

 3.1 控制结构

R 语言是块状结构程序语言,块由大括号“{ }”划分,程序语句由换行符或分号分割。程序控制结构的作用是应用 if 条件语句、循环语句等控制程序的走向。程序控制结构又称为流程控制。R 语言的基本控制结构有顺序结构、分支结构和循环结构。

3.1.1 分支结构

条件分支语句在编程语言中非常常见,R 语言中,常用的条件分支语句有 if-else 语句和 switch 语句。

1. if-else 语句

创建 if-else 结构语句的基本格式如下。

```
if ( 条件 ) {  
    //结果为 TRUE 时执行语句组 1  
} else {  
    //结果为 FALSE 时执行语句组 2  
}
```

其中,条件是布尔表达式,当结果为 TRUE 时,执行 if 语句中的代码块,否则将执行 else 语句中的代码块,这是最简单的分支结构。if-else 结构语句还可以多重条件嵌套,使用 else if 语句完成多个条件的判断。除此之外,ifelse 结构比较紧凑,格式为 ifelse(condition, statement1, statement2)。若 condition 为 TRUE,则执行第一个语句;若 condition 为 FALSE,则执行第二个语句。当希望结构的输入和输出均为向量时,可使用 ifelse 语句。相关示例如下所示。

```
> x <- c("what", "is", "the", "Truth")  
> if ("Truth" %in% x) { print("Truth is found")  
+ } else { print("Truth is not found") }  
[1] "Truth is found"
```

```

> #嵌套的 if-else 结构语句
> a <- -1
> if (a < 0) {
+   result = 0
+ } else if (a < 1) {
+   result = 1
+ } else {
+   result = 2
+ }
> result
[1] 0
> #ifelse 语句
> x <- c(1,1,1,0,0,1,1)
> ifelse(x != 1, 1, 0)           #如果 x 的值不等于 1,输出 1,否则输出 0
[1] 0 0 0 1 1 0 0

```

2. switch 分支语句

switch 语句可以直接实现多分支语句。如果条件式结果等于 n,则执行第 n 条分支的处理;如果取值不符合条件,则返回“NULL”值。其格式为 switch(expression, list),其中 expression 为表达式,list 为列表。如果表达式返回值在 1 到 length(list)之间,则返回列表相应位置的值,否则返回“NULL”值。当表达式等于 list 的变量名时,返回变量名对应的值,否则返回“NULL”值。相关示例代码如下所示。

```

> switch (2, mean(1:10), 1:5, 1:10)           #输出第 2 个向量
[1] 1 2 3 4 5
> y <- "fruit"                                #输出变量名 fruit 对应的值
> switch(y, fruit = "apple", vegetable = "broccoli",meat="beef")
[1] "apple"

```

3.1.2 循环结构

常用的循环语句主要有 for 循环、while 循环和 repeat 循环,常见的控制语句有 break 语句和 next 语句,如表 3-1 和表 3-2 所示。

表 3-1 常见的循环结构

循 环 结 构	描 述
for 循环	基本的循环语句,变量的遍历
while 循环	类似 for 语句,满足条件执行循环体
repeat 循环	多次执行,满足条件退出循环

表 3-2 常见的控制语句

控制语句	描 述
break 语句	终止循环语句,执行循环后的下一语句
next 语句	跳过本次循环,执行下一次循环

1. for 循环

for 语句用于创建循环,格式为 `for (name in expr1) {expr2}`。其中,name 为循环变量,在每次循环时从 `expr1` 中顺序取值;`expr1` 是一个向量表达式,`expr2` 通常是一组表达式,当 name 的值包含在 `expr1` 中时,执行 `expr2` 的语句,否则循环终止。在循环过程中如果需要输出每次循环的结果,则可以使用 `cat()` 函数或 `print()` 函数。`cat()` 函数的格式为 `cat (expr1, expr2, ...)`,其中,`expr1`、`expr2` 为输出的内容,可为字符串或表达式。另外,符号“\n”表示换行,“\n”后的语句将在下一行输出。

以下为两个使用 for 循环的例子。在第一个例子中,`for (i in pv)` 表示 `i`,即 `pv` 中的每一个元素值,进入循环计算,`i` 的取值是具体的成绩分值;在第二个例子中,`for (i in 1:length(pv))` 表示从 1 到 `length(pv)` 中每一个符合条件的 `i` 值进入循环计算,`i` 的取值是位置。因此,在循环中的 if-else 分支语句的条件表达式使用 `pv[i]` 来代表成绩。最后结果虽然相同,但循环因子和用法是不同的,请思考这两个例子的不同点。

```
># 例 1
> pv <- c(40, 55, 70, 95, 82, 100, 66, 90)
> m <- 1
> for (i in pv) {
+   if (i < 60) {
+     result[m] <- "需要加强"
+   } else if (i <= 80) {
+     result[m] <- "继续加油"
+   } else {
+     result[m] <- "做得很好"
+   }
+   m <- m+1
+ }
> result
[1] "需要加强" "需要加强" "继续加油" "做得很好" "做得很好" "做得很好"
[7] "继续加油" "做得很好"
```

```
># 例 2
> pv <- c(40, 55, 70, 95, 82, 100, 66, 90)
> m <- 1
> for (i in 1:length(pv)) {
```

```
+      if (pv[i] < 60) {  
+          result[i] <- "需要加强"  
+      } else if (pv[i] <= 80) {  
+          result[i] <- "继续加油"  
+      } else {  
+          result[i] <- "做得很好"  
+      }  
+  }  
> result  
[1] "需要加强" "需要加强" "继续加油" "做得很好" "做得很好" "做得很好"  
[7] "继续加油" "做得很好"
```

2. while 循环

while 语句用于创建循环,格式为 while (cond) {expr}。其中,cond 为判断条件,expr 为一个或一组表达式。while 循环重复执行语句 expr,直到条件 cond 不为真为止。注意: for 是通过遍历一个向量来控制循环的次数,while 是直接设置判断条件的范围,这是两者的主要区别。示例如下。

```
> pv <- c(40, 55, 70, 95, 82, 100, 66, 90)  
> i <- 1  
> while(i <= length(pv) ) {  
+   if (pv[i] < 60) {  
+       result[i] <- "需要加强"  
+   } else if (pv[i] <= 80) {  
+       result[i] <- "继续加油"  
+   } else {  
+       result[i] <- "做得很好"  
+   }  
+   i <- i + 1  
+ }  
> result  
[1] "需要加强" "需要加强" "继续加油" "做得很好" "做得很好" "做得很好" "继续加油" "做得很好"
```

3.1.3 控制语句

repeat 是无限循环语句,不能自动停止,需要配合使用 break 语句跳出循环,格式为 repeat { if (cond) { break } }。repeat-break 循环的相关示例如下所示。

```
> pv <- c(40, 55, 70, 95, 82, 100, 66, 90)  
> i <- 1  
> repeat{
```

```
+      if (i > length(pv)) {  
+          break  
+      }  
+      if (pv[i] < 60) {  
+          result[i] <- "需要加强"  
+      } else if (pv[i] <= 80) {  
+          result[i] <- "继续加油"  
+      } else {  
+          result[i] <- "做得很好"  
+      }  
+      i <- i + 1  
+  }  
> result  
[1] "需要加强" "需要加强" "继续加油" "做得很好" "做得很好" "做得很好" "继续加油" "做得很好"
```



3.2 函 数

R 语言大量使用函数,因此,灵活掌握函数的用法可以完成很多工作。R 语言的基础包和第三方包提供很多函数,可以直接使用。在处理复杂问题中,也可以编写自定义函数来实现所需功能。本节重点介绍 R 语言常用的数学运算函数、字符处理函数、日期处理函数、自定义函数以及函数的嵌套等。

3.2.1 数学运算函数

1. 数学函数

和其他数据分析软件一样,R 语言中也有许多应用于计算和统计分析的函数,主要分为数学函数、统计函数、概率函数等。常用的数学函数如表 3-3 所示。

表 3-3 数学函数

函 数	描 述
abs(x)	求绝对值
sqrt(x)	求平方根
ceiling(x)	求不小于 x 的最小整数
floor(x)	求不大于 x 的最大整数
trunc(x)	向 0 的方向截取 x 中的整数部分
round(x,digits=n)	将 x 舍入为指定位数的小数
signif(x,digits=n)	将 x 舍入为指定的有效数字位数

续表

函 数	描 述
$\sin(x)$, $\cos(x)$, $\tan(x)$	求正弦、余弦和正切
$\sin(x)$, $\cos(x)$, $\tan(x)$	求反正弦、反余弦和反正切
$\sinh(x)$, $\cosh(x)$, $\tanh(x)$	求双曲正弦、双曲余弦和双曲正切
$\log(x, \text{base}=n)$	对 x 取以 n 为底的对数
$\log(x)$	对 x 取自然对数
$\log_{10}(x)$	对 x 取常用对数
$\exp(x)$	指数函数

数学函数的示例如下所示。

```

> abs(-5)                                #求绝对值
[1] 5
> sqrt(16)                               #求平方根
[1] 4
> 16^(0.5)                               #和 sqrt(16) 等价
[1] 4
> ceiling(3.457)                         #求不小于 x 的最小整数
[1] 4
> floor(3.457)                           #求不大于 x 的最大整数
[1] 3
> trunc(5.99)                            #向 0 的方向截取 x 中的整数部分
[1] 5
> trunc(-5.99)                           #向 0 的方向截取 x 中的整数部分
[1] -5
> round(3.457, digits = 2)               #将 x 舍入为指定位的小数
[1] 3.46
> signif(3.457, digits = 2)              #将 x 舍入为指定的有效数字位数
[1] 3.5
> cos(2)                                 #求余弦
[1] -0.4161468
> log(10, base = 10)                     #求以 10 为底的对数
[1] 1
> log(10)                                #对 10 取自然对数
[1] 2.302585
> log10(10)                              #对 10 取常用对数
[1] 1
> exp(2.3026)                            #指数函数
[1] 10.00015

```

2. 统计函数

R 语言的统计函数在应用中非常多,常用的统计函数见表 3-4。

表 3-4 统计函数

函 数	描 述
mean(x)	求平均值
median(x)	求中位数
sd(x)	求标准差
var(x)	求方差
mad(x)	求绝对中位差
quantile(x, probs)	求分位数,其中 x 为待求分位数的数值型向量; probs 为一个由[0,1]之间的概率值组成的向量
range(x)	求值域
sum(x)	求和
min(x)	求最小值
max(x)	求最大值

统计函数的示例如下所示。

```
> x <- c(1, 2, 3, 4)                # 向量 x
> mean(x)                           # 求平均数
[1] 2.5
> median(x)                         # 求中位数
[1] 2.5
> sd(x)                             # 求标准差
1.290994
> var(x)                            # 求方差
[1] 1.666667
> mad(x)                            # 求绝对中位差
1.4826
> quantile(x, c(.3, .84))           # 求 x 的 30%和 84%分位点
30% 84%
1.90 3.52
> range(x)                          # 求值域
1 4
> sum(x)                            # 求和
10
> min(x)                            # 求最小值
1
```

```
> max(x)                                #求最大值
4
```

3.2.2 字符处理函数

1. 正则表达式

正则表达式是对字符串操作的一种逻辑公式,它不是 R 语言的专属内容,而是程序设计语言处理字符串的通用方式。正则表达式中包括普通字符,例如 a 到 z 之间的字母以及特殊字符,称为“元字符”。常见的元字符见表 3-5。

表 3-5 常见的元字符

符号	含 义	符号	含 义
\\	转义字符	()	提取匹配的字符串
	可选项	[]	选择中括号中的任意一个字符
\$	放在句尾,表示一行字符串的结束	{ }	前面的字符或表达式的重复次数
.	除了换行以外的任意字符	*	前面的字符或表达式重复 0 次或多次
^	匹配字符串的位置	+	前面的字符或表达式重复 1 次或多次
?	前面的字符或表达式重复 0 次或 1 次		

正则表达式符号的运算顺序可以总结为:圆括号内的表达式最优先;然后是表示重复次数的操作,例如 *、+、{ };接下来是连接运算符;最后是表示可选的运算符|。R 语言中,将反斜杠\作为一个转义符。输入“? Quotes”可查看转义符的用法。常用的转义符如表 3-6 所示。

表 3-6 常用的转义符

符号	含 义	符号	含 义
\t	tab	\v	vertical tab
\b	backspace	\	backslash \
\a	alert (bell)	'	ASCII apostrophe'
\f	form feed	"	ASCII quotation mark "

2. 字符处理函数

R 语言提供了很多字符处理函数,常用的函数如表 3-7 所示。

表 3-7 常用的字符处理函数

函 数	描 述
strsplit()	字符串分割
paste()	字符串连接
nchar()	计算字符数量
substr(x,start,stop)	提取或替换一个字符向量中的字符串
strsplit(x,split,fixed=FALSE)	在 split 处分隔字符向量 x 中的元素
grep()	正则表达式函数,用于查找
sub()	正则表达式函数,用于替换
chartr()	字符串替换
toupper()	切换大写字母
tolower()	切换小写字母

字符处理函数使用方法示例如下。

```

> myresult <- strsplit("123abcdefgabcdef", split = "ab") #用分割参数分割
> myresult
[1] "123" "cdefg" "cdef"
> temp_1 <- c("a", "b", "c")
> temp_2 <- c("1", "2", "3")
> paste(temp_1, temp_2, sep = "_") #两个字符串连接:sep
[1] "a_1" "b_2" "c_3"
> paste(temp_1, collapse = "_") #一个字符串向量内部连接
"a_b_c"
> paste(temp_1, temp_2, sep = "_", collapse = ":" ) #字符串内外都用的模式
[1] "a_1 : b_2 : c_3"
> nchar("abc") #求字符数量
3
> substr("abcdef", start = 2, stop = 4) #提取子串
"bcd"
> substring("abcdef", first = 2) #提取子串
"bcdef"
> temp_3 <- rep("abcdef", 3) #重复
> temp_3
"abcdef" "abcdef" "abcdef"
> nchar(temp_3)
6 6 6
> substr(temp_3, 1:3, 4:6) #头部从 1 到 3,尾部从 4 到 6 的子串截取
"abcd" "bcde" "cdef"
> x2 <- c("asfef", "qwerty", "yuiop", "b")

```

```
> chartr(old = "sey", new = "123", x2)           #字符替换
[1] "alf2f" "qw2rt3" "3uiop" "b"
```

除了 R 语言基础包中的字符处理函数外,用户也会经常使用第三方包 stringr,其函数命名为“str_XXX”形式,具有规范和标准的参数定义,能够有效地提高代码的编写效率。stringr 包常用的字符处理函数有:字符串拼接,如 str_c(),将多个字符串或向量拼接为一个字符串;str_trim(),去掉字符串的空格;str_pad(),可以补充字符串的长度;复制字符串,如 str_dup(),用于对字符串或向量的重复复制;str_sub(),用于在指定位置截取子字符串;此外,还有字符串值排序函数 str_sort()、字符串分割函数 str_split()、字符串匹配函数 str_subset()、字符串替换函数 str_replace()、字符编码转换函数 str_conv()以及字符串计算函数,如 str_count()和 str_length()等。

3.2.3 日期处理函数

日期是重要且特殊的一类数据,通常以字符串的形式输入,但字符型的日期值无法进行计算,因此 R 语言提供了相关处理函数,将字符型的日期值转换成日期变量,实现计算需求,并转换为数值形式存储。as.Date()函数用于将字符型变量转换为日期型,但是转换时需按照一定的规范格式:当输入默认格式“yyyy-mm-dd”时,字符可以自动转换为对应的日期;当输入其他格式时,需要用到格式转换。同样,可使用 as.character()函数将日期转换为字符型数据,转换后可以使用的字符处理函数,如取子集、替换、连接等函数。表 3-8 列出了常用的日期格式,表 3-9 列出了常用的日期函数。

表 3-8 常用的日期格式

符号	含 义	示例	符号	含 义	示例
%d	数字表示的日期	01~31	%Y	四位数的年份	2022
%a	缩写的星期名	Mon	%H	24 小时制小时	00~23
%A	非缩写的星期名	Monday	%I	12 小时制小时	01~12
%w	数字表示	0~6	%p	AM/PM 指示	AM/PM
%m	数字表示的月份	01~12	%M	十进制的分钟	00~60
%b	缩写的月份	Jan	%S	十进制的秒	00~60
%B	非缩写的月份	January	%y	两位数的年份	21

表 3-9 常用的日期函数

函 数	功 能
Sys.Date()	返回系统当前的日期
Sys.time()	返回系统当前的日期和时间
date()	返回系统当前的日期和时间(返回的值为字符串)

续表

函 数	功 能
as.Date()	将字符串形式的日期值转换为日期变量
as.POSIXlt()	将字符串转换为包含时间和时区的日期变量
strptime()	将字符型变量转换为包含时间的日期变量
strftime()	将日期变量转换为指定格式的字符型变量
format()	将日期变量转换为指定格式的字符串

Sys.Date()、Sys.time()和 date()三个函数都是返回当前日期和时间。format()函数主要用于将日期变量转换为指定格式的字符串。

```
> Sys.Date()
[1] "2022-01-12"
> Sys.time()
[1] "2022-01-12 09:44:58 CST"
> date()
[1] "Wed Jan 12 09:46:16 2022"
> format(Sys.time(), format = "%B-%d-%Y")
[1] "一月-12-2022"
```

as.Date()函数和 as.POSIXlt()函数也是常用的日期处理函数。as.Date()函数将字符串形式的日期值转换为日期变量,对于标准格式,即形如“yyyy-mm-dd”或“yyyy/mm/dd”格式的时间数据,可以直接转换为 Date 类型;对于非标准格式,在 as.Date()函数中可以增加一个 format 选项,通过 format 表达式读入指定的格式。as.POSIXlt()函数将字符串转换为包含时间及时区的日期变量,它以列表的形式把字符串型日期时间值分成年、月、日、时、分、秒,进行存储。在返回结果中,UTC 为世界标准时间,和 GMT (Greenwich Mean Time, 格林威治标准时间)大致等同,是世界时间参考点。CST 可同时代表不同时区的标准时间,如美国、澳大利亚、古巴或中国的标准时间,与 R 语言所使用的操作系统时间配置相一致。

```
> as.Date("2022/1/10")
[1] "2022-01-10"
> as.Date(c("2022-1-10", "2022-1-11"))
[1] "2022-01-10" "2022-01-11"
> as.Date("1/10/2022", format = "%m/%d/%Y") # 按照月/日/年的格式输入
[1] "2022-01-10"
> as.POSIXlt("1/10/2022", tz = "", format = "%m/%d/%Y")
[1] "2022-01-10 CST"
```

strptime()函数和 strftime()函数用于数据类型的转换。strptime()函数将字符型变量转换为包含时间的日期变量,strftime()函数将日期变量转换为指定格式的字符型

变量。

```
> strptime("1/10/2022", format="%m/%d/%Y", tz="")
[1] "2022-01-10 CST"
> strftime("2022-01-10 19:33:02 CST", format="%Y/%m/%d")
[1] "2022/01/10"
```

以下为日期处理函数的综合使用案例。

```
#新建一个字符型日期数据变量
> x <- c("2022-02-08 10:07:52", "2022-08-07 19:33:02")
> is.character(x)                                #字符型
[1] TRUE
> as.POSIXlt(x, tz = "", "%Y-%m-%d %H:%M:%S")
[1] "2022-02-08 10:07:52 CST" "2022-08-07 19:33:02 CST "
> as.Date(x, "%Y-%m-%d")
[1] "2022-02-08" "2022-08-07"
> (x <- strptime(x, "%Y-%m-%d %H:%M:%S"))
[1] "2022-02-08 10:07:52 CST" "2022-08-07 19:33:02 CST "
> strftime(x, format = "%Y/%m/%d")
[1] "2022/02/08" "2022/08/07"
> class(x)
[1] "POSIXlt" "POSIXt"
#输出的格式转换为 format 定义的格式
> format(x, "%d/%m/%Y")
[1] "08/02/2022" "07/08/2022"
```

3.2.4 自定义函数

R 语言可以灵活使用自定义函数来完成较大规模的程序。自定义函数的结构如下。

```
myfunction <- function ( arglist ) {
  statements
  return ( object )
}
```

其中,myfunction 为函数名称,arglist 为函数中的参数列表,大括号“{}”内的语句为函数体,函数参数是函数体内部将要处理的值。函数中的对象只能在函数内部使用。函数体通常包括三部分:

- (1) 异常处理,若输入的数据不能满足函数计算的要求,或者类型不符,则应设计相应的机制提示哪个地方出现错误;
- (2) 运算过程,包括具体的运算步骤;
- (3) 返回值,表示函数输出的结果,一般用 return() 函数给出。函数在内部处理过程

中,一旦遇到 `return()` 函数,就会终止运行,将 `return` 内的数据作为函数处理的结果返回。当没有写 `return()` 函数时,R 语言默认将最后一行作为返回值。如果函数的结果需要多个返回值,可以创建一个 `list()` 函数并返回该对象。自定义函数的示例如下。

```
> #例 1:加法运算
> S <- function(x,y) {
+   a <- x+y
+   return(a)
+ }
> S(2, 3)
5
> #例 2:求向量中的偶数个数
> Ans <- function(x) {
+   k=0
+   stopifnot(is.numeric(x))           #异常处理
+   for(i in x) {
+     if (i %% 2==0) {
+       k = k+1
+     }
+   }
+   return(k)
+ }
> Ans(1:10)
5                                     #1 到 10 里面有 5 个偶数
> #例 3:z=x 的平方+y 的平方,求 x+y+z 的值,不使用 return() 函数
> a<- function(x, y) { z <- x^2 + y^2; x+y+z }
> a(0:7, 1)                          #x=0~7, y=1,代入函数 a
[1] 2 4 8 14 22 32 44 58
> (function(x, y) { z <- x^2 + y^2; x+y+z }) (0:7, 1) #另一种写法
2 4 8 14 22 32 44 58
> #例 4:用于将矩阵与其转置相乘,符号为%*%
> norm <- function(x)  sqrt(x%*%x)
> norm(1:4)
      [,1]
[1,] 5.477226
> #可以自行分解
> (1:4)%*%(1:4)                      #向量(1,2,3,4) 与其转置相乘
      [,1]                            #结果是 1 行 1 列的矩阵
[1,] 30
> sqrt((1:4) %*% (1:4))
      [,1]
[1,] 5.477226
> class(sqrt((1:4) %*% (1:4)))
[1] "matrix" "array"                  #结果是矩阵类型
```

3.2.5 函数的嵌套

在 R 语言的数据处理中,有时为了完成某项操作通常需要使用不止一个函数,而且需要将上一个函数的结果作为下一个函数的输入,嵌套多次之后才得到最终结果,这称为函数的嵌套。在嵌套过程中,需要注意函数中变量的作用范围。变量分为全局变量和局部变量两种,在函数内部对变量赋值,则这个变量属于局部变量,仅在函数内部有效;当在函数定义之前对变量赋值,并且在函数内部使用这个变量,则这个变量属于全局变量。具体用法如下所示。

```
> # 例 1:函数中嵌套函数,x 和 y 是函数的参数
> S <- function(x,y) {
+   a <- x + y
+   b <- function() {
+     return(a * 2)
+   }
+   return(b())
+ }
> S(2,3)
[1] 10
```

```
> # 例 2:函数中调用其他自定义函数,注意变量的作用范围
> y <- 10
> f <- function(x) {
+   y <- 2
+   y^2 + g(x)
+ }
> g <- function(x) {
+   x * y
+ }
> f(5)
[1] 54
```

如例 2 所示,在 $f()$ 函数中, y 是局部变量,被赋值为 2;在 $g()$ 函数中, y 是全局变量, y 的取值是在定义 $g()$ 函数时决定的,因此 y 的取值是 10 而不是 2。求值 $f(5)$ 时,结果为 $y^2 + g(x) = 2^2 + g(5) = 2^2 + 5 * 10 = 54$,返回值为 54。



3.3 apply 函数族

3.3.1 函数族

在进行数据批量处理时,虽然可以使用 for 循环在数据对象上重复执行表达式,但实践中,for 循环往往是最后的选择,因为每次迭代重复都是相互独立的,效率比较低,所以

通常尽量使用向量化操作来代替循环操作,用更简洁、更快速的方式来实现相同的效果。向量化操作,是同时对一批值或者一批变量做相同的计算操作,这种操作效率高、快速简洁。

apply 函数族是 R 语言中数据处理的一组常用核心函数的集合,可以实现对数据的循环、分组、计算、过滤、控制,并返回结果,能够对数据进行向量化操作,解决数据 for 循环处理速度慢的问题。为了面向不同的数据类型和不同的返回值要求,这些功能类似的函数成了一个函数族,主要包括函数 apply()、lapply()、sapply()、tapply()、mapply()、rapply()、vapply()、eapply()等。apply 函数族中常用的函数如表 3-10 所示。

表 3-10 apply 函数族中常用的函数

函 数 名	使 用 对 象	返 回 结 果
apply()	矩阵、数组、数据框	向量、数组、列表
lapply()	列表、向量	列表
sapply()	列表、数据框、向量	向量、数组
tapply()	不规则数组	列表
mapply()	列表、向量	列表

3.3.2 apply ()函数

apply()函数可以对矩阵、数据框和数组按行或列进行计算并返回计算结果,是常用的代替 for 循环的函数。使用“? apply”可查看详细的说明,语法格式为 apply(x, MARGIN, FUN,...),其中,x 表示需要处理的数据;MARGIN 表示对哪个维度使用函数;FUN 则是所使用的函数,既可以是自定义的函数,也可以是 R 自带的函数;“...”表示 FUN 函数的其他参数。

```
>(x <- matrix(1:20,ncol=4))
     [,1] [,2] [,3] [,4]
[1,] 1   6   11  16
[2,] 2   7   12  17
[3,] 3   8   13  18
[4,] 4   9   14  19
[5,] 5  10  15  20
> apply(x,1,mean)                #维度为 1:行,使用 mean() 函数
[1] 8.5 9.5 10.5 11.5 12.5
> apply(x,2,mean)                #维度为 2:列
[1] 3 8 13 18
> apply(x,1,max)                 #维度为 1:行,使用 max() 函数
[1] 16 17 18 19 20
```

Iris 鸢尾花卉数据集是常用的分类实验数据集,包含 150 个数据样本,分为 3 类,每

类 50 个数据,每个数据包含 4 个属性:花萼长度、花萼宽度、花瓣长度、花瓣宽度,预测鸢尾花卉属于三个种类(Setosa,Versicolour,Virginica)中的哪一类。apply()函数在 Iris 数据集上的实验如下所示。

```
> class(iris)
[1] "data.frame"
> dimnames(iris)[[2]]           #显示列的名字,等同于 colnames(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
> head(iris)                    #显示数据集中前面几行数据
      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.1         3.5         1.4         0.2      setosa
2           4.9         3.0         1.4         0.2      setosa
3           4.7         3.2         1.3         0.2      setosa
4           4.6         3.1         1.5         0.2      setosa
5           5.0         3.6         1.4         0.2      setosa
6           5.4         3.9         1.7         0.4      setosa
> apply(iris[,1:4],1,mean)       #前四列数据,按行,求均值
[1] 2.550 2.375 2.350 2.350 2.550 2.850 2.425 2.525 2.225 2.400 2.700 2.500
[13] 2.325 2.125 2.800 3.000 2.750 2.575 2.875 2.675 2.675 2.675 2.350 2.650
[25] 2.575 2.450 2.600 2.600 2.550 2.425 2.425 2.675 2.725 2.825 2.425 2.400
[37] 2.625 2.500 2.225 2.550 2.525 2.100 2.275 2.675 2.800 2.375 2.675 2.350
[49] 2.675 2.475 4.075 3.900 4.100 3.275 3.850 3.575 3.975 2.900 3.850 3.300
[61] 2.875 3.650 3.300 3.775 3.350 3.900 3.650 3.400 3.600 3.275 3.925 3.550
[73] 3.800 3.700 3.725 3.850 3.950 4.100 3.725 3.200 3.200 3.150 3.400 3.850
[85] 3.600 3.875 4.000 3.575 3.500 3.325 3.425 3.775 3.400 2.900 3.450 3.525
[97] 3.525 3.675 2.925 3.475 4.525 3.875 4.525 4.150 4.375 4.825 3.400 4.575
[109] 4.200 4.850 4.200 4.075 4.350 3.800 4.025 4.300 4.200 5.100 4.875 3.675
[121] 4.525 3.825 4.800 3.925 4.450 4.550 3.900 3.950 4.225 4.400 4.550 5.025
[133] 4.250 3.925 3.925 4.775 4.425 4.200 3.900 4.375 4.450 4.350 3.875 4.550
[145] 4.550 4.300 3.925 4.175 4.325 3.950
> apply(iris[,1:4],2,mean)      #前四列数据,按列,求均值
Sepal.Length Sepal.Width Petal.Length Petal.Width
5.843333     3.057333     3.758000     1.199333
```

3.3.3 tapply()函数

tapply()也是常用的函数,格式为 tapply(x, INDEX, FUN=NULL, ..., simplify=TRUE),作用是把 FUN 函数根据 INDEX 索引应用到 x 数据,可以理解为将数据按照不同方式分组,生成类似列联表形式的数据结果。tapply()函数在 Iris 数据集上的实验如下所示。

```
> tapply(iris$Sepal.Length, iris$Species, mean)
setosa    versicolor  virginica
5.006      5.936      6.588
```

3.3.4 lapply() 函数

lapply()函数主要用于列表等数据结构,格式为 lapply(x, FUN, ...),作用是将函数 FUN 运用到列表的每一个元素,对列表、数据框等数据集进行循环,返回值为列表。lapply 函数在 Iris 数据集上的实验如下所示。

```
> lapply(iris[,1:4],mean)
$`Sepal.Length`
[1] 5.843333

$Sepal.Width
[1] 3.057333

$Petal.Length
[1] 3.758

$Petal.Width
[1] 1.199333
```

3.3.5 sapply() 函数

sapply()函数和 lapply()函数类似,但是返回的数据结构更灵活。sapply()函数的格式为 sapply(x, FUN, simplify=TRUE, USE.NAMES=TRUE, ...),其中,simplify 参数用来调整输出的数据格式,输入为列表,返回值为向量。sapply 函数在 Iris 数据集上的实验如下所示。

```
> sapply(iris[,1:4],mean)
Sepal.Length Sepal.Width Petal.Length Petal.Width Species
5.843333      3.057333      3.758000      1.199333      NA
```

3.4 数据输入与输出

3.4.1 数据输入

R 语言可以从键盘、文本文件、Microsoft Excel 和 Access、流行的统计软件、特殊格式的文件以及多种关系型数据库中导入数据。文本文件可被几乎所有的数据分析软件读取,其数据若为类似电子表格的数据,通常带有分隔符,包括逗号分隔值(CSV)和制表符分隔值(TXT),以后缀区分。read.table()函数读取这些文本文件,并将结果存储在一个数据框中。如果使用 Rstudio,可以使用其提供的数据导入功能。

1. 利用 RStudio 导入数据

Studio 顶部菜单选择 Tools->Import Dataset->From Local File,弹出窗口选择要导入的数据文件,然后按照提示导入。若数据文件中包含了列名,则在 Heading 选择 yes;若文件中列是用逗号分隔的,则 Separator 选择 Comma。单击 Import 即可导入数据并保存对象。

2. 利用函数导入数据

对于数据文件或结构化文本文件,主要使用 read.table()、read.csv()等函数进行操作。read.table()的参数包括 header = TRUE,表明有标题行;sep=",",表明使用逗号作为字段之间的分隔符;nrow 可以指定读取数据的行数;skip 决定跳过文件开始的多少行;fill = TRUE 表示会使用 NA 值代替缺失的值。此外还有更多选项包括覆盖默认的行名、列名和类,指定输入文件的字符编码以及输入的字符串格式的列声明等。当数据文件不在当前工作目录中,则需加上正确的相对或绝对路径。

对于非结构化文本文件,如果文件的结构松散,可先读入文件中的所有文本行,再对其内容进行分析或操作,如使用 readLines()函数读取文件、writeLines()函数执行写操作。在 Windows 系统中,可以使用 RODBC 包、xlsx 包等包来访问 Excel 文件。

```
> read.table("sample.txt", header=T, sep=",") #读入文本文件
> read.csv("sample.csv", header=T, sep=",") #读入 csv 文件
> read.spss("sample.sav") #读入 SPSS 数据
> readtemptxt <- readLines("sample.txt") #读入文件
> writeLines("sample.txt", "add a new line") #写入文件
> library(xlsx)
> read.xlsx("sample.xls") #读入 Excel 文件
```

R 语言也支持网络爬虫,即抓取网络数据,rvest 是较为常用的包。其他如 quantmod 包,用于金融建模;RCurl 包,实现 HTTP 的一些功能,如从服务器下载文件、保持连接、上传文件、采用二进制格式读取、句柄重定向、密码认证等。

3.4.2 数据输出

R 语言提供了多种数据输出方式,根据输出的形式分为以文本文件输出和以图片形式输出。

1. 以文本文件输出

使用 write.table()将内容导出为文本文件,使用 write.csv()将内容导出为 csv 文件。

```
> age <- c(22, 23)
> name <- c("ken", "john")
```

```
>f <- data.frame(age, name)
>write.table(f, file = "f.csv", row.names=FALSE, col.names=FALSE, quote =
FALSE)
```

2. 以图片形式输出

在 R 语言中绘制的图片可以用 png、jpeg、pdf 命令保存为相应格式的图片文件。

```
>png(file="myplot.png", bg="transparent") #保存为 PNG 格式:
>jpeg(file="myplot.jpeg")                 #保存为 JPEG 格式
>pdf(file="myplot.pdf")                   #保存为 PDF 格式
```



3.5 综合实验

3.5.1 实验 1: 编写自定义函数

1. 实验目标

掌握控制结构的使用方法;掌握自定义函数的方法。

2. 实验内容

- (1) 判断 101~200 有多少个素数,并输出所有素数。
- (2) 编写一个自定义函数,求两个矩阵的乘积,并找出乘积矩阵中的最大元素。

3. 实验步骤

- (1) 判断 101~200 有多少个素数,并输出所有素数。

```
> tmp <- 0 #存放素数个数
> i <- 101
> while (i <= 200) { #i 从 101 计算到 200
+     fg <- 0 #用来判断是否能输出
+     j <- 2 #用来除去要判断的数,从 2 开始
+     while (j < sqrt(i-1)) { #循环判断
+         if (i %% j == 0) { #这里求素数,能被整除则不为素数
+             fg = 1 #这里代表可以整除
+             break #这里用来停止
+         }
+         j <- j + 1
+     }
+     if (fg == 0) { #判断是否可以输出
+         print(i) #输出为素数的数
+         tmp <- tmp + 1
+     }
+     i <- i + 1
+ }
```

```

+         }
+         i <- i + 1           #进行下一个数的判断
+     }

#即将输出结果略
> tmp
[1] 23

```

(2) 编写一个自定义函数,求两个矩阵的乘积,并找出乘积矩阵中的最大元素。

```

> myfunction <- function(x, y) {
+     m1 <- ncol(x)
+     n  <- nrow(y)
+     if (m1 != n) {
+         #第一个矩阵的列数等于第二个矩阵的行数时才能相乘
+         print("error dimension is not suitable")
+         return(0)
+     }
+     m <- nrow(x)
+     n1 <- ncol(y)
+     s <- matrix(0, m, n1)
+     for (i in 1:m) {
+         for (j in 1:n1) {
+             s[i, j] <- sum(x[i, ], y[, j])
+             #相乘后第 i 行第 j 列的元素等于第一个矩阵第 i 行乘以第 j 列元素再相加
+         }
+     }
+     return(s)
+ }

> x <- matrix(c(1:10), 5, 2, byrow = TRUE)
> y <- matrix(c(1:10), 2, 5, byrow = FALSE)
> myfunction(x, y)
      [,1] [,2] [,3] [,4] [,5]
[1,]    6  10  14  18  22
[2,]   10  14  18  22  26
[3,]   14  18  22  26  30
[4,]   18  22  26  30  34
[5,]   22  26  30  34  38

> s <- myfunction(x, y)
> max <- s[1,1]           #假设 s 矩阵的第一个元素为最大值
> for(i in 1:nrow(s)) {
+     for (j in 1:ncol(s)) {
+         if (s[i, j] > max) {

```