



第3章 程序设计基础



内容导航

CONTENTS

3.1 ● 控制流

3.2 ● 函数设计

3.3 ● 编程规范与性能优化

- 简单的问题可以通过在命令行中顺序执行赋值指令解决。
- 要解决现实中的复杂问题，还需要在程序中完成对指令执行过程中的路径选择与重复执行，也就是控制流的任务。
- R语言中与控制流相关的语句包括：
 - if/else，根据条件分别执行两个不同的分支；
 - repeat，执行无限循环，除非使用break终止循环；
 - while，条件成立时执行循环；
 - for，执行给定次数的循环；
 - break，强行终止循环；
 - next，跳过下面的语句，回到循环体头部，执行下一次循环；
 - switch，根据条件选择执行一个分支。

```
> { x <- 0
+ x + 5
+ }
[1] 5
```

- 如果要完成一系列相互关联的语句的执行，可以把这些语句用大括号“{”和“}”组织起。
- 逻辑上相关的这样一组语句也被称为**代码块**。
- 系统直到读到闭括号后才执行对代码块的赋值。

```
> x <- 0

> if (x < 0) {
+ print ("Negative")
+ }else if (x > 0){
+ print ("Positive")
+ }else {
+ print ("Zero")
+ }
```

```
[1] "Zero"
```

- 条件语句if/else根据条件是否成立来选择两个分支中的一个去执行。
- else部分为可选项。
- 当条件是逻辑型或数值型向量（只使用第一个元素）时，条件判断才有效。
- 在R中，数值0对应的是逻辑值FALSE，非零数值则被视为TRUE。

```
> if (x < 0) print ("Negative") else{
+ if (x > 0) print ("Positive") else print ("Zero")}
[1] "Zero"

> x <- -2:4                                #把x设为从-2到4的整数向量

> sqrt (ifelse (x >= 0, x, 0))              #如果x < 0, 开方时返回0
[1] 0.000000 0.000000 0.000000 1.000000 1.414214 1.732051
2.000000
```

- 可以使用嵌套形式的条件判断。
- 注意条件元素选择的语句ifelse和if/else的区别。
- 例子中通过ifelse把向量x中的负数元素置为0, 避免sqrt函数出错

- 循环结构用来重复地执行一个代码块中的语句。
- R 提供了三种直接循环的方式：分别使用**repeat**、**while**和**for语句**。循环语句的返回值总会是NULL。
- 此外，还有两个循环内置结构next和break，在循环中提供了额外的控制。在循环中执行break语句会导致从当前所执行的最内层循环中直接退出；执行next语句则使控制立刻返回到循环的开头，循环体内位于next之后的语句不会被执行。
- R还提供了其他函数来间接实现循环，如函数族tapply ()、apply ()和lapply ()。
- 重要的特点是很多操作（如算数操作），可以使用向量化的方式提高效率，能够避免直接使用循环。

```
> i <- 1
```

#为i赋初值

```
> repeat {
```

```
+ print (i); i <- i + 1
```

#循环打印i的数值

```
+ if (i > 5) break}
```

#条件判断：当i > 5时，退出循环

```
[1] 1
```

```
[1] 2
```

```
[1] 3
```

```
[1] 4
```

```
[1] 5
```

- repeat让代码块中的语句被重复执行，直至明确地使用break终止循环。
- 在运用repeat时一定要避免出现无限循环。


```
> i <- 1                                #为i赋初值
> while (i <= 5) {print (i); i <- i + 1} #循环打印从1到5的整数
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

- while的句法形式为：
while (expression) statement
- 除非是有意使用无限循环，否则需要确保 expression 最终一定会赋值为FALSE。
- 如果expression是一个逻辑型或数值型向量，只使用向量的第一个元素进行循环条件判断。

```
> x <- c (2,3,5,8,13,21)      #将x设为Fibonacci数列中的几个值
> count <- 0                  #计数器初始化为0
> for (i in x) {               #在向量x中依次为i取值
+ if (i %% 2 != 0) count <- count + 1} #统计奇数的个数
> print (count)

[1] 4
```

- 使用for的形式为：
for (name in vector)
statement
- 其中，name是循环变量，依次取vector中的值。
- 副作用是当循环结束后，name中仍然保留了vector最后一个成员的值。

```
> x <- 3

> switch (x,1,2+3,rnorm(5),mean(1:20))    #x=3对应于rnorm(5)
[1] -0.27563903  0.80389074 -0.02545708  1.35181943  0.07127780

> switch (x+1,1,2+3,rnorm(5),mean(1:20))  #x+1=4对应于mean(1:20)
[1] 10.5

> y <- switch (6,1,2+3,rnorm(5),mean(1:20))  #不在list的合理区间
> y
NULL
```

- switch句法形式为：
switch (statement, list)
- 其中list是一个列表，可以用名称标签一一列举。
- 系统计算statement的值，如果在1与list的长度之间，则返回list中相应索引位置的值。
- 否则返回NULL。



内容导航

CONTENTS

3.1

控制流

3.2

函数设计

3.3

编程规范与性能优化

- R语言的**内置函数**足够解决许多基本的数据分析任务。但是在很多其他应用场景中，R用户需要编写自己的函数才能解决独特的问题。
- **函数**具有其他脚本形式无法取代的优点。首先，函数可以接受输入参数，允许用户把相同的处理逻辑封装到同一个函数，而在调用时根据不同参数去独立计算。
- 其次，R语言函数可以返回一个对象，而对象的属性在函数之外的地方能被方便地访问，这给使用函数增加了不可取代的灵活性。
- 使用函数可以把复杂的代码划分成逻辑上统一的组成部分，使得代码具有更好的可读性，还可以通过代码复用来提高效率并保障程序的质量。

- 函数定义的形式为

```
func_name <- function (argument) {  
    statement  
}
```

- func_name是由用户指定的函数名，函数名需要符合R对象命名规则。
- 关键词function表示接下来的代码用于创建一个新的函数。
- 在参数列表argument里，用逗号来把一组形式参数分开。形式参数有三种表示方式，分别是：符号、形如 “symbol = expression” 的语句，以及特殊的形式参数 “...” 。
- 函数体statement可以是任何合法的R表达式。一般情况下，函数体由一个包含在大括号（ “{” 和 “}” ）内的代码块构成。。

```
> echo <- function(x) print(x)           #定义函数echo ()
> echo ("Hello, world!")                 #调用自定义函数echo
[1] "Hello, world!"
> pow <- function(x, y) {                 #打印x^y的值
+ result <- x^y
+ print (paste(x,"^",y,"=",result)) #paste函数将参数粘贴成一个字符串
+ }
> pow (3,2)
[1] "3 ^ 2 = 9"
```

- echo ()接受一个参数值x，在函数体内仅有一条语句“print (x)”，当这个函数被调用时，就会将传入的参数值x打印到控制台。
- 在函数pow ()的声明中，变量x和y叫做形式参数，而调用函数时传递给函数的参数值则称为实际参数。

```
> pow (x=3,2)
```

```
[1] "3 ^ 2 = 9"
```

```
> pow (2,x=3)
```

```
[1] "3 ^ 2 = 9"
```

```
> pow <- function(x, y = 2) {result <- x^y
```

```
+ print (paste(x,"^",y,"=",result))}
```

```
> pow (3)
```

```
[1] "3 ^ 2 = 9"
```

#y的默认值是2

- 在调用函数时候完成了对形式参数的赋值。
- 一般地，赋值是按照位置顺序的匹配来实现的。
- 另外，还可以使用带名称标签的参数形式来精确匹配参数。(如x=3)
- 在声明函数时还可以指定参数的默认值。


```
> test <- function (x) {x<-x+1; print (x)} #函数中让x值加一
> y <- 1
> test(y)
[1] 2
> y
[1] 1
```

#形式参数的变化不会影响y

- 在R中调用函数时传递参数的方法是“按值调用”。
- 在函数体内用实际参数来给局部变量赋初值，局部变量的变量名由形式参数给定。
- 在函数体内改变参数值不会改变函数外的对象的值。

```
> test <- function(x) {
+ if (x %% 2 == 0) return("Even")      #如果x为偶数, 返回"Even"
+ else return("Odd") }                #否则返回"Odd"

> test (123)

[1] "Odd"

> y <- test (5678)                    #把返回值赋给y

> y

[1] "Even"
```

- 需要在函数完成处理之后把结果反映到函数之外, 则可以使用函数的返回值。
- **return**的句法形如下:
return (expression)
- 从函数中返回的值可以是任意的合法对象。
- 即使不直接使用return, 也可以返回结果。

```
test <- function () {
  my.name <- readline (prompt = "Enter your name here: ")
  my.age <- readline (prompt = "Enter your age here: ")
  #把字符串转换成整型数
  my.age <- as.integer (my.age)
  #paste ()把参数转换为字符向量, 然后把它们连接成一个字符串
  print (paste ("Hi", my.name, "you will be", my.age + 1, "next year"))
}
```

- 除了调用函数时传递参数之外, 用户还可以从键盘输入一些值交给函数去处理。
- 也可以把中间结果或执行过程输出到控制台上。

```
> test ()
```

```
Enter yur name here: Jane Doe
```

```
Enter your age here: 18
```

```
[1] "Hi Jane Doe you will be 19 next year"
```

```
> x <- "This is a string"           #设置字符向量
```

```
> print (toupper(x))               #toupper ()转换成大写, tolower()转换为小写
```

```
[1] "THIS IS A STRING"
```

```
> print (x, quote=FALSE)           #不打印引号
```

```
[1] This is a string
```

- 执行函数test ()时, prompt ()函数会在控制台中提示用户需要输入的内容。
- 函数print ()提供了针对不同对象的不同输出形式。

```
sink ("sink-examp.txt") #指定输出到文件sink-examp.txt
```

```
i <- 1:10
```

```
outer (i, i, "*") #计算向量的外积, 结果不会在控制台显示, 而是输出到文件
```

```
sink () #关闭对文件的输出
```

- 把R的输出打印到文件, 便于以后分析中间结果和过程。
- 使用**sink ()**函数把输出结果转移到函数参数指定的文件中, 下一次不带参数调用**sink ()**则会停止输出到文件, 而恢复在控制台上的输出。

```
> a <- 2                                #创建若干对象
> b <- 5
> f <- function(x) x<-0
> f(a)
> ls()                                  #显示当前环境中的对象，注意x不在其中
[1] "a" "b" "f"
> environment ()                        #查看当前环境
<environment: R_GlobalEnv>
```

- 环境定义了对象和函数作用范围，R语言中的每一个对象或函数都处于某一个特定的环境中。
- 函数调用时会创建一个专属环境，包含了函数所使用的局部变量。
- 可以调用environment()函数来获知当前环境是什么。

```
rec.gcd <- function (m, n) {
  if (n == 0)
    return (m)
  else
    return (rec.gcd (n, m %% n))
}
```

```
> rec.gcd (12345, 67890)      #调用递归
```

```
[1] 15
```

- 在一个函数中以直接或间接的方式调用该函数自身的做法，称为递归。
- R中支持定义递归函数。

```
rec.Fibonacci <- function (n) {
  if (n == 1)
    return (1)
  else if (n == 2)
    return (1)

  return (rec.Fibonacci(n-1) + rec.Fibonacci(n-2))
}
```

- Fibonacci序列的递推公式 $f(n) = f(n-1) + f(n-2)$ 可以直接转化为递归形式来实现。
- 在该函数的递归实现中存在大量的重复计算，因此执行效率低下。



内容导航

CONTENTS

3.1

控制流

3.2

函数设计

3.3

● 编程规范与性能优化

- 在编程中不仅仅局限于考虑得到正确的计算结果。
- 在软件开发过程中会涉及很多与复杂性相关的问题。
- 当代码规模很大和需要解决的问题本身逻辑极其复杂时，通过实施一些在软件行业得到了验证的最佳编程实践，可以更好地控制软件编程的复杂性。

```
SumSquareFunc <- function (x, y)
{
  return (x^2 + y^2)
}
```

```
> source ("SumSquareFunc.R")
> myFunc (2,2)
[1] 8
```

- 可以在RGui或者Rstudio的脚本编辑器中编写一个或多个函数，并且把它们保存在一个用户自己命名的文件中，如 SumSquareFunc.R。。
- 在控制台中用命令行函数完成加载，就可以调用自定义函数。

- 用户在编写代码时要保持良好的编程风格。
- 根据软件行业最佳实践来制订统一的编程规范，是保证软件质量和提高开发效率的重要手段之一。
- **保持对齐与缩进**：代码中不同逻辑部分的层次感以及独立性会增加代码的可读性。
- **遵守命名规则**：在命名时，同一类别或作用相似的变量和函数最好具有相似的名字。
- **添加注释**：建议在代码块、分支、循环或逻辑处理较为复杂的代码前增加注释。
- **考虑内聚与耦合**：把相同或相似的工作交给同一个函数完成，使函数内部的处理在逻辑上高度一致。
- **进行参数检验**：对参数做一些合法性的检查，当参数不合法或者不在合理范围内，则报告错误，并从函数中退出。
- **记录日志**：在代码中增加日志，记录执行过程的轨迹。

- 时间复杂度表示程序执行时间随着问题规模增加而发生的变化趋势。
- 也可以使用R语言提供的与时间有关的函数，如**计时函数**`proc.time ()`来直接测量一段代码的执行时间。
- 可以采用下列方式改进代码的性能：
 - 减少不必要的计算。
 - 减少输入/输出。
 - 保存中间结果。
 - 使用向量操作。
 - 设计并使用好的算法。