

R 语言数据分析

Lecture 3 R 编程基础

刘寅

统计与数学学院

- 1 简介
- 2 控制流
- 3 编写函数
- 4 编写脚本



1. 简介

1.1 R编程

编程是借助计算机来解决某个问题。**R编程**是在数据环境中借助计算机软件来解决问题。



1.2 R编程的目的

- ◆ 使代码更简洁
- ◆ 使代码更稳健
- ◆ 使代码运行更快



2. 控制流

2.1 分支语句

◆ if语句

用法: **if (con) expr1**

如果条件con成立, 执行expr1, 否则跳过.

```
> x <- seq(1,10,2)
> i <- 2
> if (x[i]==5) print("Today is sunny")
> i <- 3
> if (x[i]==5) print("Today is sunny")
```



◆ if...else...语句

用法: `if (con) expr1 else expr2`

如果条件con成立, 执行expr1, 否则, 执行expr2.

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



◆ if...else if... else...语句

用法: `if (con1) expr1 else if (con2) expr2 else expr3`
如果条件con1成立, 执行expr1, 如果条件con2成立, 执行expr2, 否则, 执行expr3.

```
> x <- c("what","is","truth")
> if("Truth" %in% x) {
+   print("Truth is found the first time")
+ } else if ("truth" %in% x) {
+   print("truth is found the second time")
+ } else {
+   print("No truth found")
+ }
```



◆ ifelse语句

用法: `ifelse (con, expr1, expr2)`

如果条件con成立, 执行expr1, 否则, 执行expr2.

```
> x <- 1:10
```

```
> ifelse (x<5, print("Apple"), print("orange"))
```



◆ switch语句

用法: `switch (expr, list)`

如果expr的返回值在1到length(list), 则返回list中相应位置的值, 否则跳过。当list是有名定义时, expr等于变量名时, 返回变量名对应的值, 否则跳过。

```
> switch(2, 2*3, mean(1:10), rnorm(4))
> switch(6, 2*3, mean(1:10), rnorm(4))
> x <- "fruit"
> switch(x, vegetable="bean", meat="beaf",
+       fruit="banana")
> switch("flower", vegetable="bean",
+       meat="beaf", fruit="banana")
```



练一练

3.1 将一个数值赋值给 x :

- (1) 若 x 为奇数, 输出“ x 是奇数!”
- (2) 若 x 为奇数, 输出“ x 是奇数!”; 否则, 输出“ x 是偶数!”

3.2 给`score`赋0~100中的任意一个数值, 若 $0 \leq \text{score} < 60$, 输出“不及格”; 若 $60 \leq \text{score} < 70$, 输出“及格”; 若 $70 \leq \text{score} < 80$, 输出“中等”; 若 $80 \leq \text{score} < 90$, 输出“良好”; 否则, 输出“优秀”

3.3 创建一个长度为10的数值型向量, 用`ifelse`函数判断: 若分量大于6.5, 输出向量("Apple", "Orange", "Banana")中对应元素; 否则, 输出向量("Potato", "Tomato", "Bean", "Carrot", "Mushroom")中对应元素

3.4 将一个整数值赋值给 x , 利用`switch`函数在“R”、“软”、“件”、“介”、“绍”中进行选择

2.2 循环语句

◆ for 循环语句

用法: **for (name in expr1) expr2**

name为循环变量, expr1是一个向量表达式(通常为一个序列, 例如1:20), expr2为一组表达式。name访问expr1所有可以去到的值时, 都会执行expr2。

```
> # 产生一个4阶的Hilbert矩阵
>
> X <- matrix(0, 4, 4) # 定义一个4阶0矩阵
> for (i in 1:4)
+ {
+   for (j in 1:4)
+   {
+     X[i,j] <- 1/(i+j-1)
+   }
+ }
```



◆ while 循环语句

用法: **while (con) expr**

若条件con成立, 则执行表达式expr。也可强制使用break跳出循环。

```
> x <- y <- 0.5
> tt <- 0 # 定义循环变量的初始值
> while (tt<100)
+ {
+     x <- x^2+0.5*y
+     y <- 0.5*x-0.5
+     tt <- tt+1
+ }
> c(x,y,tt)
[1] -0.250 -0.625 100.000
```



```
> x <- y <- 0.5
> tt <- 0 # 定义循环变量的初始值
> while (tt<1e3)
+ {
+     x <- x^2+0.5*y
+     y <- 0.5*x-0.5
+     tt <- tt+1
+     if (x < 0.2) break
+ }
> c(x,y,tt)
[1] 0.1250 -0.4375 2.0000
```



```
> x <- y <- 0.5
> tt <- 0 # 定义循环变量的初始值
> error <- 1
> while (tt<1e3 & error > 0.5)
+ {
+     x <- x^2+0.5*y
+     y <- 0.5*x-0.5
+     tt <- tt+1
+     error <- x-y
+ }
> c(x,y,error,tt)
[1] -0.2031250 -0.6015625  0.3984375  3.0000000
```



◆ repeat循环语句

用法: **repeat expr**

重复执行表达式expr, 依赖break跳出循环。

```
> x <- 0
> repeat
+ {
+     x <- x^2-0.5*x-1
+ }
> x <- 0
> repeat
+ {
+     x <- x^2-0.5*x-1
+     if (x > 0.2) break
+ }
> x
[1] 0.5
```



练一练

- 3.5 使用for循环计算30个Fibonacci数(黄金分割数: 0,1,1,2,3,5,8,13,...)
- 3.6 使用for循环计算1~100中整数之和
- 3.7 使用while循环计算30个Fibonacci数
- 3.8 使用while循环计算Fibonacci数, 直至新产生的数超过128时停止
- 3.9 使用while循环计算1~100中整数之和
- 3.10 使用repeat循环计算Fibonacci数, 直至新产生的数超过128时停止
- 3.11 使用repeat循环计算1~100中整数之和

2.3 向量化

◆ 使用apply类函数

分组计算

tapply

参数: vector
返回值: vector

apply

参数: list, data.frame, array
返回值: vector, matrix

多参数计算

mapply

参数: vector, 不限个数
返回值: vector, matrix

循环迭代

lapply

参数: list, data.frame
返回值: list

简化版

sapply

参数: list, data.frame
返回值: vector, matrix

可设置返回值

vapply

参数: list, data.frame
返回值: vector, matrix

递归版

rapply

参数: list
返回值: list

环境空间遍历

eapply

参数: environment
返回值: list



3. 编写函数

3.1 函数概述

- ◆ 函数是一种特殊的对象
- ◆ 函数主要用于操作处理对象
- ◆ 有系统自带函数，也可以自己定义函数
- ◆ 系统自带函数都存放在库(library)中
- ◆ 一些最基本的系统函数是直接C语言写的，其他的都是使用R语言写的，使用R语言写的和用户自定义函数没有本质区别



3.2 编写自定义函数

- ◆ R 语言允许用户创建自己的函数(function)对象。

- ◆ 自定义函数是通过下面的语句形式定义的：

```
name <- function(arg1, arg2, ...) expression
```

- ◆ 其中**arg1**, **arg2**是参数，**expression**是成组表达式。

- ◆ 参数可以被设定一些默认值。使用函数时如果默认值适合你要做的事情，则可以省略这些参数。

- ◆ 函数的返回值可以是任何R对象。尽管返回值通常为列表形式，其实返回值甚至可以是另一个函数。

- ◆ 可以通过显式地调用**return()**，把一个值返回给主调函数。

- ◆ 如果不使用这条语句，默认将会把最后执行的语句的值作为返回值。



例3.1

编写一个用二分法求非线性方程根的函数，并求方程

$$x^3 - x - 1 = 0$$

在区间 $[1,2]$ 内的根，要求精度 $\varepsilon = 10^{-6}$ 。



```

> bisect <- function(f, a, b, pre=1e-6)
+ {
+   # *****
+   # Function name: bisect(f, a, b, pre=1e-6)
+   # ----- Input -----
+   # f = function to be resolved
+   # a = lower bound
+   # b = upper bound
+   # pre = 1e-6 precision required
+   # *****
+
+   if (f(a)*f(b) > 0) {
+     print("fail to find the root!")
+   } else {
+     repeat
+     {
+       if (abs(b-a) < pre) break
+       x <- (a+b)/2
+       if (f(a)*f(x) < 0) b <- x
+       else
+         a <- x
+     }
+     result <- (a+b)/2
+   }
+   return(result)
+ }
>
> f <- function(x) x^3-x-1
> bisect(f, 1, 2)
[1] 1.324718

```



3.3 函数结果的输出：列表与数据框

3.3.1 列表

- ◆ 列表(list)是一种特别的集合对象，其元素由下标区分，但各元素的类型可以是任意对象、任意长度，不同原色不必是同一对象、同一长度。
- ◆ 用法： `list(name1=object1, name2=object2, ...)`
- ◆ 其中 **name1**, **name2** 是元素名， **object1**, **object2** 是元素。
- ◆ 列表的引用可以通过“列表名[[下标]]”或“列表名\$元素名”的格式引用。
- ◆ 列表元素的修改、增加或删除元素均可通过元素引用赋值来实现。



例3.2

编写一个求任意矩阵的转置、逆矩阵以及行列式的函数。

```
> matrix.opera <- function(A)
+ {
+   trans <- t(A)
+   p <- dim(A)[1]
+   q <- dim(A)[2]
+   if (p != q || det(A) < 1e-15) {
+     inv <- "inverse matrix does not exist!"
+     determin <- "determination does not exist!"
+   } else {
+     inv <- solve(A)
+     determin <- det(A)
+   }
+   result <- list(transpose=trans, inverse_matrix=inv,
+                 determination=determin)
+   return(result)
+ }
```



```
> X <- matrix(1:16,4,4)
> re <- matrix.opera(X)
> re
$'transpose'
```

| | [,1] | [,2] | [,3] | [,4] |
|------|------|------|------|------|
| [1,] | 1 | 2 | 3 | 4 |
| [2,] | 5 | 6 | 7 | 8 |
| [3,] | 9 | 10 | 11 | 12 |
| [4,] | 13 | 14 | 15 | 16 |

```
$inverse_matrix
```

```
[1] "inverse matrix does not exist!"
```

```
$determination
```

```
[1] "determination does not exist!"
```




```
> Y <- diag(1:4)
> re <- matrix.opera(Y)
> re
```

```
$‘transpose‘
```

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

```
$inverse_matrix
```

| | [,1] | [,2] | [,3] | [,4] |
|------|------|------|------------|------|
| [1,] | 1 | 0.0 | 0.00000000 | 0.00 |
| [2,] | 0 | 0.5 | 0.00000000 | 0.00 |
| [3,] | 0 | 0.0 | 0.33333333 | 0.00 |
| [4,] | 0 | 0.0 | 0.00000000 | 0.25 |

```
$determination
```

```
[1] 24
```



```

> re[[2]]
      [,1] [,2]      [,3] [,4]
[1,]      1 0.0 0.0000000 0.00
[2,]      0 0.5 0.0000000 0.00
[3,]      0 0.0 0.3333333 0.00
[4,]      0 0.0 0.0000000 0.25
> re$determination
[1] 24
> re$determination <- 10
> re$determination
[1] 10
> re$sum <- sum(Y)
> re$transpose <- NULL
> re
$'inverse_matrix'
      [,1] [,2]      [,3] [,4]
[1,]      1 0.0 0.0000000 0.00
[2,]      0 0.5 0.0000000 0.00
[3,]      0 0.0 0.3333333 0.00
[4,]      0 0.0 0.0000000 0.25

$determination
[1] 10

$sum
[1] 10

```



3.3.2 数据框

- ◆ 数据框(data.frame)是一种数据结构，通常以矩阵的形式展现，矩阵的各列可以是不同类型的数据。
- ◆ 数据框每一列是一个变量，每行是一个观测。作为数据框变量的元素必须具有相同的长度(行数)。
- ◆ 用法: `data.frame(name1=object1, name2=object2, ...)`
- ◆ 其中name1, name2是元素名，object1, object2是元素。
- ◆ 如果一个列表的各成分满足数据框元素的要求，可以用`as.data.frame()`函数强制转换成数据框。
- ◆ 数据框的引用与矩阵元素的引用方法相同，可使用下标或下标向量，也可使用名字或名字向量。数据框的各变量可以通过“数据框名[[下标]]”或“数据框名\$变量名”的格式引用。



例3.3

试将以下学生信息以数据框形式进行展现：

| | | | | | |
|-----|---------|---------|---------|-----------|--------|
| 姓名： | "Alice" | "Becka" | "James" | "Jeffrey" | "John" |
| 性别： | "F" | "F" | "M" | "M" | "M" |
| 年龄： | 13 | 13 | 12 | 13 | 12 |
| 身高： | 156.5 | 165.3 | 157.3 | 162.5 | 159.0 |
| 体重： | 84.0 | 98.0 | 83.0 | 84.0 | 99.5 |



```
> df <- data.frame(  
+   name = c("Alice","Becka","James","Jeffery","John"),  
+   sex = c("F","F","M","M","M"),  
+   age = c(13,13,12,13,12),  
+   height = c(156.5, 165.3, 157.3, 162.5, 159.0),  
+   wight = c(84.0, 98.0, 83.0, 84.0, 99.5)  
+ )  
> df
```

| | name | sex | age | height | wight |
|---|---------|-----|-----|--------|-------|
| 1 | Alice | F | 13 | 156.5 | 84.0 |
| 2 | Becka | F | 13 | 165.3 | 98.0 |
| 3 | James | M | 12 | 157.3 | 83.0 |
| 4 | Jeffery | M | 13 | 162.5 | 84.0 |
| 5 | John | M | 12 | 159.0 | 99.5 |



```

> df[1:2,c(1,3,5)]
  name age wight
1 Alice  13    84
2 Becka  13    98
> df$name
[1] Alice   Becka   James   Jeffery  John
> rownames(df) <- c("1","2","3","4","5")
> df
  name sex age height wight
1  Alice  F  13  156.5  84.0
2  Becka  F  13  165.3  98.0
3  James  M  12  157.3  83.0
4 Jeffery M  13  162.5  84.0
5   John  M  12  159.0  99.5
> colnames(df) <- NULL
> df
1  Alice F 13 156.5 84.0
2  Becka F 13 165.3 98.0
3  James M 12 157.3 83.0
4 Jeffery M 13 162.5 84.0
5   John M 12 159.0 99.5

```



4. 编写脚本

脚本是什么？

- 脚本是一系列命令
- 可以先批量编写程序或对他人已经编号好的程序进行修改，再输入到控制台进行调试，以满足数据分析的需求。
- 利用R自带的脚本编辑器进行编译。



例3.4

对一批涂料进行研究，确定搅拌速度对杂质含量的影响，数据如下：

转速rpm 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42

杂质率% 8.4, 9.5, 11.8, 10.4, 13.3, 14.8, 13.2, 14.7, 16.4, 16.5, 18.9, 18.5

试进行回归分析。

```
rpm <- c(20,22,24,26,28,30,32,34,36,38,40,42)
impurity <-c(8.4,9.5,11.8,10.4,13.3,14.8,13.2,
             14.7,16.4,16.5,18.9,18.5)
reg <- lm(impurity~rpm)
plot(rpm, impurity, type="p")
abline(reg, col="blue")
summary(reg)
```

将上述代码保存为example3.4.R文件。



◆ 脚本的运行通过以下三种方式:

1. 通过source()函数运行

```
source("../lecture 3/example3.4.R")
```

2. 通过R脚本编辑器运行

Ctrl+R运行

3. 直接粘贴到R控制台

Ctrl+c, Ctrl+V



找到脚本中的错误

- ◆ 在脚本编写中可能会出现错误，大多数错误通过调试来进行修改，即一些适用性的检测工作。

```
> Moose.density <- c(0.17, 0.23, 0.23, 0.26, 0.37, 0.42, 0.66,  
+                    0.80, 1.11, 1.30, 1.37, 1.41, 1.73, 2.49)  
> kill.rate <- c(0.37, 0.47, 1.90, 2.04, 1.12, 1.74, 2.78, 1.85,  
+               1.88, 1.96, 1.80, 2.44, 2.81, 3.75)  
> plot(moose.density, kill.rate, type="p")  
Error in plot(moose.density, kill.rate, type = "p") :  
  找不到对象'moose.density'  
> m <- 2.5*(0:100)/200  
> a <- 3.37  
> b <- 0.47  
> k <- a*m/(b+m)  
> points(m, k, type="l")  
Error in plot.xy(xy.coords(x, y), type = type, ...) :  
  plot.new has not been called yet
```



◆ 若可以直接定位错误的位置并进行修正，后面的错误可能可以避免

```
> Moose.density <- c(0.17, 0.23, 0.23, 0.26, 0.37, 0.42, 0.66,  
+                    0.80, 1.11, 1.30, 1.37, 1.41, 1.73, 2.49)  
> kill.rate <- c(0.37, 0.47, 1.90, 2.04, 1.12, 1.74, 2.78, 1.85,  
+               1.88, 1.96, 1.80, 2.44, 2.81, 3.75)  
> plot(Moose.density, kill.rate, type="p")  
> m <- 2.5*(0:100)/200  
> a <- 3.37  
> b <- 0.47  
> k <- a*m/(b+m)  
> points(m, k, type="l")
```



- ◆ 若可以直接定位错误的位置并进行修正，后面的错误可能可以避免

```
> Moose.density <- c(0.17, 0.23, 0.23, 0.26, 0.37, 0.42, 0.66,  
+                    0.80, 1.11, 1.30, 1.37, 1.41, 1.73, 2.49)  
> kill.rate <- c(0.37, 0.47, 1.90, 2.04, 1.12, 1.74, 2.78, 1.85,  
+               1.88, 1.96, 1.80, 2.44, 2.81, 3.75)  
> plot(Moose.density, kill.rate, type="p")  
> m <- 2.5*(0:100)/200  
> a <- 3.37  
> b <- 0.47  
> k <- a*m/(b+m)  
> points(m, k, type="l")
```

- ◆ 建议：编写代码时，逐行或小部分进行代码调试，避免大规模代码运行！



- ◆ 若可以直接定位错误的位置并进行修正，后面的错误可能可以避免

```
> Moose.density <- c(0.17, 0.23, 0.23, 0.26, 0.37, 0.42, 0.66,  
+                    0.80, 1.11, 1.30, 1.37, 1.41, 1.73, 2.49)  
> kill.rate <- c(0.37, 0.47, 1.90, 2.04, 1.12, 1.74, 2.78, 1.85,  
+               1.88, 1.96, 1.80, 2.44, 2.81, 3.75)  
> plot(Moose.density, kill.rate, type="p")  
> m <- 2.5*(0:100)/200  
> a <- 3.37  
> b <- 0.47  
> k <- a*m/(b+m)  
> points(m, k, type="l")
```

- ◆ 建议：编写代码时，逐行或小部分进行代码调试，避免大规模代码运行！
- ◆ 注意：即使脚本可以正常运行，其结果也可能错的离谱！原因在于计算本身存在错误！



利用注释对脚本内容说明

- ◆ 在脚本编写中可通过“#”添加注释内容，任何以“#”开头的命令行都会被R忽略。

```
#####  
## This is the R code for the example 9.1  
#####
```

```
## now source the function definitions:  
source("../functions/mean.std.CI.R")  
source("../functions/example9_1.R")
```

```
## save the results to results/  
# results in Example 9.1.1  
results_911 <- example9_1(1)  
write.csv(results_911, file="../results/results_911.csv")  
# results in Example 9.1.2  
results_912 <- example9_1(2)  
write.csv(results_912, file="../results/results_912.csv")  
# results in Example 9.1.3  
results_913 <- example9_1(3)  
write.csv(results_913, file="../results/results_913.csv")  
# results in Example 9.1.4  
results_914 <- example9_1(4)  
write.csv(results_914, file="../results/results_914.csv")
```





第三章结束了!

THANKS