第 1 章 R 简 介

1

R 是属于 GNU 系统的一款自由、免费且开源的高级编程语言和软件环境。它是一款集数据分析、统计计算和绘图功能于一体的整合性软件，同时也是一门功能强大的程序设计语言。R 最初由 Ross Ihaka 与 Robert Gentleman 开发，其语法与 AT & T 贝尔实验室 Rick Becker、John Chambers 与 Allan Wilks 等人开发的 S 语言相似。R 支持 Windows、UNIX、Linux 及 macOS 等多种操作系统。目前，R 的核心开发团队由来自世界各地的不同机构组成，其官方网站地址为 http://www.r-project.org。在此网站上，我们可以查阅大量关于 R 的文档、图书及其他资源。R 的应用领域广泛，包含数据分析、统计分析、数据挖掘、机器学习、推荐系统、文本挖掘及深度学习等。

1.1 R 软件介绍

读者可从 R 官方网站下载适合自己操作系统的新版 R 软件。安装 R 软件后，可以搜索并下载 RStudio。RStudio 是一个专为 R 设计的集成开发环境。R 和 RStudio 在 Windows 操作系统上的安装步骤详见附录 A 和附录 B。

R 官方网站提供了功能非常强大的工具集，我们可以从网站上安装相关的应用包（Package，或称为程序包，软件包）。R 提供了超过 1 万个免费包。当计算机连接到因特网后，若使用 Windows 版本，可以通过“程序包”菜单选项轻松安装这些免费的程序包。用户可以从该菜单中选择“加载程序包”选项来选择可用的包。在选择好所需的包之后，R 软件将自动下载并安装所选包。本书中的范例及操作皆在 Windows 操作系统下进行，如果要在 UNIX、Linux 或 macOS 操作系统上运行 R 软件，则需要进行相应的调整。我们也可以自行安装包，例如安装 C5.0 决策树包 C50（注意英文字母大小写），只需在 R 提示符“>”后输入以下指令（注意：

当提示符号为“+”时，表示程序正在执行中，或在等待未执行完成的指令）：

```
> install.packages("C50")
```

可使用以下指令来调用 C50 包中提供的函数：

```
> library(C50)
```

若要删除已安装的包，例如 C50 包，可使用代码如下：

```
> remove.packages("C50")
```

R 软件是一种语法非常简单的表达式语言（Expression Language）。它支持对象（Object），且对象名（即变量名）的第一个字母必须是英文字母或句点（.）。若对象名以句点开始，则其后不能紧跟数字。例如，.2iswrong 是无效的对象名。R 语言中的对象在使用前无须声明，但对象名称中的字母大小写会被区分，因此 X 和 x 被视为不同的对象。R 语言保留了一些标识符作为指令名（即保留字），如 c 与 NA 等。R 语言的赋值（Assignment）操作可以使用符号“<-”（也可以使用“=”），示例代码如下：

```
# 赋值表达式
> x <- 10
> x
[1] 10

> X <- x^2
> X
[1] 100
> z <- sqrt(X)
> z
[1] 10
```

可以通过对象名的数据属性（Attribute）来描述对象的特性。也就是说，对象名的数据属性决定了该对象的属性。所有对象名都有两个基本属性：数据类型（Mode）和长度（Length）。对象中的元素（Element）共有 4 种基本数据类型：数值（Numeric）、字符串（Character）、复数（Complex）和逻辑（Logical）。虽然存在其他数据类型，但它们不能用于表示数据，例如函数（Function）或表达式（Expression）。长度（Length）是指对象中元素的数量。对象的数据类型和长度可以分别通过 mode()函数和 length()函数获得。代码如下：

```
# 对象的数据类型和长度
> x <- 10
> x
[1] 10

> mode(x)
```

```
[1] "numeric"

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

如果要在同一行中运行多个表达式，可以使用分号（;）将它们隔开，代码如下：

```
> x <- 10; y <- x^2; z <- sqrt(y)
> z
[1] 10
```

注释可以放在程序中的任何地方，所有以“#”号开头的行都是注释，代码如下：

```
# 整数
> x <- 10
> x
[1] 10
> mode(x)
[1] "numeric"
> length(x)
[1] 1
# 浮点数（实数）
> y <- 10.9
> y
[1] 10.9
> mode(y)
[1] "numeric"
> length(y)
[1] 1

# 逻辑
> z <- T
> z
[1] TRUE
> mode(z)
[1] "logical"
> length(z)
[1] 1

# 字符串
> a <- "Hello"
> a
[1] "Hello"
```

```
> mode(a)
[1] "character"
> length(a)
[1] 1

# 复数
> z <- 4+2i
> z
[1] 4+2i
> mode(z)
[1] "complex"
> length(z)
[1] 1
```

1.2　R 对象介绍

R 是面向对象的程序设计语言，其常用对象有向量（Vector）、数组（Array）、矩阵（Matrix）、数据框（Data Frame）、因子（Factor）及列表（List）等。

1.2.1　向量

向量由包含相同数据类型的元素组成。R 程序中最简单的结构就是由一串有序数值构成的数值向量。假如用户要建立一个含有 6 个数值的向量 ***V***，其值分别为 10、5、3.1、6.4、9.2 和 21.7，可以在 R 程序中调用 c()函数来实现，代码如下：

```
# 向量
> V <- c(10, 5, 3.1, 6.4, 9.2, 21.7)
> V
[1] 10.0 5.0 3.1 6.4 9.2 21.7
> length(V)
[1] 6
> mode(V)
[1] "numeric"
```

也可以调用 assign()函数来实现与上述程序相同的结果，代码如下：

```
> assign("V", c(10, 5, 3.1, 6.4, 9.2, 21.7))
> V
[1] 10.0 5.0 3.1 6.4 9.2 21.7
> length(V)
```

```
[1] 6
> mode(V)
[1] "numeric"
```

在某些情况下，向量的元素可能会丢失。当向量中的元素为缺失值（Missing Value）时，可以给该元素赋值为 NA（需大写），代码如下：

```
# 缺失值
> V <- c(10, 5, NA, 6.4, 9.2, 21.7)
> V
[1] 10.0 5.0 NA 6.4 9.2 21.7
```

我们可以使用方括号“[]”来访问向量中的特定元素。值得注意的是，在 R 语言中，向量对象的第一个元素索引值（Index）默认是从 1 开始的，而不是从 0 开始，代码如下：

```
# 访问向量中的特定元素
> V[2]
[1] 5
```

R 语言中还提供 Inf、–Inf 及 NaN（Not a Number），而 NULL 表示对象的长度为 0，代码如下：

```
# Inf -Inf  NaN NULL 说明
>  V <- c(1,-2,0)
>  V/0
[1]  Inf -Inf  NaN

> V <- NULL
> length(V)
[1] 0
```

此外，还可以使用“:”操作符创建连续向量并进行访问，代码如下：

```
# 创建连续向量并进行访问
> V2=1:10
> V2
[1]  1  2  3  4  5  6  7  8  9 10

> V2[1]
[1] 1

> V2[2:4]
[1] 2 3 4
```

1.2.2　数组

数组可以看作是多维度的向量。例如，一个三维数组 ***X*** 可以用 ***X***[*i*,*j*,*k*]来指向其中的特定元素。假设数组 ***X*** 的维度向量是 ***c***(3,4,2)，则表示数组 ***X*** 中有 3×4×2 = 24 个元素，这些元素依次为 ***X***[1,1,1], ***X***[2,1,1],⋯, ***X***[2,4,2], ***X***[3,4,2]。

假设 ***X*** 是一个包含 24 个元素的向量，代码如下：

```
> X <- 1:24
```

可以调用 dim()函数来指定数组的维度（Dimension），让 ***X*** 变成一个 3×4×2 的三维数组，R 程序会按照列（Column）方式排列元素，代码如下：

```
# 调用dim()函数用于指定数组的维度
> dim(X) <- c(3,4,2)
> X
, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

我们可以让 ***X*** 变成一个 4×6 的二维数组，代码如下：

```
# 改变维度
> dim(X) <- c(4,6)
> X
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    2    6   10   14   18   22
[3,]    3    7   11   15   19   23
[4,]    4    8   12   16   20   24
```

要创建一个数组，也可以直接调用 array()函数，此函数的第一个参数用于指定数据向量，第二个参数用于指定数组的维度。假设要创建一个 3×4×2 的三维数组，代码如下：

```
# 调用array()函数来创建数组
> X <- array(1:24, dim = c(3,4,2))
> X
, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

假设要创建一个 4×6 的二维数组，代码如下：

```
> X <- array(1:24, dim = c(4,6))
> X
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    2    6   10   14   18   22
[3,]    3    7   11   15   19   23
[4,]    4    8   12   16   20   24
```

需要注意的是，可以通过以下代码创建一个所有元素的值都是 0 的三维数组：

```
> X <- array(0, dim = c(3,4,2))
> X
, , 1

     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0

, , 2

     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
```

```
[3,]    0    0    0    0
```

此外，还可以调用 rbind()函数和 cbind()函数来创建数组。rbind()函数将向量按行（Row）合并成一个数组，而 cbind()函数则将向量按列（Column）合并成一个数组，代码如下：

```
# 调用rbind()函数和cbind()函数来创建数组
> X1 <- c(1,2,3,4)
> X2 <- c(5,6,7,8)
> X  <- rbind(X1,X2)
> X
   [,1] [,2] [,3] [,4]
X1    1    2    3    4
X2    5    6    7    8

> X <- cbind(X1,X2)
> X
     X1 X2
[1,]  1  5
[2,]  2  6
[3,]  3  7
[4,]  4  8
```

1.2.3　矩阵

矩阵（Matrix）是二维数组。要创建一个矩阵，可以调用 matrix()函数。在 R 4.0 及以上的版本中，矩阵和数组的处理方式更加一致。虽然在概念上矩阵只是二维数组，但在 R 的早期版本中，矩阵和数组对象的处理方法在某些情况下并不相同。自 R 的新版本开始，矩阵对象将继承自数组类，从而消除了这种处理不一致的问题，代码如下：

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

参数说明如下：

- nrow：表示矩阵的行数。
- ncol：表示矩阵的列数。
- byrow：表示矩阵中的数据是按行（byrow=TRUE）还是按列（byrow=FALSE）的顺序排列。
- dimnames：用于为行或列命名。

```
# 调用matrix()函数创建二维数组
> X <- matrix(1:24, nrow=4, ncol=6, byrow=TRUE)
> X
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    4    5    6
[2,]    7    8    9   10   11   12
[3,]   13   14   15   16   17   18
[4,]   19   20   21   22   23   24

> X <- matrix(1:24, nrow=4, ncol=6, byrow=FALSE)
> X
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    2    6   10   14   18   22
[3,]    3    7   11   15   19   23
[4,]    4    8   12   16   20   24
```

在 R 4.0 及以上版本中，矩阵对象将继承自数组类，可调用 class()函数进行确认。

```
> class(X)
[1] "matrix" "array"
```

矩阵也可以通过调用 rbind()函数和 cbind()函数来创建。t()函数是矩阵的转置（Transposition）函数，而 nrow()函数和 ncol()函数分别返回矩阵的行数和列数。

```
# 调用rbind()函数和cbind()函数来创建矩阵和转置矩阵
> X1 <- c(1,2,3)
> X2 <- c(4,5,6)
> X3 <- c(7,8,9)
> X <- cbind(X1,X2,X3)
> X
     X1 X2 X3
[1,]  1  4  7
[2,]  2  5  8
[3,]  3  6  9
> Y=t(X)
> Y
   [,1] [,2] [,3]
X1    1    2    3
X2    4    5    6
X3    7    8    9

# 调用nrow()函数和ncol()函数返回矩阵的行数和列数
> m <- nrow(Y)
> m
```

```
[1] 3
> n <- ncol(Y)
> n
[1] 3
```

若要显示矩阵 ***X*** 的第一列元素，可执行代码如下：

```
# 显示矩阵整列的元素
> X[,1]
[1] 1 2 3
```

若要显示矩阵 ***X*** 的第二行元素，可执行代码如下：

```
# 显示矩阵整行的元素
> X[2,]
X1 X2 X3
 2  5  8
```

若要显示矩阵 ***X*** 的第一行及第三行元素，可执行代码如下：

```
# 显示矩阵部分行的元素
> X[c(1,3),]
     X1 X2 X3
[1,]  1  4  7
[2,]  3  6  9
```

若要删除矩阵 ***X*** 的第一列元素，可执行代码如下：

```
# 删除矩阵的第一列元素
> X[,-1]
     X2 X3
[1,]  4  7
[2,]  5  8
[3,]  6  9
```

若要删除矩阵 ***X*** 的第二行元素，可执行代码如下：

```
# 删除矩阵的第二行元素
> X[-2,]
     X1 X2 X3
[1,]  1  4  7
[2,]  3  6  9
```

eigen()函数可以用来计算矩阵的特征值（Eigen Value）和特征向量（Eigen Vector），代码如下：

```
# 矩阵运算
> eigen(Y)
$values
[1]  1.611684e+01 -1.116844e+00 -1.303678e-15

$vectors
          [,1]       [,2]      [,3]
[1,] -0.2319707 -0.78583024  0.4082483
[2,] -0.5253221 -0.08675134 -0.8164966
[3,] -0.8186735  0.61232756  0.4082483
```

可以用“%*%”运算符表示矩阵相乘，执行代码如下：

```
> z <- Y%*%X
> z
   X1  X2  X3
X1 14  32  50
X2 32  77 122
X3 50 122 194
```

若要修改矩阵 $\boldsymbol{Z}$ 的列名称，可执行代码如下：

```
# 修改矩阵的列名称
> colnames(z) <- c("c1","c2","c3")
> z
   c1  c2  c3
X1 14  32  50
X2 32  77 122
X3 50 122 194
```

若要修改矩阵 $\boldsymbol{Z}$ 的行名称，可执行代码如下：

```
# 修改矩阵的行名称
> rownames(z) <- c("r1","r2","r3")
> z
   c1  c2  c3
r1 14  32  50
r2 32  77 122
r3 50 122 194
```

1.2.4　数据框

数据框与矩阵的结构类似，因为两者都是二维结构。然而，与矩阵不同的是，数据框的不

同列可以包含不同的数据类型，但同一列的数据类型必须相同。数据框的每一行可视为一组观察值（Observation）或案例（Case），其变量名称由每一列的名称来定义。

使用以下方式创建数据框：

```
# 创建数据框
> id <- c(1, 2, 3, 4)
> age <- c(25, 30, 35, 40)
> sex <- c("Male", "Male", "Female", "Female")
> pay <-c (30000, 40000, 45000, 50000)
> X.dataframe <- data.frame(id, age, sex, pay)
> X.dataframe
  id age    sex   pay
1  1  25   Male 30000
2  2  30   Male 40000
3  3  35 Female 45000
4  4  40 Female 50000
```

使用以下方式获取或引用数据框中第三行第二列的元素：

```
> X.dataframe[3,2]
[1] 35
```

使用列的名称获取或引用数据框中对应列的所有元素：

```
# 使用列的名称获取列元素
> X.dataframe$age
[1] 25 30 35 40
```

使用以下方式获取或引用数据框中对应列的名称及元素：

```
# 使用列的名称获取列的名称及元素
> X.dataframe[2]
  age
1  25
2  30
3  35
4  40
```

R 程序提供了与 Excel 界面类似的编辑器来创建或修改数据框的值（见图 1-1）：

```
# 编辑或修改数据框的值
> edit(X.dataframe)
```

若确定要修改数据框的值，则需使用赋值运算符：

```
> X.dataframe <- edit(X.dataframe)
```

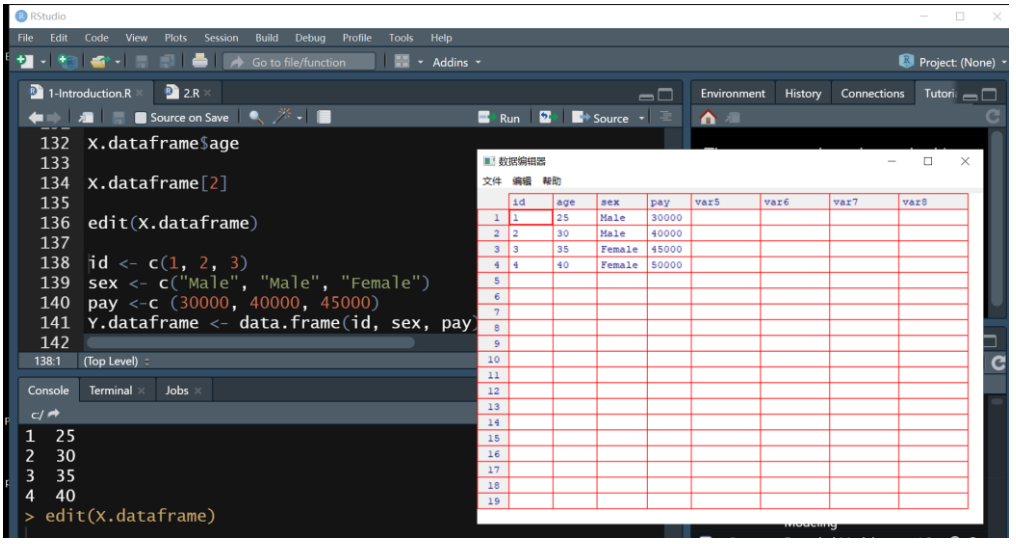

图 1-1　edit()函数

注　意

如果要修改数据框的值，必须先在要修改或编辑的字段上双击鼠标左键才可以开始修改。

1.2.5　因子

因子（Factor）是一种特别的向量，用于将长度相同的离散数据向量分组（Grouping）。在字符串向量中，每一个元素取一个离散值，因子有一个特殊属性，称为水平（Levels），表示因子变量可取的所有离散值。一旦被设置为因子，R 在打印时就不会加上双引号。

可以调用 factor()函数来创建因子，其中 Levels 代表因子可取的所有离散值：

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

1.2.6　列表

R 语言的列表是由有序序列（Order Sequence）构成的对象。列表的组成元素（Component，也简称元素）可以是异质（Heterogeneous）的对象，也就是说，各个组成元素的数据类型可以不同。一个列表中的元素可以包括数值、逻辑、字符串、复数、向量、矩阵、因子以及数据框等。

可以调用 list()函数来创建列表：

```
# 创建列表
> id <- c(1, 2, 3)
> sex <- c("Male", "Male", "Female")
> pay <-c (30000, 40000, 45000)
> Y.dataframe <- data.frame(id, sex, pay)

> gender <- factor(c("男", "男", "女"))

> Paul.Family <- list(name="Paul", wife="Iris", no.kids=3,
kids.age=c(25,28,30), gender, Y.dataframe)
> Paul.Family
$name
[1] "Paul"

$wife
[1] "Iris"

$no.kids
[1] 3

$kids.age
[1] 25 28 30

[[5]]
[1] 男 男 女
Levels: 女 男

[[6]]
  id    sex   pay
1  1   Male 30000
2  2   Male 40000
3  3 Female 45000
```

可以使用“$”符号来获取或引用列表中的某个元素。例如要获取 Paul.Family 中的第 4 个元素，可执行如下指令：

```
# 使用$符号来获取或引用列表中的元素
> Paul.Family$kids.age
[1]  25 28 30
```

也可以使用双重方括号“[[]]”及索引值来获取或引用列表中某个元素。例如要获取 Paul.Family 中的第 4 个元素，可执行如下指令：

```
# 使用双重方括号“[[]]”来获取或引用列表中的元素
> Paul.Family[[4]]
```

```
[1]  25 28 30
```

若使用方括号“[]”及索引值，则可获取或引用列表中某个位置的组成元素及其名称，执行如下命令：

```
# 使用方括号“[]”来获取或引用列表中的元素及其名称
> Paul.Family[4]
$kids.age
[1] 25 28 30
```

若使用双重方括号“[[]]”获取第二个孩子的年龄，可执行如下的命令：

```
> Paul.Family$kids.age[2]
[1] 28
```

或

```
> Paul.Family[[4]][2]
[1] 28
```

1.2.7　对象转换

R 语言提供了多个函数用于不同对象之间的转换，这些函数包括 as.vector()、as.array()、as.matrix()、as.factor()、as.data.frame()及 as.list()等。

创建数据框对象，命令如下：

```
# 转换向量->数据框->矩阵->向量
> id <- c(1, 2, 3, 4)
> x <- data.frame(id)
> x
  id
1  1
2  2
3  3
4  4
```

将数据框对象转换为矩阵对象，命令如下：

```
> matrix.x=as.matrix(x)
> matrix.x
     id
[1,]  1
[2,]  2
[3,]  3
```

```
[4,]  4
```

将矩阵对象转换为向量对象，命令如下：

```
> vector.x=as.vector(matrix.x)
> vector.x
[1] 1 2 3 4
```

1.3 习　　题

（1）使用 Repository 安装 C50 库（包），如图 1-2 所示。

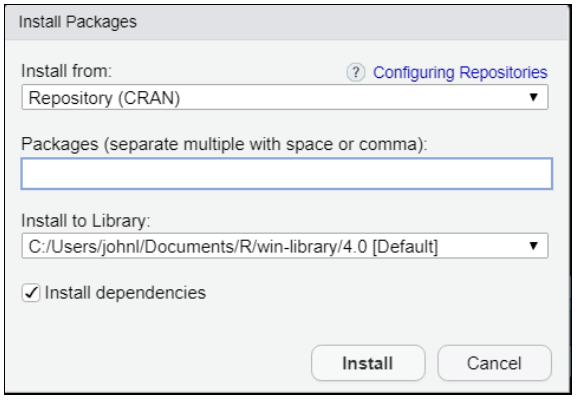

图 1-2　自动安装

（2）使用 Package Archive File 安装 RGtk2_2.20.36.zip 库（包），如图 1-3 所示。

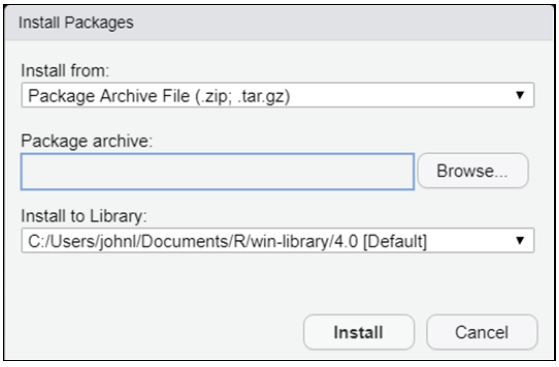

图 1-3　手动安装

第 2 章

读 写 数 据

2

对于数据的读取和写入，可以先调用 setwd（"C:/data"）函数或 setwd（"/home/R"）函数来切换工作目录，再调用 getwd()函数来确认当前工作目录。

```
> setwd("c:/")
> getwd()
[1] "c:/"
```

2.1 读 取 数 据

R 语言经常调用 read.table()、read.csv()或 scan()函数来读取存储在文本文件（ASCII）中的数据，read.table()函数主要用于数据框中的操作，可以直接将整个外部文件读入数据框对象。

外部文件常要求有特定的格式，例如：

（1）第一行（Line）可以是表头（Header），包含各列数据的变量名（Variable Name）。不过，表头也可以省略。

（2）其余各行为各行的值。

要读取 R 工作目录中的 X.csv 文件（见表 2-1），可执行如下指令：

表 2-1 X.csv 文件

id	age	sex	pay
1	25	Male	30000
2	30	Male	40000

（续表）

id	age	sex	pay
3	35	Female	45000
4	49	Female	50000

```
# 调用read.table()函数
> setwd("c:/") # 设置工作目录

# encoding表示编码格式
> X <- read.table("X.csv",sep=",",header=TRUE, encoding="UTF-8")
> X
  id age    sex   pay
1  1  25   Male 30000
2  2  30   Male 40000
3  3  35 Female 45000
4  4  49 Female 50000

> X$age
[1] 25 30 35 49

> X[1,2]
[1] 25
```

注意，CSV 文件中的数据是用逗号分隔开的，所以加入 sep=","来指定分隔符是逗号。若 header=FALSE，则使用默认的 V_1,V_2,…,$V_{\#}$来作为表头（header）的名称。

```
# 使用header参数要小心
> setwd("c:/")
> X <- read.table("X.csv",sep=",", header=F, encoding="Big5")
> X
  V1  V2    V3    V4
1 id age    sex   pay
2  1  25   Male 30000
3  2  30   Male 40000
4  3  35 Female 45000
5  4  49 Female 50000
```

也可以调用 read.csv()函数：

```
# 调用read.csv()函数
> setwd("c:/")
> X <- read.csv("X.csv", header=TRUE, encoding="GBK")
```

```
> X
  id age    sex   pay
1  1  25   Male 30000
2  2  30   Male 40000
3  3  35 Female 45000
4  4  49 Female 50000

> X <- read.csv("X.csv", header=FALSE, encoding="UTF-8")
> X
  V1  V2    V3    V4
1 id age    sex   pay
2  1  25   Male 30000
3  2  30   Male 40000
4  3  35 Female 45000
5  4  49 Female 50000
```

可以使用 Excel 表将 X.csv 文件转换为 X.txt 文件（请使用文本文件，而不要使用 Unicode 字符编码的文件），再将文件读入。若文件中有中文，则需要先确认文件中的字符编码，将转换文件的字符编码后再将文件读入。macOS 操作系统默认的字符编码是 UTF-8，而 Windows 操作系统的中文版默认的字符编码是 GBK（简体字）或 Big5（繁体字）。使用 Windows 操作系统时，可以使用 notepad 打开 CSV 文件，再另存为 UTF-8 编码的文件，而后再读入文件。

读取网络上的 iris.data，网址为 https://gairuo.com/file/data/dataset/iris.data，读者也可使用本书提供的 iris.data：

```
> X <- read.csv("https://gairuo.com/file/data/dataset/iris.data", header = TRUE,
encoding = "UTF-8")
> X
```

R 4.0及以上版本最重要的更新之一是导入的字符串数据不再被默认转换成因子变量（Factor）。过去，stringsAsFactors选项默认为TRUE，因此导入的字符串数据都会被转换成因子变量，但是在新版本中，stringsAsFactors选项默认为FALSE。

```
> mode(X$species)
[1] "character"
> class(X$species)
[1] "character"

> X <- read.table("X.txt",header=TRUE, encoding="UTF-8")
> X
  id age    sex   pay
1  1  25   Male 30000
2  2  30   Male 40000
```

```
3  3  35 Female 45000
4  4  49 Female 50000
```

scan()函数比 read.table()函数更加灵活，因为 scan()函数可接收键盘输入的数据：

```
> X <- scan("")
1: 12       # 输入数据后按Enter键
2: 10
3: 5
4: 6.3
5:          # 不再输入数据时，可再按Enter键结束输入
Read 4 items
> X
[1] 12.0 10.0  5.0  6.3
```

scan()函数也可以指定输入数据的数据类型，例如要创建列表对象：

```
> my=scan(file="",what=list(name="",pay=integer(0),sex=""))
1: peter 50000 M     # 输入数据后按Enter键
2: lisa 40000 F
3: johnson 65000 M
4:                   # 不再输入数据时，可再按Enter键结束输入
Read 3 records

> mode(my)
[1] "list"
```

参数说明如下：

- file：文件路径，file=""表示由键盘输入值。
- what：设置输入值的数据类型，上述例子为创建列表对象，且第一个元素 name=""表示字符串，第二个元素 pay=integer(0)表示整数，而第三个元素 sex=""也是字符串。

scan()函数也可以读取 CSV 文件和文本文件，表 2-2 所示为 X1.csv 文件，读取命令如下：

表2-2　X1.csv文件

id	age	pay
1	25	30000
2	30	40000
3	35	45000
4	49	50000

```
> X <- scan("X1.csv", sep=",")
```

```
Read 12 items
> X
 [1]     1    25 30000     2    30 40000     3    35 45000     4    49   50000
```

使用 Excel 表将 X1.csv 文件转换为 X1.txt 文件后再读入：

```
> X <- scan("X1.txt")
Read 12 items
> X
 [1]     1    25 30000     2    30 40000     3    35 45000     4    49   50000
```

2.2　写 入 数 据

若需将存储数据或分析结果输出至外部文件，可以调用 write.table()函数，命令如下：

```
> write.table(X,"C:/X_File.csv",row.names=FALSE,col.names=TRUE,sep=",",
+           fileEncoding="GBK")
```

参数说明如下：

- X: 表示要输出至外部文件的对象。
- "C:/X_File.csv": 用于指定要输出至外部文件的文件名和路径。
- row.names: 表示输出至外部文件是否加上行名称。
- col.names: 表示输出至外部文件是否加上列名称。
- sep=",": 表示分隔符。
- fileEncoding: 表示输出至外部文件的字符编码格式。

R语言提供了一些内建的数据集，可以通过调用data()函数来查询这些内建的数据集，如图2-1所示。

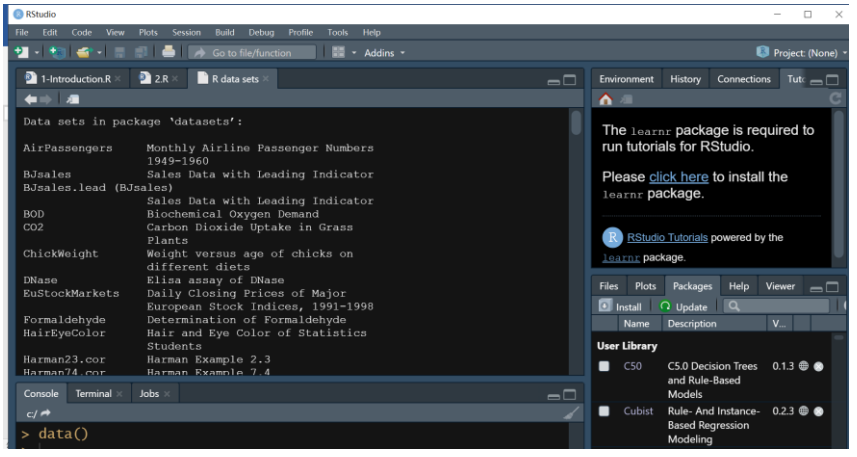

图 2-1　调用 data()函数查询内建的数据集

```
> data()
```

可调用 data()（数据集名称）函数来使用内建的数据集。例如，要使用 iris 数据集，可执行如下指令：

```
# 使用内建的iris数据集
> data(iris)
> iris
```

结果如图 2-2 所示。

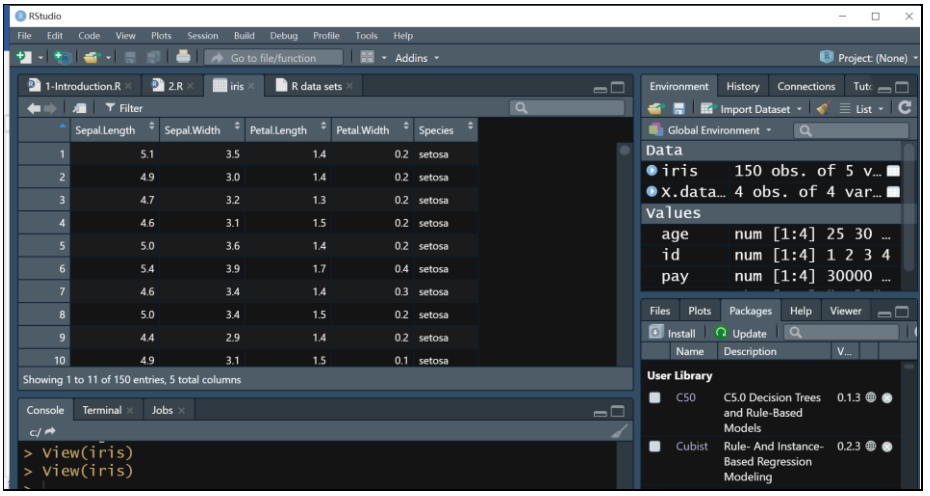

图 2-2　iris 数据集

可调用 str()函数来取得数据集的数据结构：

```
> str(iris)
'data.frame':  150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1
1 1 ...
```

2.3　读写 RData 数据

R 语言可将使用的对象存储为 RData 格式的外部文件，也可将 RData 格式的外部文件读回 R 系统中。若想把 iris 数据集存储至 c:\iris.RData，可调用 save()函数，命令如下：

```
> setwd("c:/")
```

```
> data(iris)
> save(iris,file="iris.RData")
```

若想把 iris.RData 文件读取到 R 系统中，可调用 load()函数，命令如下：

```
> getwd()
[1] "c:/"

> load("iris.RData", .GlobalEnv)
```

参数说明如下：

- .GlobalEnv：表示用户正在使用的工作空间（Workspace）。

2.4　读写 Excel 数据

首先，安装 xlsx 包，并加载 xlsx 包：

```
> install.packages("xlsx")
> library("xlsx")
```

然后，设置工作目录，使用 iris 数据集：

```
> setwd("c:/")
> data("iris")
```

接着，调用 write.xlsx()函数执行写入操作：

```
> write.xlsx(iris, file="myexcel.xlsx", sheetName="IRIS", append=T)
> write.xlsx(mtcars, file="myexcel.xlsx", sheetName="MTCARS", append=+TRUE)
> write.xlsx(Titanic, file="myexcel.xlsx", sheetName="TITANIC", appen+d=TRUE)
```

最后，调用 read.xlsx()函数将 myexcel.xlsx 文件的第一个工作表读入 res 变量：

```
# read the first sheet
> sheetIndex=1
> res <- read.xlsx("myexcel.xlsx", sheetIndex, header=TRUE)
```

2.5　习　　题

从 UCI 机器学习仓库（https://archive.ics.uci.edu/dataset/53/iris）下载 iris 数据集，读者也可使用本书提供的 iris.data 并转换成 iris.csv 文件。

第 3 章

从流程控制到函数

3

R 语言是一种表达式语言，其所有的指令都是函数或表达式。赋值表达式“<-”的返回值就是用于赋值的值。在 R 语言中，最简单的运行方式就是一行一行地输入表达式，然后显示运行的结果，例如：

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

如果想直接显示运行的结果，可将指令用圆括号（()）括起来，例如：

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

指令也可以用花括号（{}）括起来，例如{expr#1;…;expr#m}，以运行多条指令，例如：

```
> {a <- c(1,2,3);x=a+2}
> x
[1] 3 4 5
```

R 语言的流程控制提供了条件执行（Condition Execution）与循环（Loop）等结构性语法。

3.1 条 件 执 行

R 语言的条件执行包含 if-else 语句、ifelse()函数和 switch()函数。

R 语言的 if-else 语句为：

```
if (condition) expr#1 else expr#2
```

或

```
if (condition) expr#1
```

参数说明如下：

- condition: 表示条件判断表达式，必须返回一个布尔值（TRUE 或 FALSE），而&&（AND）和||（OR）逻辑运算符常用于条件判断表达式的条件控制部分。
- expr#1: 一般表达式。
- expr#2: 一般表达式。

```
> x <- 6
> if (x>5) y=2 else y=4
> y
[1] 2

> X <- 3
> if (X<5) Y=10
> Y
[1] 10
```

若有多个表达式，则可以使用花括号括起来，例如{expr#1;…;expr#m}：

```
> X <- 3
> Y <- 1
> if (X<5 && Y<5)
+ {Y <- 10; Z <- 5}
> Y
[1] 10
> Z
[1] 5
```

R 语言的 ifelse()函数可用于简单的逻辑判断，若 condition 结果为 TRUE，则返回 a；否则返回 b。其语法为：

```
ifelse (condition, a, b)

> X <- 20
> Y=ifelse(X>5, 2, 3)
> Y
[1] 2
```

R 语言的 switch()函数语法为：

```
switch (condition, expr#1,…,expr#m)
```

参数说明如下：

- condition 为正整数或文字。
  - 若 condition 的值为正整数 n，则执行表达式 expr#n，若 n 值大于 m 或小于 1 时，switch()函数无返回值。
  - 若 condition 值为文字，则执行相对应的表达式。

```
> X <- 1
> switch(X, 5, sum(1:10), rnorm(5))
[1] 5

> X <- 2
> switch(X, 5, sum(1:10), rnorm(5))
[1] 55

> X <- 3
> switch(X, 5, sum(1:10), rnorm(5))
[1] -0.185252822 -0.351313575 -0.008195255 -1.920097610 -0.680803488

> X <- 4
> switch(X, 5, sum(1:10), rnorm(5))    # 无返回值
>

> Y <- 1
> switch(Y, juice="Apple", meat="Pork")
[1] "Apple"
```

switch()函数也可以使用文字，例如：

```
> Y <- "juice"
> switch(Y, juice="Apple", meat="Pork")
[1] "Apple"
```

3.2　循 环 控 制

R 语言的循环控制语句包含 for、while 及 repeat。在循环中可使用 break 指令跳出循环，或使用 next 跳过当前一轮循环尚未执行的语句，直接进入当前循环体的下一轮循环。

R 语言的 for 循环语句的语法为：

```
for (index in expr#1) expr#2
```

或

```
for (index in expr#1) {expr#2;…;expr#m}
```

参数说明如下：

- index：表示循环索引。
- expr#1：表示数值或文字向量，例如 1:5 或 c("A","O","B","AB")。
- expr#2：根据 index 设计的区块表达式。for 循环会将 expr#1 向量中的每个元素按照顺序以一次一个的方式指定给 index，每指定一次，index 就会运行一次对应的 expr#2 表达式。
- {expr#2;…;expr#m}：多个表达式。

例如：

```
> X <- 0
> for(i in 1:5) X <- X+i
> X
[1] 15
```

```
> X <- 0
> Y <- 0
> for(i in 1:5) { X<- X+i; Y <- i^2}
> X
[1] 15
> Y
[1] 25
```

R 语言的 while 循环语句的语法为：

```
while (condition) expr#1
```

或

```
while (condition) {expr#1;…;expr#m}
```

参数说明如下：

- condition：当 condition 值为 TRUE 时，运行循环体内的表达式，并重复运行循环体内的指令，直到 condition 值为 FALSE。
- expr#1：一般表达式。
- {expr#1;…;expr#m}：多个表达式。

例如，求 1+2+…+9+10=55。

```
> sum <- 0
> i <- 1
> while (i <= 10) {sum <- sum + i; i <- i + 1} # condition i <= 10
```

```
> sum
[1] 55
```

Repeat 循环会重复运行表达式，通常在循环中设置检查控制循环的条件，并结合 break 指令。break 可用于结束循环，它是结束 repeat 循环的唯一方法。

R 语言的 repeat 循环语句的语法为：

```
repeat expr
```

参数说明如下：

- expr: 一个用花括号括起来的区块表达式，必须设置检查循环控制条件。若符合特定的循环控制条件，则利用 break 指令结束循环。

例如，求 1+2+…+9+10=55。

```
> sum <- 0
> i <- 1
> repeat {
+ sum <- sum + i
+ i <- i + 1
+ if ( i > 10 ) break  # 结束循环
+ }
> sum
[1] 55
```

break 指令可用于结束循环，它是结束 repeat 循环的唯一办法；而 next 指令可用来跳过当前一轮循环尚未执行的语句，直接进入下一轮循环。

例如，求 1+3+…+47+49=625。

```
> sum <- 0
> for (i in 1:50)
+ {
+ if ( i %% 2 == 0 ) next   # %%是求偶数
+ sum <- sum + i            # 若i是偶数，则不运行sum <- sum + i
+ }
> sum
[1] 625
```

同其他程序设计语言一样，R 语言也常需要用到循环，但 R 语言的循环运行效率较差，因此应尽量避免使用循环。在 R 语言中，有些函数如 apply()、lapply()和 sapply()等，可以更高效地执行类似循环的指令。

apply(x, MARGIN, FUN, …)函数的作用是将一个指定函数的计算运用于数组或矩阵的每一列或每一行。

参数说明如下：

- x: 表示要进行计算的目标数组或矩阵。
- MARGIN: 其值为 1 或 2。1 表示行，2 表示列。
- FUN: 表示指定的函数。

例如，调用 sum()函数求出数组的每一行的总数。

```
> X <- array(1:24, dim = c(4,6))
> X
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    2    6   10   14   18   22
[3,]    3    7   11   15   19   23
[4,]    4    8   12   16   20   24
> apply(X,1,sum)
[1] 66 72 78 84
```

例如，调用 sum()函数求出数组的每一列的总数。

```
> X <- array(1:24, dim = c(4,6))
> X
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    2    6   10   14   18   22
[3,]    3    7   11   15   19   23
[4,]    4    8   12   16   20   24
> apply(X,2,sum)
[1] 10 26 42 58 74 90
```

lapply(X, FUN, …)函数的作用是将一个指定函数的计算运用于列表对象 X 的每一元素，并返回一个列表对象。返回的列表对象的长度与原列表对象 X 的长度一致。

例如：

```
> X <- list(a=1:10, b=exp(-1:1))
> lapply(X,sum)
$a
[1] 55

$b
[1] 4.086161
```

sapply(X,FUN,…)函数的功能与 lapply(X,FUN,…)函数的功能类似，但 sapply(X,FUN,…)函数返回一个向量或矩阵对象。

```
> X <- list(a=1:10, b=exp(-1:1))
> sapply(X,sum)
       a        b
55.000000  4.086161
```

3.3 函　数

R 语言提供的常用函数可参考附录 C。我们也可以自定义函数，其语法如下：

```
> myfun <- function(arg#1,arg#2,…) {expr#1;…;expr#m}
```

参数说明如下：

- arg#1,arg#2,…: 自变量（Argument），可以有多个。
- expr#1;…;expr#m: 表达式。自定义函数可以不返回函数值，R 语言默认将函数的最后一个表达式的结果作为返回值，也可以调用 return()函数返回函数的值。

例如：

```
> X <- 1
> myfun <- function(X) { Y <- X+2; return (Y) }
> myfun(X)
[1] 3
```

R 语言的自定义函数允许自变量有默认值。若调用函数时没有给自变量传入值，则使用默认值作为自变量的传入值；若自变量的传入值与默认值不同，则使用传入的值。

例如：

```
> X <- 2        # 自变量的传入值与默认值不同
> myfun <- function(X=1) { Y <- X+2; return (Y) }
> myfun(X)
[1] 4

> myfun <- function(X=1) { Y <- X+2; return (Y) }
> myfun()      # 若没有给自变量传入值，则使用默认值X=1
[1] 3
```

在 R 语言的自定义函数中，如果要修改函数外部对象的值，则只有使用“<<-”才能改变自定义函数外部对象的值。

例如：

```
> x <- 1
> myfun <- function(x) { x <- 2; print(x) }
```

```
> myfun(x)
[1] 2        # myfun中x的值
> x          # 无法改变外部对象x的值
[1] 1

> x <- 1
> myfun <- function(x) { x <<- 2; print(x) }
> myfun()
[1] 2        # myfun中改变外部对象x的值
> x          # 外部对象x的值已经改变
[1] 2
```

3.4 习　　题

（1）使用 `while（i <= 9）{ i <- i + 1;sum <- sum + i}`完成 1+2+…+9+10=55。

（2）使用 while 循环语句和 break 指令完成 1+2+…+9+10=55。