从 POSIXlt 格式日期的列表变量中获得。

#### 转换为日期

- 用 `as.POSIXct` 把年月日格式的日期转换为 R 的标准日期, 没有时间部分就认为时间在午夜。如:



```
> as.POSIXct(c('1998-03-16'))  
[1] "1998-03-16 CST"  
> as.POSIXct(c('1998/03/16'))  
[1] "1998-03-16 CST"
```

但是转换时年月日之间的分隔符不能既有减号又有斜杠。

### 转换为日期时间

- 待转换的日期时间字符串，可以是年月日之后隔一个空格以时:分:秒格式带有时间。如

```
> as.POSIXct(c('1998-03-16 13:15:45'))  
[1] "1998-03-16 13:15:45 CST"
```

- 可以同时转换多项日期时间，如

```
> as.POSIXct(c('1998-03-16 13:15:45',  
+              '2015-11-22 9:45:3'))  
[1] "1998-03-16 13:15:45 CST"  
[2] "2015-11-22 09:45:03 CST"
```

### 日期计算

- 所有的比较运算都适用于日期类型。
- 可以给一个日期加减一定的秒数，如

```
> as.POSIXct(c('1998-03-16 13:15:45')) - 30  
[1] "1998-03-16 13:15:15 CST"  
> as.POSIXct(c('1998-03-16 13:15:45')) + 10  
[1] "1998-03-16 13:15:55 CST"
```

但是日期不能再加另一个日期。

- 给一个日期加减一定天数，可以通过加减秒数实现，如：

```
> as.POSIXct(c('1998-03-16 13:15:45')) + 3600*24*2
[1] "1998-03-18 13:15:45 CST"
```

例子结果推后了两天。

### 日期差

- 用 `difftime(time1, time2, units='days')` 计算 `time1` 减去 `time2` 的天数，如：

```
> x <- as.POSIXct(c('1998-03-16', '2015-11-22'))
> c(difftime(x[2], x[1], units='days'))
[1] 6460
```

函数结果用 `c()` 包裹以转换为数值，否则会带有单位。

- 如果 `time1` 和 `time2` 中含有时间部分，则间隔天数也会带有小数部分。  
如

```
> x <- as.POSIXct(c('1998-03-16 13:15:45',
+                    '2015-11-22 9:45:3'))
> c(difftime(x[2], x[1], units='days'))
[1] 6459.854
```

- `difftime` 中 `units` 选项还可以取为 `'secs'`, `'mins'`, `'hours'` 等。

### 日期显示格式

- 用 `as.character` 函数把日期型数据转换为字符型，如

```
> x <- as.POSIXct(c('1998-03-16', '2015-11-22'))
> as.character(x)
[1] "1998-03-16" "2015-11-22"
```

- 在 `as.character` 中可以用 `format` 参数指定显示格式，如：

```
> as.character(x, format='%m/%d/%Y')
[1] "03/16/1998" "11/22/2015"
```

这里“%Y”代表四位的公元年号，“%m”代表两位的月份数字，“%d”代表两位的月内日期号。

- 又如：

```
> x <- as.POSIXct(c('1998-03-16', '2015-11-22'))
> Sys.setlocale('LC_TIME', 'C')
[1] "C"
> as.character(x, format='%b%y')
[1] "Mar98" "Nov15"
```

这里用 `Sys.setlocale` 设置了本地化的时间设置为 C 语言库设置，格式中“%b”是本地化月份简写，“%y”是两位数年份。

- 应该避免使用两位数年份，因为两位数年份有时难以区分前面应该是 20 还是 19。
- 包含时间的转换如：

```
> x <- as.POSIXct('1998-03-16 13:15:45')
> as.character(x)
[1] "1998-03-16 13:15:45"
> as.character(x, format='%H:%M:%S')
[1] "13:15:45"
```

这里“%H”代表小时（按 24 小时制），“%M”代表两位的分钟数字，“%S”代表两位的秒数。

### 读入非标准格式日期

- 在 `as.POSIXct` 函数中用 `format` 参数指定一个日期格式。如

```
> as.POSIXct('3/13/15', format='%m/%d/%y')  
[1] "2015-03-13 CST"
```

- 如果日期仅有年和月，必须添加日（添加 01 为日即可）才能读入。比如用 '1991-12' 表示 1991 年 12 月，则如下程序将其读入为 '1991-12-01'：

```
> as.POSIXct(paste('1991-12', '-01', sep=''),  
+           format='%Y-%m-%d')  
[1] "1991-12-01 CST"
```

- 又如：

```
> Sys.setlocale('LC_TIME', 'C')  
[1] "C"  
> as.POSIXct(paste('01', 'DEC91', sep=''),  
+           format='%d%b%y')  
[1] "1991-12-01 CST"
```

把 'DEC91' 转换成了 '1991-12-01'。

### 取出日期时间的组成值

- 把一个 R 日期时间值用 `as.POSIXlt` 转换为 `POSIXlt` 类型，就可以用列表元素方法取出其组成的年、月、日、时、分、秒等数值。
- 如

```
> x <- as.POSIXct('1998-03-16 13:15:45')  
> y <- as.POSIXlt(x)  
> cat(1900+y$year, y$mon+1, y$mday,  
+     y$hour, y$min, y$sec, '\n')  
1998 3 16 13 15 45
```

- 注意 year 要加 1900, mon 要加 1。
- 对多个日期, 取出年号如

```
> x <- as.POSIXct(c('1998-03-16', '2015-11-22'))
> as.POSIXlt(x)$year + 1900
[1] 1998 2015
```

这是因为 `as.POSIXlt` 把多个日期转换成了一个列表, 列表中元素 `year` 是所有日期减去 1900 的值组成的向量。

### 时区问题

- 以上的例子没有考虑时区、夏时制等问题。
- 如果明确地知道时区, 在 `as.POSIXct()` 和 `as.POSIXlt()` 中可以加选项 `tz=` 字符串。
- 选项 `tz` 的缺省值为空字符串, 这一般对应于当前操作系统的默认时区。
- 但是, 有些操作系统和 R 版本不能使用默认值, 这时可以为 `tz` 指定时区, 比如北京时间可指定为 `tz='Etc/GMT+8'`。如:

```
as.POSIXct('1940-01-01', tz='Etc/GMT+8')
```

### 练习

- 把 “patients.csv” 读入为 R 数据框 `pa.tab`, 取出其中前 10 条的出生日期、发病日期、诊断时间到变量中, 用如下程序:

```
pa.tab <- read.csv("patients.csv", header=TRUE)
date1 <- pa.tab[1:10, '出生日期']
date2 <- pa.tab[1:10, '发病日期']
date3 <- pa.tab[1:10, '诊断日期']
```

- (1) 把 date1、date2、date3 转换为 R 的 POSIXct 日期型。
- (2) 求 date1 中的各个出生年。
- (3) 计算发病时的年龄，以周岁论（过生日才算）。
- (4) 把 date2 中发病年月转换为'monyy' 格式，这里 mon 是如 FEB 这样英文三字母缩写，yy 是两数字的年份。

### 思考题

1. 假设向量  $x$  长度为  $n$ , 其元素是  $\{1, 2, \dots, n\}$  的一个重排。可以把  $x$  看成一个  $i$  到  $x[i]$  的映射 ( $i$  在  $\{1, 2, \dots, n\}$  中取值)。求向量  $y$ , 保存了上述映射的逆映射, 即: 如如果  $x[i]=j$ , 则  $y[j]=i$ 。
2. 对诸如'FEB91', 'OCT15' 这样的年月数据, 假设 00—20 表示 21 世纪年份, 21—99 表示 20 实际年份。编写 R 函数, 输入这样的字符型向量, 返回相应的 POSIXct 格式日期, 具体日期都取为相应月份的 1 号。
3. 对 R 的 POSIXct 日期, 写函数转换成'FEB91', 'OCT15' 这样的年月表示, 假设 00—20 表示 21 世纪年份, 21—99 表示 20 实际年份。
4. 给定两个 POSIXct 日期向量 birth 和 work, birth 为生日, work 是入职日期, 编写 R 函数, 返回相应的入职周岁整数 (不到生日时周岁值要减一)。

### 2.1.7 因子

#### 因子

- R 中用因子代表数据中分类变量的值。
- 如性别、省份、职业。
- 有序因子代表有序量度的变量值, 如打分结果, 疾病严重程度等。
- 例:

```
> x <- c("男", "女", "男", "男", "女")
> sex <- factor(x)
> sex
[1] 男 女 男 男 女
Levels: 男 女
```

### 因子的类型转换

- 因子在数据框中保存时保存为 1,2,3 这样的数值型序号以节省存储空间, 在显示时才显示为其对应的值。
- 可以用 `as.numeric` 把因子转换为其保存用的编号数值, 用 `as.character` 把因子转换为其表示用的形式, 也不再是因子。如

```
> as.numeric(sex)
[1] 1 2 1 1 2
> as.character(sex)
[1] "男" "女" "男" "男" "女"
```

### factor 函数

- factor 函数的一般形式:

```
factor(x, levels = sort(unique(x),
      na.last = TRUE),
      labels, exclude = NA,
      ordered = FALSE)
```

- 可以自行指定各离散取值 (水平 levels), 不指定时由 x 的不同值来求得。
- labels 可以用来指定各水平的标签, 不指定时用各离散取值的对应字符串。
- exclude 参数用来指定要转换为缺失值 (NA) 的元素值集合。

- 如果指定了 levels, 则因子向量元素中的某个元素当它等于水平中第 j 个时元素值取”j”, 如果它的值没有出现在 levels 中则对应因子元素值取 NA。
- ordered 取真值时表示因子水平是有次序的 (按编码次序)。

### table 函数

- 用 table 函数统计因子各水平的出现次数。
- 如

```
> table(sex)
sex
男 女
3  2
```

- 对一个变量用 table 函数计数的结果是一个有元素名的向量, 元素名是分析的变量的取值, 结果的元素值是其元素名对应的变量值的频数。
- 一般的向量也可以用 table 函数统计各不同值的出现次数。

### tapply 函数

- 可以按照因子分组然后每组计算另一变量的概括统计。
- 如

```
> h <- c(165, 170, 168, 172, 159)
> tapply(h, sex, mean)
      男      女
168.3333 164.5000
```

- 这里身高向量 h 与因子 factor 是等长的, 对应元素分别为同一人的身高和性别, tapply 函数分男女两组计算了身高平均值。



## 练习

- 把 “class.csv” 读入为 R 数据框 d.class, 其中的 sex 列已经自动转换为因子。取出其中的 sex 和 age 列到变量 sex 和 age 中, 用如下程序:

```
d.class <- read.csv('class.csv', header=TRUE)
sex <- d.class[, 'sex']
age <- d.class[, 'age']
```

- (1) 统计并显示列出 sex 的不同值频数;
- (2) 分男女两组分别求年龄最大值;
- (3) 把 sex 变量转换为一个新的因子, F 变成 “女”, M 变成 “男”。

## 2.1.8 R 矩阵和多维数组

## R 矩阵

- 矩阵用 matrix 函数定义。缺省为按列存储。

```
> A <- matrix(1:6, nrow=3, ncol=2)
> print(A)
      [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> B <- matrix(c(1,-1, 1,1),
+   nrow=2, ncol=2, byrow=TRUE)
> print(B)
      [,1] [,2]
[1,]    1   -1
[2,]    1    1
```

### 矩阵运算

- 可以用`%%` 表示矩阵乘法，如

```
> C1 <- A %% B
> print(C1)
      [,1] [,2]
[1,]    5    3
[2,]    7    3
[3,]    9    3
```

### 矩阵运算

- 可以与标量四则运算，结果为每个元素进行相应运算，如：

```
> C2 <- A + 2
> print(C2)
      [,1] [,2]
[1,]    3    6
[2,]    4    7
[3,]    5    8
> C3 <- A / 2
> print(C3)
      [,1] [,2]
[1,]  0.5  2.0
[2,]  1.0  2.5
[3,]  1.5  3.0
```

### 矩阵运算

- 相同形状的两个矩阵可以作四则运算，表示对应元素的运算，如

```
> C4 <- C2 * C3
> print(C4)
      [,1] [,2]
```

```
[1,]  1.5 12.0  
[2,]  4.0 17.5  
[3,]  7.5 24.0
```

- 要注意，两个矩阵用星号连接不表示矩阵乘法，而是对应元素相乘；矩阵乘法用`%*%`表示。

### 矩阵下标

- `A[1,]` 取出 A 的第一行。
- `A[,1]` 取出 A 的第一列。
- `A[1:3,1:2]` 取出子矩阵。
- 用列名访问矩阵：

```
colnames(A) <- c("x", "y")  
A[, "y"]
```

- `rownames(A)` 访问 A 的行名。

### `rbind` 和 `cbind` 函数

- `rbind(A, B)` 把两个列数 (宽度) 相同的矩阵上下合并。
- `cbind(A, B)` 把两个行数 (高度) 相同的矩阵左右合并。
- `cbind(x)` 把向量 `x` 转换为列向量 (一个列数为 1 的矩阵)。
- 注意向量 (vector) 既不是行向量，也不是列向量；既可以当成行向量，也可以当成列向量。

### 矩阵求逆

- `solve(B)` 返回 B 的逆矩阵。
- `solve(B, c(1,2))` 解线性方程组

$$\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} x = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

**apply() 函数**

- `apply(x, 2, FUN)` 把矩阵 `x` 的每一列分别输入到 `FUN` 中，得到对应于每一列的结果，如

```
> A <- rbind(c(1, 5, 2), c(7, 4, 9)); A
      [,1] [,2] [,3]
[1,]    1    5    2
[2,]    7    4    9
> apply(A, 2, mean)
[1] 4.0 4.5 5.5
```

- `apply(x, 1, FUN)` 把矩阵 `x` 的每一行分别输入到 `FUN` 中，得到与每一行对应的结果，如

```
> print(A)
      [,1] [,2] [,3]
[1,]    1    5    2
[2,]    7    4    9
> apply(A, 1, sum)
[1] 8 20
```

- 如果函数 `FUN` 返回多个结果，则 `apply(x, 2, A)` 结果为矩阵，矩阵的每一列是输入矩阵的 `FUN` 结果，如

```
> print(A)
      [,1] [,2] [,3]
[1,]    1    5    2
[2,]    7    4    9
> apply(A, 2, range)
      [,1] [,2] [,3]
[1,]    1    4    2
[2,]    7    5    9
```

- 如果函数 FUN 返回多个结果，为了对每行计算 FUN 的结果，应该用 `t(apply(x, 1, FUN))` 格式，如

```
> print(A)
      [,1] [,2] [,3]
[1,]    1    5    2
[2,]    7    4    9
> t(apply(A, 1, range))
      [,1] [,2]
[1,]    1    5
[2,]    4    9
```

- 结果是一个矩阵，矩阵的每行是输入矩阵每行用 FUN 汇总的结果。

### 多维数组

- 矩阵是多维数组的特例。矩阵是  $x_{ij}, i = 1, 2, \dots, n, j = 1, 2, \dots, m$  这样的两下标数据的存储格式，三维数组是  $x_{ijk}, i = 1, 2, \dots, n, j = 1, 2, \dots, m, k = 1, 2, \dots, p$  这样的三下标数据的存储格式， $s$  维数组则是有  $s$  个下标的数据的存储格式。
- 多维数组的一般定义语法为

```
数组名 <- array(数组元素,
  dim=c(第一下标个数, 第二下标个数, ...,
    第 s 下标个数))
```

- 其中数组元素的次序是第一下标变化最快，第二下标次之，最后一个下标是变化最慢的。这种次序称为 FORTRAN 次序。
- 下面是一个三维数组定义例子。

```
> ara <- array(1:24, dim=c(2,3,4)); ara
, , 1
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
, , 2
```

```

      [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3

      [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18

, , 4

      [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24

```

- 这样的数组保存了  $x_{ijk}, i = 1, 2, j = 1, 2, 3, k = 1, 2, 3, 4$ 。
- 三维数组 ara 可以看成是 4 个  $2 \times 3$  矩阵。取出其中一个如 `ara[, , 2]` (取出第二个矩阵):

```

> print(ara[, , 2])
      [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

```

- 但是，多维数组可以进行一般的子集操作，比如 `ara[, 2, 2:3]` 是  $x_{ijk}, i = 1, 2, j = 2, k = 2, 3$  的值，结果是一个  $2 \times 2$  矩阵:

```

> print(ara[, 2, 2:3])
      [,1] [,2]
[1,]    9   15
[2,]   10   16

```

### 数组降维

- 取数组（包括矩阵）子集时，如果某个下标是单一值，结果中该维消失。

- 比如，设  $A$  是一个矩阵， $A[,1]$  取出  $A$  的第一列，返回为一个向量，不再是列向量； $A[1,]$  取出  $A$  的第一行，返回为一个向量，不再是行向量。
- 为了在取子集时不降低维数，在方括号中加 `drop=FALSE` 选项。这样， $A[,1,drop=FALSE]$  把  $A$  的第一列取出作为一个列数等于 1 的矩阵。
- 在 R 编程中取数组下标子集时，如果对某个下标不确定会不会仅有一个下标值，应该加上 `drop=FALSE` 以保持结果维数的一致性。

### 2.1.9 数据框

#### 数据框

- 统计分析中最常见的原始数据形式是类似于数据库表或 Excel 数据表的形式。
- R 中叫做数据框 (`data.frame`)。
- 数据框类似于一个矩阵，有  $n$  行、 $p$  列，但各列允许有不同类型：数值型向量、因子、字符型向量。
- 函数 `data.frame` 可以生成数据框，如：

```
> d <- data.frame(  
+   name=c(" 李明", " 张聪", " 王建"),  
+   age=c(30, 35, 28),  
+   height=c(180, 162, 175))  
> print(d)  
  name age height  
1 李明  30   180  
2 张聪  35   162  
3 王建  28   175
```

#### 数据框内容访问

- 数据框可以用矩阵格式访问，如

```
> d[2,3]
[1] 162
> d[,2]
[1] 30 35 28
```

- 数据框每一列叫做一个变量，必须有变量名。按列名访问列可用如

```
> d[, "age"]
[1] 30 35 28
> d$age
[1] 30 35 28
```

- 可以同时取行子集和列子集，如

```
> d[1:2, 'age']
[1] 30 35
```

- 又如

```
> d[1:2, c('age', 'height')]
      age height
李明   30    180
张聪   35    162
```

- 数据框每一行可以有行名，这是重要的技术。比如，每一行定义行名为身份证号，则可以唯一识别各行。如

```
> rownames(d) <- d$name
> d$name <- NULL
> d
      age height
李明   30    180
张聪   35    162
```



```
王建 28 175
> d["张聪", "age"]
[1] 35
```

### 数据框与矩阵的区别

- 数据框不能作为矩阵参加矩阵运算。
- 可以用 `as.matrix` 函数转换数据框或数据框的子集为矩阵。
- 如

```
> d2 <- as.matrix(d[,c("age", "height")])
> d3 <- crossprod(d2)
> d3

      age height
age    2909 15970
height 15970 89269
```

这里 `crossprod(A)` 表示  $A^T A$ 。

### gl 函数

- `gl` 函数可以用来生成多个因子的均匀组合。
- 如

```
d <- data.frame(
  group=gl(3, 10, length=30),
  subgroup=gl(5,2,length=30),
  obs=gl(2,1,length=30))
print(d)
```

结果的数据框 `d` 有三个变量: `group` 是大组, 共分 3 个大组, 每组 10 个观测; `subgroup` 是子组, 在每个大组内分为 5 个子组, 每个子组 2 个观测。共有  $3 \times 5 \times 2 = 30$  个观测 (行)。

- `gl` 第一个参数是因子水平个数，第二个参数是同一因子水平连续重复次数，第三个参数是总共需要的元素个数，所有水平都出现后则重复整个模式直到长度满足要求。

### 练习

- 把 “class.csv” 读入为 R 数据框 `d.class`, 其中的 `sex` 列已经自动转换为因子。
  - (1) 显示 `d.class` 中年龄至少为 15 的行子集;
  - (2) 显示女生且年龄至少为 15 的学生姓名和年龄;
  - (3) 取出数据框中的 `age` 变量赋给变量 `x`。

### 2.1.10 列表

#### 列表

- R 中列表 (`list`) 类型来保存不同类型的数据。
- 一个主要目的是提供 R 分析结果输出包装：输出一个变量，这个变量包括回归系数、预测值、残差、检验结果等等一系列不能放到规则形状数据结构中的内容。
- 实际上，数据框也是列表的一种，但是数据框要求各列等长，而列表不要求。
- 定义列表用函数 `list`, 如

```
> rec <- list(name=" 李明", age=30,
+             scores=c(85, 76, 90))
> rec
$name
[1] " 李明"

$age
[1] 30

$scores
[1] 85 76 90
```

### 列表元素访问

- 列表的一个元素也可以称为列表的一个“变量”。
- 用两重方括号格式访问，如：

```
> rec[[3]]
[1] 85 76 90
> rec[[3]][2]
[1] 76
> rec[["age"]]
[1] 30
```

- 也可以用“\$”格式访问，如：

```
> rec$age
[1] 30
```

### 列表元素名处理

- 用 `names(rec)` 函数查看和修改元素名。如

```
> names(rec)
[1] "name"  "age"   "scores"
> names(rec)[names(rec)=='scores'] <- ' 三科分数 '
> names(rec)
[1] "name"      "age"        " 三科分数 "
> rec[[" 三科分数"]]
[1] 85 76 90
```

### 修改列表元素

- 可以修改列表元素内容。如

```
> rec[[" 三科分数"]][2] <- 77
> print(rec)
$name
[1] " 李明"

$age
[1] 30

$ 三科分数
[1] 85 77 90
```

### 添加列表元素

- 直接给列表不存在的元素名定义元素值就添加了新元素，如

```
> rec[[' 身高']] <- 178
> print(rec)
$name
[1] " 李明"

$age
[1] 30

$ 三科分数
[1] 85 77 90

$ 身高
[1] 178
```

### 删除列表元素

- 把某个列表元素赋值为 `NULL` 就删掉这个元素。`NULL` 是 R 的一个特殊值，表示“不存在”，注意这与缺失值不同，缺失值可理解为存在但是没有记录到的值。如

```
> rec[['age']] <- NULL
> print(rec)
$name
[1] " 李明"

$ 三科分数
[1] 85 77 90

$ 身高
[1] 178
```

### 返回列表的函数示例—`lm()`

- 列表的重要作用是把分析函数的多项输出集中到一个变量中。
- 比如`lm`函数的结果：

```
> d.class <- read.csv(file='class.csv',
+                      header=TRUE)
> lm1 <- lm(weight ~ height + age,
+           data=d.class)
> names(lm1)
[1] "coefficients" "residuals"
[3] "effects"      "rank"
[5] "fitted.values" "assign"
[7] "qr"           "df.residual"
[9] "xlevels"      "call"
[11] "terms"        "model"
```

### 返回列表的函数示例—`eigen()`

- 计算特征值和特征向量的 `eigen` 函数返回两个元素 `values` 和 `vectors` 的列表：

```

> ev <- eigen((1:3) %o% (1:3))
> ev
$values
[1] 1.400000e+01 5.329071e-15 1.484923e-15

$vectors
      [,1]      [,2]      [,3]
[1,] -0.2672612  0.9636241  0.0000000
[2,] -0.5345225 -0.1482499 -0.8320503
[3,] -0.8017837 -0.2223748  0.5547002

```

- values 是特征值，vectors 是一个矩阵，矩阵每一列是一个特征向量。

### 返回列表的函数示例—strsplit()

- strsplit() 输入一个字符型向量并指定一个分隔符，返回一个项数与字符型向量元素个数相同的列表，列表每项对应于字符型向量中一个元素的拆分结果。
- 如

```

> x <- c('10, 8, 7',
+       '5, 2, 2',
+       '3, 7, 8',
+       '8, 8, 9')
> strsplit(x, ',')
[[1]]
[1] "10" " 8" " 7"

[[2]]
[1] "5" " 2" " 2"

[[3]]
[1] "3" " 7" " 8"

[[4]]
[1] "8" " 8" " 9"

```

### 2.1.11 R 工作空间

#### R 工作空间

- R 把在命令行定义的变量都保存到工作空间中，在退出 R 时可以选择是否保存工作空间。

- 用 `ls()` 命令可以查看工作空间中的内容，如：

```
> ls()
[1] "d"          "d.class" "h"          "name"
[5] "pa.tab"     "rec"      "sex"        "x"
```

### 工作空间管理

- 随着多次在命令行使用 R，工作空间的变量越来越多，使得重名的可能性越来越大，而且工作空间中变量太多也让我们不容易查看其内容。
- 可以用 `rm` 函数删除工作空间中的变量，格式如

```
> rm(d, h, name, rec, sex, x)
> ls()
[1] "d.class" "pa.tab"
```

### 用自定义函数避免工作空间杂乱

- 要避免工作空间杂乱，最好的办法还是所有的运算都写到自定义函数中。
- 自定义函数中定义的变量都是临时的，不会保存到工作空间中。
- 这样，仅需要时才把变量值在命令行定义，这样的变量一般是读入的数据或自定义的函数（自定义函数也保存在工作空间中）。

### 避免工作空间杂乱的草稿函数

- 定义如下的 `sandbox` 函数：

```
sandbox <- function(){
  cat(' 沙盘： 接连的空行回车可以退出。\\n')
  browser()
  browser()
}
```

- 运行 `sandbox()` 函数，将出现如下的 browser 命令行：

```
> sandbox()  
沙盘：接连的空行回车可以退出。  
Called from: sandbox()  
Browse[1]>
```

在这样的 browser 命令行中随意定义变量，定义的变量不会保存到工作空间中。

## 2.2 R 输入输出

### 2.2.1 输入输出的简单方法

#### R 简单输出

- 显示某个变量：`print(x)`。
- `cat()` 函数：用如

```
cat("x =", x, "\n")
```

- 写入指定文件中并且在文件尾部添加：

```
cat("x =", x, "\n",  
    file="res.txt", append=TRUE)
```

#### 输出记录

- 开始把输出分流到指定的文件中：

```
sink("allres.txt", split=TRUE)
```

- 终结前面的 `sink()` 的作用：

```
sink()
```



### 输入向量

- 用 `scan()` 函数输入文本文件中的数值向量，数值之间以空格分开。  
如

```
> cat(1:12, file="d:/work/x.txt")
> x <- scan("d:/work/x.txt")
Read 12 items
> x
[1] 1 2 3 4 5 6 7 8 9 10 11 12
> A1 <- matrix(x, nrow=4, ncol=3, byrow=TRUE)
> A1
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
```

- 没有提供输入文件的 `scan()` 函数从命令行读入数据。
- 没有提供输入文件的 `scan()` 函数从命令行读入数据。

### 读入 csv 文件

- 用 `read.csv()` 把一个第一行为变量名的 CSV 格式的数据表读入为 R 的数据框 (data frame)。如

```
> d.class <- read.csv("class.csv", header=TRUE)
> head(d.class)
  name sex age height weight
1 Alice  F  13   56.5   84.0
2 Becka  F  13   65.3   98.0
3  Gail   F  14   64.3   90.0
4 Karen  F  12   56.3   77.0
5 Kathy  F  12   59.8   84.5
6  Mary   F  15   66.5  112.0
```

### 2.2.2 文本格式数据访问

#### 文本文件读入

- 主要支持.CSV 文件，用 `read.csv` 读取。
- 对于空格分开的文件，可以用 `read.table` 读取。
- 使用其它分隔符如分号、制表符的文本也可以用 `read.table` 读取。
- 每列上下对齐的文本文件可以用 `read.fwf` 读取。
- `scan` 函数可以从命令行读入向量，从文件读入向量，也可以读入复杂结构的文本文件。

#### `read.csv` 函数

- 用 `read.csv` 函数把.csv 格式的文件读入为 R 数据框。
- 用 `header=TRUE` 指明文件第一行作为变量名。
- `read.csv` 会把字符型的列转换为数据框内的因子列，为了保持原来的字符型列仍为字符型，用选项 `stringsAsFactors=FALSE`。

### 2.2.3 Excel 表访问

#### Excel 表格数据访问

- 基本办法是转换为其他格式如 CSV 格式，用 `read.csv` 和 `write.csv` 读写。
- RODBC 包也可以支持 Excel 文件的读写。
- 小量数据可以选中一个矩形区域复制，然后在 R 中用如

```
myDF <- read.delim("clipboard")
```

- 也可以从 R 中复制数据集到 Excel 中:

```
write.table(iris, "clipboard", sep = "\t",  
            col.names = NA)  
## 然后在空 Excel 表中粘贴就可以。
```

- 从 R 中复制数据框到 Excel 中, 可以自己定义这样一个函数:

```
write.table(iris, "clipboard", sep = "\t",  
            col.names = NA)  
## 然后在空 Excel 表中粘贴就可以。
```

### 用 write.csv 函数把数据框保存为.csv 文件

- 先生成模拟测试数据, 然后用 write.csv 保存 csv 文件:

```
d1 <- data.frame(' 学号'=c("a", "b", "c"),  
                 ' 数学'=1:3, ' 语文'=11:13)  
write.csv(d1, file='tmp1.csv', row.names=FALSE)
```

### 用 RODBC 包读取 Excel 文件

- 使用 RODBC 包可以直接读取 Excel 文件中的工作簿。
- 使用样例:

```
> library(RODBC)  
> d1 <- sqlFetch(odbcConnectExcel("class.xls"),  
+               sqtable = "Sheet1", na.strings = "NA",  
+               as.is = T)  
> odbcCloseAll()  
> head(d1, 3)  
  name sex age height weight  
1 Alice  F  13   56.5     84  
2 Becka  F  13   65.3     98  
3 Gail   F  14   64.3     90
```

- 其中 “class.xls” 是一个 Excel 文件, 有一个 “Sheet1” 工作簿, 工作簿中第一行是变量名。
- sqlFetch 函数把工作簿内容读入到数据框中。
- odbcClassAll 函数关闭打开的数据库连接。

### 用 RODBC 包保存到 Excel 文件

- 先生成模拟测试数据:

```
d1 <- data.frame(' 学号'=c("a", "b", "c"),  
                  ' 数学'=1:3, ' 语文'=11:13)  
d2 <- data.frame(" 学号"=c("x", "y", "z"),  
                  " 数学"=101:103, " 语文"=211:213)
```

- 写入 Excel 文件。先确保没有旧文件:

```
require(RODBC)  
  
fname <- "testwrite.xls"  
if(file.exists(fname)) file.remove(fname)
```

- 打开可写的连接后把 d1 和 d2 保存为两个工作簿, 并关闭数据库连接:

```
con <- odbcConnectExcel(fname, readOnly=F)  
res <-  
  sqlSave(con, d1, tablename=" 理科成绩",  
           rownames=F, colnames=F,  
           safer=T)  
res <-  
  sqlSave(con, d2, tablename=" 文科成绩",  
           rownames=F, colnames=F,  
           safer=T)  
close(con)
```

### 2.2.4 Access 数据库访问

#### Access 数据库访问

- 假设有 Access 数据库在文件 c:/Friends/birthdays.mdb 中，内有两个表 Men 和 Women，每个表包含域 Year, Month, Day, First Name, Last Name, Death。域名应尽量避免用空格。
- 读入所有女性记录的程序：

```
require(RODBC)
con <- odbcConnectAccess("c:/Temp/birthdays.mdb")
women <- sqlQuery(con, 'select * from Women')
print(women)
close(con)
```

- 因为 sqlQuery 支持对数据库用 SQL 查询，所以 RODBC 包可以连接数据库并实现复杂查询访问。
- RODBC 对有定义的 ODBC 数据源都可以访问。

### 2.2.5 Oracle 数据库访问

#### Oracle 数据库访问

- Oracle 是最著名的数据库服务器软件。
- 为了访问 Oracle 数据库服务器中的数据库，在 R 中需要安装 ROracle 包。这是一个源代码扩展包，需要用户自己编译安装。
- 在 MS Windows 环境下，需要安装 R 软件和 RTools 软件包（在 CRAN 网站的 Windows 版本软件下载栏目中）。
- 用如下命令编译 R 的 ROracle 扩展包：

```
set OCI_LIB32=D:\oracle\product\10.2.0\db_1\bin
set OCI_INC=D:\oracle\product\10.2.0\db_1\oci\include
set PATH=D:\oracle\product\10.2.0\db_1\bin;\
C:\Rtools\bin;C:\Rtools\gcc-4.6.3\bin;"%PATH%"
C:\R\R-3.2.0\bin\i386\rcmd INSTALL ROracle_1.2-1.tar.gz
```

- 其中的前三个 set 命令设置了 Oracle 数据库程序或客户端程序的查找路径，第三个 set 命令还设置了 RTools 编译器的路径。第三行和第四行应该是连在一起的。这些路径需要根据实际情况修改。最后一个命令进行编译。

### 从 R 中访问 Oracle 数据库的代码

- 为了从 R 中访问 Oracle 数据库的表，示例如下：

```
require(ROracle)
drv <- dbDriver("Oracle")

conn <- dbConnect(drv, username="test",
  password="oracle",
  dbname="orcl")

rs <- dbSendQuery(conn, "select * from testtab")
d <- fetch(rs)
print(d)
```

- 其中 orcl 是本地安装的 Oracle 数据库软件或客户端软件定义的本地或远程 Oracle 数据库的标识，test 和 oracle 是此数据库的用户名和密码，testtab 是此数据库中的一个表。
- 用 dbGetTable 取出一个表并存入 R 数据框中。

### 2.2.6 MySQL 数据库访问

#### R 访问 MySQL 数据库

- MySQL 是高效、免费的数据库服务器软件，在很多行业尤其是互联网行业占有很大的市场。
- 为了在 R 中访问 MySQL 数据库，只要安装 RMySQL 扩展包（有二进制版本）。
- 假设服务器地址在 192.168.1.111，可访问的数据库名为 world，用户为 test，密码为 mysql。设 world 库中有表 country。

- 创建连接的程序如下：

```
con <- dbConnect(RMySQL::MySQL(),
  dbname='world',
  username='test', password='mysql',
  host='192.168.1.111')
```

- 下列代码列出 world 库中的所有表，然后列出其中的 country 表的所有变量：

```
dbListTables(con)
dbListFields(con, 'country')
```

- 下列代码取出 country 表并存入 R 数据框 d.country 中。

```
d.country <- dbReadTable(con, 'country')
```

- 下列代码把 R 中的示例数据框 USArrests 写入 MySQL 库 world 的表 arrests 中：

```
data(USArrests)
dbWriteTable(con, 'arrests', USArrests,
  overwrite=TRUE)
```

- 可以用 dbGetQuery 执行一个 SQL 查询并返回结果，如

```
> dbGetQuery(con, 'select count(*) from arrests')
  count(*)
1       50
```

- 当表很大时，可以用 `dbSendQuery()` 发送一个 SQL 命令，返回一个查询结果指针对象，用 `dbFetch()` 从指针对象位置读取指定行数，用 `dbHasCompleted()` 判断是否已读取结束。如

```
res <- dbSendQuery(con, "SELECT * FROM country")
while(!dbHasCompleted(res)){
  chunk <- dbFetch(res, n = 5)
  print(chunk[,1:2])
}
dbClearResult(res)
```

- 数据库使用完毕时，需要关闭用 `dbConnect()` 打开的连接：

```
dbDisconnect(con)
```

### 2.2.7 格式化输出

#### L<sup>A</sup>T<sub>E</sub>X 介绍

- L<sup>A</sup>T<sub>E</sub>X 是一个文档排版系统，功能强大，结果美观，设计合理。
- 缺点是需要学习类似于 HTML 的另一种语言。
- L<sup>A</sup>T<sub>E</sub>X 源文件主要是编译为 PDF 和 PostScript，也可以编译为 HTML。

#### knitr 介绍

- knitr 是一种 L<sup>A</sup>T<sub>E</sub>X 模板，其中可以包含 R 代码，直接把 R 变量和运行结果、图形插入生成的 L<sup>A</sup>T<sub>E</sub>X 文件中。
- 是一种自动生成重复报告的好办法。
- 用 `\Sexpr` 插入一个 R 表达式的值。
- 插入 R 代码以 `<<>=` 行开始，以 `@` 行结束。
- 结果中的图形可以自动插入到结果中。



- 在<<>>= 中间可以插入选项:
- `echo=FALSE`: 结果中不要插入程序代码本身;
- 在 R 中, 用

```
knit('testsw.Rnw')
```

转换为  $\text{L}^{\text{T}}\text{E}_X$  文件, 然后再编译为 PDF 即可。

- knitr 也可以生成 HTML、Markdown 等格式。

### 2.2.8 文件输入输出

#### 文件输入输出

- 输入输出可以针对命令行, 针对文件, R 支持扩展的文件类型, 称为“connection”。
- 用 `open` 命令打开连接, 返回一个句柄, 用 `close` 关闭打开的连接。
- 在打开的时候文件可以顺序访问。
- 函数 `file` 打开一个普通文件。函数 `url` 打开一个用 URL 指定的远程文件。函数 `file` 打开一个字符串用于读写。函数 `gzfile`, `bzfile`, `xzfile`, `unz` 支持对压缩过的文件的访问 (不是压缩包, 只对一个文件压缩)。

#### 打开文件连接的函数

```
file("path", open="", blocking=T,
     encoding = getOption("encoding"),
     raw = FALSE)

url(description, open = "", blocking = TRUE,
     encoding = getOption("encoding"))

textConnection(description, open="r",
               local = FALSE,
               encoding = c("", "bytes", "UTF-8"))
```

```
gzfile(description, open = "",
        encoding = getOption("encoding"),
        compression = 6)

bzfile(description, open = "",
        encoding = getOption("encoding"),
        compression = 9)

xzfile(description, open = "",
        encoding = getOption("encoding"),
        compression = 6)

unz(description, filename, open = "",
     encoding = getOption("encoding"))
```

### 连接访问

- 连接函数不自动打开连接。
- 给定一个未打开的连接，读取函数从中读取时会自动打开连接，函数结束时自动关闭连接。
- 要多次从一个连接读取时就应该先打开连接。读取完毕用 `close` 函数关闭。
- 连接函数有一个 `open` 参数如果指定值则会打开连接。

### 打开连接的不同模式

- 参数 `open` 的取值如下：
- `"r"`—文本型只读；
- `"w"`—文本型只写；
- `"a"`—文本型末尾添加；
- `"rb"`—二进制只读；

- "wb"—二进制只写;
- "ab"—二进制末尾添加;
- "r+" 或 "r+b"—允许读和写;
- "w+" 或 "w+b"—允许读和写, 但刚打开时清空文件;
- "a+" 或 "a+b"—末尾添加并允许读。

### 文本文件访问

- 函数 `readLines`, `scan` 可以从一个文本型连接读取。
- 给定一个打开的连接 `con`, 用 `readLines` 函数可以把文件各行读入为字符型向量的各个元素。可以指定要读的行数。如

```
ll <- readLines(file('class.csv'))
print(ll)
```

- 用 `writeLines` 函数可以把一个字符型向量各元素作为不同行写入一个文本型连接。如

```
vnames <- strsplit(ll, ',')[[1]]
writeLines(vnames, con='class-names.txt')
```

其中的 `con` 参数应该是一个打开的文本型写入连接, 但是可以直接给出一个要写入的文件名。

### 二进制文件访问

- 函数 `save` 用来保存 R 变量到文件, 函数 `load` 用来从文件中读取保存的 R 变量。
- 函数 `readBin` 和 `writeBin` 对 R 变量进行二进制文件存取。
- 如果要访问其它软件系统的二进制文件, 请参考 R 手册中的“R Data Import/Export Manual”。

### 字符型连接

- 函数 `textConnection` 打开一个字符串用于读取或写入，是很好用的一个 R 功能。
- 可以把一个小文件存放在一个长字符串中，然后用 `textConnection` 读取，如

```
fstr <-  
"name,score  
王芳,78  
孙莉,85  
张聪,80  
"  
d <- read.csv(textConnection(fstr), header=T)  
print(d)
```

- 读取用的 `textConnection` 的参数是一个字符型变量。

### 字符型连接

- 在整理输出结果时，经常可以向一个字符型变量连接写入，最后再输出整个字符串值。
- 例如：

```
tc <- textConnection("sres", open="w")  
cat('Trial of text connection.\n', file=tc)  
cat(1:10, '\n', file=tc, append=T)  
close(tc)  
print(sres)
```

- 注意写入用的 `textConnection` 的第一个参数是保存了将要写入的字符型变量名的字符串，而不是变量名本身，第二个参数表明是写入操作，使用完毕需要用 `close` 关闭。

### 2.2.9 目录和文件管理

#### 目录和文件管理函数

- `getwd()`—返回当前工作目录。
- `setwd(path)`—设置当前工作目录。
- `list.files()` 或 `dir()`—查看目录中内容。`list.files(pattern='.*[.]r$')` 可以列出所有以“.r”结尾的文件。
- `file.path()`—把目录和文件名组合得到文件路径。
- `file.info(filename)`—显示文件的详细信息。
- `file.exists()`—查看文件是否存在。
- `file.access()`—考察文件的访问权限。
- `create.dir()`—新建目录。
- `file.create()`—生成文件。
- `file.remove()` 或 `unlink()`—删除文件。`unlink()` 可以删除目录。
- `file.rename()`—为文件改名。
- `file.append()`—把两个文件相连。
- `file.copy()`—复制文件。
- `basename()` 和 `dirname()`—从一个全路径文件名获取文件名和目录。

## 2.3 程序控制结构

### 2.3.1 表达式

#### 程序控制结构

- S 是一个表达式语言, 其任何一个语句都可以看成是一个表达式。
- 表达式之间以分号分隔或用换行分隔。

- 表达式可以续行, 只要前一行不是完整表达式 (比如末尾是加减乘除等运算符, 或有未配对的括号) 则下一行为上一行的继续。
- 若干个表达式可以放在一起组成一个复合表达式, 作为一个表达式使用。组合用大括号表示, 如:

```
{  
  x <- 15  
  x  
}
```

### 2.3.2 分支结构

#### 分支结构

- 分支结构包括 if 结构:  
if (条件) 表达式 1
- 或  
if (条件) 表达式 1 else 表达式 2
- 其中的“条件”为一个标量的真或假值, 表达式可以用大括号包围的复合表达式。
- 如

```
if(is.na(lambda)) lambda <- 0.5
```

- 又如

```
if(x>1) {  
  y <- 2.5  
} else {  
  y <- -y  
}
```

### 用逻辑下标代替分支结构

- R 是向量化语言，尽可能少用标量运算。
- 比如，x 为一个向量，要定义 y 与 x 等长，且 y 的每一个元素当且仅当 x 的对应元素为正数时等于 1，否则等于零。
- 这样是错误的：

```
if(x>0) y <- 1 else y <- 0
```

- 正解为：

```
y <- numeric(length(x))
y[x>0] <- 1
y[x<=0] <- 0
y
```

### 2.3.3 循环结构

#### 计数循环

- 为了对向量每个元素、矩阵每行、矩阵每列循环处理，语法为

for(循环变量 in 下标列表) 表达式

- 如：

```
y <- numeric(length(x))
for(i in seq(length(x))) {
  if(x[i]>0) y[i] <- 1 else y[i] <- 0
}
```

- 其中的 seq(length(x)) 可以改写成 seq(along=x)，也能产生 x 的下标序列。
- 但是，for 循环应尽量避免：其速度比向量化版本慢一个数量级以上，而且写出的程序不够典雅。

### while 循环和 repeat 循环

- 用  
`while(条件) 表达式`  
进行当型循环。
- 用  
`repeat 表达式`  
进行无条件循环（一般在循环体内用 `if` 与 `break` 退出）。
- `break` 语句退出所在的循环。
- `next` 语句进入所在循环的下一轮。

### R 中判断条件

- `if` 语句和 `while` 语句中用到条件。
- 条件必须是标量值，而且必须为 `TRUE` 或 `FALSE`，不能为 `NA` 或零长度。
- 这是比较容易出错的程序做法。

### 2.3.4 函数

#### R 函数

- 编程语言中函数的优点：代码复用、模块化设计。
- 把编程任务分解成小的模块，每个模块用一个函数实现，可以降低复杂性，防止变量混杂。
- 函数的自变量是只读的，函数中定义的局部变量只在函数运行时起作用，不会与外部或其它函数中同名变量混杂。
- 函数返回一个对象作为输出，如果需要返回多个变量，可以用列表进行包装。



### 函数定义

- 例如

```
fsub <- function(x, y=0){  
  cat("x=", x, " y=", y, "\n")  
  x - y  
}
```

函数体内最后一个表达式为返回值。也可以用 `return(x)` 返回值。

- 定义时可以规定可选参数如上面的 `y`。
- 注意：函数定义是一个表达式，函数作为一个对象被赋值给了变量 `fsub`。

### 函数调用

- 函数调用时可选参数可以省略，可以按次序对准，也可以用名字对准。
- 如

```
> fsub(5,3)  
x= 5  y= 3  
[1] 2  
> fsub(5)  
x= 5  y= 0  
[1] 5  
> fsub(x=5, y=3)  
x= 5  y= 3  
[1] 2  
> fsub(y=3, x=5)  
x= 5  y= 3  
[1] 2
```

### 变量作用域

- 函数的参数和函数中定义（即赋值）的变量都是局部的。
- 这一点符合函数式编程 (functional programming) 的要求。
- 函数的自变量是只读的，在函数内部对自变量作任何修改在函数运行完成后都不影响原来的变量值。
- 在所有函数外面（如 R 命令行）定义的变量是全局变量。
- 函数内部可以访问全局变量的值，但一般不能修改全局变量的值：
- 全局变量总是可读有时会造成难以察觉的错误。
- 在函数内部如果要修改全局变量的值，用 `<<-` 代替 `<-` 进行赋值。

### 修改自变量

- 为了修改某个自变量，在函数内修改其值并将其作为函数返回值，赋值给原变量。
- 比如定义了如下函数：

```
f <- function(x, inc=1){  
  x <- x + inc  
  x  
}
```

- 调用如

```
x <- 100  
cat('Old x=', x, '\n')  
x <- f(x)  
cat('New x=', x, '\n')
```

### 2.3.5 程序调试

### 程序调试

- 函数定义一般都包含多行，所以一般不在命令行定义函数，而是把函数定义写在源程序文件中，用 `source` 命令调入。
- 考虑如下函数定义：

```
f <- function(x){  
  for(i in 1:n){  
    s <- s + x[i]  
  }  
}
```

- 运行发现有错误：

```
> f(1:5)  
Error in f(1:5) : 找不到对象'n'
```

- 用 `browser()` 函数帮助调试程序。`browser()` 函数可以令程序进入跟踪运行状态，在跟踪运行状态可以查看变量值，用 `n` 命令逐句运行，用 `c` 命令恢复正常运行。
- 为调试函数 `f` 的定义，在定义中插入对 `browser()` 的调用：

```
f <- function(x){  
  browser()  
  for(i in 1:n){  
    s <- s + x[i]  
  }  
}
```

- 再次运行：

```
> f(1:5)  
Called from: f(1:5)  
Browse[1]> n
```

```
debug 在 #3: for (i in 1:n) {
  s <- s + x[i]
}
Browse[2]> n
Error in f(1:5) : 找不到对象'n'
```

- 发现是在 `for(i in 1:n)` 行遇到未定义的变量 `n`。
- 在源文件中把出错行改为 `for(i in 1:length(x))`，再次运行：

```
> f(1:5)
Called from: f(1:5)
Browse[1]> n
debug 在 #3: for (i in 1:length(x)) {
  s <- s + x[i]
}
Browse[2]> n
debug 在 #4: s <- s + x[i]
Browse[2]>
错误: 找不到对象's'
```

- 发现在运行 `s <- s + x[i]` 行时，遇到未定义的变量 `s`。这是忘记初始化引起的。
- 在 `for` 语句前添加 `s <- 0` 语句，函数定义变成：

```
f <- function(x){
  browser()
  s <- 0
  for(i in 1:length(x)){
    s <- s + x[i]
  }
}
```

- 再次运行，在 `Browse[1]>` 提示下命令 `c` 表示不跟踪运行，程序不显示错误但是没有显示求和结果。

- 错误是忘记把函数返回值写在函数定义最后。
- 在函数定义最后添加 `s` 一行，函数定义变成：

```
f <- function(x){  
  browser()  
  s <- 0  
  for(i in 1:length(x)){  
    s <- s + x[i]  
  }  
  s  
}
```

- 再次运行，程序结果与手工验算结果一致：

```
> f(1:5)  
Called from: f(1:5)  
Browse[1]> c  
[1] 15
```

- 自定义函数应该用各种不同输入测试其正确性和稳定性。如

```
> x <- 1:5; f(x[x>6])  
Called from: f(x[x > 6])  
Browse[1]> c  
numeric(0)
```

- 程序输入了零长度自变量，我们期望其输出为零而不是 `numeric(0)`。在自变量 `x` 为零长度时，函数中 `for(i in 1:length(x))` 应该一次都不进入循环，跟踪运行发现实际对 `i=1` 和 `i=0` 共运行了两轮循环。把这里的 `1:length(x)` 改成 `seq(along=x)` 解决了问题，`seq(along=x)` 生成 `x` 的下标序列，如果 `x` 是零长度的则下标序列为零长度向量。
- 函数最终修改为：

```
f <- function(x){  
  s <- 0  
  for(i in seq(along=x)){  
    s <- s + x[i]  
  }  
  s  
}
```

注意程序调试成功后对 `browser()` 的调用就删除了（也可以注释掉）。

- 这里只是用这个简单函数演示如何调试程序，求向量和本身是不需要我们去定义新函数的，`sum` 函数本来就是完成这样的功能。
- 实际上，许多我们认为需要自己编写程序作的事情，在 R 网站都能找到别人已经完成的程序。

### 出错调试选项

- 进行如下设置：

```
options(error=recover)
```

- 则在出错后可以选择进入出错的某一层函数内部，在 `browser` 环境中跟踪运行。
- 例如刚开始的有错误的函数定义

```
f <- function(x){  
  for(i in 1:n){  
    s <- s + x[i]  
  }  
}
```

- 在运行时出现调试选择：

```
> f(1:5)
Error in f(1:5) : 找不到对象'n'

Enter a frame number, or 0 to exit

1: f(1:5)

Selection: 1
Called from: top level
Browse[1]> print(x)
[1] 1 2 3 4 5
Browse[1]> print(n)
收捲时出错: 找不到对象'n'
Browse[1]> Q
```

- 选择要进入的函数定义对应序号进入该函数内部跟踪。用 Q 终止运行并退出整个 browser 跟踪。

### 警告处理

- 有些警告信息实际是错误，用 `options()` 的 `warn` 参数可以设置警告级别，如设置 `warn=2` 则所有警告当作错误处理。
- 设置如

```
options(warn=2)
```

## 2.4 R 程序效率

### 2.4.1 向量化编程

#### 向量化编程

- R 是解释型语言，在执行单个运算时，效率与编译代码相近；在执行迭代循环时，效率较低，与编译代码的速度可能相差几十倍。

- R 以向量、矩阵为基础运算单元，在进行向量、矩阵运算时效率很高，应尽量采用向量化编程。
- 在计算总和、元素乘积或者每个向量元素的函数变换时，应使用相应的函数，如 `sum`, `prod`, `sqrt`, `log` 等。

### 向量化编程例 1

- 假设要计算如下的偏度估计：

$$\text{Skewness} = \frac{n}{(n-1)(n-2)} \sum_{i=1}^n \left( \frac{(x_i - \bar{x})^3}{s_x} \right)^3, \quad (2.1)$$

其中  $\bar{x}$  和  $s_x$  为样本均值和样本方差。

- 利用传统思维，程序可写成：

```
skew <- function(x){
  n <- length(x)
  xbar <- mean(x)
  s <- sd(x)
  su <- 0
  for(i in seq(along=x)){
    su <- su + ((x[i] - xbar) / s)^3
  }
  su <- su * n / ((n-1)*(n-2))
  su
}
```

- 利用 R 向量化计算，程序简化为

```
skew <- function(x){
  n <- length(x)
  res <- ( n / ((n-1)*(n-2))
    * sum( ((x - mean(x)) / sd(x))^3 ) )

  res
}
```



## 向量化编程例 2

- 假设要编写函数计算

$$f(x) = \begin{cases} 1 & x \geq 0, \\ 0 & \text{其它} \end{cases} \quad (2.2)$$

- 利用传统思维，程序写成

```
f <- function(x){  
  n <- length(x)  
  y <- numeric(n)  
  
  for(i in seq(along=x)){  
    if(x[i] >= 0) y[i] <- 1  
    else y[i] <- 0  
  }  
  
  y  
}
```

- 利用向量化与逻辑下标，程序写成:

```
f <- function(x){  
  n <- length(x)  
  y <- numeric(n)  
  y[x >= 0] <- 1  
  ##y[x < 0] <- 0  
  
  y  
}
```

## 向量化编程例 3

- 考虑一个班的学生存在生日相同的概率。

- 假设一共有 365 个生日 (只考虑月、日)。
- 设一个班有  $n$  个人, 当  $n$  大于 365 时 {至少两个人有生日相同} 是必然事件。
- 当  $n$  小于等于 365 时:

$$\begin{aligned}
 &P(\text{至少有两人同生日}) \\
 &= 1 - \Pr(n \text{ 个人生日彼此不同}) \\
 &= 1 - \frac{365 \times 364 \times \cdots \times (365 - (n - 1))}{365^n} \\
 &= 1 - \frac{365 - 0}{365} \cdot \frac{365 - 1}{365} \cdots \frac{365 - (n - 1)}{365}
 \end{aligned}$$

- 对  $n = 1, 2, \dots, 365$  来计算对应的概率。
- 完全用循环 (两重循环):

```

x <- numeric(365)
for(n in 1:365){
  x[n] <- 1
  for(j in 0:(n-1)){
    x[n] <- x[n] * (365-j)/365
  }
  x[n] <- 1 - x[n]
}

```

- 重复一千次用了 125 秒。
- 用 prod 函数向量化内层循环:

```

x <- numeric(365)
for(n in 1:365){
  x[n] <- 1 - prod((365:(365-n+1))/365)
}

```

- 重复一千次用了 1.7 秒, 速度提高了 70 倍。

- 注意：中间的语句不能写成

```
x[n] <- 1 - prod((365:(365-n+1)))/365^n
```

否则可能会数值溢出。

- 用 `cumprod` 函数可以完全避免循环：

```
x <- 1 - cumprod((365:1)/365)
```

- 重复十万次才用了 0.60 秒，速度比第二个版本提高 280 倍，比第一个版本提高 2 万倍！

### 2.4.2 减少显式循环

#### 代替显式循环的 R 函数

- 显式循环是 R 运行速度较慢的部分。
- R 中的有些运算可以用已有函数完成，如 `sum`, `prod`, `cumsum`, `cumprod`, `mean`, `var`, `sd` 等。
- R 的 `sin`, `sqrt`, `log` 等函数都是向量化的，可以直接对输入向量的每个元素进行变换。
- 对矩阵，用 `apply` 函数汇总矩阵每行或每列。`colMeans`, `rowMeans` 可以计算矩阵列平均和行平均，`colSums`, `rowSums` 可以计算矩阵列和与行和。
- `apply` 类函数有多个，包括 `apply`, `sapply`, `lapply`, `tapply`, `vapply`, `replicate` 等。

#### `sapply()` 和 `lapply()` 函数

- 对列表，使用 `sapply` 函数或 `lapply` 函数操作列表每个元素。格式为

```
sapply(列表, 函数)
```

或

```
lapply(列表, 函数)
```

- `sapply` 和 `lapply` 函数都会对输入列表的每一元素应用输入的函数进行变换，但是 `lapply` 总是返回一个列表，其元素与输入列表元素相对应；`sapply` 试图简化输出结果为向量或矩阵，在不可行时才和 `lapply` 返回列表结果。
- `sapply` 和 `lapply` 并不能提高运行效率，只会令程序更简洁。

### `sapply` 和 `lapply` 示例 1: 数据框变量类型

- 例如，`typeof` 函数求变量的存储类型，如

```
> d.class <- read.csv('class.csv', header=TRUE)
> typeof(d.class[, 'age'])
[1] "integer"
```

- 设 `d.class` 是一个数据框，数据框也是一个列表，每个元素是数据框的一列。如下程序可以求每一列的存储类型：

```
> sapply(d.class, typeof)
      name      sex      age      height      weight
"integer" "integer" "integer" "double"  "double"
```

- 注意因为 CSV 文件读入为数据框时把姓名、性别都转换成了因子，所以这两个变量的存储类型也是整数。
- 用 `mode` 也是求变量存储类型，但与 `typeof` 略有差别。`typeof` 区分 `double` 和 `integer`，`mode` 把这两种都算是 `numeric`。如

```
> sapply(d.class, mode)
      name      sex      age      height      weight
"numeric" "numeric" "numeric" "numeric" "numeric"
```

- 用 `class()` 可以获得变量的类型如整数、数值、因子等。如

```
> sapply(d.class, class)
      name      sex      age      height      weight
"factor" "factor" "integer" "numeric" "numeric"
```

- 关于数据框变量类型，用 `str()` 函数可以显示更详细的结果：

```
> str(d.class)
'data.frame':  19 obs. of  5 variables:
 $ name  : Factor w/ 19 levels "Alfred","Alice",...:
      2 3 5 10 11 12 15 16 17 1 ...
 $ sex   : Factor w/ 2 levels "F","M":
      1 1 1 1 1 1 1 1 1 2 ...
 $ age   : int   13 13 14 12 12 15
      11 15 14 14 ...
 $ height: num   56.5 65.3 64.3 56.3
      59.8 66.5 51.3 62.5 62.8 69 ...
 $ weight: num   84 98 90 77 84.5 ...
```

#### sapply 和 lapply 示例 2: `strsplit()` 函数结果处理

- 假设有 4 个学生的 3 次小测验成绩，每个学生的成绩记录到了一个以逗号分隔的字符串中，如：

```
> s <- c('10, 8, 7',
+       '5, 2, 2',
+       '3, 7, 8',
+       '8, 8, 9')
```

- 对单个学生，可以用 `strsplit()` 函数把三个成绩拆分并转为数值型，如：

```
> strsplit(s[1], ',', fixed=TRUE)[[1]]
[1] "10" " 8" " 7"
```

```

> as.numeric(strsplit(s[1], ',',
+                     fixed=TRUE)[[1]])
[1] 10  8  7
> sum(as.numeric(strsplit(s[1], ',',
+                     fixed=TRUE)[[1]]))
[1] 25

```

- 注意这里 `strsplit()` 的结果是仅有一个元素的列表，用了 “[[...]]” 格式取出其元素。
- 用 `strsplit()` 处理有 4 个字符串的 `s`, 结果是长度为 4 的列表：

```

> tmp.res <- strsplit(s, ',', fixed=TRUE)
> tmp.res
[[1]]
[1] "10" " 8" " 7"

[[2]]
[1] "5" " 2" " 2"

[[3]]
[1] "3" " 7" " 8"

[[4]]
[1] "8" " 8" " 9"

```

- 用 `sapply()` 和 `as.numeric()` 可以把列表中所有字符型转为数值型，并以矩阵格式输出：

```

> sapply(tmp.res, as.numeric)
      [,1] [,2] [,3] [,4]
[1,]   10    5    3    8
[2,]    8    2    7    8
[3,]    7    2    8    9

```

- 当 `sapply()` 对列表每一项都有多个输出且输出个数不变时，输出为矩阵，而且矩阵的行数与每项输出个数相同，矩阵每一列对应于输入列表的每一项。
- 这里可能转置结果更合理:

```
> t(sapply(tmp.res, as.numeric))
      [,1] [,2] [,3]
[1,]   10    8    7
[2,]    5    2    2
[3,]    3    7    8
[4,]    8    8    9
```

### `sapply` 和 `lapply` 示例 3: 不等长结果

- 调用 `sapply(列表, 函数)` 时，如果“函数”结果长度有变化，结果只能以列表输出。
- 例如，数据框 `d` 中有两列数，希望将每列变成没有重复值的。数据例子如下:

```
> d <- data.frame(x1=c(1,3,3,2), x2=c(3,5,5,3))
> d
  x1 x2
1  1  3
2  3  5
3  3  5
4  2  3
```

- 因为 `x1` 和 `x2` 两列的无重复值个数不同，结果只能是列表:

```
> sapply(d, unique)
$ x1
[1] 1 3 2

$ x2
[1] 3 5
```

- 但是，如果无重复个数相同，结果就是矩阵：

```
> d <- data.frame(x1=c(1,3,3,2), x2=c(3,5,5,7))
> sapply(d, unique)
      x1 x2
[1,]  1  3
[2,]  3  5
[3,]  2  7
```

- 这样使用 `sapply()` 会造成编程判断困难，所以在不能确定函数结果长度是否保持不变时，应该用 `lapply()` 代替 `sapply()`, `lapply()` 总是返回列表。
- 如

```
> d <- data.frame(x1=c(1,3,3,2), x2=c(3,5,5,7))
> lapply(d, unique)
$x1
[1] 1 3 2

$x2
[1] 3 5 7
```

#### sapply 和 lapply 示例 4: 无名函数

- `sapply` 和 `lapply` 中要做的函数变换可以当场定义，不需要函数名。
- 仍使用示例 2 的数据，任务是从输入的逗号分隔成绩中求每个学生的三科总分。
- 用 `strsplit()` 拆分可得列表，每个列表是由三个成绩字符串的字符型向量。如下代码可以求得总分：



```

> s <- c('10, 8, 7',
+       '5, 2, 2',
+       '3, 7, 8',
+       '8, 8, 9')
> sapply( strsplit(s, ',', fixed=TRUE),
+       function(ss) sum(as.numeric(ss)) )
[1] 25  9 18 25

```

### replicate() 函数

- `replicate()` 函数用来执行某段程序若干次，类似于 `for()` 循环但是没有计数变量。常用于随机模拟。
- `replicate()` 的缺省设置会把重复结果尽可能整齐排列成一个多维数组输出。
- 语法为  
`replicate(重复次数, 要重复的表达式)`
- 其中的表达式可以是复合语句，也可以是执行一次模拟的函数。

### replicate 示例 1—正态统计量

- 例如，设总体  $X$  为  $N(0,1)$ ，取样本量  $n = 5$ ，重复地生成模拟样本共  $B = 6$  组，输出每组样本的样本均值和样本标准差。
- 模拟可以用如下的 `replicate()` 实现：

```

> replicate(6, {
+   x <- rnorm(5, 0, 1);
+   c(mean(x), sd(x)) })
      [,1]      [,2]      [,3]
[1,] -0.6149395 -0.7964398 -0.3146326
[2,]  0.5630400  1.2359015  1.4808345
      [,4]      [,5]      [,6]
[1,] -0.4606527 -0.04231328  0.3024232
[2,]  0.7614834  1.40925525  0.7193673

```

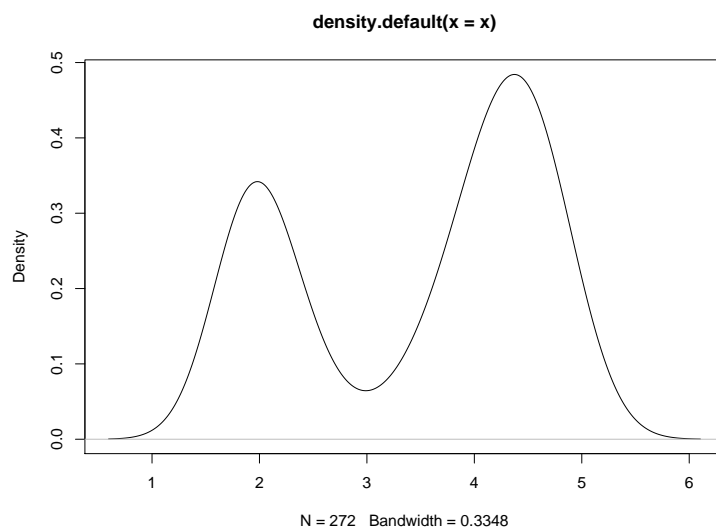
- 结果是一个矩阵，矩阵行数与每次模拟的结果（均值、标准差）个数相同，矩阵每一列对应于一次模拟。此结果转置可能更合适。

### replicate 示例 2—核密度的 bootstrap 置信区间

- R 自带的数据框 `faithful` 内保存了美国黄石国家公园 Faithful 火山的 272 次爆发持续时间和间歇时间。
- 为估计爆发持续时间的密度，可以用核密度估计方法，R 函数 `density` 可以执行此估计，返回  $N$  个格子点上的密度曲线坐标：

```
x <- faithful$eruptions
est0 <- density(x)
plot(est0); locator(1)
```

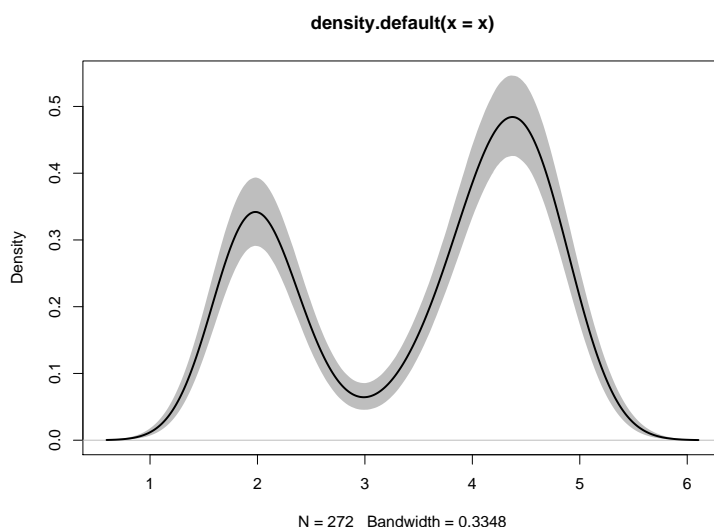
- 图形如下：



- 核密度估计是统计估计，希望得到其置信区间。采用非参数 bootstrap 方法。

- 重复  $B = 10000$  次, 每次从原始样本中有重复地抽取大小相同的一组样本, 对这组样本计算核密度估计, 结果为  $(x_i, y_i^{(j)}), i = 1, 2, \dots, N, j = 1, 2, \dots, B$ , 每组样本估计  $N$  个格子点的密度曲线坐标, 横坐标不随样本改变。
- 对每个横坐标  $x_i$ , 取 bootstrap 得到的  $B$  个  $y_i^{(j)}, j = 1, 2, \dots, B$  的 0.025 和 0.975 样本分位数, 作为真实密度  $f(x_i)$  的 bootstrap 置信区间。
- 注意, 此置信区间是点点意义下的, 不是整体的。
- 程序和图形如下。

```
set.seed(1)
resm <- replicate(10000, {
  x1 <- sample(x, replace=TRUE)
  density(x1, from=min(est0$x),
          to=max(est0$x))$y
})
CI <- apply(resm, 1, quantile, c(0.025, 0.975))
plot(est1, ylim=range(CI), type='n')
polygon(c(est0$x, rev(est0$x)),
        c(CI[1,], rev(CI[2,])),
        col='grey', border=FALSE)
lines(est0, lwd=2)
```



- 程序中，`set.seed()` 调用保证了每次运行程序得到完全相同的结果。
- `replicate()` 重复了 10000 次 bootstrap 抽样和密度估计，得到的结果 `resm` 有 10000 列，每列是一组 bootstrap 样本的核密度估计曲线的格子点纵坐标，格子点横坐标采用了和原始数据相同的格子。
- 对 `resm` 矩阵，用 `apply()` 函数对每个格子点横坐标（`resm` 的每行），估计 10000 个密度估计纵坐标的样本 0.025 和 0.975 分位数，结果 `CI` 为两行，第一行是各格子点横坐标对应的 0.025 分位数，第二行是各格子点横坐标对应的 0.975 分位数。
- `polygon()` 函数用来画阴影区域，习惯做法是 `polygon(c(x, rev(x)), c(y1, rev(y2)))`，这里 `x` 是界定阴影区域的上下两条曲线的共同的横坐标，`y1` 是下边缘曲线纵坐标，`y2` 是上边缘曲线纵坐标。

### 2.4.3 R 的计算函数

#### R 的计算函数

- R 中提供了大量的数学函数、统计函数和特殊函数，可以打开 R 的 HTML 帮助页面，进入“Search Enging & Keywords”链接，查看其中与算数、数学、优化、线性代数等有关的专题。
- 这里简单列出一部分常用函数，对函数 `filter`, `fft`, `convolve` 进行说明。

### 数学函数

- 常用数学函数包括 `abs`, `sign`, `log`, `log10`, `sqrt`, `exp`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`。还有 `gamma`, `lgamma`(伽玛函数的自然对数)。
- 这些函数是向量化的一元函数。

### 概括函数

- `sum` 对向量求和, `prod` 求乘积。
- `cumsum` 和 `cumprod` 计算累计, 得到和输入等长的向量结果。
- `mean` 计算均值, `var` 计算样本方差, `sd` 计算样本标准差, `median` 计算中位数, `quantile` 计算样本分位数。

### 最值

- `max` 和 `min` 求最大和最小, `cummax` 和 `cummin` 累进计算。
- `range` 返回最小值和最大值两个元素。
- `max`, `min`, `range` 如果有多个自变量可以把这些自变量连接起来后计算。
- `pmax(x1,x2,...)` 对若干个等长向量计算对应元素的最大值, 不等长时短的被重复使用。`pmin` 类似。比如, `pmax(0, pmin(1,x))` 把 `x` 限制到 `[0,1]` 内。

### 排序

- `sort` 返回排序结果。可以用 `decreasing=TRUE` 选项进行降序排序。
- `sort` 可以有一个 `partial=` 选项, 这样只保证结果中 `partial=` 指定的下标位置是正确的。比如:

```
> sort(c(3,1,4,2,5), partial=3)
[1] 2 1 3 4 5
```

只保证结果的第三个元素正确。可以用来计算样本分位数估计。

- 选项 `na.last` 指定缺失值的处理, 取 `NA` 则删去缺失值, 取 `TRUE` 则把缺失值排在最后面, 取 `FALSE` 则把缺失值排在最前面。
- `order` 返回排序用的下标序列, 它可以有多个自变量, 按这些自变量的字典序排序。可以用 `decreasing=TRUE` 选项进行降序排序。如果只有一个自变量, 可以使用 `sort.list` 函数。
- `rank` 计算秩统计量, 可以用 `ties.method` 指定同名次处理方法, 如 `ties.method="min"` 取最小秩。
- `order`, `sort.list`, `rank` 也可以有 `na.last` 选项, 只能为 `TRUE` 或 `FALSE`。
- `rev` 反转序列。

### 离散傅立叶变换 `fft`

- R 中 `fft` 函数使用快速傅立叶变换算法计算离散傅立叶变换。
- 设 `x` 为长度 `n` 的向量, `y=fft(x)`, 则

$$y[k] = \sum_{j=0}^{n-1} x[j] \exp(-i2\pi \frac{jk}{n}), \quad k = 0, 1, \dots, n-1.$$

- 即

$$y_{k+1} = \sum_{j=0}^{n-1} x_{j+1} \exp\left(-i2\pi \frac{jk}{n}\right), \quad k = 0, 1, \dots, n-1.$$

注意没有除以 `n`。

- 另外, 若 `y=fft(x)`, `z=fft(y, inverse=T)`, 则 `x == z/length(x)`。

### 用 `filter` 函数作迭代

- R 在遇到向量自身迭代时很难用向量化编程解决, `filter` 函数可以解决其中部分问题。
- `filter` 函数可以进行卷积型或自回归型的迭代。语法为

```
filter(x, filter,
       method = c("convolution", "recursive"),
       sides=2, circular =FALSE, init)
```

**filter 示例 1: 双侧滤波**

- 对输入序列  $x_t, t = 1, 2, \dots, n$ , 希望进行如下滤波计算:

$$y_t = \sum_{j=-k}^k a_j x_{t-j}, \quad k+1 \leq t \leq n-k-1, \quad (2.3)$$

其中  $(a_{-k}, \dots, a_0, \dots, a_k)$  是长度为  $2k+1$  的向量。

- 注意公式中  $a_j$  与  $x_{t-j}$  对应。
- 假设  $x$  保存在向量  $x$  中,  $(a_{-k}, \dots, a_0, \dots, a_k)$  保存在向量  $f$  中,  $y_{k+1}, \dots, y_{n-k}$  保存在向量  $y$  中, 无定义部分取 NA, 程序可以写成

```
y <- filter(x, f, method="convolution", sides=2)
```

- 比如, 设  $x = (1, 3, 7, 12, 17, 23)$ ,  $(a_{-1}, a_0, a_1) = (0.1, 0.5, 0.4)$ , 则

$$y_t = 0.1 \times x_{t+1} + 0.5 \times x_t + 0.4 \times x_{t-1}, \quad t = 2, 3, \dots, 5$$

- 程序为

```
> y <- filter(c(1,3,7,12,17,23),
+   c(0.1, 0.5, 0.4),
+   method="convolution",
+   sides=2)
> print(y)
Time Series:
Start = 1
End = 6
Frequency = 1
[1]  NA  2.6  5.9 10.5 15.6  NA
```

**filter 示例 2: 单侧滤波**

- 对输入序列  $x_t, t = 1, 2, \dots, n$ , 希望进行如下滤波计算:

$$y_t = \sum_{j=0}^k a_j x_{t-j}, \quad k+1 \leq t \leq n, \quad (2.4)$$

其中  $(a_0, \dots, a_k)$  是长度为  $k+1$  的向量。

- 注意公式中  $a_j$  与  $x_{t-j}$  对应。
- 假设  $x$  保存在向量  $x$  中,  $(a_{-k}, \dots, a_0, \dots, a_k)$  保存在向量  $f$  中,  $y_{k+1}, \dots, y_n$  保存在向量  $y$  中, 无定义部分取 NA, 程序可以写成

```
y <- filter(x, f, method="convolution", sides=1)
```

- 比如, 设  $x = (1, 3, 7, 12, 17, 23)$ ,  $(a_0, a_1, a_2) = (0.1, 0.5, 0.4)$ , 则

$$y_t = 0.1 \times x_t + 0.5 \times x_{t-1} + 0.4 \times x_{t-2}, \quad t = 3, 4, \dots, 6$$

- 程序为

```
> y <- filter(c(1,3,7,12,17,23),
+           c(0.1, 0.5, 0.4),
+           method="convolution",
+           sides=1)
> print(y)
Time Series:
Start = 1
End = 6
Frequency = 1
[1] NA NA 2.6 5.9 10.5 15.6
```

### filter 示例 3: 自回归迭代零初值

- 设输入  $x_t, t = 1, 2, \dots, n$ , 要计算

$$y_t = \sum_{j=1}^k a_j y_{t-j} + x_t, \quad t = 1, 2, \dots, n,$$

其中  $(a_1, \dots, a_k)$  是  $k$  个实数,  $(y_{-k+1}, \dots, y_0)$  已知。

- 设  $x$  保存在向量  $x$  中,  $(a_1, \dots, a_k)$  保存在向量  $f$  中,  $(y_1, \dots, y_n)$  保存在向量  $y$  中。
- 如果  $(y_{-k+1}, \dots, y_0)$  都等于零, 可以用程序:



```
filter(x, f, method="recursive")
```

- 如果  $(y_0, \dots, y_{-k+1})$  保存在向量  $b$  中 (注意与时间顺序相反), 可以用程序:

```
filter(x, f, method="recursive", inits=b)
```

- 比如, 设  $x = (1, 3, 7, 12, 17, 23)$ ,  $(a_1, a_2) = (0.9, 0.1)$ ,  $y_{-1} = y_0 = 0$ , 则

$$y_t = 0.9 \times y_{t-1} + 0.1 \times y_{t-2} + x_t, \quad t = 1, 2, \dots, 6$$

- 程序为

```
> y <- filter(c(1,3,7,12,17,23),
+           c(0.9, 0.1),
+           method="recursive")
> print(y, digits=4)
Time Series:
Start = 1
End = 6
Frequency = 1
[1] 1.00 3.90 10.61 21.94 37.81 59.22
```

#### filter 示例 4: 自回归迭代有初值

- 在前面的示例 3 中, 如果已知  $y_0 = 200, y_{-1} = 100$ , 程序改为

```
> y <- filter(c(1,3,7,12,17,23),
+           c(0.9, 0.1),
+           method="recursive",
+           init=c(200, 100))
> print(y, digits=4)
Time Series:
Start = 1
```

```
End = 6  
Frequency = 1  
[1] 191.0 194.9 201.5 212.8 228.7 250.1
```

## 第三章 R 数据整理与概括

### 3.1 数据整理

#### 3.1.1 数据读取技巧

`read.csv` 使用技巧

- 最基本用法如:

```
> d.class <- read.csv('class.csv', header=TRUE)
> print(head(d.class))
  name sex age height weight
1 Alice  F  13   56.5   84.0
2 Becka  F  13   65.3   98.0
3 Gail   F  14   64.3   90.0
4 Karen  F  12   56.3   77.0
5 Kathy  F  12   59.8   84.5
6 Mary   F  15   66.5  112.0
```

- `head=TRUE` 指定 CSV 文件第一行是变量名。
- `read.csv` 可能会把字符型列自动转为因子。对于 `sex` 变量，这是合适的；对于 `name` 变量，转为因子生成了一个没有重复值的因子，这样的因子没有什么用处。
- 用 `stringsAsFactors=FALSE` 选项抑制这种转换：

```
d.class <- read.csv('class.csv', header=TRUE,  
  stringsAsFactors=FALSE)
```

- 需要转换为因子的，可以自己用程序转换，如：

```
d.class[, 'sex'] <- factor(d.class[, 'sex'])
```

### 读入日期

- 设文件 “dates.csv” 中包含如下内容：

```
序号, 出生日期, 发病日期  
1, 1941/3/8, 2007/1/1  
5, 1943/3/10, 2007/1/1  
6, 1940/1/8, 2007/1/1  
9, 1961/6/23, 2007/1/2  
10, 1949/1/10, 2007/1/2  
15, 1930/7/1, 2007/1/3  
18, 1989/8/2, 2007/1/4  
19, 1949/7/9, 2007/1/4  
22, 1948/4/17, 2007/1/4  
23, 1957/2/3, 2007/1/5
```

- 先把日期当作字符串读入：

```
d.dates <- read.csv('dates.csv', header=TRUE,  
  stringsAsFactors=FALSE)
```

- 然后用 `as.POSIXct` 函数转换为 R 日期类型：

```
d.dates[, '出生日期'] <-  
  as.POSIXct(d.dates[, '出生日期'],  
    format='%Y/%m/%d')
```

```
d.dates[, '发病日期'] <-
  as.POSIXct(d.dates[, '发病日期'],
    format='%Y/%m/%d')
```

- 读入效果:

```
> head(d.dates)
  序号  出生日期  发病日期
1    1 1941-03-08 2007-01-01
2    5 1943-03-10 2007-01-01
3    6 1940-01-08 2007-01-01
4    9 1961-06-23 2007-01-02
5   10 1949-01-10 2007-01-02
6   15 1930-07-01 2007-01-03
```

### 日期差计算

- R 的日期可以用 `difftime` 计算差值。为了计算发病时的年龄，包括小数部分，可以这样计算：

```
d.dates[, '发病年龄（带小数年）'] <-
  as.numeric(difftime(d.dates[, '发病日期'],
    d.dates[, '出生日期'], units='days')/365.25)
```

### 计算周岁

- 如果按照我们通常计算周岁的方法计算年龄，算法就不仅包括年的差，还要判断是否到了本年的生日。用函数实现周岁计算如下：

```
## 给定生日和现在日期，计算周岁年龄。
age.int <- function(birth, now){
  date1 <- as.POSIXlt(birth)
  date2 <- as.POSIXlt(now)
  age <- date2$year - date1$year
```

```

sele <- (date2$mon * 100 + date2$mday
        < date1$mon * 100 + date1$mday)
## sele 是那些没有到生日的人
age[sele] <- age[sele] - 1

age
}

```

- d.dates 中计算发病时周岁年龄：

```

d.dates[, '发病年龄'] <-
  age.int(d.dates[, '出生日期'],
          d.dates[, '发病日期'])

```

- 结果：

```

> print(head(d.dates))
  序号  出生日期  发病日期  发病年龄（带小数年）  发病年龄
1    1 1941-03-08 2007-01-01          65.81805         65
2    5 1943-03-10 2007-01-01          63.81394         63
3    6 1940-01-08 2007-01-01          66.98163         66
4    9 1961-06-23 2007-01-02          45.52783         45
5   10 1949-01-10 2007-01-02          57.97673         57
6   15 1930-07-01 2007-01-03          76.50924         76

```

### 缺失值处理

- 设有如下的“bp.csv”数据：

```

序号，收缩压
1,145
5,110
6，未测
9,150
10，拒绝
15,115

```

- 其中的血压有非数值内容。
- 直接用 `read.csv` 读入:

```
> d.bp <- read.csv('bp.csv', header=TRUE,
+   stringsAsFactors=FALSE)
> print(head(d.bp))
  序号 收缩压
1    1    145
2    5    110
3    6   未测
4    9    150
5   10   拒绝
6   15    115
> is.character(d.bp[, '收缩压'])
[1] TRUE
```

- 读入的收缩压被当成了字符型列，无法进行计算。如果没有加 `stringsAsFactors` 则可能被读成因子，也不能作为数值计算。
- 把字符型的收缩压转换为数值型:

```
> d.bp[, '收缩压数值'] <-
+   as.numeric(d.bp[, '收缩压'])
Warning message:
强制改变过程中产生了 NA
> print(head(d.bp))
  序号 收缩压 收缩压数值
1    1    145        145
2    5    110        110
3    6   未测         NA
4    9    150        150
```

5	10	拒绝	NA
6	15	115	115

- 收缩压中非数值的项被转换为数值型缺失值，并在转换时有警告信息。
- 注意这里同时保存了原始输入的收缩压和转换为数值的收缩压，这样便于数据核对。

### 练习

- 读入 patients.csv，把其中的日期转换为 R 日期类型，计算发病年龄。保存为数据框 d.patients。
- 读入 cancer.csv，保存为 d.cancer。v0 是放疗前肿瘤体积，v1 是放疗后肿瘤体积。计算放疗后肿瘤缩减率。

### 3.1.2 数据框子集与合并

#### 列子集

- 数据框单个列取出后为向量。访问如 `d.class[, 'height']`, `d.class[, 4]`, `d.class$height`。
- 数据框多个列取出后仍为数据框。访问如 `d.class[, c('height', 'weight')]`, `d.class[, 4:5]`。

#### 行子集

- 数据框的任何行子集仍为数据框，即使只有一行而且都是数值也是如此。
- 行子集如 `d.class[1:5,]`, `head(d.class)`, `tail(d.class)`。
- 选择行子集，方法如

```
sele <- d.class[, 'sex'] == 'F'
d.class[sele,]
```

在行下标处可以使用一个逻辑表达式表示每行是否入选。



### 用 subset 函数取行列子集

- 函数 `subset` 可以更方便地提取数据框行子集，且行选择条件中如果有缺失值则缺失值当作不选入。如

```
> subset(d.bp, 收缩压数值>120)
  序号 收缩压 收缩压数值
1     1     145         145
4     9     150         150
```

注意收缩压数值缺失的没有选入。

- `subset` 的第一参数是一个数据框或矩阵，第二参数是一个选择行的逻辑表达式（注意列名可以直接当作变量名使用）。
- `subset` 还可以加一个 `select` 参数选择列子集，列子集可以用列编号或列名指定，如

```
subset(d.class, sex=='F',
       select=3:5)
subset(d.class, sex=='F',
       select=c('age','height','weight'))
```

### 数据框纵向合并

- 矩阵或数据框要纵向合并，使用 `rbind` 函数即可。
- 要求所有列是一一对应的。
- 比如有如下两个分开男生、女生的数据框：

```
> d1 <- subset(d.class,sex == 'F')
> print(head(d1))
  name sex age height weight
1 Alice  F  13   56.5   84.0
2 Becka  F  13   65.3   98.0
3 Gail   F  14   64.3   90.0
4 Karen  F  12   56.3   77.0
```

```
5 Kathy    F  12   59.8   84.5
6  Mary    F  15   66.5  112.0
```

```
> d2 <- subset(d.class, sex == 'M')
> print(head(d2))
      name sex age height weight
10 Alfred  M  14   69.0  112.5
11  Duke   M  14   63.5  102.5
12  Guido  M  15   67.0  133.0
13  James  M  12   57.3   83.0
14 Jeffrey M  13   62.5   84.0
15  John   M  12   59.0   99.5
d <- rbind(d1, d2)
```

- 合并行如下:

```
> d <- rbind(d1, d2)
> print(dim(d))
[1] 19  5
```

### 横向合并

- 简单的对应行合并，可以用 `cbind` 函数。
- 但是，待合并的两个数据框一般要求按照某列合并。
- 用 `merge` 函数实现按照关键列横向合并。
- 比如下面的程序把 `d.class` 数据框拆分为包含姓名、性别的数据框 `d1` 和包含姓名、年龄、身高、体重的数据框 `d2`，并把 `d2` 重排次序:

```
> print(head(d.class))
      name sex age height weight
1 Alice    F  13   56.5   84.0
2 Becka    F  13   65.3   98.0
```

3	Gail	F	14	64.3	90.0
4	Karen	F	12	56.3	77.0
5	Kathy	F	12	59.8	84.5
6	Mary	F	15	66.5	112.0

```
> d1 <- d.class[,c(1,2)]
> print(head(d1))
      name sex
1 Alice   F
2 Becka   F
3  Gail    F
4 Karen    F
5 Kathy    F
6  Mary    F
```

```
> d2 <- d.class[,c(1,3:5)]
> d2 <- d2[order(d2[, 'age']),]
> print(head(d2))
      name age height weight
7  Sandy  11   51.3   50.5
18 Thomas 11   57.5   85.0
4   Karen 12   56.3   77.0
5   Kathy 12   59.8   84.5
13 James 12   57.3   83.0
15  John 12   59.0   99.5
```

- 横向合并程序:

```
> d3 <- merge(d1, d2, by='name')
> print(head(d3))
      name sex age height weight
1 Alfred   M  14   69.0  112.5
2  Alice    F  13   56.5   84.0
3  Becka    F  13   65.3   98.0
```

4	Duke	M	14	63.5	102.5
5	Gail	F	14	64.3	90.0
6	Guido	M	15	67.0	133.0

- 为了容易标识数据框每行，经常在数据框中设置身份证号、学号、病历号这样的唯一标志列，或把数据框的行名设为这样的身份标志。

### 一对多横向合并

- `merge` 支持一对多横向合并。
- 比如，如下的数据框为性别指定了不同显示方式:

```
d.sexec <- data.frame(
  sexe=c('F', 'M'),
  sexc=c('女', '男'))
```

- 用 `merge` 进行一对多横向合并，参数 `by.x` 指定左数据框的关键列，参数 `by.y` 指定右数据框的关键列:

```
d.class2 <- merge(d.class, d.sexec,
  by.x = 'sex', by.y='sexe')
```

- 合并后:

```
> head(d.class2)
  sex  name age height weight sexc
1  F Alice  13   56.5   84.0   女
2  F Becka  13   65.3   98.0   女
3  F Gail  14   64.3   90.0   女
4  F Karen 12   56.3   77.0   女
5  F Kathy 12   59.8   84.5   女
6  F Mary  15   66.5  112.0   女
```

### 3.1.3 排序与标准化

#### 数据框排序

- 用 `order` 函数可以返回按照某个变量排序的下标序列。
- 比如，为了按年龄排序，用如

```
d.class[order(d.class[, 'age']),]
```

- 用 `decreasing=TRUE` 选项可以从大到小排序。
- 为了按性别和年龄排序，用如

```
d.class[with(d.class, order(sex, age)),]
```

#### 标准化

- 设 `x` 是各列都为数值的列表或数值型矩阵，用 `scale(x)` 可以把每一列都标准化，即每一列都减去该列的平均值，然后除以该列的样本标准差。
- 用 `scale(x, center=TRUE, scale=FALSE)` 仅中心化而不标准化。
- 为了把每列变到 `[0, 1]` 内，可以用如下的方法：

```
scale(x,  
      center=apply(x, 2, min),  
      scale=apply(d1, 2, max)  
        - apply(d1, 2, min))
```

- 函数 `sweep()` 可以执行对每列更一般的变换。

### 3.1.4 长表与宽表的变形与整理

### 长表样例

- 设数据框 `d.long` 变量为 `subject`(病人号, 有 10 个不同值), `time`(随访序号, 取 1,2,3,4), 变量 `x, y`(每个病人每次随访的两个测量指标值)。数据框为  $40 \times 4$  大小。读入:

```
d.long <- read.csv('longtab.csv', header=TRUE)
```

- 实际中, 常常需要把每个病人的 4 次随访的 8 个测量指标合并到一行当中, 这称为长表变宽表问题; 反之则称为宽表变长表问题。
- 用 R 的 `reshape` 扩展包可以比较容易地进行长、宽变换。

### d.long 数据

	subject	time	x	y
1	1	1	5	9
2	1	2	7	7
3	1	3	8	6
4	1	4	6	9
5	2	1	8	1
6	2	2	2	1
7	2	3	10	9
8	2	4	4	5
9	3	1	7	10
10	3	2	2	8
.....				

### melt 函数

- `reshape` 包用 `melt()` 函数把数据框转换为一个容易变形的格式, 称为“融化”。
- `melt()` 函数把变量分为两种: 分组用 (`id`), 测量用 (`measured`)。
- 如下程序把 `d.long` 转化为“融化”格式:

```
require(reshape)
melt.long <- melt(
  d.long,
  id.vars=c('subject', 'time'),
  measure.vars=c('x', 'y'))
print(melt.long)
```

融化后的部分结果

	subject	time	variable	value
1	1	1	x	5
2	1	2	x	7
3	1	3	x	8
4	1	4	x	6
5	2	1	x	8
6	2	2	x	2
7	2	3	x	10
8	2	4	x	4
.....				
41	1	1	y	9
42	1	2	y	7
43	1	3	y	6
44	1	4	y	9
.....				

- 可见，融化的数据框是最长表，包括分组变量的每种组合，另外有特殊变量“variable”和“value”，variable 是某个测量变量名，value 是对应的值。

#### cast 函数

- 用 `cast()` 函数把融化的数据框转换为要求的格式。
- 例如，下面的程序把 `cast.long` 重新转化成了原来 `d.long` 的格式：

```
d1 <- cast(melt.long,
            subject + time ~ variable)
print(d1)
```

- `cast()` 的第二自变量是公式，波折号左边是分组变量，右边是要从纵向转为横向的变量，这里把 `variable` 中的 `x` 和 `y` 转为横向，相应的变量值从 `value` 中取出。

### 转为宽表

- 如下的程序把融化后的数据框转换为宽表，每个病人 4 次随访的 8 个测量值都合并到同一行中：

```
d2 <- cast(melt.long,
            subject ~ variable + time)
print(d2)
```

- 这里公式以病人作为仅有的分类，而变量 `x`, `y` 和 4 个 `time`(时间) 都变成了横向。

### 宽表结果

	subject	x_1	x_2	x_3	x_4	y_1	y_2	y_3	y_4
1	1	5	7	8	6	9	7	6	9
2	2	8	2	10	4	1	1	9	5
3	3	7	2	5	6	10	8	3	7
4	4	1	5	6	7	10	1	1	7
5	5	9	7	10	7	8	8	10	8
6	6	9	2	2	1	8	5	4	5
7	7	6	4	7	1	10	5	4	9
8	8	4	7	4	7	6	2	9	10
9	9	3	4	7	1	8	2	5	9
10	10	7	2	6	9	8	2	6	1



### 宽表反向变换

- 函数 `cast()` 的结果会带有恢复信息，所以可以很容易地用 `melt()` 恢复成融化状态。
- 融化时，`x_1` 这样的变量名自动地解释为变量 `x` 的时间 1 的值。
- 如：

```
d2a <- melt(d2, id.vars='subject')
print(d2a)
```

- 结果数据框与 `melt.long` 基本相同，只有行名和列的次序有差别。

### 宽表变长表

- 但是，如果 `d2` 不再是 `cast` 的结果，而只是一个普通的数据框：

```
d3 <- as.data.frame(d2)
```

- 这时 `melt` 不再默认 `x_1` 表示 `x` 变量的时间 1 的值：

```
d3a <- melt(d3, id.vars='subject')
print(d3a)
```

- 这里没有用 `measure.vars`，则除 `id` 变量之外都是测量变量。结果没有进行时间的拆分。

### 简单 `melt` 的结果

	subject	variable	value
1	1	x_1	5
2	2	x_1	8
3	3	x_1	7
.....			
11	1	x_2	7
12	2	x_2	2
13	3	x_2	2
.....			

- 对于以上的融化不好的结果，可以用 reshape 包的 `colsplit()` 函数把 `x_1` 这样的变量名字重新拆分为测量变量名和测量时间两部分。
- 如：

```
d3b <- cbind(d3a[,-2],
             colsplit(d3a['variable'], split='_',
                     names=c('variable', 'time'))
             )
print(d3b)
```

#### d3b 内容

	subject	value	variable	time
1	1	5	x	1
2	2	8	x	1
3	3	7	x	1
.....				
10	10	7	x	1
11	1	7	x	2
12	2	2	x	2
.....				
41	1	9	y	1
42	2	1	y	1
43	3	10	y	1
.....				

- 结果的 d3b 数据框已经符合融化数据框要求。
- 所以，可以用 `cast()` 转换：

```
d3c <- cast(d3b, subject + time ~ variable)
print(d3c)
```

### 变量名与时间的一般拆分

- 考虑更复杂的情况。假设变量名与时间之间没有下划线这样的分隔符，而且不同测量变量的测量时间也不完全相同的情形。
- 如下程序产生模拟例子：

```
d4 <- d3[,c('subject', 'x_1', 'x_2', 'y_1')]
names(d4) <- c('subject', 'x1', 'x2', 'y1')
print(d4)
```

### d4 的内容

	subject	x1	x2	y1
1	1	5	7	9
2	2	8	2	1
3	3	7	2	10
4	4	1	5	10
5	5	9	7	8
6	6	9	2	8
7	7	6	4	10
8	8	4	7	6
9	9	3	4	8
10	10	7	2	8

- 这样的数据在 melt 后，仍可以用 colsplit() 函数处理：

```
d4a <- melt(d4, id.vars='subject',
            measure.vars=c('x1', 'x2', 'y1'))
print(d4a)
d4b <- cbind(d4a[,-2], # 去掉了 variable 列
            colsplit(d4a['variable'], split=' ',
                    names=c('variable', 'time')))
print(d4b)
```

- 然后就可以用 `cast()` 变成类似于 `d.long` 的格式了，注意这时变量 `y` 的 `time=2` 的值缺失。
- 如：

```
d4c <- cast(d4b, subject + time ~ variable)
print(d4c)
```

`d4c` 的结果

```
  subject time x  y
1        1   1  5  9
2        1   2  7 NA
3        2   1  8  1
4        2   2  2 NA
5        3   1  7 10
6        3   2  2 NA
.....
```

变形同时进行概括

- 如果 `cast` 每组有不只一个值，可以指定一个统计函数进行汇总。
- 下面的程序计算每个病人的所有随访的 `x` 平均值和 `y` 平均值：

```
cast(melt.long, subject ~ variable, mean)
```

平均值结果

```
  subject    x    y
1        1 6.50 7.75
2        2 6.00 4.00
3        3 5.00 7.00
```

4	4	4.75	4.75
5	5	8.25	8.50
6	6	3.50	5.50
7	7	4.50	7.00
8	8	5.50	6.75
9	9	3.75	6.00
10	10	6.00	4.25

- 下面的程序对每个病人的每个随访时间，计算 x 和 y 的最大值：

```
cast(melt.long, subject + time ~ ., max)
```

- 注意公式一端没有变量指定时用句点。

求 x 与 y 最大值的结果

	subject	time	(all)
1	1	1	9
2	1	2	7
3	1	3	8
4	1	4	9
5	2	1	8
.....			

## 3.2 数据概括

### 3.2.1 数据框变量简单概括

癌症数据

- 在 “cancer.csv” 中保存了 34 个病人的代码 (id)、年龄 (age)、性别 (sex)、病理类型 (type)、放疗前肿瘤体积 (v0)、放疗后肿瘤体积 (v1)。

- 其中, sex、type 用来作为分类, 年龄可以作为连续型取值变量, 也可以分组后作为分类。体积是连续型取值变量。
- 读入:

```
d.cancer <- read.csv('cancer.csv', header=TRUE)
```

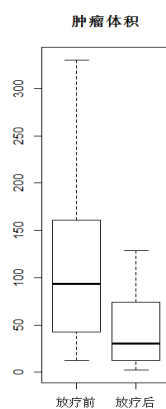
### 用 summary 函数概括数值型量

- 对向量 x, 用 summary(x) 可以获得变量的平均值、中位数、最小值、最大值、四分之一和四分之三分位数。如

```
> with(d.cancer, summary(v0))
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
12.58  43.77   93.40  110.10  157.20  330.20
> with(d.cancer, summary(v1))
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 2.30  12.74   30.62   44.69   72.94  128.30
```

- 可以看出放疗后体积减小了很多。
- 可以用盒形图表现类似的信息, 如

```
with(d.cancer, boxplot(v0))
with(d.cancer, boxplot(list('放疗前'=v0,
                             '放疗后'=v1), main='肿瘤体积'))
```



### 连续型变量概括函数

- 对连续取值的变量 `x`，可以用 `mean`, `std`, `var`, `sum`, `prod`, `min`, `max` 等函数获取基本统计量。加 `na.rm=TRUE` 选项可以仅对非缺失值计算。
- `sort(x)` 返回排序后的结果。
- `rev(x)` 把 `x` 所有元素次序颠倒后返回。
- `quantile(x, c(0.05, 0.95))` 可以求 `x` 的样本分位数。
- `rank(x)` 对 `x` 求秩得分（即名次，但从最小到最大排列）。

### 用 `summary` 函数概括分类变量

- 对因子 `x`，用 `summary(x)` 可以获得各个水平值的出现次数（频率）。如

```
> with(d.cancer, summary(sex))
  F  M
13 21
> with(d.cancer, summary(type))
鳞癌 腺癌
 22  12
```

一般用 `table` 函数作这样的计数。

- 分类变量频数统计结果可以用 `barplot` 画图，见前面关于 `barplot` 的部分。

### 用 `summary` 函数概括数据框所有列

- 对一个数据框 `d`，用 `summary(d)` 可以获得每个连续型变量的基本统计量，和每个离散取值变量的频率。如

```
> summary(d.cancer)
      id      age      sex
Min.   : 1.00  Min.   :49.00 F:13
1st Qu.: 9.25  1st Qu.:55.00 M:21
Median :17.50  Median :67.00
Mean   :17.50  Mean   :64.13
```

```

3rd Qu.:25.75    3rd Qu.:70.00
Max.      :34.00    Max.      :79.00
          NA's    :11
   type          v0          v1
鳞癌:22  Min.    : 12.58  Min.    :  2.30
腺癌:12  1st Qu.: 43.77  1st Qu.: 12.73
          Median : 93.40  Median : 30.62
          Mean   :110.08  Mean   : 44.69
          3rd Qu.:157.18  3rd Qu.: 72.94
          Max.   :330.24  Max.   :128.34

```

### 数据框变量信息

- 对数据框 `d`，用 `str(d)` 可以获得各个变量的类型和取值样例。如

```

> str(d.cancer)
'data.frame':   34 obs. of  6 variables:
 $ id  : int   1 2 3 4 5 6 7 8 9 10 ...
 $ age : int   70 70 69 68 67 75 52 71 68 79 ...
 $ sex : Factor w/ 2 levels "F","M": 1 1 1 2 2 1 2 2 2 2 ...
 $ type: Factor w/ 2 levels "鳞癌","腺癌": 2 2 2 2 1 2 1 1 1 1 ...
 $ v0  : num   26.5 135.5 209.7 61 237.8 ...
 $ v1  : num    2.91 35.08 74.44 34.97 128.34 ...

```

- 用 `head(d)` 可以返回数据框（或向量、矩阵）的前几行，用 `tail(d)` 可以返回数据框的后几行。如

```

head(d.class)
tail(d.class, 1)

```

函数的第二个参数表示提取几行。

### 3.2.2 分类变量概括

#### 分类变量频数

- 分类变量一般输入为因子。
- 对因子 `x`，`table(x)` 返回 `x` 的每个不同值的频率（出现次数），结果为一个类（class）为 `table` 的一维数组。每个元素有对应的元素名，为 `x` 的各水平值。



- 如

```
> res <- with(d.cancer, table(sex))
> res
sex
  F  M
13 21
> res['F']
  F
13
>
```

#### 概括结果转为数据框

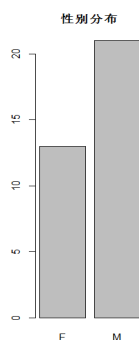
- 对单个分类变量, `table` 结果是一个有元素名的向量。
- 用 `as.data.frame` 函数把 `table` 的结果转为数据框:

```
> res <- with(d.cancer, table(sex))
> as.data.frame(res)
  sex Freq
1   F   13
2   M   21
```

#### 用概括结果绘制条形图

- `table` 作的单变量频数表可以用 `barplot` 表现为图形, 如:

```
barplot(res, main=' 性别分布')
```



### 列联表

- 对两个分类变量  $x_1$  和  $x_2$ , 其每个组合的出现次数可以用 `table(x1, x2)` 函数统计, 结果叫做列联表。
- 如

```
> res <- with(d.cancer, table(sex, type))
> res
      type
sex 鳞癌 腺癌
F      4    9
M     18    3
```

- 结果是一个类为 `table` 的二维数组 (矩阵), 每行以第一个变量  $x_1$  的各水平值为行名, 每列以第二个变量  $x_2$  的各水平值为列名。

### 列联表转为数据框

- 对两个分类变量, `table` 结果是一个矩阵。
- 用 `as.data.frame` 函数把 `table` 的结果转为数据框:

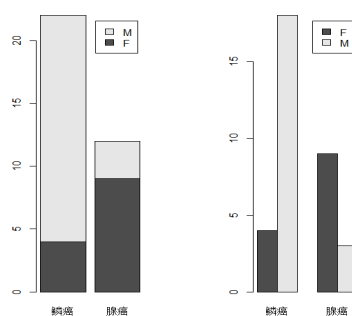
```
> res <- with(d.cancer, table(sex, type))
> as.data.frame(res)
  sex type Freq
1  F 鳞癌     4
2  M 鳞癌    18
```

3	F	腺癌	9
4	M	腺癌	3

### 对列联表绘制条形图

- 列联表的结果可以用条形图表示。如

```
barplot(res, legend=TRUE)
barplot(res, beside=TRUE, legend=TRUE)
```



### 列联表的变换

- 对于 `table` 的结果列联表，可以用 `addmargins` 函数增加行和与列和。如

```
> addmargins(res)
      type
sex  鳞癌 腺癌 Sum
F      4    9  13
M     18    3  21
Sum    22   12  34
```

- 用 `margin.table` 可以计算列联表行或列的和并返回，如

```
> margin.table(res, 1)
sex
  F  M
13 21
> margin.table(res, 2)
type
鳞癌 腺癌
  22  12
```

- 用 `prop.table(res)` 把一个列联表 `res` 转换成百分比表。如

```
> print(res)
      type
sex 鳞癌 腺癌
  F    4    9
  M   18    3
> prop.table(res)
      type
sex      鳞癌      腺癌
  F 0.11764706 0.26470588
  M 0.52941176 0.08823529
```

- 用 `prop.table(res,1)` 把列联表 `res` 转换成行百分比表。用 `prop.table(res,2)` 把列联表 `res` 转换成列百分比表。如

```
> prop.table(res, 1)
      type
sex      鳞癌      腺癌
  F 0.3076923 0.6923077
  M 0.8571429 0.1428571
> prop.table(res, 2)
      type
sex      鳞癌      腺癌
```

```
F 0.1818182 0.7500000
M 0.8181818 0.2500000
```

### 多个分类变量的交叉分类频数

- 在有多个分类变量时，用 `as.data.frame(table(x1, x2, x3))` 形成多个分类变量交叉分类的频数统计数据框。
- 比如，`d.cancer` 数据框中按年龄分为“中青年”和“老年”如下：

```
> d.cancer[, 'ageg'] <- cut(d.cancer[, 'age'],
+   breaks=c(0, 60, 150),
+   include.lowest=TRUE,
+   labels=c(' 中青年', ' 老年'))
```

- 有三个交叉分类变量的频数统计如下：

```
> cross3.tab <- with(d.cancer,
+   table(sex, ageg, type))
> cross3.tab
, , type = 鳞癌

   ageg
sex 中青年 老年
F    0     0
M    5     8

, , type = 腺癌

   ageg
sex 中青年 老年
F    1     6
M    1     2
```

- 用 `as.data.frame` 函数可以转换这样的结果为数据框：

```
> cross3.df <- as.data.frame(cross3.tab)
> cross3.df
   sex  ageg type Freq
1  F  中青年 鳞癌    0
```

2	M	中青年	鳞癌	5
3	F	老年	鳞癌	0
4	M	老年	鳞癌	8
5	F	中青年	腺癌	1
6	M	中青年	腺癌	1
7	F	老年	腺癌	6
8	M	老年	腺癌	2

- 用 `ftable` 函数可以转换这样的结果为最后一个分类变量变成列变量的表:

```
> ftable(cross3.tab)
      type 鳞癌 腺癌
sex ageg
F   中青年      0   1
     老年      0   6
M   中青年      5   1
     老年      8   2
```

3.2.3 数据框概括

数据框概括

- 用 `colMeans()` 对数据框或矩阵的每列计算均值, 用 `colSums()` 对数据框或矩阵的每列计算总和。用 `rowMeans()` 和 `rowSums()` 对矩阵的每行计算均值或总和。
- 数据框与矩阵有区别, 某些适用于矩阵的计算对数据框不适用, 例如矩阵乘法。用 `as.matrix()` 把数据框的数值子集转换成矩阵。

apply 函数

- 对矩阵, 用 `apply(x, 1, FUN)` 对矩阵 `x` 的每一行使用函数 `FUN` 计算结果, 用 `apply(x, 2, FUN)` 对矩阵 `x` 的每一列使用函数 `FUN` 计算结果。

- 如果 `apply(x,1,FUN)` 中的 `FUN` 对每个行变量得到多个  $m$  结果，结果将是一个矩阵，行数为  $m$ ，列数等于 `nrow(x)`。例如：

```
t(apply(as.matrix(iris[,1:4]), 2,
      function(x)
        c(n=sum(!is.na(x)),
          mean=mean(x, na.rm=TRUE),
          sd=sd(x, na.rm=TRUE))))
```

- 如果 `apply(x,2,FUN)` 中的 `FUN` 对每个列变量得到多个  $m$  结果，结果将是一个矩阵，行数为  $m$ ，列数等于 `ncol(x)`。

### 3.2.4 分类概括

#### 用 `tapply` 分组概括向量

- 用 `tapply` 函数进行分组概括：

```
tapply(X, INDEX, FUN)
```

其中 `X` 是一个向量，`INDEX` 是一个分类变量，`FUN` 是概括统计函数。

- 如

```
> with(d.cancer, tapply(v0, sex, mean))
      F      M
113.2354 108.1214
> with(d.cancer, tapply(v0, type, mean))
      鳞癌      腺癌
116.0000  99.2175
```

#### 用 `aggregate` 分组概括数据框

- `aggregate` 函数对输入的数据框用指定的分组变量（或交叉分组）分组进行概括统计。
- 如

```
> res <- aggregate(d.cancer[,c('age', 'v0', 'v1')],
+   list(sex=d.cancer[, 'sex']), mean, na.rm=TRUE)
> res
   sex      age      v0      v1
1  F 66.14286 113.2354 42.65538
2  M 63.25000 108.1214 45.95524
```

按性别分组计算其它变量的平均值。

- 可以同时计算多个概括统计量，如：

```
> res <- aggregate(d.cancer[,c('age', 'v0', 'v1')],
+   list(sex=d.cancer[, 'sex']), summary)
```

结果略。

- 可以交叉分组后概括，如

```
> res <- with(d.cancer,
+   aggregate(cbind(v0, v1),
+     list(sex=sex, type=type), mean))
> res
   sex type      v0      v1
1  F 鳞癌 126.99250 45.54750
2  M 鳞癌 113.55722 49.65556
3  F 腺癌 107.12111 41.37000
4  M 腺癌  75.50667 23.75333
```

### 用 split 函数分组后概括

- split 函数可以把数据框的各行按照一个或几个分组变量分为子集的列表，然后可以用 sapply 对每组进行概括。
- 如



```
> sp <- split(d.cancer[,c('v0','v1')],
+           d.cancer[, 'sex'])
> sapply(sp, colMeans)
           F           M
v0 113.23538 108.12143
v1  42.65538  45.95524
```

返回矩阵，行为变量 v0, v1，列为不同性别，值为相应的变量在各性别组的平均值。

- `colMeans` 函数计算数据框每列的平均值。

### 练习

- 把“patients.csv”读入“d.patients”中，并计算发病年龄、发病年、发病月、发病年月（格式如“200702”表示 2007 年 2 月份）。
- 把“现住地址国标”作为字符型，去掉最后两位，仅保留前 6 位数字，保存到变量“地址编码”中。
- （1）按照地址编码和发病年月交叉分类汇总发病人数，保存到数据框 d.pas1 中，然后保存为 CSV 文件“分区分年月统计.csv”中。要求结果有三列：“地址编码”、“发病年月”、“发病人数”。
- （2）按照地址编码和发病月分类汇总发病人数，保存到数据框 d.pas2 中，然后保存为 CSV 文件“分区分月统计.csv”中。要求每个地址编码占一行，各列为地址编码以及 1、2、……、12 各月份，每行为同一地址编码各月份的发病数。
- （3）按发病年月和性别汇总发病人数，并计算同年月不分性别的发病总人数。结果保存到数据框 d.pas3 中，然后保存到 CSV 文件“分年月分性别统计.csv”中。要求每个不同年月占一行，变量包括年月、男性发病数、女性发病数、总计。
- （4）分析病人的职业分布，保存到数据框 d.pas4 中，然后保存到 CSV 文件“职业构成.csv”中。要求各列为职业、发病人数、百分比（结果乘以 100 并保留一位小数）。

- (5) 把年龄分成 0—9, 11—19, …… , 70 以上各段, 保存为 “年龄段” 变量。用年龄段和性别交叉汇总发病人数和百分比 (结果乘以 100 并保留一位小数), 制作如下表格, 保存到 “年龄性别分布.csv” 中。

年龄段	男性		女性	
	发病数	发病率	发病数	发病率
0—9	4	0.2	2	0.1
10—19	18	0.9	11	0.5
20—29	111	5.3	47	2.2
.....				

## 第四章 R 绘图

### 4.1 常用图形

#### R 的图形支持

- R 支持多种图形系统，其中传统的是 base 部分的图形，比较新的有 lattice, ggplot2 等。
- 介绍 R 传统的 base 图形。
- 有两类图形函数：高级图形函数，直接针对某一绘图任务作出完整图形；低级图形函数，在已有图形上添加内容。
- 具备有限的与图形交互的能力（函数 locator 和 identify）。

#### 4.1.1 条形图

##### 条形图

- 设 “class.csv” 内容读入到了数据框 d.class 中，有 name, sex, age, height, weight 等变量。
- 用 `table(d.class[, 'sex'])` 可以统计数据框中变量 sex 的不同取值的频数。
- 用 `barplot` 函数可以对这样的频数结果绘制条形图。程序为

```
d.class <- read.csv('class.csv', header=TRUE)
res <- table(d.class[, 'sex']); print(res)
barplot(res)
```

## 性别的条形图



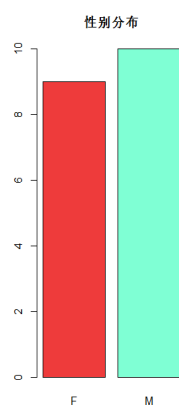
- 用 `main=` 选项指定一个标题。
- 可以自己指定不同条形的颜色。如

```
barplot(res, main=" 性别分布",  
        col=c("brown2", "aquamarine1"))
```

- R 的 `colors()` 函数返回 R 中可用的颜色名称表，如：

```
> head(colors(), 6)  
[1] "white"          "aliceblue"  
[3] "antiquewhite"   "antiquewhite1"  
[5] "antiquewhite2"  "antiquewhite3"
```

## 性别的条形图（调整）



### 列联表的条形图

- 设 d.class 中按把年龄 11, 12 归为'幼', 把年龄 13,14,15,16 归为'长'。  
办法如

```
> d.class[, 'ageg'] <- cut(d.class[, 'age'],
+   breaks=c(10, 12.5, 17),
+   labels=c('幼', '长')); d.class[, 'ageg']
[1] 长 长 长 幼 幼 长 幼 长 长 长 长
[13] 幼 长 幼 长 幼 幼 长
Levels: 幼 长
```

- cut 函数输入一个连续取值的变量, 给定划分区间端点 (包括最左和最右), 可以把连续取值的变量变成相应分段值。
- 可以用 table 函数按性别和年龄长幼交叉分组:

```
> res <- with(d.class, table(sex, ageg)); res
      ageg
sex 幼 长
F   3  6
M   4  6
```

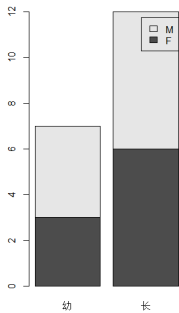
- 结果是一个矩阵表, 每行对应于 sex 的不同值, 每列对应于 ageg 的不同值。

列联表的条形图—分段

- 用如下程序把上述列联表用条形图表示：

```
barplot(res, legend=TRUE)
```

- 图中以列联表列变量为大类，每一大类画一个条形；以列联表行变量为小类，每一条形中按小类分段。

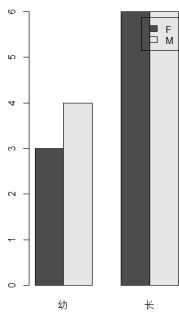


列联表的条形图—并排

- 用如下程序把上述列联表用条形图表示：

```
barplot(res, beside=TRUE, legend=TRUE)
```

- 图中以列联表列变量为大类，每一大类画一组条形；以列联表行变量为小类，在大类的组内按小类画频数条形图，各组条形左右并排分隔放置。

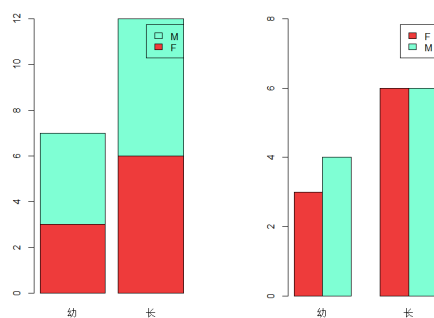


## 列联表的条形图—颜色

- 如下程序增加了小类颜色设置：

```
barplot(res, legend=TRUE,
        col=c("brown2", "aquamarine1"))
barplot(res, beside=TRUE, legend=TRUE,
        ylim=c(0, 8),
        col=c("brown2", "aquamarine1"))
```

第二个 barplot 还用了 ylim 选项使得标注与图形分隔开。

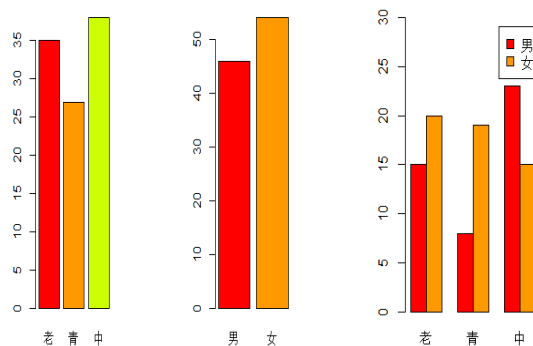


## 练习

- 设如下程序模拟生成了 100 个人的年龄组信息 g 和性别信息 s:

```
set.seed(1)
g <- sample(c('青', '中', '老'),
            size=100, replace=TRUE)
s <- sample(c('男', '女'),
            size=100, replace=TRUE)
```

- (1) 分别作年龄分布与性别分布的条形图。使用不同颜色。
- (2) 作年龄与性别交叉分组频数结果的条形图，以年龄作为大类，性别作为小类，作并排的条形图。男女的条形采用不同颜色。

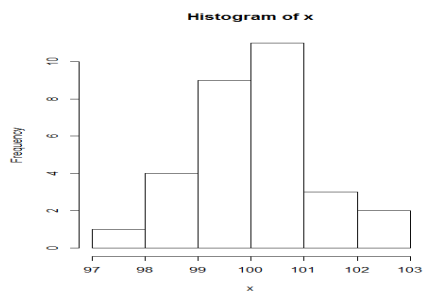


### 4.1.2 直方图和密度估计图

#### 直方图

- 如下程序模拟了正态分布  $N(100, 2^2)$  的 30 个点, 并用 `hist(x)` 对 `x` 作频数直方图, 直方图自动等距分段, 纵坐标为每段的样本点个数:

```
> x <- rnorm(30, mean=100, sd=1)
> print(round(x,2))
[1] 100.06 100.19 100.10 99.56 98.81
[6] 99.71 98.78 98.74 102.06 101.45
[11] 99.22 101.51 99.91 99.10 99.18
[16] 100.27 99.87 99.00 100.09 100.09
[21] 102.64 99.10 100.57 100.81 97.93
[26] 100.17 98.67 101.11 100.60 100.81
> hist(x)
```

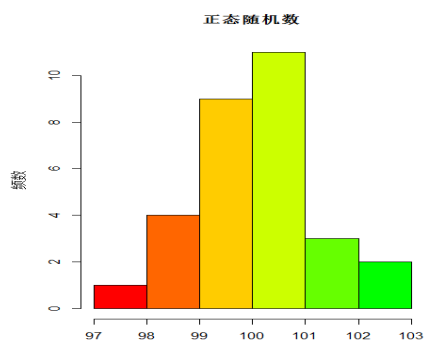




## 颜色、标题

- 指定了颜色、标题和 x、y 轴标签的程序:

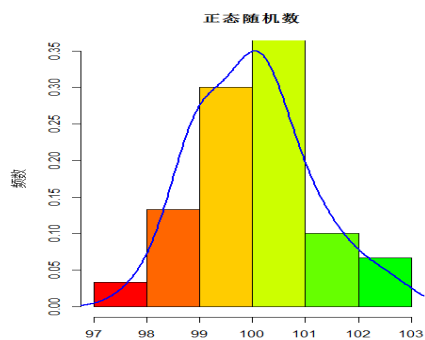
```
hist(x, col=rainbow(15),  
     main=' 正态随机数', xlab='', ylab=' 频数')
```



## 密度估计图

- 直方图可以用来估计分布密度，在 `hist` 中加入 `freq=FALSE` 选项，并可叠加用核密度估计方法做的密度估计曲线:

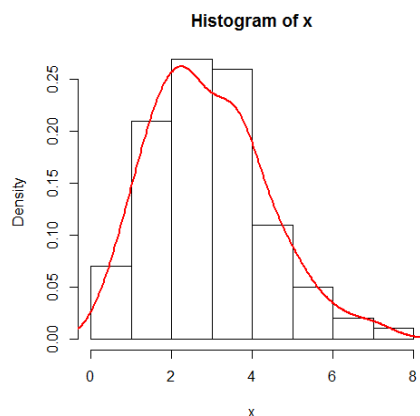
```
tmp.dens <- density(x)  
hist(x, freq=FALSE, ylim=c(0, max(tmp.dens$y)),  
     col=rainbow(15), main=' 正态随机数',  
     xlab='', ylab=' 频数')  
lines(tmp.dens, lwd=2, col='blue')
```



## 练习

- 仿照上面的程序编写一个自定义函数，输入自变量  $x$  (为数值型向量) 后自动做出类似的密度估计图，包括密度直方图和叠加的核密度估计曲线。
- 如下程序生成了 100 个服从伽马分布的随机数。用上面的自定义函数作密度估计图。

```
set.seed(1)
x <- rgamma(100, shape=3, rate=1)
```



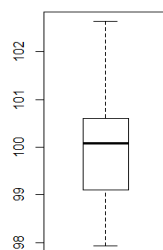
## 4.1.3 盒形图

## 盒形图 (boxplot)

- 盒形图可以简洁地表现变量分布，比如前面对直方图中的正态随机数  $x$ :

```
boxplot(x)
```

- 其中中间粗线是中位数，盒子上下边缘是  $\frac{3}{4}$  和  $\frac{1}{4}$  分位数，两条触须线延伸到取值区域的边缘。

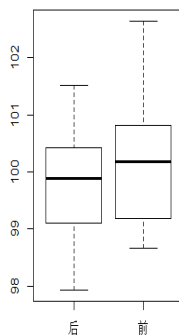


### 分组并排盒形图

- 假设前面的正态随机数  $x$  代表了 30 个学生的分数，现在又有变量  $g$  代表每个学生的座位前后，用如下程序模拟生成：

```
> g <- sample(c(' 前', ' 后'), size=30, replace=TRUE)
> g
[1] " 后" " 前" " 后" " 后" " 后" " 后" " 后" " 后"
[8] " 前" " 前" " 后" " 后" " 后" " 后" " 后" " 后"
[15] " 前" " 后" " 后" " 后" " 后" " 前" " 后" " 前"
[22] " 后" " 后" " 后" " 后" " 后" " 前" " 前" " 后"
[29] " 前" " 前"
> boxplot(x ~ g)
```

- 程序中 `boxplot` 对  $g$  的每一组画了一个  $x$  的盒形图，不同组左右并列。这种并排盒形图可以很容易地比较不同组的取值分布。

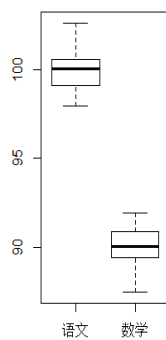


### 多变量并排盒形图

- 假设前面的正态随机数  $x$  代表了 30 个学生的语文分数，随机数  $y$  代表他们的数学分数（在下面的程序中模拟生成），如下的程序中 `boxplot` 并排地画了这两科成绩的盒形图：

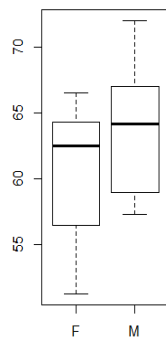
```
y <- rnorm(30, mean=90, sd=1)
boxplot(list('语文'=x, '数学'=y))
```

- 这种并排盒形图可以很容易地比较不同变量的取值分布。当然，这两个变量应该是可比的。



### 练习

- 对 `d.class` 数据框，分男女两组作身高的并排盒形图。



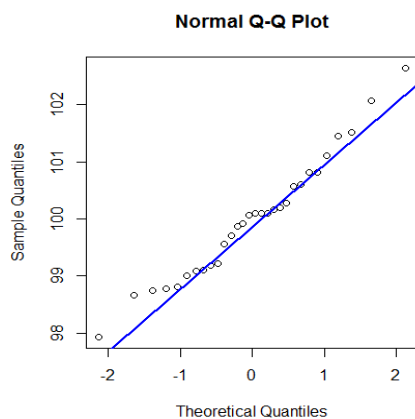
### 4.1.4 QQ 图

#### 正态 QQ 图

- 从直方图与盒形图可以模糊地判断可能的分布类型。
- 为了确定数值型变量是否来自正态分布，可以做正态 QQ 图。
- 比如前面的 30 正态随机数  $x$ ，做正态 QQ 图：

```
qqnorm(x)
qqline(x, lwd=2, col='blue')
```

- 图中散点与直线走向很接近时，可以认为数据是正态分布的。

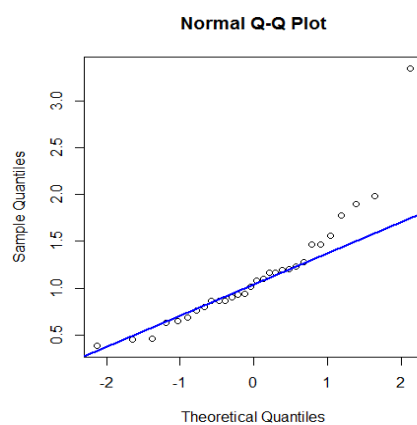


#### 正态 QQ 图—反例

- 下面的程序模拟生成了服从对数正态分布的 30 个随机数存入向量  $z$  中，并做正态 QQ 图：

```
z <- 10^rnorm(30, mean=0, sd=0.2)
qqnorm(z)
qqline(z, lwd=2, col='blue')
```

- 图中散点呈现碗底型弯曲，这是所谓“右偏”分布的特征。右偏分布的分布密度右尾长而左尾短。典型右偏分布包括居民收入、医院化验指标等。



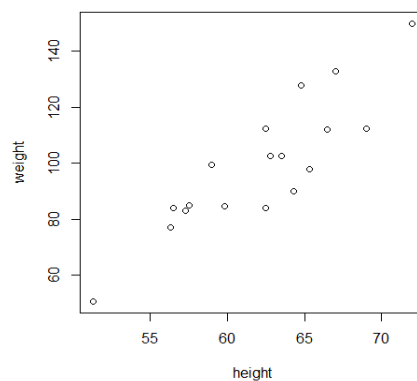
### 4.1.5 散点图

#### 散点图

- 考察数据框 `d.class` 中 19 个学生的身高和体重的关系，可以做散点图。
- 最简单的散点图：

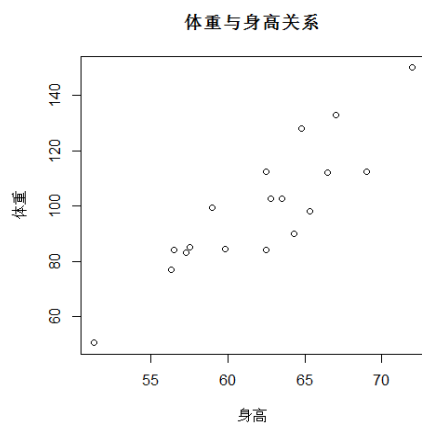
```
with(d.class, plot(height, weight))
```

- 横、纵坐标自动用变量名标注。



- 在 `plot` 函数内用 `main` 参数增加标题，用 `xlab` 参数指定横轴标注，用 `ylab` 参数指定纵轴标注，如

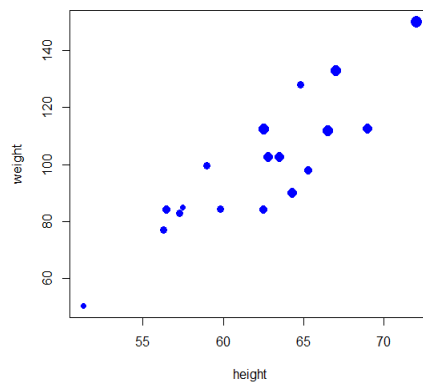
```
with(d.class,  
      plot(height, weight,  
            main=' 体重与身高关系 ',  
            xlab=' 身高', ylab=' 体重'))
```



- 用 `pch` 参数指定不同散点形状，用 `col` 参数指定颜色，用 `cex` 参数指定散点大小，如：

```
with(d.class,  
      plot(height, weight,  
            pch=16, col='blue',  
            cex=1 + (age - min(age))  
                  /(max(age)-min(age))))
```

- 这里点的大小代表了年龄大小。

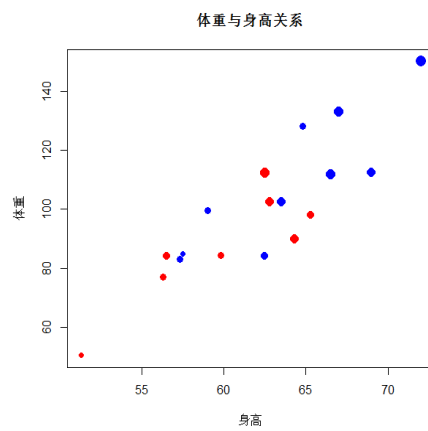


- 下面的程序做的散点图中红色点为女生，蓝色点为男生，点大小代表了年龄大小：

```
with(d.class,
      plot(height, weight,
            main=' 体重与身高关系',
            xlab=' 身高', ylab=' 体重',
            pch=16,
            col=ifelse(sex=='M', 'blue', 'red'),
            cex=1 + (age - min(age))
                  /(max(age)-min(age))))
```

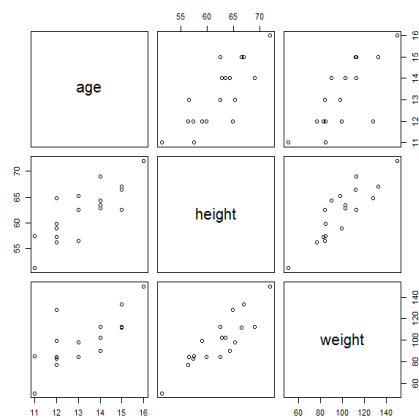
- 这里点的大小代表了年龄大小。





### 散点图矩阵

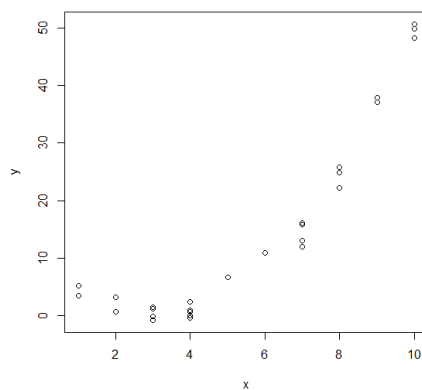
- `pairs` 函数对矩阵或数据框的各列两两做散点图，构成一个散点图矩阵，如：



### 练习

- 设如下程序生成了 `x` 和 `y` 变量，作散点图。

```
set.seed(1)
x <- sample(1:10, size=30, replace=TRUE)
y <- (x-3)^2 + rnorm(30, 0, 2)
```

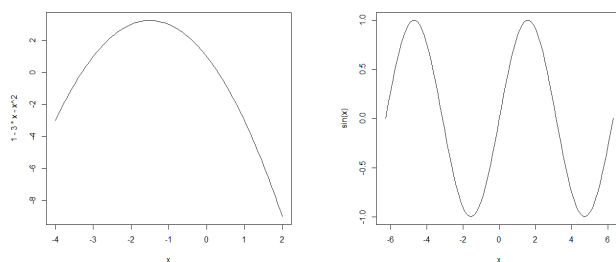


### 4.1.6 曲线图

#### 函数曲线

- `curve(expr, from, to)` 可以对以 `x` 为自变量的表达式做函数曲线, 或者对某个函数作函数曲线。如

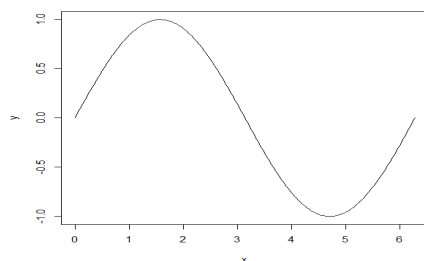
```
curve(1 - 3*x - x^2, -4, 2)
curve(sin, -2*pi, 2*pi)
```



#### 折线图

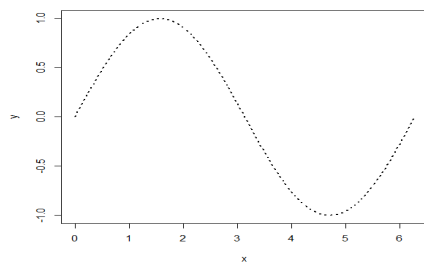
- 给定曲线上一组坐标点时, 把相邻点连接起来的折线图如果比较光滑也可以叫做曲线图。
- 在 `plot` 函数中使用 `type='l'` 参数可以作折线图, 如

```
x <- seq(0, 2*pi, length=200)
y <- sin(x)
plot(x,y, type='l')
```



- 除了仍可以用 `main`, `xlab`, `ylab`, `col` 等参数外, 还可以用 `lwd` 指定线宽度, `lty` 指定虚线, 如

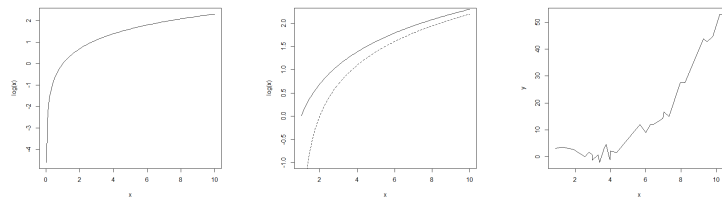
```
plot(x,y, type='l',
      lwd=2, lty=3)
```



### 练习

- 作  $\log(x)$  函数在  $(0.01, 10)$  范围的函数曲线。
- 在同一坐标系作  $\log(x)$  函数和  $\log(x-1)$  函数在  $(1.01, 10)$  范围的函数曲线。
- 设如下程序生成了  $x, y$  坐标对, 作折线图。

```
set.seed(1)
x <- (sample(1:10, size=30, replace=TRUE)
      + rnorm(30, 0, 0.5))
x <- sort(x)
y <- (x-3)^2 + rnorm(30, 0, 2)
```



#### 4.1.7 三维曲面图

##### 三维曲面图

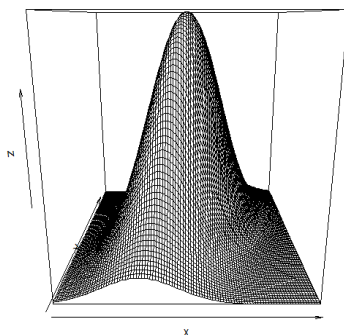
- 用 `persp` 函数作三维曲面图, `contour` 作等值线图, `image` 作色块图。
- 坐标 `x` 和 `y` 构成一张平面网格, 数据 `z` 是包含 `z` 坐标的矩阵, 每行对应一个横坐标, 每列对应一个纵坐标。
- 如

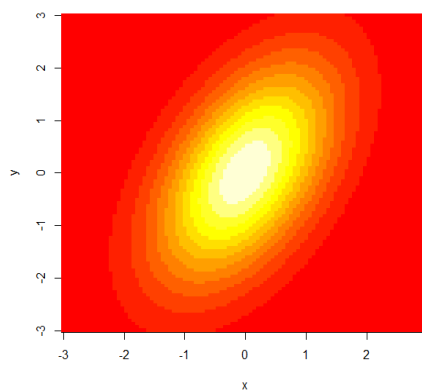
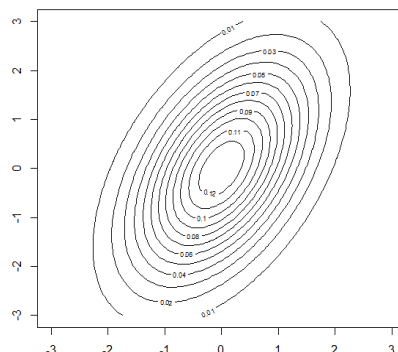
```
x <- seq(-3,3, length=100)
y <- x
f <- function(x,y,ssq1=1, ssq2=2, rho=0.5){
  det1 <- ssq1*ssq2*(1 - rho^2)
  s1 <- sqrt(ssq1)
  s2 <- sqrt(ssq2)
  1/(2*pi*sqrt(det1)) *
  exp(-0.5 / det1 * (
    ssq2*x^2 + ssq1*y^2 - 2*rho*s1*s2*x*y))
}
z <- outer(x, y, f)
persp(x, y, z)
contour(x, y, z)
image(x, y, z)
```

### 三维曲面图

- 用 `persp` 函数作三维曲面图, `contour` 作等值线图, `image` 作色块图。
- 坐标 `x` 和 `y` 构成一张平面网格, 数据 `z` 是包含 `z` 坐标的矩阵, 每行对应一个横坐标, 每列对应一个纵坐标。
- 如

```
x <- seq(-3,3, length=100)
y <- x
f <- function(x,y,ssq1=1, ssq2=2, rho=0.5){
  det1 <- ssq1*ssq2*(1 - rho^2)
  s1 <- sqrt(ssq1)
  s2 <- sqrt(ssq2)
  1/(2*pi*sqrt(det1)) *
  exp(-0.5 / det1 * (
    ssq2*x^2 + ssq1*y^2 - 2*rho*s1*s2*x*y))
}
z <- outer(x, y, f)
persp(x, y, z)
contour(x, y, z)
image(x, y, z)
```





## 4.2 图形定制

### 4.2.1 低级图形函数

#### 低级图形函数

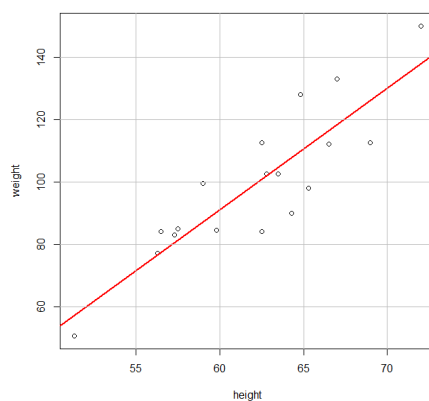
- `barplot`, `plot`, `hist`, `boxplot` 等函数可以直接绘制一幅新图形，称为高级图形函数。
- `abline`, `points`, `lines` 等函数在已有图形中添加新内容，称为低级图形函数。

#### 用 `abline` 函数增添直线

- 用 `abline` 函数在图中增加直线。可以用参数 `a`, `b` 指定截距和斜率，或用参数 `v` 指定横坐标画竖线，用参数 `h` 指定纵坐标画水平线。

- 如

```
with(d.class, plot(height, weight))
abline(-143, 3.9, col="red", lwd=2)
abline(v=c(55,60,65,70), col="gray")
abline(h=c(60,80,100,120,140), col="gray")
```



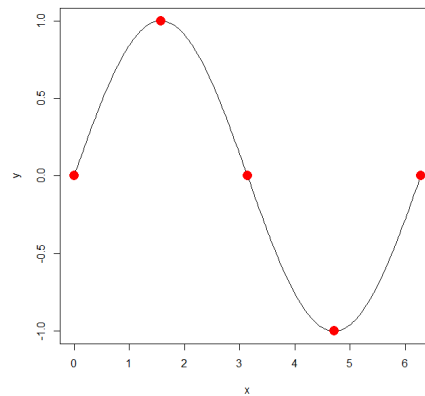
### 用 points 函数增添散点

- 用 `points(x,y)` 在指定的坐标位置增加散点
- 如

```
x <- seq(0, 2*pi, length=200)
y <- sin(x)
special <- list(x=(0:4)*pi/2, y=sin((0:4)*pi/2))
plot(x, y, type='l')
points(special$x, special$y,
       col="red", pch=16, cex=2)
```

- 选项 `pch` 指定一种绘点符号（可选 1:16），`cex` 指定绘点符号大小倍数。
- 因为上面定义的 `special` 是一个包含变量 `x, y` 的列表，所以最后的 `points` 语句也可以写成：

```
points(special, col="red", pch=16, cex=2)
```

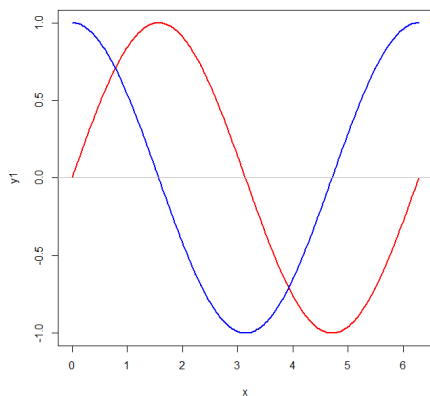


### 用 lines 函数增添线

- 用 `points` 函数增加散点，用 `lines` 函数增加曲线。
- 如

```
x <- seq(0, 2*pi, length=200)
y1 <- sin(x)
y2 <- cos(x)
plot(x, y1, type='l', lwd=2, col="red")
lines(x, y2, lwd=2, col="blue")
abline(h=0, col='gray')
```



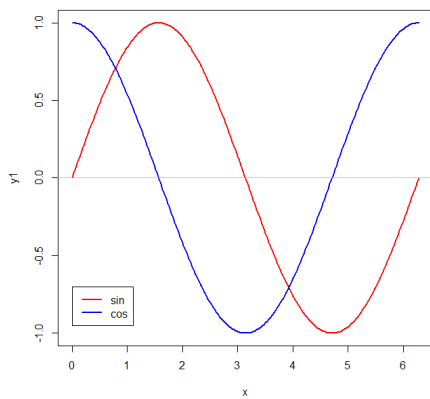


### 用 legend 函数增加图例

- 可以用 `legend` 函数增加图例，比如上个正弦、余弦曲线图，为了区分两条曲线，用如下语句增加图例：

```
legend(0, -0.7, col=c("red", "blue"),  
      lty=c(1,1), lwd=c(2,2),  
      legend=c("sin", "cos"))
```

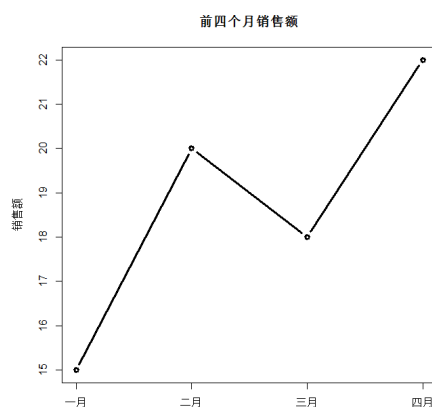
- `legend` 前两个参数是图例的横纵坐标位置，然后用 `col`、`lty`、`lwd`、`legend` 参数分别规定各曲线的颜色、线型、粗细、标注内容。
- 图例也可以对线和符号混合的情形进行标注。



### 用 axis 函数控制坐标轴

- 在 plot 函数中用 axes=FALSE 可以取消自动的坐标轴。
- 用 box 函数画坐标边框。
- 用 axis 函数单独绘制坐标轴。axis 的第一个参数取 1, 2, 3, 4, 分别表示横轴、纵轴、上方和右方。
- axis 的参数 at 为刻度线位置, labels 为标签。
- 如

```
x <- c(' 一月 '=15, ' 二月 '=20,  
      ' 三月 '=18, ' 四月 '=22)  
plot(seq(along=x), x, axes=FALSE,  
      type='b', lwd=3,  
      main=' 前四个月销售额',  
      xlab='', ylab=' 销售额')  
box(); axis(2)  
axis(1, at=seq(along=x), labels=names(x))
```

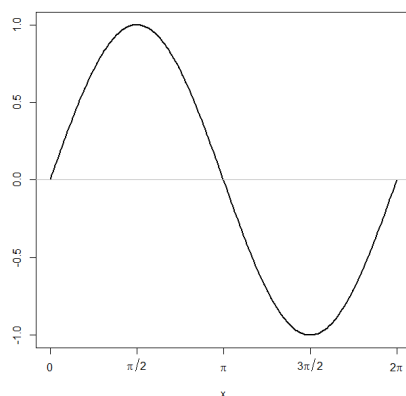


### 数学公式

- R 的 expression 支持把数学公式显示到坐标轴中。比如:

```
x <- seq(0, 2*pi, length=200)
y1 <- sin(x)
plot(x, y1, type='l', lwd=2,
      axes=FALSE,
      xlab='x', ylab='')
abline(h=0, col='gray')
box()
axis(2)
axis(1, at=(0:4)/2*pi,
      labels=c(0, expression(pi/2),
               expression(pi), expression(3*pi/2),
               expression(2*pi)))
```

- 详见 R 中 plotmath 的帮助。



### 用 text 函数增添文字

- 用 text 函数在坐标区域内标注文本。前两个参数是坐标位置，第三个参数 labels 是要标注的文字。
- 如

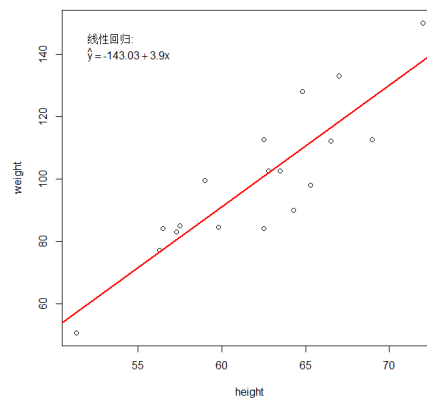
```
with(d.class, plot(height, weight))
lm1 <- lm(weight ~ height, data=d.class)
abline(lm1, col='red', lwd=2)
```

```

a <- coef(lm1)[1]
b <- coef(lm1)[2]
text(52, 145, adj=0, ' 线性回归:')
text(52, 140, adj=0,
      substitute(hat(y) == a + b*x,
                  list(a=round(coef(lm1)[1], 2),
                       b=round(coef(lm1)[2], 2))))

```

- 如果要标注数学公式，做法比较复杂，详见关于 plotmath 的帮助。



### locator 函数

- `locator` 函数在执行时等待用户在图形的坐标区域内点击并返回点击处的坐标。可以用参数 `n` 指定要点击的点的个数。不指定个数则需要用右键菜单退出。如

```

x <- seq(0, 2*pi, length=200)
y1 <- sin(x); y2 <- cos(x)
plot(x, y1, type='l',
      col="red")
lines(x, y2, col="blue")
legend(locator(1), col=c("red", "blue"),
       lty=c(1,1), legend=c("sin", "cos"))

```

- 这也可以用来要求用户点击以进行到下一图形。

### locator 函数

- `locator` 函数在执行时等待用户在图形的坐标区域内点击并返回点击处的坐标。可以用参数 `n` 指定要点击的点的个数。不指定个数则需要用右键菜单退出。如

```
x <- seq(0, 2*pi, length=200)
y1 <- sin(x); y2 <- cos(x)
plot(x, y1, type='l',
      col="red")
lines(x, y2, col="blue")
legend(locator(1), col=c("red", "blue"),
       lty=c(1,1), legend=c("sin", "cos"))
```

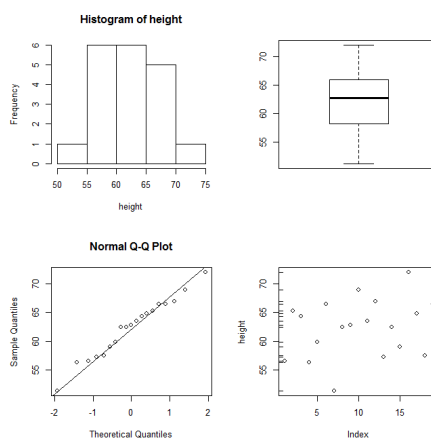
- 这也可以用来要求用户点击以进行到下一图形。

### 4.2.2 图形参数

#### 图形参数

- 用图形参数可以选择点的形状、颜色、线型、粗细、坐标轴做法、边空、一页多图等。
- 有些参数直接用在绘图函数内，如 `plot` 函数可以用 `pch`、`col`、`cex`、`lty`、`lwd` 等参数。
- 有些图形参数必须使用 `par` 函数指定。`par` 函数指定图形参数并返回原来的参数值，所以在修改参数值作图后通常应该恢复原始参数值，做法如

```
opar <- par(mfrow=c(2,2))
with(d.class, {hist(height);
  boxplot(height);
  qqnorm(height); qqline(height);
  plot(height); rug(height,side=2)})
par(opar)
```



### 图形参数分类

- 图形元素控制;
- 坐标轴与坐标刻度;
- 图形边空;
- 一页多图。

### 图形元素参数示例

- `pch=16` 参数。散点符号, 取 0 ~ 18 的数。
- `lty=2` 参数。线型, 1 为实线, 从 2 开始为各种虚线。
- `lwd=2` 参数, 线的粗细, 标准粗细为 1。
- `col="red"` 参数, 颜色, 可以是数字 1 ~ 8, 或颜色名字字符串如 "red", "blue" 等。用 `colors()` 函数查询有名字的颜色。用 `rainbow(n)` 函数产生连续变化的颜色。
- `font=2` 参数, 字体, 一般 `font=1` 是正体, 2 是粗体, 3 是斜体, 4 是粗斜体。
- `adj=-0.1` 指定文本相对于给定坐标的对齐方式。取 0 表示左对齐, 取 1 表示右对齐, 取 0.5 表示居中。此参数的值实际代表的是出现在给定坐标左边的文本的比例。
- `cex=1.5` 绘点符号大小倍数, 基本值为 1。

## 坐标轴与坐标刻度参数示例

- `mgp=c(3,1,0)` 坐标轴的标签、刻度值、坐标轴线到实际的坐标轴位置的距离，以行高为单位。经常用来缩小坐标轴所占的空间，如 `mgp=c(1.5, 0.5, 0)`。
- `lab=c(5,7,12)` 提供刻度线多少的建议，第一个数为 x 轴刻度线个数，第二个数为 y 轴刻度线个数，第三个数是坐标刻度标签的字符宽度。
- `las=1` 坐标刻度标签的方向。0 表示总是平行于坐标轴，1 表示总是水平，2 表示总是垂直于坐标轴。
- `tck=0.01` 坐标轴刻度线长度，以绘图区域大小为单位 1。

- `xaxs="s", yaxs="e"`: 控制 x 轴和 y 轴标刻度的方法。

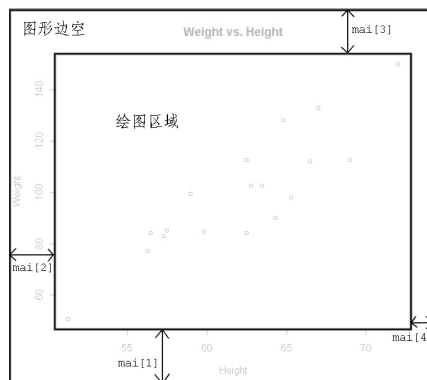
取"`s`"(即 standard) 或"`e`"(即 extended) 的时候数据范围控制在最小刻度和最大刻度之间。取"`e`" 时如果有数据点十分靠近边缘轴的范围会略微扩大。

取值为"`i`" (即 internal) 或"`r`" (此为缺省) 使得刻度线都落在数据范围围内部，而"`r`" 方式所留的边空较小。

取值设为"`d`" 时会锁定此坐标轴，后续的图形都使用与它完全相同的坐标轴，这在要生成一系列可比较的图形的时候是有用的。要解除锁定需要把这个图形参数设为其它值。

## 图形边空

- 一个单独的图由绘图区域 (绘图的点、线等画在这个区域中) 和包围绘图区域的边空组成，边空中可以包含坐标轴标签、坐标轴刻度标签、标题、小标题等，绘图区域一般被坐标轴包围。

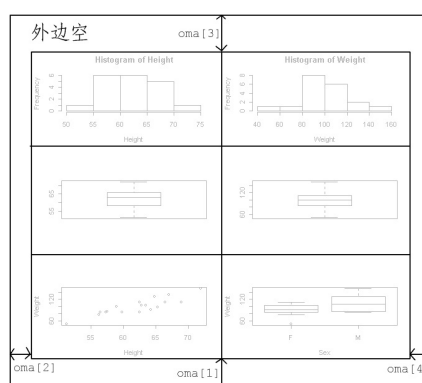


- 边空的大小由 `mai` 参数或 `mar` 参数控制, 它们都是四个元素的向量, 分别规定下方、左方、上方、右方的边空大小, 其中 `mai` 取值的单位是英寸, 而 `mar` 的取值单位是文本行高度。
- 例如:

```
opar <- par(mar=c(2,2,0.5,0.5),
             mgp=c(0.5, 0.5, 0), tck=0.005)
with(d.class, plot(height, weight,
                   xlab='', ylab=''))
par(opar)
```

### 一页多图

- R 可以在同一页面开若干个按行、列排列的窗格, 在每个窗格中可以作一幅图。
- 每个图有自己的内边空, 而所有图的外面可以包一个“外边空”。
- 如



- 一页多图用 `mfrow` 参数或 `mfcol` 参数规定, 如:

```
opar <- par(mfrow=c(2,2),
             oma=c(0,0,2,0))
```



```
with(d.class, {hist(height);  
  boxplot(height);  
  qqnorm(height); qqline(height);  
  plot(height); rug(height,side=2)})  
mtext(side=3, text=' 身高分布', cex=2, outer=T)  
par(opar)
```

- 用 `oma` 指定四个外边空的行数。
- 用 `mtext` 加 `outer=T` 指定在外边空添加文本。如果没有 `outer=T` 则在内边空添加文本。

### 4.2.3 图形设备

#### 图形设备

- 只要启用了高级绘图函数会自动选用当前绘图设备，缺省为屏幕窗口。
- 用 `pdf` 函数可以指定输出到 PDF 文件。如

```
pdf(file='fig-hw.pdf', height=10/2.54,  
    width=10/2.54, family='GB1')  
with(d.class, plot(height, weight,  
                    main=' 体重与身高关系'))  
dev.off()
```

- 用 `dev.off()` 关闭当前设备并生成输出文件（如果是屏幕窗口则没有保存结果）。
- 用 `jpeg` 函数启用 JPEG 图形设备。
- 用 `bmp` 函数启用 BMP 图形设备。
- 用 `png` 函数启用 PNG 图形设备。
- 用 `postscript` 函数启用 PostScript 图形设备。

### 4.2.4 更多中文字体

#### 用 showtext 扩展包支持更多字体

- 在 R 图形中，如果是窗口图形，可以使用汉字，但是不能指定字体。如果是 PDF 设备，使用 `family='GB1'` 可以使用中文字体，但是只有一种。
- 为了能够使用 MS Windows 系统字体，一种办法是安装 showtext 包。该包替换画图时的添加文本函数命令，把文本内容替换成多边形（PDF 或 PS 图）或点阵（点阵图）。
- 安装 showtext 包需要作的设置：
- 查看 Windows 的 font 目录内容，看文件名与字体名的对应关系。
- 找到自己希望使用的中文字体的文件名。
- 用 `font.add()` 命令，增加一套自定义字体 family，一套中可以指定四种：常规 (regular), 粗体 (bold), 斜体 (italic), 粗斜体 (bolditalic)。
- 程序中用 `require` 调入 showtext 包并运行 `showtext.auto()` 命令，这个命令使得文本命令采用 showtext 包。
- 用 `par(family=)` 指定自定义的字体 family。
- 作图（主要是 PDF），然后关闭图形设备。
- 下面是一个例子：

```
test.chinese <- function(){
  require(showtext); showtext.auto()

  ## 建立文件名到字体名对照表
  fmap <- c(
    'msyh'=' 微软雅黑常规',
    'msyhbd'=' 微软雅黑粗体',
    'msyhl'=' 微软雅黑细体',
    'simsun'=' 宋体',
    'simfang'=' 仿宋',
    'simkai'=' 楷体',
    'simhei'=' 黑体',
```

```

'SIMLI'=' 隶书',
'SIMYOU'=' 幼圆',
'STSONG'=' 华文宋体',
'STZHONGS'=' 华文中宋',
'STFANGSO'=' 华文仿宋',
'STKAITI'=' 华文楷体',
'STXIHEI'=' 华文细黑',
'STLITI'=' 华文隶书',
'STXINGKA'=' 华文行楷',
'STXINWEI'=' 华文新魏',
'STCAIYUN'=' 华文彩云',
'STHUPO'=' 华文琥珀'
)

fmapr <- names(fmap); names(fmapr) <- unname(fmap)
cat('==== 字体文件名与字体名称对应:\n')
print(fmap)
cat('==== 字体名与字体文件名对应:\n')
print(fmapr)

## 找到某个字体的字体文件
## font.name 是字体名称
find.font <- function(font.name){
  fname <- fmapr[font.name]
  flist0 <- font.files()
  flist1 <- sapply(strsplit(flist0, '[.]'),
    function(it) it[1])
  flist0[flist1==fname]
}

ff1 <- find.font(' 宋体')
ff2 <- find.font(' 黑体')
ff3 <- find.font(' 仿宋')
ff4 <- find.font(' 隶书');

font.add('cjk4',
  regular=ff1,
  bold=ff2,

```

```
        italic=ff3,
        bolditalic=ff4)
##browser()

pdf('test-chinese.pdf'); on.exit(dev.off())
par(family='cjk4')

plot(c(0,1), c(0,1), type='n',
      axes=FALSE, xlab='', ylab='')
text(0.1, 0.9, ' 正体', font=1)
text(0.1, 0.8, ' 粗体', font=2)
text(0.1, 0.7, ' 斜体', font=3)
text(0.1, 0.6, ' 粗斜体', font=4)
## 注意: 图形参数 font=1 表示正体,
## font=2 表示粗体, font=3 表示斜体, font=4 表示粗斜体
}
test.chinese()
```

# 第五章 用 R 作随机模拟

## 5.1 随机模拟

### 5.1.1 介绍

#### 随机模拟

- 随机模拟是统计研究的重要方法，另外许多现代统计计算方法（如 MCMC）也是基于随机模拟的。
- R 中提供了多种不同概率分布的随机数函数，可以批量地产生随机数。
- 一些 R 扩展包利用了随机模拟方法，如 boot 包进行 bootstrap 估计。

#### 随机数

- 所谓随机数，实际是“伪随机数”，是从一组起始值（称为种子），按照某种递推算法向前递推得到的。所以，从同一种子出发，得到的随机数序列是相同的。
- 为了得到可重现的结果，随机模拟应该从固定不变的种子开始模拟。用 `set.seed(k)` 指定一个编号为 k 的种子，这样每次从编号 k 种子运行相同的模拟程序可以得到相同的结果。
- 还可以用 `set.seed()` 加选项 `kind=` 指定后续程序要使用的随机数发生器名称，用 `normal.kind=` 指定要使用的正态分布随机数发生器名称。

#### R 的随机数函数

- R 提供了多种分布的随机数函数，如 `runif(n)` 产生 n 个标准均匀分布随机数，`rnorm(n)` 产生 n 个标准正态分布随机数。
- 例如：

```
> round(runif(5), 2)
[1] 0.44 0.56 0.93 0.23 0.22
> round(rnorm(5), 2)
[1] -0.20 1.10 -0.02 0.16 2.02
```

注意因为没有指定种子，每次运行会得到不同的结果。

- 在 R 命令行运行

```
?Distributions
```

可以查看 R 中提供的不同概率分布。

### sample() 函数

- `sample()` 函数从一个有限集合中无放回或有放回地随机抽取，产生随机结果。
- 例如，为了设随机变量  $X$  取值于 {正面, 反面}，且  $P(X = \text{正面}) = 0.7 = 1 - P(X = \text{反面})$ ，如下程序产生  $X$  的 10 个随机抽样值：

```
> sample(c(' 正面', ' 反面'), size=10,
+        prob=c(0.7, 0.3), replace=TRUE)
[1] " 反面" " 反面" " 反面" " 反面" " 正面"
[6] " 正面" " 正面" " 正面" " 反面" " 反面"
```

- 选项 `size` 指定抽样个数，`prob` 指定每个值的概率，`replace=TRUE` 说明是有放回抽样。
- 如果要做无放回等概率的随机抽样，可以不指定 `prob` 和 `replace` (缺省是 `FALSE`)。
- 比如，下面的程序从 1:10 随机抽取 4 个：

```
> sample(1:10, size=4)
[1] 1 5 8 10
```

- 如果要从  $1:n$  中等概率无放回随机抽样直到每一个都被抽过，只要用如：

```
> sample(10)
[1] 3 5 9 2 10 7 4 1 6 8
```

这实际上返回了  $1:10$  的一个重排。

### 5.1.2 随机模拟示例

随机模拟示例：线性回归模拟

- 考虑如下线性回归模型

$$y = 10 + 2x + \varepsilon, \varepsilon \sim N(0, 0.5^2).$$

- 假设有样本量  $n = 10$  的一组样本，R 函数 `lm()` 可以得到截距  $a$ , 斜率  $b$  的估计  $\hat{a}, \hat{b}$ , 以及相应的标准误差  $SE(\hat{a}), SE(\hat{b})$ 。样本可以模拟产生。

产生模拟样本

- 模型中的自变量  $x$  可以用随机数产生，比如，用 `sample()` 函数从  $1:10$  中随机有放回地抽取  $n$  个。
- 模型中的随机误差项  $\varepsilon$  可以用 `rnorm()` 产生。
- 产生一组样本的程序如：

```
x <- sample(1:10, size=n, replace=TRUE)
eps <- rnorm(n, 0, 0.5)
y <- a + b * x + eps
```

## 计算回归结果

- 如下程序计算线性回归:

```
lm(y ~ x)
```

- 如下程序计算线性回归的多种统计量:

```
summary(lm(y ~ x))
```

- 如下程序返回一个矩阵, 包括  $a, b$  的估计值、标准误差、t 检验统计量、检验 p 值:

```
summary(lm(y ~ x))$coefficients
```

- 如下程序把上述矩阵的前两列拉直成一个向量返回:

```
c(summary(lm(y ~ x))$coefficients[,1:2])
```

这样得到  $\hat{a}, \hat{b}, SE(\hat{a}), SE(\hat{b})$  这四个值。

## 用 replicate() 进行重复模拟

- 用 replicate(次数, 复合语句) 执行多次模拟, 返回向量或矩阵结果, 返回矩阵时, 每列是一次模拟的结果。
- 下面是线性回归整个模拟程序, 写成了一个函数。

```
reg.sim <- function(a=10, b=2, sigma=0.5,
                    n=10, B=1000){
  set.seed(1)
  resm <- replicate(B, {
    x <- sample(1:10, size=n, replace=TRUE)
    eps <- rnorm(n, 0, 0.5)
    y <- a + b * x + eps
    c(summary(lm(y ~ x))$coefficients[,1:2])
  })
}
```



```
  })  
  resm <- t(resm)  
  colnames(resm) <- c('a', 'b', 'SE.a', 'SE.b')  
  cat(B, ' 次模拟的平均值:\n')  
  print( apply(resm, 2, mean) )  
  cat(B, ' 次模拟的标准差:\n')  
  print( apply(resm, 2, sd) )  
}
```

### 运行结果

- 结果:

```
1000 次模拟的平均值:  
          a          b          SE.a          SE.b  
9.99624063 1.99889264 0.36333861 0.05948963  
1000 次模拟的标准差:  
          a          b          SE.a          SE.b  
0.39281551 0.06369336 0.12802849 0.01945043
```

- 可以看出, 标准误差作为  $\hat{a}, \hat{b}$  的标准差估计, 与多次模拟得到多个  $\hat{a}, \hat{b}$  样本计算得到的标准差估计是比较接近的。



## 第六章 用 Rcpp 连接 C++ 代码

### 6.1 Rcpp 介绍

#### 高性能计算

- 向量化编程;
- apply 族函数;
- 内建函数 sum, prod, cumsum, cumprod, filter, fft 等。
- Rcpp 包把低效部分变成 C++ 代码。
- 并行计算。

#### Rcpp

- Rcpp 可以很容易地把 C++ 代码与 R 程序连接在一起，可以从 R 中直接调用 C++ 代码而不需要用户关心那些繁琐的编译、链接、接口问题。可以在 R 数据类型和 C++ 数据类型之间容易地转换。
- 安装要求：除了要安装 Rcpp 包之外，MS Windows 用户还需要安装 RTools 包，这是用于 C, C++, Fortran 程序编译链接的开发工具包，是自由软件。用户的应用程序路径 (PATH) 中必须有 RTools 包可执行程序的路径 (安装 RTools 可以自动设置)。
- Rcpp 支持把 C++ 代码写在 R 源程序文件内，执行时自动编译连接调用；也支持把 C++ 代码保存在单独的源文件中，执行 R 程序时自动编译连接调用；对较复杂的问题，应制作 R 扩展包，利用构建 R 扩展包的方法实现 C++ 代码的编译连接，这时接口部分也可以借助 Rcpp 属性功能或模块功能完成。

## Rcpp 的用途

- 把已经用 R 代码完成的程序中运行速度瓶颈部分改写成 C++ 代码，提高运行效率。
- 对于 C++ 或 C 程序源代码或二进制代码提供的函数库，可以用 Rcpp 编写 C++ 界面程序进行 R 与 C++ 程序的输入、输出的传送，并在 C++ 界面程序中调用外来的函数库。
- 注意，用 Rcpp 编写 C++ 程序，不利于把程序脱离 R 运行或被其他的 C++ 程序调用。当然，可以只把 Rcpp 作为界面，主要的算法引擎完全不用 Rpp 的数据类型。
- RInside 扩展包支持把 R 嵌入到 C++ 主程序中。

### 6.1.1 简单样例

#### 用 cppFunction() 转换简单的 C++ 函数—Fibonacci 例子

- 考虑用 C++ 程序计算 Fibonacci 数的问题。Fibonacci 数满足  $f_0 = 0, f_1 = 1, f_t = f_{t-1} + f_{t-2}$ 。
- 可以使用如下 R 代码，其中有一部分 C++ 代码，用 cppFunction 转换成了 R 可以调用的同名 R 函数。

```
cppFunction(code='
  int fibonacci(const int x){
    if(x < 2) return x;
    else
      return ( fibonacci(x-1) + fibonacci(x-2) );
  }
')
print(fibonacci(5))
```

- 编译、链接、导入是在后台由 Rcpp 控制自动进行的，不需要用户去设置编译环境，也不需要用户执行编译、链接、导入 R 的工作。
- 在没有修改 C++ 程序时，同一 R 会话期间重新运行不必重新编译。
- 上面的 Fibonacci 函数仅接受标量数值作为输入，不允许向量输入。

- 从算法角度评价，这个算法是极其低效的，其算法规模是  $O(2^n)$ ， $n$  是自变量值。

用 `sourceCpp()` 转换 C++ 程序——正负交替迭代例子

- 设  $x_t, t = 1, 2, \dots, n$  保存在 R 向量 `x` 中，令

$$y_1 = x_1 \quad (6.1)$$

$$y_t = (-1)^t y_{t-1} + x_t, \quad t = 2, \dots, n \quad (6.2)$$

- 希望用 C++ 函数对输入序列 `x` 计算输出 `y`，并用 R 调用这样的函数。
- 下面的程序用 R 函数 `sourceCpp()` 把保存在 R 字符串中的 C++ 代码编译并转换为同名的 R 函数。

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector iters(NumericVector x){
    int n = x.size();
    NumericVector y(n);

    y[0] = x[0];
    double sign=-1;
    for(int t=1; t<n; t++){
        sign *= -1;
        y[t] = sign*y[t-1] + x[t];
    }

    return y;
}
')
print(iters(1:5))
```

## 说明

- C++ 程序可以直接写在 R 程序文件内 (保存为 R 多行字符串), 用 `sourceCpp()` 函数编译;
- Rcpp 包为 C++ 提供了一个 `NumericVector` 数据类型, 用来存储数值型向量。用成员函数 `size()` 访问其大小, 用方括号下标访问其元素。
- C 程序中定义的函数可以返回 `NumericVector` 数据类型, 将自动转换为 R 的数值型向量。
- 特殊的注释 `//[[Rcpp::export]]` 用来指定哪些 C++ 函数是要转换为 R 函数的。这叫做 Rcpp 属性 (attributes) 功能。

用 `sourceCpp()` 转换 C++ 源文件中的程序—正负交替迭代例子

- 直接把 C++ 代码写在 R 源程序内部的好处是不用管理多个源文件, 缺点是当 C++ 代码较长时, 不能利用专用 C++ 编辑环境和调试环境, 出错时显示的错误行号不好定位, 而且把代码保存在 R 字符串内, C++ 代码中用到字符时需要特殊语法。
- 稍复杂的 C++ 代码应该放在单独的 C++ 源文件内。
- 假设上面的 `iters` 函数的 C++ 代码单独存入了一个 `iters.cpp` 源文件中。用如下的 `sourceCpp()` 函数把 C++ 源文件中代码编译并转换为 R 可访问的同名函数, 测试调用:

```
> sourceCpp(file='iters.cpp')
> print(iters(1:5))
[1] 1 3 0 4 1
```

用 `sourceCpp()` 转换 C++ 源程序文件—卷积例子

- 考虑向量  $\mathbf{x} = (x_0, x_1, \dots, x_{n-1})$ ,  $\mathbf{y} = (y_0, y_1, \dots, y_{m-1})$ , 定义  $\mathbf{x}$  与  $\mathbf{y}$  的离散卷积  $\mathbf{z} = (z_0, z_1, \dots, z_{n+m-2})$  为

$$z_k = \sum_{(i,j): i+j=k} x_i y_j = \sum_{i=\max(0, k-m+1)}^{\min(k, n)} x_i y_{k-i}. \quad (6.3)$$



卷积演示 (2)

$x$		1	2	3	4	5	
$\text{rev}(y)$	3	2	1				
<hr/>							
$\times, +$		2	2				$\rightarrow 4$

卷积演示 (3)

$x$		1	2	3	4	5	
$\text{rev}(y)$	3	2	1				
<hr/>							
$\times, +$		3	4	3			$\rightarrow 10$

卷积演示 (4)

$x$		1	2	3	4	5	
$\text{rev}(y)$			3	2	1		
<hr/>							
$\times, +$			6	6	4		$\rightarrow 16$

卷积演示 (5)

$x$		1	2	3	4	5	
$\text{rev}(y)$				3	2	1	
<hr/>							
$\times, +$				9	8	5	$\rightarrow 22$

卷积演示 (6)

$x$		1	2	3	4	5	
$\text{rev}(y)$					3	2	1
<hr/>							
$\times, +$					12	10	$\rightarrow 22$



## 卷积演示 (7)

$$\begin{array}{rcccccc}
 & x & & 1 & 2 & 3 & 4 & 5 \\
 \text{rev}(y) & & & & & & 3 & 2 & 1 \\
 \hline
 \times, + & & & & & & 15 & & \rightarrow 15
 \end{array}$$

## 6.1.2 R 与 C++ 的类型转换

用 `wrap()` 把 C++ 变量返回到 R 中

- 在 R API 中用 `.Call()` 函数调用 C 程序库函数时，R 对象的数据类型一般是 `SEXP`。
- `Rcpp` 提供了模板化的 `wrap()` 函数把 C++ 的函数返回值转换成 R 的 `SEXP` 数据类型。此函数的声明为

```
template <typename T> SEXP wrap(const T& object);
```

`wrap()` 能转换的类型

- 把 `int`, `double`, `bool` 等基本类型转换为 R 的原子向量类型（所有元素数据类型相同的向量）；
- 把 `std::string` 转换为 R 的字符型向量；
- 把 STL 容器如 `std::vector<T>` 或 `std::map<T>` 转换成基本类型为 T 的向量，条件是 T 能够转换；
- 把 STL 的映射 `std::map<std::string, T>` 转换为基本类型为 T 的有名向量，条件是 T 能够转换；
- 可以转换定义了 `operator SEXP()` 的 C++ 类的对象；
- 可以转换专门化过 `wrap` 模板的 C++ 对象。
- 是否可用 `wrap()` 转换是在编译时确定的。

### 用 `as()` 函数把 R 变量转换为 C++ 类型

- Rcpp 提供了模板化的 `as()` 用来把 SEXP 类型转换成适当的 C++ 类型。
- `as()` 函数的声明为:

```
template <typename T> T as(SEXP x);
```

- `as()` 可以把 R 对象转换为基本的类型如 `int`, `double`, `bool`, `std::string` 等, 可以转换到元素为基础类型的 STL 向量如 `std::vector<double>` 等。
- 如果 C++ 类定义了以 SEXP 为输入的构造函数也可以利用 `as()` 来转换。
- `as()` 可以针对用户自定义类作专门化。

### `as()` 和 `wrap()` 的隐含调用

- 当 C++ 中赋值运算的右侧表达式是一个 R 对象或 R 对象的部分内容时, 可以隐含地调用 `as()` 将其转换成左侧的 C++ 类型。
- 当 C++ 中赋值运算的左侧表达式是一个 R 对象或其部分内容时, 可以隐含地调用 `wrap()` 将右侧的 C++ 类型转换成 R 类型。
- 在用 R 属性声明的 C++ 函数中, 可以直接以 `IntegerVector`, `NumericVector`, `CharacterVector`, `Function` 等作为自变量类型或返回值, 可以与 R 中相应的类型直接对应。

## 6.2 Rcpp 属性

### 6.2.1 介绍

#### Rcpp 属性介绍

- Rcpp 属性 (attributes) 用来简化把 C++ 函数变成 R 函数的过程。
- 做法是在 C++ 源程序中加入一些特殊注释, 利用其指示自动生成 C++ 与 R 的接口程序。属性是 C++11 标准的内容, 现在的编译器支持还不多, 所以在 Rcpp 支持的 C++ 程序中写成了特殊格式的注释。

- Rcpp 属性的优点：
  - 降低了同时使用 R 与 C++ 的学习难度；
  - 取消了很多繁复的接口代码；
  - 可以在 R 会话中很简单地调用 C++ 代码，不需要用户自己考虑编译、连接、接口问题；
  - 可以先交互地调用 C++，成熟后改编为 R 扩展包而不需要修改界面代码。

### Rcpp 属性的主要组成部分

- 在 C++ 中，提供 `Rcpp::export` 标注要输出到 R 中的 C++ 函数。
- 在 R 中，提供 `sourceCpp()`，用来自动编译连接保存在文件或 R 字符串中的 C++ 代码，并自动生成界面程序把 C++ 函数转换为 R 函数。
- 在 R 中，提供 `cppFunction()` 函数，用来把保存在 R 字符串中的 C++ 函数自动编译连接并转换成 R 函数。提供 `evalCpp()` 函数，用来把保存在 R 字符串中的 C++ 代码片段自动编译连接并执行。
- 在 C++ 中，提供 `Rcpp::depends` 标注，说明编译连接时需要的外部头文件和库的位置。
- 在构建 R 扩展包时，提供 `compileAttributes()` R 函数，自动给 C++ 函数生成相应的 `extern "C"` 声明和 `.Call` 接口代码。

### 在 C++ 源程序中指定要导出的 C++ 函数

- 用特殊注释 `//[[Rcpp::export]]` 说明某 C++ 函数需要在编译成动态链接库时，把这个函数导出到链接库的对外可见部分。
- 例如

```
//[[Rcpp::export]]
NumericVector convolveCpp(
    NumericVector a, NumericVector b){
    .....
}
```

具体程序参见前面“用 `sourceCpp()` 转换 C++ 源程序文件—卷积例子”。

- 假设此 C++ 源程序保存到了当前工作目录的 `conv.cpp` 源文件中，为了在 R 中调用此 C++ 程序，只要用如：

```
sourceCpp(file='conv.cpp')
convolveCpp(1:3, 1:5)
```

- 注意 `sourceCpp()` 把 C++ 源程序自动进行了编译链接并转换成了同名的 R 函数。
- 在同一 R 会话内，如果源程序和其依赖资源没有变化（根据文件更新时间判断），就不重新编译 C++ 源代码。

### 在 C++ 源程序中修改导出的 C++ 函数名

- 在用特殊注释说明要导出的 C++ 函数时，可以用特殊的 `name=` 参数指定函数导出到 R 中的 R 函数名。如果不指定，R 函数名和 C++ 函数名是相同的。
- 例如

```
//[[Rcpp::export(name="conv")]]
NumericVector convolveCpp(
    NumericVector a, NumericVector b){
    .....
}
```

则 C++ 函数 `convolveCpp` 导入到 R 中后，改名为“conv”。

### 可导出的 C++ 函数的限制条件

- 必须在全局名字空间中定义，而不能在某个 C++ 名字空间声明内定义。

- 自变量必须能够用 `Rcpp::as` 转换成 C++ 类型，返回值必须是空值或者能够用 `Rcpp::wrap` 转换成 R 类型。
- 在自变量和返回值类型说明中，必须使用完整的类型，比如 `std::string` 不能简写成 `string`。Rcpp 提供的类型如 `NumericVector` 可以不必用 `Rcpp::` 修饰。

### 6.2.2 R 中编译链接 C++ 代码

#### `sourceCpp()`

- `sourceCpp()` 函数可以用 `code=` 指定一个 R 字符串，字符串的内容是 C++ 源程序，其中还是用特殊注释 `//[[Rcpp::export]]` 标识要导出的 C++ 函数。如

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector convolveCpp(
    NumericVector a, NumericVector b){
    .....
}
')
convolveCpp(1:3, 1:5)
```

#### `cppFunction()`

- 对于比较简单的单个 C++ 函数，可以用 `cppFunction()` 函数的 `code=` 指定一个 R 字符串，字符串的内容是一个 C++ 函数定义，转换为一个 R 函数。例如

```
cppFunction(code='
int fibonacci(const int x){
    if(x < 2) return x;
    else
```

```
        return ( fibonacci(x-1) + fibonacci(x-2) );  
    }  
' )  
print(fibonacci(5))
```

### evalCpp()

- 为了在 R 中计算一个简单的 C++ 表达式，可以用 `evalCpp('C++ 表达式内容')`，如

```
evalCpp('std::numeric_limits<double>::max()')
```

函数将返回该 C++ 表达式的值。

### 指定要链接的其它库

- 在 `cppFunction()` 和 `evalCpp()` 中，可以用 `depends=` 参数指定要链接的其它库，如

```
sourceCpp(depends='RcppArmadillo', code='.....')
```

在编译代码时与 `RcppArmadillo` 的动态连接库连接。

- 也可以把这样的链接依赖关系写在特殊的 C++ 注释中，如

```
sourceCpp(depends='RcppArmadillo', code='.....')
```

- 这样的注释仅对 `sourceCpp()` 和 `cppFunction()` 有效，在编译 R 扩展包时，仍需要把依赖的包列在 `DESCRIPTION` 文件的 `Imports` 中，把要链接的包列在 `LinkingTo` 中。

### 6.2.3 Rcpp 属性的其它功能

### 自变量有缺省值的函数

- 借助于 Rcpp, 自变量有缺省值的 C++ 函数可以自动转换成自变量有缺省值的 R 函数。
- 定义时要符合 C++ 语法, 比如带缺省值的自变量都要在不带缺省值的自变量的后面, 缺省值不能有变量。
- 如

```
DataFrame readData(  
    CharacterVector file,  
    CharacterVector colNames = CharacterVector::create(),  
    std::string comment = "#",  
    bool header = true){ ... }
```

转换到 R 中, 相当于

```
function(file, colNames=character(), comment="#", header=TRUE)
```

- 能自动转换到 R 中的缺省值类型包括:
- 用双撇号界定的字符串常量;
- 十进数值如 10, 4.5;
- 预定义的常数如 true, false, R\_NilValue, NA\_STRING, NA\_INTEGER, NA\_REAL, NA\_LOGICAL;
- 用 create 生成的基础向量类实例, 如 CharacterVector, IntegerVector, NumericVector 等。

### 允许用户中断

- 在 C++ 代码中进行长时间的计算时, 应该允许用户可以中断计算。Rcpp 的办法是在 C++ 计算过程中每隔若干步循环就插入一个 `Rcpp::checkUserInterrupt();` 语句。

### 把 R 代码写在 C++ 源文件中

- 正常情况下，应该把 R 代码和 C++ 代码写在分别的源程序中，当 C++ 代码比较短时，也可以把 C++ 代码写在 R 源程序中作为一个字符串。
- Rcpp 允许把 C++ 代码和 R 代码都写在一个 C++ 源文件中，R 代码作为特殊的注释，以 `/** R` 行开头，以正常的 `*/` 结束。在 R 中用 `sourceCpp()` 调用这个 C++ 源文件，就可以编译 C++ 后执行其中特殊注释内的 R 代码。这样的特殊注释可以有多个。
- 例如，下述内容保存在文件 `fibonacci.cpp` 中：

```
/**[Rcpp::export]
int fibonacci(const int x){
    if(x < 2) return x;
    else
        return ( fibonacci(x-1) + fibonacci(x-2) );
}

/** R
# 调用 C++ 中的 fibonacci() 函数
print(fibonacci(10))
*/
```

- 只要在 R 中运行

```
sourceCpp(file='fibonacci.cpp')
```

就可以编译连接此 C++ 文件，把其中用 `/**[Rcpp::export]` 标识的函数转换为 R 函数，并在 R 中执行源文件内特殊注释中的 R 代码。

### 在 C++ 中调用 R 的随机数发生器

- 在 C 或 C++ 中调用 R 的随机数发生器，需要能够同步地更新随机数发生器状态。
- 如果利用 Rcpp 属性编译 C++ 源程序，则 Rcpp 属性会自动添加一个 `RNGScope` 实例进行随机数发生器状态的同步。



## 6.3 Rcpp 提供的向量类的 C++ 数据类型

### 6.3.1 RObject 类

#### Rcpp 提供的 C++ 数据类型及其运算

- Rcpp 包为 C++ 定义了 NumericVector, IntegerVector, CharacterVector, Matrix 等新数据类型, 可以直接与 R 的 numeric, character, matrix 对应。
- Rcpp 最基础的 R 数据类型是 RObject, 这是 NumericVector, IntegerVector 等的基类, 通常不直接使用。
- RObject 包裹了原来 R 的 C API 的 SEXP 数据结构, 并且提供了自动的内存管理, 不再需要用户自己处理建立内存和消除内存的工作。
- RObject 存储数据完全利用 R C API 的 SEXP 数据结构, 不进行额外的复制。

#### RObject 类的成员函数

- 因为 RObject 类是基类, 所以其成员函数也适用于 NumericVector 等类。
- isNULL, isObject, isS4 可以查询是否 NULL, 是否对象, 是否 S4 对象。
- inherits 可以查询是否继承自某个特定类。
- 用 attributeNames, hasAttribute, attr 可以访问对象的属性。
- 用 hasSlot, slot 可以访问 S4 对象的插口 (slot)。

#### RObject 的导出类

- IntegerVector: 整数向量;
- NumericVector: 数值向量;
- LogicalVector: 逻辑向量;
- CharacterVector: 字符型向量;
- GenericVector: 列表;

- ExpressionVector: 表达式向量;
- RawVector: 元素为 raw 类型的向量。
- IntegerMatrix, NumericMatrix: 整数值或数值矩阵。

### 与 R 原子向量对应的类

- 在 R 向量中, 如果其元素都是同一类型 (如整数、双精度数、逻辑、字符型), 则称为原子向量。
- Rcpp 提供了 IntegerVector, NumericVector, LogicalVector, CharacterVector 等数据类型与 R 的原子向量类型对应。
- 在 C++ 中可以用 [] 运算符存取向量元素, 也可以用 STL 的迭代器。用 .begin(), .end() 等界定范围, 用循环或或者 accumulate 等 STL 算法处理整个向量。

### 6.3.2 IntegerVector 类

#### IntegerVector 类

- 在 R 中通常不严格区分整数与浮点实数, 但是在与 C++ 交互时, C++ 对整数与实数严格区分, 所以 RCpp 中整数向量与数值向量是区分的。
- 在 R 中, 如果定义了一个仅有整数的向量, 其类型是整数 (integer) 的, 否则是数值型 (numeric) 的, 如:

```
> x <- 1:5
> class(x)
[1] "integer"
> y <- c(0, 0.5, 1)
> class(y)
[1] "numeric"
```

- 用 as.integer() 和 as.numeric() 函数可以显式地确保其自变量转为需要的整数型或数值型。

- RCpp 可以把 R 的整数向量传递到 C++ 的 IntegerVector 中，也可以把 C++ 的 IntegerVector 函数结果传递回 R 中变成一个整数向量。
- 也可以在 C++ 中生成 IntegerVector 向量，填入整数值。

#### IntegerVector 示例 1: 返回完全数

- 如果一个正整数等于它所有的除本身以外的因子的和，称这个数为完全数。如

$$6 = 1 + 2 + 3$$

$$28 = 1 + 2 + 4 + 7 + 14$$

是完全数。

- 任务：用 C++ 程序输入前 4 个完全数偶数，返回到 R 中。这 4 个数为 6, 28, 496, 8182。
- 程序：

```
require(Rcpp)
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
IntegerVector wholeNumberA(){
    IntegerVector epn(4);
    epn[0] = 6;    epn[1] = 28;
    epn[2] = 496;  epn[3] = 8182;

    return epn;
}
')
print(wholeNumberA())
## [1]    6   28 496 8182
```

- 可以在 C++ 中建立一个 IntegerVector, 需指定大小。
- 可以逐个填入数值。
- 直接返回 IntegerVector 到 R 即可, 不需显式地转换。

### IntegerVector 示例 2: 输入整数向量

- 任务: 用 C++ 编写函数, 从 R 中输入整数向量, 计算其元素乘积 (与 R 的 `prod()` 函数功能类似)。
- 程序:

```
require(Rcpp)
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
IntegerVector prod1(IntegerVector x){
    int prod = 1;
    for(int i=0; i < x.size(); i++){
        prod *= x[i];
    }
    return wrap(prod);
}
')
print(prod1(1:5))
```

- 用 IntegerVector 从 R 中接受一个整数值向量时, 不需要显式地转换。
- 把一个 C++ 整数值返回给 R 时, 必须用 IntegerVector 返回, 这需要用 `Rcpp::wrap()` 转换一下。在 `sourceCpp` 中可以省略 `Rcpp::` 部分。
- 还可以用 C++ STL 的算法库进行这样的累计乘积计算, `std::accumulate()` 可以对指定范围进行遍历累计运算。前两个参数是一个范围, 用迭代

器 (iterators) 表示开始和结束，第三个参数是初值，第四个参数是对每个元素累计的计算。

- 程序：

```
require(Rcpp)
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
IntegerVector prod2(IntegerVector x){
  int prod = std::accumulate(
    x.begin(), x.end(),
    1, std::multiplies<int>());
  return wrap(prod);
}
')
print(prod2(1:5))
```

### 错误的输入

- 在以上的输入 IntegerVector 的 C++ 程序中，如果从 R 中输入了实数型的向量，则元素被转换成整数型。比如 `prod2(seq(1,1.9,by=0.1))` 结果将等于 1。
- 如果输入了无法转换为整数型向量的内容，比如 `prod2(c('a', 'b', 'c'))`，程序会报错。

### 6.3.3 NumericVector 类

#### NumericVector 类

- NumericVector 类在 C++ 中保存双精度型一维数组，可以与 R 的实数型向量 (class 为 numeric) 相互转换。
- 这是自己用 C++ 程序与 R 交互时最常用到的数据类型。

示例 1: 计算元素  $p$  次方的和

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector ssp(
    NumericVector vec, double p){
    double sum = 0.0;
    for(int i=0; i < vec.size(); i++){
        sum += pow(vec[i], p);
    }
    return(wrap(sum));
}
')
```

```
ssp(1:4, 2)
## [1] 30
ssp((1:4)/10, 2.2)
## [1] 0.2392496
sum( ((1:4)/10)^2.2 )
## [1] 0.2392496
```

### 用 Rcpp 属性编译时 C++ 函数的输入与返回值类型

- R 中数值型向量在 C++ 中可以用 NumericVector 接收;
- R 中单个的实数在 C++ 中可以用 double 来接收;
- 为了返回单个的实数, 在 C++ 中需要以 NumericVector 为返回类型, 可以从一个 double 型用 wrap() 自动转换。
- C++ 中如果返回 NumericVector, 在 R 中转换为数值型向量。

**示例 2: clone 函数**

- 在自定义 R 函数时，输入的自变量的值不会被改变，相当于自变量都是局部变量。
- 如果在自定义函数中修改了自变量的值，实际上只能修改自变量的一个副本的值。如

```
x <- 100
f <- function(x){
  print(x); x <- 99; print(x)
}
c(f(x), x)
## [1] 100
## [1] 99
## [1] 99 100
```

- 但是，在用 Rcpp 编写 R 函数时，因为 RObject 传递的是指针，并不会自动复制自变量值，所以修改自变量值会真的修改原始的自变量变量值。

- 如：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector f2(NumericVector x){
  x[0] = 99.0;
  return(x);
}
')
x <- 100
c(f2(x), x)
```

- 可见自变量的值被修改了。
- 当然，对这个问题而言，因为输入的是一个标量，只要函数自变量不是 NumericVector 类型而是用 double 类型，则自变量值会被复制，达到值传递的效果，自变量值也就不会被真的修改。

如

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector f3(double x){
    x = 99.0;
    return(wrap(x));
}
')
x <- 100
c(f3(x), x)
## [1] 99 100
```

- 下面的程序把输入向量每个元素平方后返回，为了不修改输入自变量的值而是返回一个修改后的副本，使用了 Rcpp 的 clone 函数：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector square(NumericVector x){
    NumericVector y = clone(x);
    for(int i=0; i < x.size(); i++){
        y[i] = x[i]*x[i];
    }
}
```



```

    return(y);
}
')
```

```

x <- c(2, 7)
cbind(square(x), x)
##           x
## [1,]  4  2
## [2,] 49  7
```

### 示例三：把输入矩阵制作副本计算平方根

- NumericMatrix 是 Rcpp 提供的元素为双精度型的矩阵。
- 下面的例子输入一个 R 矩阵，输出其元素的平方根，用了 clone 函数来避免对输入的直接修改。

```

sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericMatrix matSqrt(NumericMatrix x){
    NumericMatrix y = clone(x);
    std::transform(y.begin(), y.end(),
        y.begin(), ::sqrt);
    return(y);
}
')
```

```

x <- rbind(c(1,2), c(3,4))
cbind(matSqrt(x), x)
##           [,1]      [,2] [,3] [,4]
## [1,] 1.000000 1.414214    1    2
## [2,] 1.732051 2.000000    3    4
```

- 在上面的 C++ 程序中，NumericMatrix 看成了一维数组，用 STL 的 iterater 遍历，用 STL 的 transform 对每个元素计算变换。

### 6.3.4 Rcpp 的其它向量类

#### Rcpp 的 LogicalVector 类

- LogicalVector 类可以存储 C++ 值 true, false, 还可以保存缺失值 NA\_REAL, R\_NaN, R\_PosInf, 但是这些不同的缺失值转换到 R 中都变成 NA。
- 如:

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
LogicalVector f4(){
  LogicalVector x(5);
  x[0] = false; x[1] = true;
  x[2] = NA_REAL;
  x[3] = R_NaN; x[4] = R_PosInf;
  return(x);
}
```

```
f()
## [1] FALSE TRUE NA NA NA
identical(f(), c(FALSE, TRUE, rep(NA,3)))
## [1] TRUE
```

**Rcpp 的 CharacterVector 类型**

- CharacterVector 类型可以与 R 的字符型向量相互交换信息，在 C++ 中其元素为字符串。
- 字符型缺失值在 C++ 中为 R\_NaString。
- R 的字符型向量也可以转换为 `std::vector<std::string>`。
- 如：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
CharacterVector f5(){
    CharacterVector x(3);
    x[0] = "This is a string";
    x[1] = "Test";
    x[2] = R_NaString;
    return(x);
}
')
```

```
f()
## [1] "This is a string" "Test"          NA
```

**6.4 Rcpp 提供的其它数据类型****6.4.1 Named 类型****Named 类型**

- R 中的向量、矩阵、数据框可以有元素名、列名、行名。这些名字可以借助 Named 类添加。
- 例如：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector f6(){
    NumericVector x = NumericVector::create(
        Named("math") = 82,
        Named("chinese") = 95,
        Named("English") = 60);
    return(x);
}
')
```

```
f6()
##   math chinese English
##    82      95      60
```

- “Named("元素名")”可以简写成“\_<sup>下划线</sup>["元素名]”。
- 如：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector f6b(){
    NumericVector x = NumericVector::create(
        _["math"] = 82,
        _["chinese"] = 95,
        _["English"] = 60);
}
```

```

    return(x);
}
')
```

### 6.4.2 List 类型

#### List 类型

- Rcpp 提供的 List 类型对应于 R 的 list(列表) 类型，在 C++ 中也可以写成 GenericVector 类型。其元素可以不是同一类型，在 C++ 中可以用方括号和字符串下标的格式访问其元素。
- 例如，下面的函数输入一个列表，列表元素 vec 是数值型向量，列表元素 multiplier 是数值型标量，返回一个列表，列表元素 sum 为 vec 元素和，列表元素 dsum 为 vec 元素和乘以 multiplier 的结果：

```

sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
List f7(List x){
    NumericVector vec = as<NumericVector>(x["vec"]);
    double multiplier = as<double>(x["multiplier"]);
    double y = 0.0, y2;
    for(int i=0; i<vec.length(); i++){
        y += vec[i];
    }
    y2 = y*multiplier;
    return(List::create(Named("sum")=y,
        Named("dsum")=y2));
}
')
```

```

f7(list(vec=1:5, multiplier=10))
## $sum
```

```
## [1] 15
##
## $dsum
## [1] 150
```

### List 生成和使用

- 上面的程序用了 `Rcpp::List::create()` 当场生成 List 类型，因为用 Rcpp 属性功能编译所以可以略写 `Rcpp::`。
- 也可以在程序中预先生成指定大小的列表，然后再给每个元素赋值，元素值可以是任意能够转化为 SEXP 的类型，如：

```
.....
List gv(2);
gv[0] = "abc";
gv[1] = 123;
```

- 可以用 List 的 `reserve` 函数为列表指定元素个数。

### 6.4.3 Rcpp 的 DataFrame 类

#### DataFrame 类

- Rcpp 的 DataFrame 类用来与 R 的 `data.frame` 交换信息。
- 示例如：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
DataFrame f8(){
    IntegerVector vec =
        IntegerVector::create(7,8,9);
    std::vector<std::string> s(3);
```

```
s[0] = "abc"; s[1] = "ABC"; s[2] = "123";
return(DataFrame::create(
  Named("x") = vec,
  Named("s") = s));
}
```

```
f8()
##   x   s
## 1 7 abc
## 2 8 ABC
## 3 9 123
```

#### 6.4.4 Rcpp 的 Function 类

##### Function 类

- Rcpp 的 Function 类用来接收一个 R 函数，并且可以在 C++ 中调用这样的函数。
- 示例如：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
SEXP dsort(Function sortFun, SEXP x){
  return sortFun(x, Named("decreasing", true));
}
```

```
dsort(sort, c('bb', 'ab', 'ca'))
## [1] "ca" "bb" "ab"
```

```
## dsort(sort, c(2,1,3))
## [1] 3 2 1
```

- 程序用 Function 对象 sortFun 接收从 R 中传递过来的排序函数，实际调用时传递过来的是 R 的 sort 函数。
- 在 C++ 中调用 R 函数时，有名的自变量用 “Named(自变量字符串, 自变量值)” 的格式给出。
- 程序中的待排序的向量与排序后的向量都用了 SEXP 来说明，即直接传送原始 R API 指针，这样可以不管原来类型是什么，在 C++ 中完全不进行类型转换或复制。
- 从运行例子看出数值和字符串都正确地按照降序排序后输出了。

#### 在 C++ 中直接调用 R 中已有的函数

- R 函数可以不作为 C++ 的函数自变量传递进来，而是直接调用 R 的函数。
- 下面的函数直接调用 rt 函数生成 2 个自由度为 3 的 t 分布随机数：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector rt23(){
    Function rt("rt");
    return rt(2, 3);
}
')
```

```
set.seed(1)
rt23()
## [1] -0.7027211 -0.5693196
rt23()
## [1] 0.6842766 -0.3620012
```



- 使用了 Rcpp 属性时，生成的界面程序会自动生成一个 RNGScope 的实例，用来保存当前的随机数发生器状态，在解构时将自动更新随机数发生器状态。
- 用 rt 这个 R 函数名初始化了一个 C++ 的 Function 对象 rt，然后就可以调用这个 C++ 函数了。
- 从程序可以看出，连续两次调用的结果不同。

### 6.4.5 Rcpp 的 Environment 类

#### Environment 类

- R 的环境是分层的，可以逐层查找变量名对应的内容。
- Rcpp 的 Environment 类用来与 R 环境对应。
- 可以利用 Environment 来定位 R 的扩展包中的函数或数据，例如下面的程序片段在 C++ 中定位了 stats 扩展包中的 rnorm 函数并进行了调用：

```
Environment stats("package:stats");  
Function rnorm = stats["rnorm"];  
return rnorm(3, Named("sd", 100.0));
```

- 当然，也可以用 Function 对象直接从各环境中搜索 rnorm 函数名，但是这样指定环境更可靠。

#### 访问全局环境示例

- 下面的例子访问了 R 全局环境，取出了全局变量 x 的值存入 C++ 的双精度型 STL 向量中，并把一个 C++ 的 STL map 型数据转换成有名字符型向量存到了全局变量 y 中。
- 示例：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
void f9(){
    Environment global = Environment::global_env();
    std::vector<double> vx = global["x"];
    std::map<std::string, std::string> ma;
    ma["foo"] = "abc";
    ma["bar"] = "123";
    global["y"] = ma;
    return;
}
')
```

```
x <- c(1,5)
f9()
y
##   bar   foo
## "123" "abc"
```

#### 6.4.6 Rcpp 的 S4 类和 ReferenceClass 类

##### R 的面向对象介绍

- R 是基于 S 语言的，S 语言最初的面向对象机制是 S3 类，S3 类主要用了方法派送 (method dispatch) 机制，使用了共性函数 (generic function)，同一函数对不同的数据类型可以执行不同的操作，实际是调用了不同的具体函数。S3 类还在广泛使用。
- S 语言较现代的面向对象使用了 S4 类，S4 提供了更丰富的数据结构，要求更严谨。
- Rcpp 提供了 S4 数据类型，可以读取和写入 R 的 S4 类的插口 (slot)。

- 下面的示例在 C++ 中从输入对象定义了一个 S4 类并返回到 R 中：
- 下例中，如果一个的 RObject 对象传送了一个 S4 对象，可以测试插口的存在，并可访问插口内容：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
List f10(RObject x){
    List res(3);
    res[0] = x.isS4();
    res[1] = x.hasSlot("z");
    res[2] = x.slot("z");
    return res;
}
')
```

### ReferenceClass 类介绍

- 除了 S3 类、S4 类，R 还增加了一种参考类。
- 参考类与其它语言的类更为相近，作为函数自变量时可以按引用传递而不需制作副本，并且其方法与对象本身相连，而不是与函数相连。
- Rcpp 提供了 ReferenceClass 类。

#### 6.4.7 R 的数学函数库

##### R 的数学函数库

- R 提供了许多数学和统计相关的函数，这些函数可以编译成一个独立的函数库，供其它程序链接使用，函数内容在 R 的 Rmath.h 头文件中有详细列表。
- Rcpp 中可以调用这些函数。最简单的一种方法是直接在 Rcpp 名字空间中使用和 R 中相同的函数名和调用方法，类似 sqrt 这样的函数允许输入一个向量，对向量的每个元素计算。

- 比如，下面的程序输入一个向量，计算相应的标准正态分布函数值：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector f11(NumericVector x){
  NumericVector y(x.size());
  y = pnorm(x);
  return y;
}
')
f11(c(0, 1.96, 2.58))
## [1] 0.5000000 0.9750021 0.9950600
```

- 如果需要对单个的数值计算，可以使用 Rmath.h 中定义的带有 Rf\_ 的版本，如：

```
y = ::Rf_pnorm5(x, 0.0, 1.0, 1, 0);
```

这里用:: 使用了缺省的名字空间。注意所有自变量都不能省略。

- Rcpp 还提供了一个 R 名字空间，可以用不带 Rf\_ 前缀的函数，但是自变量也不能省略。如

```
y = R::pnorm(x, 0.0, 1.0, 1, 0);
```

- 在 Rcpp 的 R 名字空间中有许多的数学和统计相关的函数，各函数的自变量参见 Rcpp 的 Rmath.h 文件。

## 6.5 Rcpp 糖

### Rcpp 糖

- 在 C++ 中，向量和矩阵的运算通常需要逐个元素进行，或者调用相应的函数。
- Rcpp 通过 C++ 的表达式模板 (expression template) 功能，可以在 C++ 中写出像 R 中对向量和矩阵运算那样的表达式。这称为 Rcpp 糖。
- R 中的很多函数如 `sin` 等是向量化的，Rcpp 糖也提供了这样的功能。Rcpp 糖提供了一些向量化的函数如 `ifelse`, `sapply` 等。
- 比如，两个向量相加可以直接写成 `x + y` 而不是用循环或迭代器 (iterator) 逐元素计算; 若 `x` 是一个 `NumericVector`, 用 `sin(x)` 可以返回由 `x` 每个元素的正弦值组成的 `NumericVector`。
- Rcpp 糖不仅简化了程序，还提高了运行效率。

#### 简单示例

- 比如，函数

$$f(x, y) = \begin{cases} x^2 & x < y, \\ -y^2 & x \geq y \end{cases}$$

- 如下的程序可以在 C++ 中定义一个向量化的版本:

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector f12(NumericVector x, NumericVector y){
  return ifelse(x < y, x*x, -(y*y));
}
')
f12(c(1, 3), c(4,2))
## [1] 1 -4
```

- 上面简单例子中，`x < y` 是向量化的，`x * x`, `y * y`, `-(y * y)` 都是向量化的。
- `ifelse` 也是向量化的。

### 6.5.1 向量化的运算符

#### 向量化的四则运算

- Rcpp 糖使得向量化了  $+$ ,  $-$ ,  $*$ ,  $/$ 。
- 设  $x, y$  都是 `NumericVector`, 则  $x + y, x - y, x * y, x / y$  将返回对应元素进行四则运算后的 `NumericVector` 变量。
- 向量与标量运算, 如  $x + 2.0, 2.0 - x, x * 2.0, 2.0 / x$  将返回标量与向量每个元素进行四则运算后的 `NumericVector` 变量。

- 还可以进行混合四则运算, 如:

```
NumericVector res = x * y + y / 2.0;  
NumericVector res = x * (y - 2.0);  
NumericVector res = x / (y * y);
```

- 参加四则运算的或者都是同基本类型的向量而且长度相等, 或者一边是向量, 一边是同类型的标量。
- 注意: 对向量整体赋一个相同的值, 不能简单地写成如  $x=0$ ; 这样的赋值, 需要用循环或 STL 的 `fill` 算法, 如 `std::fill(x.begin(), x.end(), 0);`。

#### 向量化的二元逻辑运算

- Rcpp 糖扩展了两个元素的比较到两个等长向量的比较, 以及一个向量与一个同类型标量的比较, 结果是一个同长度的 `LogicalVector`。
- 比较运算符包括  $<$ ,  $>$ ,  $<=$ ,  $>=$ ,  $==$ ,  $!=$ 。
- 也可以使用嵌套的表达式, 比如, 设  $x, y$  是两个 `NumericVector`, 可以:

```
LogicalVector res = (x + y) < (x * x);
```

### 向量化的一元运算符

- 对数值型向量或返回数值型向量的表达式前面加负号，可以返回每个元素取相反数的结果。比如，设 `x` 是 `NumericVector`，可以：

```
NumericVector res = -x;  
NumericVector res = -x * (x + 2.0);
```

- 对逻辑型向量或返回逻辑型向量的表达式前面加叹号，可以返回每个元素取反的结果，如：

```
NumericVector y, z;  
NumericVector res = ! ( y < z );
```

## 6.5.2 Rcpp 糖对调用 R 函数的改进

### Rcpp 糖对调用 R 函数的改进

- C 和 C++ 中可以调用 R 中的函数，但是格式比较复杂，而且原来不支持向量化。
- Rcpp 糖则使得从 C++ 中调用 R 函数变得和在 R 调用函数格式类似。

### 返回单一逻辑值的函数

### 返回单一逻辑值的函数

- 在 R 中，`any()` 和 `all()` 对一个逻辑向量分别判断是否有任何真值，以及所有元素为真值。
- Rcpp 糖在 C++ 中也提供了这样的 `any()` 和 `all()` 函数。
- 如

```
sourceCpp(code='  
#include <Rcpp.h>  
using namespace Rcpp;
```

```
//[[Rcpp::export]]
List f13(){
  IntegerVector x = seq_len(1000);
  LogicalVector res1 = any( x*x < 3 );
  LogicalVector res2 = all( x*x < 3 );
  return List::create(Named("any")=res1,
    Named("all")=res2);
}
```

```
f13()
## $any
## [1] TRUE
##
## $all
## [1] FALSE
```

- `any()` 和 `all()` 不是直接返回逻辑值，而是返回一个类对象，该类定义了 `is_true`, `is_false`, `is_na` 方法与向 SEXP 转换的运算符。
- 此种结果可以保存到 `LogicalVector` 中，但不能赋值到 `bool` 类型，因为可能有缺失值。
- 逻辑型糖表达式结果可以用 `is_true`, `is_false`, `is_na` 来判断结果。
- 如

```
bool res1 = is_true( any( x < y ) );
bool res2 = is_na( all( x < y ) );
```

- 在求 `any()` 结果时，一旦遇到一个真值结果就为真值，即使后面有缺失值也没有关系。
- 在求 `all()` 结果时，一旦遇到一个假值结果就为假值，即使后面有缺失值也没有关系。



### 6.5.3 返回糖表达式的函数

#### is\_na

- `is_na` 以任何糖表达式为输入，输出一个逻辑类型的元素个数相同的糖表达式。结果每个元素当输入中对应元素缺失时为 TRUE, 否则为 FALSE。
- 如

```
IntegerVector x = IntegerVector::create(0, 1, NA_INTEGER, 3);
LogicalVector res1 is_na(x);
LogicalVector res2 = all( is_na(x) );
if( is_true( any( ! is_na(x) ) ) ) ...
```

#### seq\_along

- `seq_along` 输入一个向量，输出一个元素为该向量的各个下标值的糖表达式。
- 如

```
IntegerVector x = IntegerVector::create(0, 1, NA_INTEGER, 3);
seq_along(x);
seq_along(x*x*x*x*x);
```

- 注意上述程序不会计算  $x*x*x*x*x$  的值而只利用其结果的元素个数。所以两次调用的计算量是一样的。

#### seq\_len

- `seq_len` 自变量为个数，返回一个元素值为正数的元素个数等于自变量值的糖表达式，第  $i$  个元素等于  $i$ 。
- 常可与 `sapply`, `lapply` 配合使用。
- 如

```

sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
List f14(){
  IntegerVector x =seq_len(3);
  List y = lapply( seq_len(3), seq_len);
  return List::create(Named("x")=x,
    Named("y")=y);
}
')
```

```

f14()
## $x
## [1] 1 2 3
##
## $y
## $y[[1]]
## [1] 1
##
## $y[[2]]
## [1] 1 2
##
## $y[[3]]
## [1] 1 2 3
##
```

### pmin 和 pmax

- 自变量为两个等长的向量或糖表达式，返回长度相同的结果，结果元素只等于对应元素的最小值或最大值。
- 自变量也可以取一个向量一个标量，向量的每个元素与标量比较得到最小值或最大值。

- 如

```
IntegerVector x =seq_len(10);  
pmin(x, x*x);  
pmin(x*x, 2);
```

### ifelse

- **ifelse** 有三个自变量，第一自变量是一个逻辑型向量值的糖表达式，第二和第三自变量或者是两个同类型并与第一自变量等长的糖表达式，或者其中一个是同类型标量，结果仍为向量型糖表达式，第  $i$  元素当第一自变量第  $i$  元素为真时取第一自变量的第  $i$  元素，当第一自变量第  $i$  元素为假时取第二自变量的第  $i$  元素，当第一自变量第  $i$  元素为缺失值时取缺失值。
- 如

```
IntegerVector x, y;  
IntegerVector res1 = ifelse( x < y, x, (x+y)*y );  
IntegerVector res2 = ifelse( x > y, x, 2 );
```

### sapply 函数

- **sapply** 函数第一自变量是一个向量或列表，第二自变量是一个函数。
- 返回值类型在编译时从函数的结果类型导出。
- 第二自变量可以是任意的 C++ 函数，比如，可以是如下的重载的模板化函数：

```
template <typename T>  
T square( const T& x ){  
    return x * x;  
}  
sapply( seq_len(4), square<int> );
```

- 下面的例子中 `sapply` 第二自变量使用了 functor，这是一种能够产生函数的函数：

```
template <typename T>
struct square : std::unary_function<T,T> {
    T operator() (const T& x) {
        return x * x;
    }
}
sapply( seq_len(4), square<int>() );
```

### `lapply` 函数

- `lapply` 函数与 `sapply` 函数基本相同，只不过 `lapply` 函数总是返回列表，列表在 Rcpp 中为 `List` 或 `GenericVector`，在 R API 中类型为 `VECSXP`。

### `sign` 函数

- `sign` 函数输入一个数值型或整型表达式，返回各元素值在  $\{1, 0, -1, NA\}$  中取值的糖表达式，可以保存到 `IntegerVector` 中。
- 结果各元素的取值表示输入中对应元素为正、零、负和缺失。
- 如

```
IntegerVector x;
IntegerVector res1 = sign( x );
IntegerVector res2 = sign( x*x );
```

### `diff` 函数

- 自变量为数值型或整数型向量或有这样结果的糖表达式，输出后一元素减去前一元素的结果，结果长度比输入长度少一。

- 如

```
IntegerVector res = diff( seq_len(5) );
```

### 6.5.4 数学函数

#### 向量化的数学函数

- Rcpp 把 R 中的数学函数在 C++ 中向量化了，输入一个向量，结果是对应元素为函数值的向量。
- 自变量类型可以是数值型或整数型。
- 这些数学函数包括 abs, exp, floor, ceil, pow 等。

#### 分布密度、分布函数、分位数函数

- R 中提供了许多分布的分布密度（概率质量函数）、分布函数、分位数函数，分布密度函数和概率质量函数命名类似 dxxxx，分布函数命名类似 pxxxx，分位数函数命名类似 qxxxx。
- 这些函数可以直接用类似 R 中的格式调用。
- 如

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
List f15(){
  NumericVector x =
    NumericVector::create(0, 1.96, 2.58);
  NumericVector p =
    NumericVector::create(0.95, 0.975, 0.995);
  NumericVector y1 = dnorm(x, 0.0, 1.0);
  NumericVector y2 = pnorm(x, 0.0, 1.0);
  NumericVector y3 = qnorm(p, 0.0, 1.0);
```

```
return List::create(Named("y1")=y1,
  Named("y2")=y2, Named("y3")=y3);
}
```

```
f15()
## $y1
## [1] 0.39894228 0.05844094 0.01430511
##
## $y2
## [1] 0.5000000 0.9750021 0.9950600
##
## $y3
## [1] 1.644854 1.959964 2.575829
##
```

### Rcpp 与随机数发生器函数

- R 中的 `rxxxx` 类的函数可以产生各种分布的随机数向量，随机数向量与当前种子有关。
- Rcpp 属性会自动地维护随机数发生器的状态使其与 R 同步。
- 如

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector f16(){
  NumericVector x = rnorm(10, 0.0, 1.0);
  return x;
}
```

```
round(f16(), 2)
## [1]  0.62 -0.06 -0.16 -1.47 -0.48
## [6]  0.42  1.36 -0.10  0.39 -0.05
```

### R 与 Rcpp 不同语法示例

- RCpp 数据类型通过一些现代的 C++ 技术，支持部分的向量化运算，比如，NumericVector 与标量相加，两个等长 NumericVector 相加，对 NumericVector 计算如绝对值这样的数学函数值等。
- 这称为 Rcpp 的糖 (sugar)。
- 但是，C++ 毕竟和 R 有很大差别，要区分 Rcpp 能做的和不能做的。
- NumericVector 为了把向量所有元素都改成同一值，不能直接用等于赋值。可以用 `std::fill(y.begin(), y.end(), 99.99)` 这样的做法。

## 6.6 用 Rcpp 帮助制作 R 扩展包

### 6.6.1 介绍

#### R 扩展包

- R 扩展包是把解决某种问题的可复用代码、文档整合在一起的最好的方法。
- 写成 R 扩展包后，可以自己用，也可以利用 CRAN 分发。扩展包用户一般不用自己编译。
- 使用扩展包，多个源程序、头文件之间的依赖关系可以自动得到处理。
- 扩展包提供了测试、文档和一致性检查的统一框架。
- 代码可以仅有 R 程序，也可以包括 C 程序、C++ 程序、Fortran 程序。
- 如果仅有 R 代码，就不需要借助于 Rcpp，可以使用 `package.skeleton()` 函数生成一个扩展包框架。

- 如果有 C++ 代码，就可以用 Rcpp 作为接口，并用 Rcpp 提供的 `Rcpp.package.skeleton()` 函数制作扩展包框架。
- Rcpp 属性的 Exports 注释仍可在制作扩展包时指定如何输出 C++ 中定义的函数使其在 R 中可调用。

### 不用扩展包共享 C++ 代码的方法

- Rcpp 属性的 `sourceCpp()` 通常只适用于写在 R 程序内部的简短 C++ 代码，或者写在一个单独 C++ 文件中，不依赖于其它 C++ 程序的单独代码。
- 如果有多个 C++ 源程序、头文件，彼此有依赖关系，最好使用扩展包。
- 在多个单独的 C++ 文件共享某些简单的代码，彼此不互相依赖时，可以用 C++ 的预处理 `include` 命令共享这些代码。

### 简单代码共享示例

- 比如，有多个 C++ 源程序都用到如下的代码：

```
#ifndef __UTILITIES__
#define __UTILITIES__
inline double timesTwo(double x) {
    return x * 2;
}
#endif // __UTILITIES__
```

- 假设这段代码保存到当前子目录的“utilities.hpp”文件中。
- 则在每个需要用到这段代码的 C++ 源程序中，插入如：

```
#include "utilities.hpp"
//[[Rcpp::export]]
double transformValue(double x){
    return timesTwo(x) * 10;
}
```



### 6.6.2 生成扩展包

利用已有基于 Rcpp 属性的源程序制作扩展包

- 假设在当前目录中有了若干个 C++ 文件，其中需要转换到 R 中的 C++ 函数已经用 `Rcpp::export` 声明过。其中一个是 `conv1.cpp`。
- 从当前目录启动 R，运行

```
Rcpp.package.skeleton("testpack",  
  example_code=FALSE,  
  attributes=TRUE,  
  cpp_files=c("conv1.cpp"))
```

- 运行显示：

```
Creating directories ...  
Creating DESCRIPTION ...  
Creating NAMESPACE ...  
Creating Read-and-delete-me ...  
Saving functions and data ...  
Making help files ...  
Done.  
Further steps are described in './testpack/Read-and-delete-me'.  
  
Adding Rcpp settings  
>> added Imports: Rcpp  
>> added LinkingTo: Rcpp  
>> added useDynLib directive to NAMESPACE  
>> added importFrom(Rcpp, evalCpp) directive to NAMESPACE  
>> copied conv1.cpp to src directory
```

**Rcpp.package.skeleton** 的结果解释

- 运行完后，在当前目录生成了一个 `testpack` 子目录，这是要制作的扩展包的名字。

- 在 testpack 子目录中, 有文件 DESCRIPTION, NAMESPACE, Read-and-delete-me, 有子目录 src, R, man。
- 子目录 src 中为 C++ 和 C 源程序、头文件。
- 子目录 R 中为 Rcpp 从 C++ 程序转换过来的 R 接口程序, 用户自己的 R 程序也可以放在这里。
- 子目录 man 是特殊格式的文档, 其格式类似 L<sup>A</sup>T<sub>E</sub>X。

### DESCRIPTION 文件

- 在 DESCRIPTION 文件中, 有扩展包名称、版本、日期、作者姓名、维护者姓名和联系方式、简单描述、授权, 还有 Imports 和 LinkingTo 两项。
- 除此之外, 还有许多可选的域, 如 Depends。

### DESCRIPTION 文件中的 Imports 域

- Imports 给出本软件包要调用的其它扩展包, 但是这些扩展包并不随本扩展包一起调入, 仅是会调入其名字空间。这里的值为

```
Imports: Rcpp (>= 0.12.3)
```

### DESCRIPTION 文件中的 LinkingTo 域

- LinkingTo 指定在编译本软件包的 C、C++、Fortran 等源程序时, 会用到哪些其它扩展包的头文件。这里的值为

```
NumericVector y, z;
NumericVector res = ! ( y < z );
```

这些扩展包一般是编译时才有用的, 所以一般不会出现在 Depends 和 Imports 域中。

- LinkingTo 只解决了头文件的问题, 要链接除了 Rcpp 之外的二进制库文件, 还需要手工编辑 src/Makevars 和 src/Makevars.win 文件。

### DESCRIPTION 文件的 Depends 域

- 和 Imports 有些相像的 DESCRIPTION 域是 Depends，指定调入本扩展包时必须预先调入的软件包。这里“调入”是指用 `library()` 或 `require()` 调入扩展包。
- 多个扩展包名用逗号分开，可以在扩展包名字后面加圆括号，在圆括号内写上 `>=` 某个版本号，如 “MASS(`>=3.1-20`)”。
- Depends 也可以指定依赖于某个 R 版本之后，如 “R(`>=2.14.0`)”。
- DESCRIPTION 文件中的 Suggests 与和 Depends 域类似，但不是本扩展包必须的，比如仅用在某个例子中或测试中、仅用来编译 vignettes。

### NAMESPACE 文件

- NAMESPACE 文件如下：

```
useDynLib(testpack)
exportPattern("^[:alpha:]+$")
importFrom(Rcpp, evalCpp)
```

- 第一行指定调用本软件包时，需要调入的本扩展包的动态链接库。
- 第二行指定扩展包需要对外部可见的 R 函数是所有函数名字以字母开头的 R 函数。用户可以自己指定其它的模式或者指定固定的若干个函数。
- 第三行说明了需要从 Rcpp 包导入 evalCpp 函数。

### 重新编译

- 修改了扩展包中的 C++ 源程序后，需要重新编译。
- 只要在 R 中把工作目录设为软件包的子目录内，运行

```
compileAttributes()
```

- 这会自动生成两个文件，一个是 `src/RcppExports.cpp`，是 C++ 程序的接口函数。

- 另一个是 `R/RcpExports.R`，用 `.Call` 来调用 C++ 接口函数，转换成 R 函数。
- 这两个文件不要自己修改。

### 建立 C++ 用的接口界面

- 利用了 Rcpp 的软件包的，其 C++ 程序中的函数通常是转换成 R 函数的。
- 在 C++ 源程序中加入特殊注释

```
//[[Rcpp::interfaces(r, cpp)]]
```

则软件包在编译时也会生成该源程序文件中函数的外部可访问的接口，这些接口的界面会在安装后的包的 `include` 子目录中出现，在开发时出现在 `inst/include` 子目录中。

- 设要生成的扩展包名为 `testpack`，则界面文件包括 `include` 子目录中的 `testpack_RcppExports.h` 文件和 `testpack.h` 文件，`testpack.h` 文件仅用来包含 `testpack_RcppExports.h` 文件。
- 如果需要添加自己的一些界面程序，可以修改 `testpack.h` 文件，这时需要去掉文件开始的自动生成标记，并且保留对 `testpack_RcppExports.h` 文件的包含。
- 导出的 C++ 界面都在与制作的扩展包同名的名字空间中，比如，如果制作的软件包名为 `testpack`，其中导出的一个 C++ 函数为 `convolveCpp`，则在别的包 C++ 中调用时，应该包含 `testpack.h` 文件，并用 `testpack::convolveCpp()` 格式调用。
- 如果自己本扩展包需要在编译时包含这些头文件，需要自己编辑 `src` 子目录中的 `Makevars` 文件和 `Makevars.win` 文件，添加行：

```
PKG_CPPFLAGS += -I../inst/include/
```

- abc