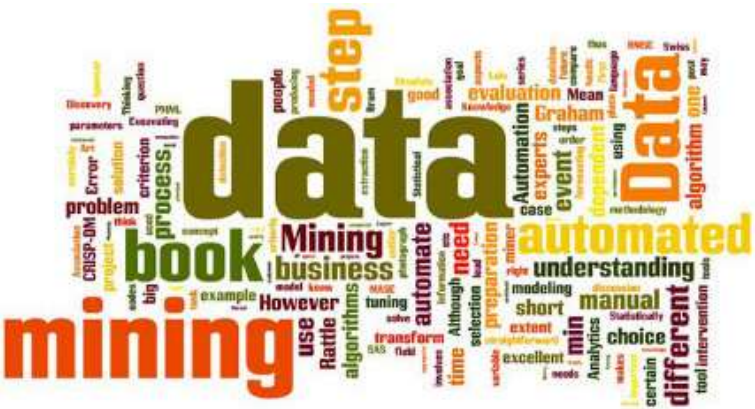
R数据分析：商业数据分析全景之 R语言编程



肖柳青 博士 教授

主要内容

- 基本数据元素
- 基本数据类型
- R程序控制
- R的函数与包
- R的时间与日期类型
- 在R中读取数据

R的基本数据类型

对象(Objects)

- R中5种基本的数据类型:
  - character-字符
  - numeric (real numbers)-数字
  - Integer-整数
  - Complex-复数
  - logical (True/False)-二元逻辑

R 语言数据类型

```
> c<-"aaa"
> c
[1] "aaa"
> mode(c)
[1] "character"
> d<-'123'
> d
[1] "123"
> mode(d)
[1] "character"
> e<-123
> e
[1] 123
> mode(e)
[1] "numeric"
> f<-T
> mode(f)
[1] "logical"
```

R 语言数据类型

还要注意数据的两种特殊的数据类型，即数据的缺失NA 和空值NULL。

```
> i<-NA
> mode(i)
[1] "logical"
> 3+i
[1] NA
> 4-i
[1] NA
> j<-c(10,12,12,11,NULL)
```

```
> a <- c(NA,2,5,NA)
> is.na(a)
[1]  TRUE FALSE FALSE  TRUE
> a[!is.na(a)]
[1] 2 5
```

数据类型的转换

Table: R 常见数据类型的判断与转换函数表

数据类型	中文含义	判断函数	转换函数
character	字符串	is.character( )	as.character( )
numeric	数值	is.numeric( )	as.numeric( )
logical	逻辑	is.logical( )	as.logical( )
complex	复数	is.complex( )	as.complex( )
NA	缺失	is.na( )	as.na( )

R的数据结构

数据结构

- R中6种基本的数据结构:
  - 向量（Vector）
  - 矩阵（Matrix）
  - 数组（Array）
  - 因子（Factor）
  - 列表（List）
  - 数据框（Data Frame）

数据结构-向量

- c() 函数用于创建向量

```
> a<-c(1,2,3)
> a
[1] 1 2 3
> i<-c(1:10)
> i
[1]  1  2  3  4  5  6  7  8  9 10
> i[1]
[1] 1
```

- vector() 函数也可以用于创建向量

```
> x <- vector("numeric", length = 10)
> x
[1] 0 0 0 0 0 0 0 0 0 0
```

数据结构-矩阵

```
> matrix1<-matrix(c(1,2,3,4,5,6),nrow=2,ncol=3)
> matrix1
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> matrix2<-matrix(c(1,2,3,4,5,6),nrow=2,byrow=TRUE)
> matrix2
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> matrix3<-matrix(c("a","b","c","d"),nrow=2)
> matrix3
     [,1] [,2]
[1,] "a"  "c"
[2,] "b"  "d"
```

数据结构-矩阵

```
> matrix4 <- matrix(nrow=2,ncol =2)
> matrix4
     [,1] [,2]
[1,] NA   NA
[2,] NA   NA
> matrix4[1,1] <- 3
> matrix4[2,1] <- 5
> matrix4[1,2] <- 10
> matrix4[2,2] <- 14
> matrix4
     [,1] [,2]
[1,]    3    5
[2,]   10   14
```

数据结构-矩阵

```
> matrix5<-matrix(1:15,nrow=3)
> matrix5
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
> matrix5[2,]
[1]  2  5  8 11 14
> matrix5[,4]
[1] 10 11 12
> matrix5[2,5]
[1] 14
```

数据结构-数组

```
> array(vector , dimensions , dimnames)
> array1<-array(1:18,c(3,3,2))
> array1
, , 1
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
, , 2
     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18
```

数据结构-数组

```
> dim1<-c("A1","A2","A3")
> dim2<-c("B1","B2","B3")
> dim3<-c("C1","C2")
> array3<-array(1:18,c(3,3,2),dimnames=list(dim1,dim2,
    dim3))
> array3
, , C1
   B1 B2 B3
A1  1  4  7
A2  2  5  8
A3  3  6  9
, , C2
   B1 B2 B3
A1 10 13 16
A2 11 14 17
A3 12 15 18
```

数据结构-数组

```
> a2<-array(1:6,6)
> a2
[1] 1 2 3 4 5 6
> v<-c(1:6)
> v
[1] 1 2 3 4 5 6
> is.array(a2)
[1] TRUE
> is.vector(a2)
[1] FALSE
> is.vector(v)
[1] TRUE
> is.array(v)
[1] FALSE
```

数据结构-数组

```
> array1<-array(1:18,c(3,3,2))
> array1
, , 1
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
, , 2
     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18
> array1[2,,]
     [,1] [,2]
[1,]    2   11
[2,]    5   14
[3,]    8   17
```

数据结构-数组

```
> array1[,1:2,]
, , 1
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
, , 2
     [,1] [,2]
[1,]   10   13
[2,]   11   14
[3,]   12   15
> array1[,,2]
     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18
```

数据结构-因子

```
> factor(x = character( ), levels, labels = levels,
+    exclude = NA, ordered = is.ordered(x), nmax = NA)
> performance<-c("bad","good","good","bad",
+    "excellent","bad")
> performance
[1] "bad"    "good"   "good"  "bad"   "excellent" "bad"
> f1<-factor(performance)
> f1
[1] bad    good  good   bad  excellent bad
Levels: bad excellent good
> levels(f1)
[1] "bad" "excellent" "good"
> f1[2]
[1] good
Levels: bad excellent good
> f1[6]
[1] bad
Levels: bad excellent good
```

数据结构-数据框

```
> name<-c("Jane","Bob","Elena","Lily","Max")
> English<-c(84,86,78,90,88)
> Math<-c(80,85,90,87,85)
> Art<-c(78,80,80,85,86)
> Score<-data.frame(name,English,Math,Art)
> Score
   name English Math Art
1  Jane      84   80  78
2   Bob      86   85  80
3 Elena      78   90  80
4  Lily      90   87  85
5   Max      88   85  86
```

数据结构-数据框

```
> v2<-c(3,4)
> v3<-c("a","b")
> df1<-data.frame(v2,v3)
> df1
  v2 v3
1  3  a
2  4  b
> class(df1$v3)
[1] "factor"
```

数据结构-数据框

```
> dm2[1,]
  X1 X2 X3 X1.1 X2.1 X3.1   v3
1  1  3  5    5    7    9 TRUE
> dm2[,4]
[1] 5 6
> Score[[3]]
[1] 80 85 90 87 85
> Score$Math
[1] 80 85 90 87 85
```

数据结构-列表

```
> v5<-c(2:8)
> v5
[1] 2 3 4 5 6 7 8
> v6<-c("aa","bb","cc")
> v6
[1] "aa" "bb" "cc"
> m6<-matrix(c(1:9),nrow=3)
> m6
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

数据结构-列表

```
> f2<-factor(c("high","low","low","high"))
> mylist<-list(v5,v6,m6,f2)
> mylist
[[1]]
[1] 2 3 4 5 6 7 8
[[2]]
[1] "aa" "bb" "cc"
[[3]]
[,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
[[4]]
[1] high low  low  high
Levels: high low
```

数据结构-列表

```
> list1[[1]]
[1] 2 3 4 5 6 7 8

> list1[[3]]
[,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> list2[["matrix"]]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> list2$factor
[1] high low  low  high
Levels: high low
```

R程序控制

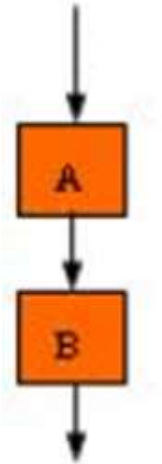
流程图基本元素

判断

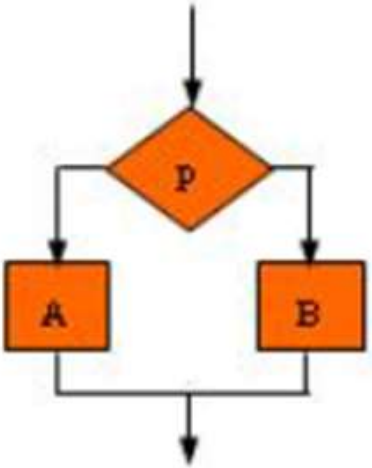
流程

方向

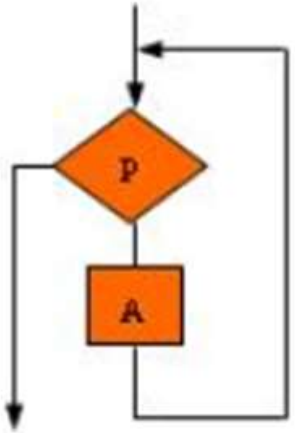
三种基本的编程结构



顺承结构



分支结构



循环结构

控制语句

- R中的控制语句
  - if, else: 条件控制
  - for: 循环控制
  - while: 条件循环控制
  - repeat: 无限重复循环
  - break: 中断本循环
  - next: 跳过指定的循环

If控制语法

```
if(<condition>) {
        ## do something
} else {
        ## do something else
}

if(<condition1>) {
        ## do something
} else if(<condition2>) {
        ## do something different
} else {
        ## do something different
}
```

If

- 有效的if控制.

```
if(x > 3) {
y <-10
} else {
y <-0
}
```

- 有效的if控制.

```
y <-if(x > 3) {
10
} else {
0
}
```

for

- for 循环使用迭代器变量，并指派连续数值型向量或序列
- For 循环常用于遍历某个对象的所有取值(list, vector等)

```
for(i in 1:10) {
print(i)
}
```

- 上例中，程序实现按顺序打印1到10的所有元素。

for

#1 遍历索引并输出取值

```
x <-c("a", "b", "c", "d")
for(i in 1:4) {
print(x[i])
}
```

#2 枚举

```
for(letter in x) {
print(letter)
}
```

For嵌套循环

- 嵌套循环示例

```
x <-matrix(1:6, 2, 3)
for(I in seq_len(nrow(x))) {
    for(j in seq_len(ncol(x))) {
        print(x[i, j])
    }
}
```

- 在使用嵌套循环应注意，三重或更多的嵌套循环一般难以理解。

while

- While 循环首先会检查一定的条件，如果满足条件,其将会被循环执行.

```
count <-0
while(count < 10) {
    print(count)
    count <-count + 1
}
```

- While 循环使用不慎可能会导致死循环，使用时应特别注意。

While

- 有时while循环的条件不止一个，也可以执行.

```
z <-5
while(z >= 3 && z <= 10) {
        print(z)
        coin <-rbinom(1, 1, 0.5)
        if(coin == 1) { ## random walk
                z <-z + 1
        } else {
                z <-z -1
        }
}
```

- 条件语句一般从左到右进行执行

repeat, break

- Repeat 执行一个无限的循环; 在统计分析中一般很少使用，但是这个函数也有用处. 结束repeat函数的唯一语句是break。
-

```
x0 <-1
tol<-1e-8
repeat {
        x1 <-computeEstimate()
        if(abs(x1 -x0) < tol) {
                break
        } else {
                x0 <-x1
        }
}
```

repeat

- Repeat函数通常会造成无限循环，所以需要设置一个循环次数的上限以终止循环.

next

- next 用于跳过指定的循环

```
for(i in 1:100) {
    if(i<= 20) {
        ## Skip the first 20 iterations
        next
    }
        ## Do something here
}
```

R的控制语句

- 总结
- if, while, for 用于控制或条件控制R的程序
- 避免出现死循环, 即使它们理论上是对的
- 与循环语句类似的过程控制语句 apply 函数更加有用

R的函数与包

函数

- R可直接调用所需要的函数，其在R中其实只是一个普通的对象，只不过其类型为函数（”function”）

```
f <-function(<arguments>) {
## Do something interesting
}
```

- 函数特点
  - 某函数可以是其他函数的参数
  - 函数可以嵌套，用户可以自己定义函数
  - 函数最终会返回某个结果

函数的参数

函数的参数都有默认的参数。

- 函数在定义的时候可以创建参数
- 可以在帮助中查看函数的所有参数及其意义
- 在R中调用函数并没有必要使用全部参数
- 函数的可选参数可以缺失

参数匹配

- R函数的参数可以按照匹配，R 函数参数可以按照位置匹配或名称匹配。

```
> mydata<-rnorm(100)
> sd(mydata)
> sd(x = mydata)
> sd(x = mydata, na.rm = FALSE)
> sd(na.rm = FALSE, x = mydata)
> sd(na.rm = FALSE, mydata)
```

- 尽管上述代码都能够得出正确结果，但不建议打乱次序。

参数匹配

- 可以使用args函数查看指定函数的参数情况，再通过名称匹配的方式输入相应的参数.

```
> args(lm)
function (formula, data, subset, weights, na.action,
method = "qr", model = TRUE, x = FALSE,
y = FALSE, qr= TRUE, singular.ok= TRUE,
contrasts = NULL, offset, ...)
```

- 下面两个函数是等价的

```
lm(data = mydata, y ~ x, model = FALSE, 1:100)
lm(y ~ x, mydata, 1:100, model = FALSE)
```

参数匹配

- 在大多情况下,采用名称匹配的方式设置函数的参数，如果函数待设置的参数太多，可以考虑使用次序匹配函数。
- 名称匹配参数可以使用户更好的记住参数的取值，尤其在绘图时。

定义函数

```
f <-function(a, b = 1, c = 2, d = NULL) {
}
```

- 定义函数格式如上，有时不用指定参数的取值,有时可以设置参数值为空(NULL).

定义函数

- 在函数中可以定义多余参数

```
f <-function(a, b) {
a^2
}
f(2)
## [1] 4
```

- 上述函数不会用到参数B, 所以调用f(,2) 会产生错误；调用f(2,)会提示参数，但不会对结果产生影响

R的时间与日期类型

R的时间与日期类型

- R 里的日期与时间类型很独特
  - 日期被代表为日期类
- 时间被代表为POSIXct或POSIXlt类
- 日期起始计算时间为1970-01-01
  - 时间被存储的日期为1970-01-01的秒的格式

R的时间与日期类型

- 日期被代表为日期类，可以使用as.Date() 函数将字符串变量转换为日期变量。

```
x <-as.Date("1970-01-01")
x
## [1] "1970-01-01"
unclass(x)
## [1] 0
unclass(as.Date("1970-01-02"))
## [1] 1
```

R的时间与日期类型

- 时间类变量可以使用 as.POSIXlt 或 as.POSIXct 生成.

```
x <-Sys.time()
x
## [1] "2013-01-24 22:04:14 EST"
p <-as.POSIXlt(x)
names(unclass(p))
## [1] "sec" "min" "hour" "mday" "mon"
## [6] "year" "wday" "yday" "isdst"
p$sec
## [1] 14.34
```

R的时间与日期类型

- POSIXct 格式.

```
x <-Sys.time()
x ## Already in ‘POSIXct’ format
## [1] "2013-01-24 22:04:14 EST"
unclass(x)
## [1] 1359083054
x$sec
## Error: $ operator is invalid for atomic vectors
p <-as.POSIXlt(x)
p$sec
## [1] 14.37
```

对Dates 和Times变量进行操作

- 示例

```
x <-as.Date("2012-03-01")
y <-as.Date("2012-02-28")
```

- x-y

```
## Time difference of 2 days
x <-as.POSIXct("2012-10-25 01:00:00")
y <-as.POSIXct("2012-10-25 02:00:00", tz = "GMT")
y-x
## Time difference of 9 hours
```

2.1.6 在R中读取数据

读取数据

- R中读取数据的函数.
  - read.table, read.csv, 读取二维表数据
  - *readLines, 按行读取文本
  - * source, 读取R代码
  - * dget, 读取R代码文件
  - * load, 读取工作空间
  - * unserialize, 读取R对象(二进制)

写入数据

写入数据函数

- writeLines
- dump
- dput
- save
- serialize

read.table函数

- read.table函数参数介绍:
  - file, 文件名或路径
  - header, 是否读取表头
  - sep, 指定分隔符
  - colClasses, 指定数据集中的类别变量
  - nrows, 指定读取行的个数
  - comment.char, 指定注释
  - skip, 跳过读取指定的行
  - stringsAsFactors, 将字符型列进行因子转换

read.table

- 读取适量数据时调用read.table函数，不用指定太多的参数

  EX： tab <- read.table("D:/R/hsb2.txt",header=TRUE)

  上例中R 将会自动确定：

  - 跳过表头读取数据
  - 读取的行数
  - 数据的类型

  read.csv 和read.table 类似，除了默认的分隔符为逗号(半角)

  EX： csv2 <- read.csv("D:/R/hsb2.csv",header=TRUE)

“foreign” 包

- read.dta() 读取 Stata 5-11 版二进制格式数据(dataframe)
- read.ssd() 读取 SAS 二进制格式的数据(dataframe)
- read.spss() 读取SPSS 二进制格式的数据(list), 添加“to.data.frame = TRUE”参数，读取成数据框

“openxlsx” 包

- 目前可以稳定的读取EXCEL的包是openxlsx。
- library(openxlsx)
- data=read.xlsx("hsb2.xlsx",sheet=1)

read.table读取大型数据

- read.table读取大型数据应注意：
- 查看read.table帮助页
- 估计一下内存的承受能力，如果数据大小超过了内存，那么最好停止读取这样的数据。

read.table读取大型数据

- 使用colClasses 参数. 指明分类变量以加快读取的速度。在使用此参数时候，需要了解数据集中哪些变量是分类变量。如果所有列都是数值型，可以设置colClasses= “numeric”。快速了解数据集中分类变量情况的语法如下：
  - **initial <-read.table("hsb2.txt",header=TRUE, nrows= 10)**
  - **classes <-sapply(initial, class)**
  - **tabAll<-read.table("hsb2.txt",header=TRUE, colClasses= classes)**
- 设置行读取限制(nrows=)，会节省内存的使用。

了解操作系统

- 一般，在读取大型数据的时候，需要对操作系统有所了解:
  - 可用内存有多少？
  - 其他应用占用了多少内存
  - 是否有其他用户正在使用这个系统
  - 操作系统
  - 操作系统类型(32bit 64bit)

计算内存需要

- 例如一个 1,500,000 行 120列的数据，列为数字型. 那么大概需要多少内存空间?

1,500,000 ×120 ×8 bytes/numeric

= 1440000000 bytes

= 1440000000 / bytes/MB

= 1,373.29 MB

= 1.34 GB