```
---
title: "R Advance"
output: ioslides_presentation
---

```{r setup, include = FALSE}
knitr::opts_chunk$set(echo    = T, collapse = T)
```
```

##R 进阶

```
<center><img src = "image/8.jpg" height = 400/></center>
```
让我们来欢脱的使用 R 吧

##主要内容

- 控制结构
- 函数创建和参数
- 循环函数(apply 函数族)
- 常见数据清理

#控制结构

控制结构

顺序执行, 判断, 循环
R 语言中的控制结构可以用来控制程序的执行流程, 默认是从上到下执行的，其他基本的控制结构包括:

* if-else 结构: 用于测试逻辑条件
* for 结构: 用于执行固定次数的循环
* while 结构: 用于在某个条件成立时执行循环
* repeat 结构: 用于直接执行无限循环
* break 语句: 用于终止并跳出循环
* next 语句: 用于跳过循环中的当前迭代
* return 语句: 用于从函数中退出

if...else..., if...else if ... else...

仅在满足一个条件时执行, 即条件只能得到一个逻辑值, else 一定要跟在大括号后面, else 语句并不是必须的.可以多个 if 语句

注意 if 的条件必须是单个值或表达式的运算, 不能是向量运算.
if 一般会配合循环使用.

```{r, eval = FALSE}
if (条件) { 语句
    } else { 语句 } # 注意如果 else 中的语句写在 2 行中, else 之前一定要有}, 如果把}放在第一行结束, 表示 if 判断语句已经结束了.

if (条件 1) { 语句
} else if (条件 2) {语句
        } else {语句}
```

```{r}
#创建 1 个均值为 3, 标准差为 1 的随机数
x <- rnorm(1, 3, 1); x
[1] 4.586588

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

```
[1] 10

if (x>3) y <- 10 else y <- 0; y
[1] 10

if (x>3) {y <- 10} else {y <- 0}; y
[1] 10

y <- if (x>3) {10} else {0}; y
[1] 10

if (rnorm(1, 3, 1)>3) {
  y <- 10
  print("随机数>3"); y
} else {
  y <- 0
  print("随机数< = 3"); y
}
[1] "随机数< = 3"
[1] 0

if (rnorm(1, 3, 1)>3) {
  y <- 10
  print("随机数>3"); y
} else {
  y <- 0
  print("随机数< = 3"); y
}
[1] "随机数>3"
[1] 10

#一般更关心 if 中要执行的语句, 而不是关心 if 函数的返回值.
y <- if (rnorm(1, 3, 2) 3) 10 else 0; y
[1] 10
```

```

## ifelse

if 的条件语句中只能产生一个的逻辑值, 想要在条件中产生多个逻辑值, 就要使用 ifelse.

ifelse 是 if…else…结构比较紧凑的向量化版本(输入输出均为向量, 向量可以有多个元素),
```

根据条件执行后面两个语句中的一个

```{r}
ifelse(rbinom(1, 1, 0.5), "字", "花") #rbinom, 投硬币, 返回投 1 次得到向上的次数, 只有 0 或 1 两个结果

ifelse(rnorm(1)>0, "a", "b")

#前面的逻辑向量, 只要是 T 的就用"a"代替, 只要是 F 的就用"b"代替, 即 ifelse 的条件可以是逻辑向量
ifelse(c(T, F, T, F), "a", "b")
[1] "a" "b" "a" "b"

#A 表示某人违约超过 180 天, B 表示某人连续 3 个月违约, AB 两个条件只要满足 1 个就可以把此人加入信用的黑名单中, 把他的状态设置为 1, C 表示正常还款或违约小于 30 天, 就不加入信用黑名单, 把他的状态设置为 0. D 表示某人的状态未知, 把他设置为 NA. 这样就能把原向量变为包含 NA 值的 1, 0 向量. 这个向量就是逻辑回归时使用的二分类向量.
x <- c("A", "B", "C", "D", "C", "B", "D", "A")
x <- ifelse(x == "C", 0, x)
x <- ifelse(x == "D", NA, x)
x <- ifelse(x == "A" | x == "B", 1, x)
x
[1] "1" "1" "0" NA "0" "1" NA "1"

#使用 ifelse 的嵌套语句来完成
x <- c("A", "B", "C", "D", "C", "B", "D", "A")
x <- ifelse(x == "C", 0, ifelse(x == "D", NA, 1))
x
[1] 1 1 0 NA 0 1 NA 1
```
ifelse 一般只能用于多条件替换值时使用, 不能像 if 那样执行特定的语句.

switch

- switch 根据后面一个表达式的值选择语句执行: switch(expr, ...),
- 表达式是数字则按顺序, 是字符则按名字, 表达式长度必须为 1
- 按名字匹配都匹配不到, 且最后一句没有给名字, 则执行最后一句

```{r}
switch(2, "不", "要")
[1] "要"
switch("median", median = median(c(-1:9, 50)), mean = mean(c(-1:9, 50)))
[1] 4.5
```

```
switch("aaa", median = median(c(-1:9, 50)), mean = mean(c(-1:9, 50)), print("nothing"))
[1] "nothing"
```

for

while 和 for 都是控制流，不能算是函数，使用?while 是不能得到帮助信息的. 而是使用自动补全.

?while (condition) {

}

?for (variable in vector) {

}

最常见的循环运算符. 基本思想是：设置一个循环下标(通常为 i)，然后循环下标遍历一个向量，每取一个值就代入循环中的语句计算一次，直到遍历完向量中所有元素.

for 循环的效率并不高，尤其是对大型数据时，所以一般不使用 for 循环来处理纯数值的运算.

```{r}
for(i in 1:2) {print(i)}
[1] 1
[1] 2

x <- c("a", "b")
for(i in 1:2) {print(x[i])}
[1] "a"
[1] "b"

x <- c("a", "b")
seq_along(x)
[1] 1 2
# seq_along 比 seq 更有效率，输入向量，输出等长的数列
for(i in seq_along(x)){
    print(x[i])}
[1] "a"
[1] "b"

x <- c("a", "b")
for(letter in x) print(letter)
[1] "a"
[1] "b"
```

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

for 一般与其它如取子集的方法配合使用

使用循环对 airquality 求列均值

```{r}
# 定义一个数值型的空向量
cn <- numeric()
for(i in 1:ncol(airquality)){
    cn[i] <- mean(airquality[, i], na.rm = T)
}
cn

[1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

for 嵌套循环

嵌套越多给理解带来的难度越大，尽量不要超过两层.

```{r}
#对一个 2*3 的矩阵，逐行打印出所有的元素

x <- matrix(1:6, 2, 3); x
     [, 1] [, 2] [, 3]
[1, ]    1    3    5
[2, ]    2    4    6

for(i in seq_len(nrow(x))) {   #先定位到每一行
    for(j in seq_len(ncol(x))) {   #再定位到每一行中的元素，打印该元素
        print(x[i, j])
    }
}

[1] 1
[1] 3
[1] 5
[1] 2
```

```
[1] 4
[1] 6

# 也可以直接把矩阵强制转换为向量
```

while

从条件语句开始，得到真执行后面的执行语句，执行完后继续判定条件，直到条件得到假，循环结束. (可能由于循环条件的设置不当而造成死循环)
使用 while 时一定要注意循环的下标，要设定循环下标的初始值才能正确开始执行循环.

```{r}
count <- 0
while(count < 6){
    print(count)
    count <- count + 1
}
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5

```

while 求列均值
```{r}
i <- 1
cn1 <- numeric()
while(i < = ncol(airequality)){
    cn1[i] <- mean(airquality[, i], na.rm = T)
    i <- i+1
}
cn1
[1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
i  # 查看 i 的值

```

while 多条件

条件语句常常多于一句，一般由左至右执行

```{r}
# 根据投掷一枚硬币的正反进行随机游走，直到 z 跳出某一个范围
z <- 5
while(z == 3 && z <= 10) {
    print(z)
    coin <- rbinom(1, 1, 0.5)  ## 投一枚均匀硬币
    if(coin == 1) {
        z <- z + 1
    } else {
        z <- z - 1
    }  ##小型的随机游走
}
z

z <- 5
while(z == 3 && z <= 10) {
  print(z)
  coin <- rbinom(1, 1, 0.5)  ## 投一枚均匀硬币
  z <- z + ifelse(coin, 1, -1)
}
z

```

repeat

无限循环，在统计应用中使用较少，但是偶尔会用到. 终止的办法是使用 break 函数

```{r, eval = F}
# 使用自定义的函数计算出 x1 的值，把 x1 和初始值 x0 做比较，当二者的差值小于定义的误差 tol 时就跳出循环. 如果二者的差值大于定义的误差 tol 时就把 x1 赋值给 x0，然后再计算出一个新的 x1，把新的 x1 与 x0 做比较，也就是和上一步计算得到的 x1 做比较，如果二者的差值仍然很大，就再次计算新的 x1，也上一次的进行比较，直到连接两次计算的 x1 的差值小于给定的误差，才跳出循环. 这是一个逐步优化的过程，类似于梯度下降的思想.

x0 <- 1
tol <- 1e-8
repeat{
```

```
    x1 <- computeEstimate()    #computeEstimate 是自己写的一个函数，先运算这个函数计算 x1 的估计值
    if(abs(x1 - x0) < tol) {
        break
    } else {
        x0 <- x1
    }
}
```

以上是计算算法收敛常用的函数，但是并不是所有的算法都是收敛的. 算法的效果还部分取决于容差，容差越小，循环的时间越长. 通常来说，由于难以预测循环的时间，就会存在无限循环的危险.

最好使用 for 循环，for 循环具有对该循环可运行迭代次数的硬性限制. 可有效防止死循环.

在 repeat 函数中，如果算法没有收敛也不会出现警告，而是一直循环下去.

next, return

next 可以用在任何一种循环中，用来跳过某一次迭代

```{r, eval = F}
for (i in 1:10) {
  if (i 3 && i < 6) {
    next   ##跳过中间的 4, 5 次循环
  }
  print(i)   ##其他想要执行的语句
}
[1] 1
[1] 2
[1] 3
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

break 函数则是完全跳出循环

return 也可以退出循环，但它主要用来退出函数，会结束整个函数，返回一个值.

Practices 1

1. 转换为二变量向量

1. 对于向量 loan <- c("A", "B", "D", "C", "C", "B", "A", "B", "A", "D"), 其中 A, C 表示客户违约的两种状态, 即均违约, 用 "是" 代替, D 表示客户未违约用 "否" 代替, B 表示状态不明用 NA 代替

```{r, echo = F}
loan <- c("A", "B", "D", "C", "C", "B", "A", "B", "A", "D")
ifelse(loan == "A" | loan == "C", "是", ifelse(loan == "D", "否", NA))
ifelse(loan == "D", "否", ifelse(loan == "B", NA, "是"))
```

```{r}
loan <- c("A", "B", "D", "C", "C", "B", "A", "B", "A", "D")

x <- character()
for (i in 1:length(loan)) {
    if (loan[i] == "A" | loan[i] == "C") {
        x[i] <- "是"
    } else if (loan[i] == "D") {
        x[i] <- "否"
    } else {
        x[i] <- NA
    }
}
x
```

提示: 可以使用 ifelse 函数嵌套, 可以使用 for 循环加 if-else 或者 switch 函数

2 . 生成九九乘法表

2. 写出如下格式的九九乘法表, 提示这里当然可以使用循环, 但是可否不用循环呢? 如何打印出上三角或下三角.

使用循环

1. 生成完整的乘法表

```
# 生成 9*9 的 NA 矩阵
m <- matrix(NA, 9, 9); m
```

```
     [, 1] [, 2] [, 3] [, 4] [, 5] [, 6] [, 7] [, 8] [, 9]
[1, ]    NA    NA    NA    NA    NA    NA    NA    NA    NA
[2, ]    NA    NA    NA    NA    NA    NA    NA    NA    NA
[3, ]    NA    NA    NA    NA    NA    NA    NA    NA    NA
[4, ]    NA    NA    NA    NA    NA    NA    NA    NA    NA
[5, ]    NA    NA    NA    NA    NA    NA    NA    NA    NA
[6, ]    NA    NA    NA    NA    NA    NA    NA    NA    NA
[7, ]    NA    NA    NA    NA    NA    NA    NA    NA    NA
[8, ]    NA    NA    NA    NA    NA    NA    NA    NA    NA
[9, ]    NA    NA    NA    NA    NA    NA    NA    NA    NA
```

```
# i, j 分别控制
for (i in 1:9) {
    for (j in 1:9) {
        m[i, j] <- i * j
    }
}
m
```

```
     [, 1] [, 2] [, 3] [, 4] [, 5] [, 6] [, 7] [, 8] [, 9]
[1, ]    1    2    3    4    5    6    7    8    9
[2, ]    2    4    6    8   10   12   14   16   18
[3, ]    3    6    9   12   15   18   21   24   27
[4, ]    4    8   12   16   20   24   28   32   36
[5, ]    5   10   15   20   25   30   35   40   45
[6, ]    6   12   18   24   30   36   42   48   54
[7, ]    7   14   21   28   35   42   49   56   63
[8, ]    8   16   24   32   40   48   56   64   72
[9, ]    9   18   27   36   45   54   63   72   81
```

 # 2. 得到下半部分的乘法表，上三角为 NA，方法 1

```
m <- matrix(NA, 9, 9); m
for(i in 1:9){
    for(j in 1:9){
        if (i < j) next
        m[i, j] <- i * j
    }
}
m
```

```
     [, 1] [, 2] [, 3] [, 4] [, 5] [, 6] [, 7] [, 8] [, 9]
```

```
 [1, ]    1   NA   NA   NA   NA   NA   NA   NA   NA
 [2, ]    2    4   NA   NA   NA   NA   NA   NA   NA
 [3, ]    3    6    9   NA   NA   NA   NA   NA   NA
 [4, ]    4    8   12   16   NA   NA   NA   NA   NA
 [5, ]    5   10   15   20   25   NA   NA   NA   NA
 [6, ]    6   12   18   24   30   36   NA   NA   NA
 [7, ]    7   14   21   28   35   42   49   NA   NA
 [8, ]    8   16   24   32   40   48   56   64   NA
 [9, ]    9   18   27   36   45   54   63   72   81
```

 # 3. 得到下半部分的乘法表，上三角为 NA，方法 2

```
m <- matrix(NA, 9, 9); m #9*9 的 NA 矩阵
for (i in 1:9) {
    for (j in 1:i) {
        m[i, j] <- i * j
    }
}
m
```

```
     [, 1] [, 2] [, 3] [, 4] [, 5] [, 6] [, 7] [, 8] [, 9]
 [1, ]    1   NA   NA   NA   NA   NA   NA   NA   NA
 [2, ]    2    4   NA   NA   NA   NA   NA   NA   NA
 [3, ]    3    6    9   NA   NA   NA   NA   NA   NA
 [4, ]    4    8   12   16   NA   NA   NA   NA   NA
 [5, ]    5   10   15   20   25   NA   NA   NA   NA
 [6, ]    6   12   18   24   30   36   NA   NA   NA
 [7, ]    7   14   21   28   35   42   49   NA   NA
 [8, ]    8   16   24   32   40   48   56   64   NA
 [9, ]    9   18   27   36   45   54   63   72   81
```

 # 4. 得到上半部分的乘法表，下三角为 NA

```
m <- matrix(NA, 9, 9); m
for (i in 1:9) {
    for (j in i:9) {
        m[i, j] <- i * j
    }
}
m
```

```
     [, 1] [, 2] [, 3] [, 4] [, 5] [, 6] [, 7] [, 8] [, 9]
 [1, ]    1    2    3    4    5    6    7    8    9
```

```
[2, ]   NA    4    6    8   10   12   14   16   18
[3, ]   NA   NA    9   12   15   18   21   24   27
[4, ]   NA   NA   NA   16   20   24   28   32   36
[5, ]   NA   NA   NA   NA   25   30   35   40   45
[6, ]   NA   NA   NA   NA   NA   36   42   48   54
[7, ]   NA   NA   NA   NA   NA   NA   49   56   63
[8, ]   NA   NA   NA   NA   NA   NA   NA   64   72
[9, ]   NA   NA   NA   NA   NA   NA   NA   NA   81
```

使用矩阵乘法

矩阵运算在很多时候都能够简化运算, 并且代码和思路都显得很清晰, 所以一定要重视矩阵运算.

A(m, n)*B(n, p) = C(m, p)

```
| 1 |  *   | 1   2 |   =  | 1   2 |
| 2 |  *                  | 2   4 |
```

9 个元素的行矩阵乘以 9 个元素的列矩阵得到的结果就是 9*9 的矩阵

```
# 把向量 1:9 强制转换为矩阵
as.matrix(1:9)
     [, 1]
[1, ]    1
[2, ]    2
[3, ]    3
[4, ]    4
[5, ]    5
[6, ]    6
[7, ]    7
[8, ]    8
[9, ]    9

# 把向量 1:9 强制转换为矩阵再转置
t(1:9)
     [, 1] [, 2] [, 3] [, 4] [, 5] [, 6] [, 7] [, 8] [, 9]
[1, ]     1     2     3     4     5     6     7     8     9

# 1:9 也会强制转换为矩阵
1:9  %*%  t(1:9)
     [, 1] [, 2] [, 3] [, 4] [, 5] [, 6] [, 7] [, 8] [, 9]
[1, ]    1    2    3    4    5    6    7    8    9
[2, ]    2    4    6    8   10   12   14   16   18
[3, ]    3    6    9   12   15   18   21   24   27
```

```
[4, ]    4    8   12   16   20   24   28   32   36
[5, ]    5   10   15   20   25   30   35   40   45
[6, ]    6   12   18   24   30   36   42   48   54
[7, ]    7   14   21   28   35   42   49   56   63
[8, ]    8   16   24   32   40   48   56   64   72
[9, ]    9   18   27   36   45   54   63   72   81
```

矩阵运算具有很多优点. 简单, 思路清晰, 所以现在都很强调矩阵运算.

```{r, echo = F}
matrix(rep(1:9, 9), 9, 9) * matrix(rep(1:9, 9), 9, 9, byrow = T)
```

```
     [, 1] [, 2] [, 3] [, 4] [, 5] [, 6] [, 7] [, 8] [, 9]
[1, ]    1    2    3    4    5    6    7    8    9
[2, ]    2    4    6    8   10   12   14   16   18
[3, ]    3    6    9   12   15   18   21   24   27
[4, ]    4    8   12   16   20   24   28   32   36
[5, ]    5   10   15   20   25   30   35   40   45
[6, ]    6   12   18   24   30   36   42   48   54
[7, ]    7   14   21   28   35   42   49   56   63
[8, ]    8   16   24   32   40   48   56   64   72
[9, ]    9   18   27   36   45   54   63   72   81
```

```

## ##Practices 2

写出如下格式的九九乘法表

提示: 结果格式不规整的时候需要用到循环, 通过查看?print 查看如何不打印引号

```{r, echo = F}
for (i in 1:9) {
    a <- vector()
    for (j in 1:i) {
        a[j] <- paste0(i, "*", j, " = ", i*j)
    }
    print(a, quote = F)
}
```

```
[1] 1*1 = 1
[1] 2*1 = 2 2*2 = 4
[1] 3*1 = 3 3*2 = 6 3*3 = 9
[1] 4*1 = 4  4*2 = 8  4*3 = 12 4*4 = 16
```
```

```
[1] 5*1 = 5  5*2 = 10 5*3 = 15 5*4 = 20 5*5 = 25
[1] 6*1 = 6  6*2 = 12 6*3 = 18 6*4 = 24 6*5 = 30 6*6 = 36
[1] 7*1 = 7  7*2 = 14 7*3 = 21 7*4 = 28 7*5 = 35 7*6 = 42 7*7 = 49
[1] 8*1 = 8  8*2 = 16 8*3 = 24 8*4 = 32 8*5 = 40 8*6 = 48 8*7 = 56 8
*8 = 64
[1] 9*1 = 9  9*2 = 18 9*3 = 27 9*4 = 36 9*5 = 45 9*6 = 54 9*7 = 63 9
*8 = 72 9*9 = 81
```

函数

函数在编程中扮演了非常重要的角色，是必须熟悉并且掌握的编程技巧. 也可以把函数放在R 包里.
R 函数中不一定要用 return 返回值，R 函数会自动返回最后一个表达式的值.

函数示例

定义两个数加和函数 add2

```{r}
add2 <- function(x, y) {x + y}

add2 <- function(x, y) {
    out = x + y  # out 为自由变量, 局部变量, 在函数内部定义的变量. 函数内部也可以看成是一个封闭的环境, 自由变量只在函数内部有效.
    return(out)  # 如果没有 return, 会自动返回最后一个表达式的值
}

# 调用函数
add2(3, 5)
```

取向量中大于指定数字的元素的函数 above

```{r}
above <- function(x, n = 10) {
    use <- x>n
    x[use]
}

above <- function(x, n = 10) {
```

```
    x[x>n]
}

x <- 1:20
#只给出向量 x, 不指定 n
above(x)
[1] 11 12 13 14 15 16 17 18 19 20
above(x, 12)
[1] 13 14 15 16 17 18 19 20
```

```
# 计算每列均值的函数 ColumnMean

```{r}
ColumnMean <- function(y, removeNA = TRUE) {
    nc <- ncol(y)
    means <- numeric(nc)
    for (i in 1:nc) {
        means[i] <- mean(y[, i], na.rm = removeNA)
    }
    return(means)
}

ColumnMean <- function(y, removeNA = TRUE) {
    x <- numeric()
    for (i in 1:ncol(y)) {
        x[i] <- mean(y[, i], na.rm = removeNA)
    }
    return(x)
}

ColumnMean(airquality)
```

记得经常保存文件: ctrl+S

函数

函数通过 function()指令创建，被当做 R 的一个对象存储. 函数属于 R 的对象类别“函数类”
```

```{r, eval = F}
f <- function(<参数>) {
    ##show your capability
}
```

在 R 中函数被认为是一级对象，所以函数的处理方式和其他类似于向量的处理方式是一样的.

* 可以把函数作为其他函数的参数
* 函数可以嵌套，既可以在一个函数中定义一个函数

函数参数

函数包含已经命名的参数，这些参数可以赋值默认值.

* 函数定义里的参数称为形参
* formals()函数可以返回一个函数所有的形参（函数名作为输入）
* 并不是每次调用函数都会用到所有的形参
* 函数参数可以缺失或者有缺省值

参数匹配

函数参数可以通过位置或者名字匹配.
以下对 sd()函数的调用都是等价的:

?sd
Standard Deviation
Description
This function computes the standard deviation of the values in x. If na.rm is TRUE then missing values are removed before computation proceeds.
Usage
sd(x, na.rm = FALSE)

```{r, collapse = T}
mydata <- rnorm(100)
sd(mydata)
sd(x = mydata)
sd(x = mydata, na.rm = T)
sd(na.rm = T, x = mydata)
sd(na.rm = T, mydata)
```

参数匹配可以名字匹配和位置匹配混合使用，先匹配名字，剩下的参数按照位置匹配.

```{r}
args(rnorm)
function (n, mean = 0, sd = 1)
NULL

rnorm(mean = 4, 10, 1)
 [1] 4.604309 4.548788 4.916433 6.661566 3.819743 4.685015 7.266415 4.560
600 3.930983 3.027557
rnorm(me = 4, sd = 2, 9)
[1] 2.9068268 0.6226153 0.8552546 3.1900257 4.6385728 4.0808554 3.2199809
 0.3615555 5.3183614
```

```{r}
args(lm)

lm
Fitting Linear Models
Description
lm is used to fit linear models. It can be used to carry out regression, single stratum analysis of variance and analysis of covariance (although aov may provide a more convenient interface for these).
Usage
lm(formula, data, subset, weights, na.action,
    method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
    singular.ok = TRUE, contrasts = NULL, offset, ...)

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

下面两个调用是等价的.

```{r, eval = F}
lm(data = mydata, y~x, model = F, 1:100)
lm(y~x, mydata, 1:100, model = F)
```

R 中匹配参数的顺序是:

1. 寻找参数名的精确匹配
2. 寻找部分匹配，缩写匹配，但一定要能匹配到唯一的值.

3. 寻找位置匹配

惰性取值

函数参数的取值是惰性的，只有需要的时候才会求值.

```{r}
f <- function(a, b) {
    a^2
}
f(3)
```

以上函数并没有使用参数 b，所以调用 f(3)的时候并不会返回错误，因为 3 是通过位置匹配到 a 的.

```{r, error = T}
f <- function(a, b) {
    print(a)
    print(b)
}
f(45)
```
先打印了 45，再触发了错误，因为参数 b 在打印 a 之前不需要取值. 只有当函数试图打印 b 的时候才会触发错误.

参数 ...

参数 ... 作用 1

参数 ... 用来表明可以传递给另一个函数的一些参数.

... 常用于需要拓展到另一个函数，而又不想复制原函数的整个参数列表.

?read.csv
read.table 和 read.csv 中参数的个数和名称都是相同的, 只有一些参数的默认值是不同的, read.csv 中的 ... 表示把 read.table 中其它未在 read.csv 中定义的参数全部复制到 read.csv 中. ... 也是一个参数, 这个参数的作用是传递其它所有的参数.

```{r}
myplot <- function(x, y, type = "l", ...) {
    plot(x, y, type = type, ...)
}
```

```

* ...使用在泛型函数(Generic function)中，使得附加函数可以传递给方法(methods)
    - 泛型函数不做任何事，它的作用是根据数据类型选择合适的方法.

```{r}
mean
```

参数 ... 作用 2

* 当传递到函数的参数数量不能事先确定的时候需要使用...

```{r}
args(paste)
paste 可以把若干个向量粘贴到一起, 其中的… 就表示不确定数量的参数.
#function (..., sep = " ", collapse = NULL)

args(cat)
#function (..., file = "", sep = " ", fill = FALSE, labels = NULL, append = FALSE)
```

... 之后的参数

使用 ... 的一个陷阱是任何出现在 ... 之后的参数都需要明确的给出名称，不能进行部分匹配. 这样做是为了让 R 区分对象传递给的参数是 ... 还是后面的参数. ... 表示任意多个向量

```{r}
args(paste)

paste("a", "b", "c", sep = ":")  # 是位置匹配的方式, 把 a b c 三个参数都传递给 ..., ... 后面的参数必须给出全名, 才能正确匹配.
paste("a", "b", "c", se = ":")  #如果不使用全名匹配, 会把 se = ":" 也传递给...
```

编码标准

* 使用文本编辑器并且将代码保存为文本文件
```

* 代码缩进(4 列/8 列)
* 限制代码宽度(一本限制在 80 列)
* 限制函数的长度

Practices 3

- 创建一个奇偶项相加的函数 sumforfun, 输入一个数值向量(如果输入错误返回提示信息), 输出一个数值向量, 如果输入的向量长度为奇数, 则最后一个值忽略, 处理过程是每个奇数项加后面紧跟的偶数项, 函数输出结果如下

```{r, echo = F}
sumforfun <- function(x) {
    if (!is.numeric(x)) {   #如果不是数值型向量
        return("x should be a numeric vector")
    } else {   #如果是数值型向量
        y <- x[-1] + x[-length(x)]   #如果是奇数个, 去掉最后一个
        return(y[rep(c(T, F), length = length(y))])
    }
}


sumforfun <- function(x) {
    if (!is.numeric(x)) {   #如果不是数值型向量
        return("x should be a numeric vector")
    } else {   #如果是数值型向量
        n <- length(x)
        n <- n%/%2 *2
        n <- x[1:n]
        m <- matrix(x, 2)   #把向量转化为矩阵, 这样奇数项都放在第 1 行, 偶数项放在第 2 行, 再求列和
        return(colSums(m))
    }
}

sumforfun(1:5)
sumforfun(c(1, 2, 3, 4, 99, 100))
sumforfun(c("a"))
```

- 提示: 用到条件选择, 可以用循环, 但是思考有没有不用循环就能实现的方式.

apply 系列循环函数

apply 系列函数

在处理大型的数据时, 循环的效率并不高, 在进行数据处理的时候, 尽量不要直接使用循环来处理每一行或每一列的数据, 而是把循环作为一个辅助的工具, 配合其它的方法来完成循环的功能.

之前介绍的 for, while 循环在交互式命令行操作时不够简洁, 在 R 中有几种循环函数, 通常名字里都带有 apply 一词. 内部真正的循环是通过 C 代码实现的, 所以执行的效率要比循环快一些.

- lapply: list apply. 用途, 有一个对象列表, 遍历这个对象列表, 对列表的每个元素运用函数.
- sapply: simple lapply. 是 lapply 的一个变体, 简化了 lapply()的结果.
- apply: 对数组进行行或列运算的函数, 对矩阵和高维数组做总结
- tapply: table apply()的缩写, 将函数应用于向量的子集
- mapply: multi lapply. 是 lapply 的多变量版本
- split: 不对对象做任何操作, 将对象分成子块, 和 lapply, sapply 结合使用

?lapply

lapply {base} R Documentation

Apply a Function over a List or Vector

Description

lapply returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.

sapply is a user-friendly version and wrapper of lapply by default returning a vector, matrix or, if simplify = "array", an array if appropriate, by applying simplify2array(). sapply(x, f, simplify = FALSE, USE.NAMES = FALSE) is the same as lapply(x, f).

vapply is similar to sapply, but has a pre-specified type of return value, so it can be safer (and sometimes faster) to use.

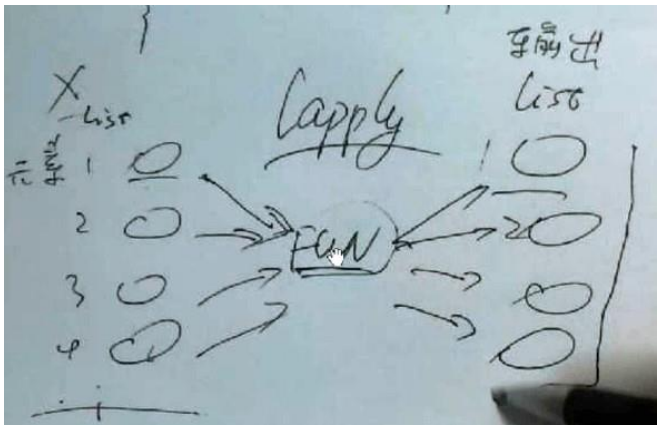
replicate is a wrapper for the common use of sapply for repeated evaluation of an expression (which will usually involve random number generation).

simplify2array() is the utility called from sapply() when simplify is not false and is similarly called from mapply().

lapply(X, FUN, ...)
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
replicate(n, expr, simplify = "array")
simplify2array(x, higher = TRUE)

lapply

lapply 有三个参数:
(1) 列表 x, 如果不是列表会强制转换成列表 as.list, 不成功则报错;
(2) 函数或者函数名;
(3) 其他参数. 可以传递给参数..., 参数...传递给之前的函数
输入的列表中的每一个元素都经过函数的处理输出到结果列表中.



lapply 总是返回一个列表, 不管输入的是什么类型

```{r}
x <- list(a = 1:5, d = rnorm(100, 5)); x

lapply(x, mean)
$a
[1] 3

$d
[1] 4.981718

#对 list 使用 lapply 结果还是 list
class(lapply(x, mean))
[1] "list"

sapply(x, mean)
        a         d
```

```
3.000000  4.981718

#对 list 使用 sapply 会把结果转化为向量
class(sapply(x, mean))
[1] "numeric"

#先把向量转化为列表, 再使用 runif 来产生均匀分布的随机数, 1 产生 1 个随机数, 2 产生 2 个随机数, 3 产生 3 个随机数
x <- 1:3; lapply(x, runif)
[[1]]
[1] 0.03910006

[[2]]
[1] 0.5045050 0.5784826

[[3]]
[1] 0.8393039 0.6544450 0.9445283

class(lapply(x, runif))
[1] "list"

#使用 sapply 的结果与 lapply 一致, 因为元素个数不同, 所以不能进行简化, 结果还是 list
x <- 1:3; sapply(x, runif); class(sapply(x, runif))
[[1]]
[1] 0.2964848

[[2]]
[1] 0.1325919 0.2659109

[[3]]
[1] 0.005727543 0.819601821 0.174661679

[1] "list"
```

lapply 中函数的参数

lapply(X, FUN, ...)
lapply 中的…参数, 就是用来放 FUN 中除了第 1 个参数 X 之外的其它所有参数.

```
# 把 x 转化为列表, 再把列表中的每个元素作为参数传递给 runif, 得到 runif(1), runif(2), runif(3).
x <- 1:3; lapply(x, runif)
[[1]]
```

```
[1]  0.08016928

[[2]]
[1]  0.1165686  0.9251926

[[3]]
[1]  0.8788442  0.1632076  0.5315639
```

#直接把 runif 的参数写在函数的后面，创建从 0 到 10 之间均匀分布的随机数. runif 函数的第 1 个参数是从 x 中依次取它的元素，而其它的参数就写在函数的后面即可. 其它参数是通过...来传递的.

```
x <- 1:3; lapply(x, runif, min = 0, max = 10)
[[1]]
[1]  9.393948

[[2]]
[1]  0.4604055  5.3486173

[[3]]
[1]  4.426608  9.063655  8.810424
```

lapply 与匿名函数

lapply 和相关的函数充分利用了所谓的匿名函数，匿名函数没有函数名，可以直接创建函数

```
x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2)); x
$a
     [, 1] [, 2]
[1, ]    1    3
[2, ]    2    4

$b
     [, 1] [, 2]
[1, ]    1    4
[2, ]    2    5
[3, ]    3    6
```

现在要提取这两个矩阵的第一列，没有现成的函数，要自己构建一个函数.

#自定义了一个函数，取输入的矩阵的第一列.

```
col <- function(x) x[, 1]
lapply(x, col)
$a
[1] 1 2
```

```
$b
[1] 1 2 3
```

还可以使用匿名函数，这个函数在 lapply()之外是不存在的，被调用完后函数就消失了

```
lapply(x, function(x) x[, 1])
lapply(x, function(x) {x[, 1]})
lapply(x, function(x) {n <- x[, 1]; return(n)})
lapply(x, function(elt) {n <- elt[, 1]; return(n)})
```

如果没有 return，函数会返回最后一个表达式的值，最后一个表达式最好不要有赋值号，否则可能出错，如果要使用赋值号，要使用 return 返回值.

```
lapply(x, function(elt) {n <- elt[, 1]; return(n)})
$a
[1] 1 2

$b
[1] 1 2 3
```

sapply

sapply 会使 lapply 的输出结果尽可能简化，如果能把列表简化为数据框就转化为数据框，如果能简化成向量就简化成向量，能简化为矩阵就简化为矩阵. 一般会简化为数据框和向量

- 结果是每个长度都是 1 的列表，会返回一个向量
- 结果是每个元素长度相同(>1)的列表，会返回一个矩阵
- 如果没办法简化会返回一个 list

```
x <- list(a = 1:5, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5)); x

sapply(x, mean)
         a          b          c          d
3.00000000 -0.05609066  0.95660224  5.09248215

mean(x)
[1] NA
Warning message:
In mean.default(x) ： 参数不是数值也不是逻辑值：回覆 NA
```

sapply 对 airquality 的每一列求均值

```
sapply(airquality, mean, na.rm = T)
```

```
     Ozone       Solar.R          Wind          Temp         Month           Day
 42.129310  185.931507     9.957516    77.882353     6.993464    15.803922
```

apply

apply 函数可以把一个函数应用在一个数组的各个维度上，通常这个函数是匿名函数

- 通常的应用对象是矩阵的行或列 (矩阵作为一个二维数据，是最常见的数组类型)
- 可以应用在一般的数组中，例如对一个数组取平均值
- 相较于通过 for 写循环并没有更快，只是输入变得更少(程序员都有手癌，输入越少越好)

```
str(apply)
function (X, MARGIN, FUN, ...)
```

- X 是一个数组或矩阵，矩阵可以看成是维度为 2 的数组, apply 的对象通常是矩阵的行或列，是把函数作用在矩阵的每一行或每一列中. 如果 x 是一个数据框，并且这个数据框可以转换为矩阵，则会自动转换为矩阵. 数据框在每一列的数据类型都一致时可以转换为矩阵. FUN 是应用的函数, ...是传递给 FUN 的其他参数
- MARGIN 是一个整数向量，指定哪个边界保留. MARGIN = 1，就是把函数作用在每一行上，也就是保留行边界. MARGIN = 2，就是把函数作用在每一列上，即保留列边界. (注意 MARGIN 为一个数时与向量时理解的不同, MARGIN 为向量时可以理解为压缩, MARGIN 为单个数时表示为行列.)

```{r}
x <- matrix(rnorm(200), 20, 10); x
apply(x, 1, mean)  #MARGIN = 1 对每一行求均值. 相当于把函数作用在每一行上，可以看成是把每一行的数据都压缩到列的维度上，成为一个列向量，再把函数作用在这个向量上.
apply(x, 2, sum)  #MARGIN = 2 对每一列求和. 相当于把函数作用在每一列上，可以看成是把每一列的数据都压缩到行的维度上，成为一个行向量，再把函数作用在这个向量上.
```

MARGIN 等于数组时的用法

```
# 定义一个 2*2*2 的三维数组
y <- array(1:8, c(2, 2, 2)); y
,, 1

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

,, 2
```

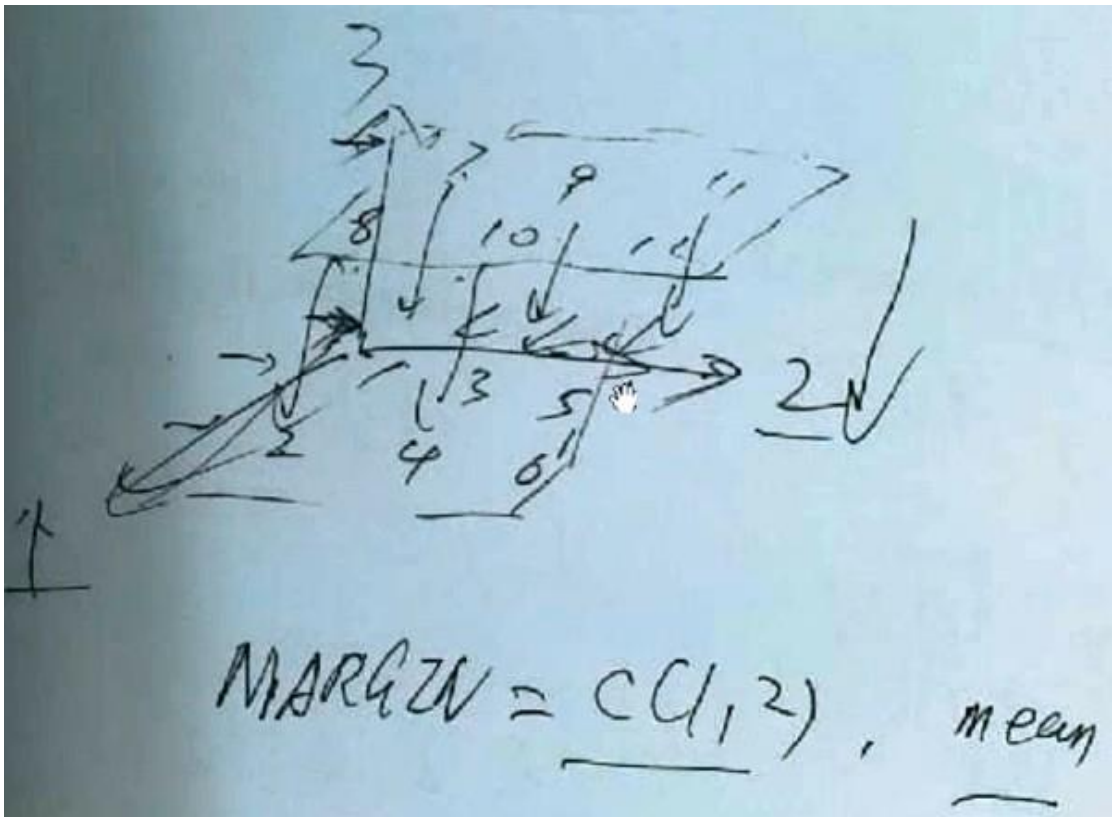
```
     [, 1] [, 2]
[1, ]    5    7
[2, ]    6    8
```

apply(y, c(1, 2), sum) # 对于 3 维的数据, apply 中的 c(1, 2)中可以取 1, 2, 3, 分别对应 x, y, z 3 个维度. MARGIN 等于 c(1, 2)时表示保留 1, 2 维, 可以理解为把 3 维上的数据压缩到向量中定义的 2 个维度上, 在这 2 个维度的 4 个元素上都得到一个向量.

#(1, 5) (3, 7)

#(2, 6) (4, 8)

再把函数作用在这些向量上. 得到一个向量中定义的维度的矩阵.



还可以把三维的数据看成是垂直于 z 轴的两个平面上的数据组成的关于 z 的一个一维向量. 向量有 2 个元素.

第一个元素是

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

第二个元素是

```
     [, 1] [, 2]
[1, ]    5    7
[2, ]    6    8
```

apply(y, c(1, 2), sum)就相当于压缩 z 轴, 把两个平面上对应的数字作为在 FUN 上.

当定义的维度等于数据的维度时，会把原数据输出，而不会进行任何处理．如

```
apply(y, c(1, 2, 3), sum)
, , 1

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

, , 2

     [, 1] [, 2]
[1, ]    5    7
[2, ]    6    8
```

向量中定义的维度哪个写在第 1 位，哪个就成为结果的第 1 维，如此类推．如 apply(y, c(2, 1, 3), sum)就是把原来的 1 维和 2 维的数据进行交换．

```
apply(y, c(2, 1, 3), sum)
, , 1

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

, , 2

     [, 1] [, 2]
[1, ]    5    6
[2, ]    7    8
```

apply(y, c(2, 3), sum)　#保留 2, 3 维，把 1 维的数压缩．把 FUN 函数作用在垂直于 x 轴的两个平面上对应的数据上．

```
#z
# (5, 6) (7, 8)
# (1, 2) (3, 4)
#0                y

#  并且把第 2 维的数据作为结果的 1 维，把第 3 维的数据作为结果的第 2 维，得到
#(1, 2) (5, 6)
#(3, 4) (7, 8)

#       [, 1] [, 2]
#[1, ]      3     11
```

#[2,] 7 15

apply(y, c(3, 2), sum)

#把 1 维的数据压缩，得到

#z
(5, 6) (7, 8)
(1, 2) (3, 4)
#0 y

#并且把第 3 维的数据作为结果的 1 维，把第 2 维的数据作为结果的第 2 维. c(3, 2)中定义的向量哪个写在前面，就把哪个作为结果的 1 维. 得到
#(1, 2)(3, 4)
#(5, 6)(7, 8)

结果为:
[, 1] [, 2]
#[1,] 3 7
#[2,] 11 15

```
y <- array(1:8, c(2, 2, 2)); y
,, 1

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

,, 2

     [, 1] [, 2]
[1, ]    5    7
[2, ]    6    8
```

apply(y, 1, sum) #对行求和. 先把 z 轴数据压缩到 xy 面上，得到
#(1, 5) (3, 7)
#(2, 6) (4, 8)
#再对行求和即可，可以看成是把数据压缩到 y 轴上，得到
#(1, 5, 3, 7)
#(2, 6, 4, 8)
#求和，得到数值型向量(16, 20).

#只保留 x 维度，也可以理解为直接把 y, z 平面上的数据都压缩到一维上，得到(1, 3, 5, 7) (2,

4, 6, 8). 理解上可以用压缩的方式进行理解, 实际上是的运算是把垂直于 x 轴的两个平面上的元素转换为矩阵, 传递给 FUN 函数进行处理的, 这时就可以把垂直于 x 轴的两个平面上的所有元素作为一个一维的向量, 两个元素分别是
| 5 7 | | 6 8 |
| 1 3 | | 2 4 |

所以当 MARGIN 是一个维度时, 压缩的另外两个维度的数据, FUN 函数必须要能够处理矩阵.
```

```{r}
apply(airquality, 1, mean, na.rm = T)  #等价于 rowMeans(). 对每行求均值. airquality 这个数据比较特殊, 都是数值型的元素, 所以可以转化为矩阵
apply(airquality, 2, mean, na.rm = T)  #等价于 colMeans(). 对 airquality 的每列求均值

apply(airquality, 1, sum, na.rm = T)  #等价于 rowSums(). 对每行求和.
apply(airquality, 2, sum, na.rm = T)  #等价于 colSums(). 对 airquality 的每列求和

# apply 的效率没有 solSum, colMeans 高. 但直接对行列计算的函数只有 rowMeans, colMeans, rowSums, colSums, 而使用 apply 则可以组合出无数个对行列进行的操作.

quantile(1:10, probs = c(0, 0.25, 0.5, 0.75, 1))  # 求 quantile 向量的分位数.

apply(airquality, 2, quantile, probs = c(0.25, 0.75), na.rm = T)  #求 airquality 每一列的 1/4, 3/4 分位数, probs 和 na.rm = T 都是 quantile 函数的参数. 因为输入的是矩阵, 输出的也是矩阵.
```

对于计算行列和或均值的简单操作, 有经过优化的专用函数, 可以快速实现这个功能, 他们比 apply 函数运行的更快, 尤其矩阵较大时

- rowSums = apply(x, 1, sum)
- rowMeans = apply(x, 1, mean)
- colSums = apply(x, 2, sum)
- colMeans = apply(x, 2, mean)

apply 函数的应用举例

1. 可以将 apply 应用在其他类型的函数上, 如下求矩阵每行的分位点

x <- matrix(rnorm(200), 20, 10)
apply(x, 1, quantile, probs = c(0.25, 0.75))

结果得到一个矩阵, 矩阵的每列就是对 apply 函数对 x 的每行执行 quantile 函数的结果
```

2. 计算三维数列的均值矩阵

```{r}
a <- array(rnorm(2*2*10), c(2, 2, 10)); a

apply(a, c(1, 2), mean)
          [, 1]      [, 2]
[1, ] 0.1373085 -0.5651948
[2, ] 0.2586311  0.5268097

rowMeans(a, dims = 2)
          [, 1]      [, 2]
[1, ] 0.1373085 -0.5651948
[2, ] 0.2586311  0.5268097
```

mapply

mapply 是 lapply 和 sapply 的多变量版本，即把一个函数并行的应用到一组不同的参数上，可以把好几个列表作为参数

```
str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

- FUN: 待应用的函数
- ... 包含需要应用的所有参数，参数数量是可变的参数的数量，至少应该大于等于传递给 mapply 的列表的数量
- MoreArgs: 传递给函数的其他参数列表
- SIMPLIFY: 指出结果是否需要简化

冗余的形式

```
list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
```

```
[1] 4
```

```
rep(1:4, 4:1)
[1] 1 1 1 1 2 2 2 3 3 4
```

这里可以应用 mapply, 把 1:4 和 4:1 中对应的 4 对元素分别作为两个参数传递给 rep 函数. 每一对参数生成一组数据, 作为结果列表中的一个元素. 与直接使用 rep(1:4, 4:1)的结果类似, 只是以列表的形式返回结果.

```
mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4
```

mapply 函数的应用: 将函数向量化, 如下随机生成一些正态噪音

```
noise <- function(n, mean, sd) {rnorm(n, mean, sd)}

noise(5, 1, 2)
[1]  0.7529086  1.2947858 -1.4819324  5.0084869 -0.5439591

set.seed(1); noise(1:5, 1:5, 2)
set.seed(1); rnorm(5, 1:5, 2)  #二者等价
[1] -0.2529076  2.3672866  1.3287428  7.1905616  5.6590155

list(noise(1, 1, 2), noise(2, 2, 2), noise(3, 3, 2))
[[1]]
[1] 2.014136

[[2]]
[1] 1.815193 1.693501

[[3]]
[1] 1.831636 3.951789 3.879710
```

使用 mapply 向量化,

```
mapply(noise, 1:5, 1:5, 2)
[[1]]
[1] 3.97239

[[2]]
[1] -2.239116  2.043825

[[3]]
[1] 1.812876 2.444062 2.420144

[[4]]
[1] 2.5418642 4.5870830 2.5655389 0.8696945

[[5]]
[1] 7.169760 4.476527 4.599566 4.911308 2.691305
```

这就是将不同的向量参数的函数瞬间向量化的方法

tapply 分组汇总

分组汇总, 只能对向量作, 通过一个因子把向量分成若干组, 再对每一组执行函数.

tapply 可以把一个函数应用在向量的子集上, 即通过另一个变量或者对象对向量中各元素分组, 对于每一组计算一个概要统计量

```
str(tapply)
function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

- X 是一个数值类型或者其他类型的向量
- INDEX, 和第一个向量长度相同的因子或一列因子(会被强制转化为因子), 指明第一个向量中的各元素属于哪一组
- FUN 想要应用的函数
- ...传递进该函数的其他变量
- simplify 表明你是否想要简化结果, 默认为 TRUE

取每组均值

```{r}
x <- c(rnorm(10), runif(10, 0, 10), rbinom(10, 5, 0.5)); x
[1] -0.04422838 -0.42532983 -1.63893147  0.06526015 -0.77870021 -0.41627648
[7] -0.35819274 -1.68124046  0.45756422  0.20284981  6.56099161  3.57506583
```

```
[13]  2.34025480  9.16430336  0.30555000  4.39074042  9.73690321  3.99140808
[19]  9.19769866  0.98763876  2.00000000  3.00000000  1.00000000  4.00000000
[25]  2.00000000  5.00000000  2.00000000  4.00000000  0.00000000  2.00000000
```

##使用 gl 函数创建因子变量，使用因子对数据进行分组. 如 3 个班级的各科成绩，以 3 个班的班号 1，2，3 对成绩进行分组. f 中的前 10 个 1 表示 x 中的前 10 个数，f 中的 10 个 2 表示 x 中的中间 10 个数，如此类推.

```
f <- gl(3, 10); f
[1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
Levels: 1 2 3

# 分组求均值
tapply(x, f, mean, na.rm = T)
          1           2           3
-0.4617225  5.0250555  2.5000000

#分组求最大最小值.
tapply(x, f, range)
$`1`
[1] -1.6812405  0.4575642

$`2`
[1] 0.305550 9.736903

$`3`
[1] 0 5

#使用 quantile 求 0%分位数和 100%分位数，也可以得到最大值和最小值
tapply(x, f, quantile, probs = 0:1)
$`1`
         0%         100%
-1.6812405  0.4575642

$`2`
       0%       100%
0.305550 9.736903

$`3`
  0% 100%
   0    5
```

split

tapply 根据分组应用函数，然后再整合输出, split 不是循环函数，可以和 sapply, lapply 一起使用
tapply 是分组汇总, split 只做分组，不做汇总，没有函数. split 的对象可以是向量也可以是数据框，而 tapply 则只能用于对向量进行分组汇总.
str(split)
function (x, f, drop = FALSE, ...)

- x 是一个向量或者数据框
- f 是一个因子变量，或者强制转换为因子变量，或者是一列分组变量，用来被指定分组的水平
- drop 指明是否丢掉空的因子等级

split 的作用就是根据因子对向量或数据框进行分组.

```{r}
x <- c(rnorm(10), runif(10), rnorm(10, 1)); f <- gl(3, 10)
split(x, f)
$`1`
 [1] -0.4097968 -0.8914867 -0.7327580  1.1131105  0.6825504  1.7518865
 [7]  1.0165373  0.1400945 -0.6452599  0.2132057

$`2`
 [1] 0.60853986 0.02086326 0.38279984 0.38161991 0.06740730 0.28922640
 [7] 0.76474291 0.80158476 0.20403125 0.91786007

$`3`
 [1] -0.2693335  1.0336735  0.5820239  1.9490491  0.5772901  2.1173191
 [7]  1.2894542  1.2004781  0.7737755  2.1931774
```

split 总会返回一个列表，因子中每一组都是列表中的一个元素. 对列表操作可以用 lapply() 或 sapply()，所以 split 总是和 lapply()或 sapply 嵌套使用的.

lappy 和 sapply 是对列表中的每个元素进行循环操作, apply 是对每行或每列进行循环操作, tapply 是对每一个分类进行循环操作.

x <- c(rnorm(10), runif(10), rnorm(10, 1)); f <- gl(3, 10)

#常常将 split 放在 lapply 里
lapply(split(x, f), mean)
$`1`
[1] 0.3388931

```
$`2`
[1] 0.3772923

$`3`
[1] 0.5866041
```

```
sapply(split(x, f), range)
                1           2           3
[1, ] -0.6974966 0.07993552 -0.5624307
[2, ]  1.9988306 0.74185807  2.2298231
```

split 可以用来分解类型更加复杂的对象

```
library(datasets); head(airquality, 3)
```

可以对数据框进行分组

```
#对 airquality 根据月份进行分组. 得到一个列表, 列表中的每一个元素都是一个数据框. 如果要使用 lapply 和 sapply, 就要使用可以对数据框进行操作的函数
s <- split(airquality, airquality$Month); s
```

```
sapply(s, colMeans, na.rm = T)
              5         6         7         8         9
Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
Wind     11.62258  10.26667   8.941935   8.793548  10.18000
Temp     65.54839  79.10000  83.903226  83.967742  76.90000
Month     5.00000   6.00000   7.000000   8.000000   9.00000
Day      16.00000  15.50000  16.000000  16.000000  15.50000
```

```
# 分组后在 sapply 中使用匿名函数, s 作为匿名函数 function(x)的参数传递给匿名函数, 取出来 Ozone, Solar.R, Wind 这 3 列求列均值.
sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
              5         6         7         8         9
Ozone        NA        NA        NA        NA        NA
Solar.R      NA 190.16667 216.483871        NA 167.4333
Wind   11.62258  10.26667   8.941935  8.793548   10.1800
```

```
sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")], na.rm = T))
              5         6         7         8         9
Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
Wind     11.62258  10.26667   8.941935   8.793548  10.18000
```

lapply 使用匿名函数.

```
lapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
$`5`
    Ozone   Solar.R      Wind
       NA        NA  11.62258

$`6`
     Ozone     Solar.R      Wind
        NA  190.16667  10.26667

$`7`
      Ozone       Solar.R      Wind
         NA  216.483871    8.941935

$`8`
    Ozone   Solar.R      Wind
       NA        NA  8.793548

$`9`
    Ozone   Solar.R      Wind
       NA  167.4333  10.1800
```

可见 split 可以根据因子的水平来分解任意的对象，再对分解后列表中的元素应用任意类型的函数

aggegate 能够对数据框进行分组汇总，可以看成是 tapply 的升级版函数
dplyr 包中有一套自己数据清洗的逻辑和方法，其中也有分组汇总的工具.

split 根据多个因子分组

可能有多个因子，想要观察这些因子产生的不同水平的组合.

```
f1 <- gl(2, 5); f2 <- gl(5, 2); f1; f2
[1] 1 1 1 1 1 2 2 2 2 2
Levels: 1 2
[1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5

##把所有的水平组合起来，即求 2 个水平的交集，得出来理论上的等级
interaction(f1, f2)
[1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 2.4 1.5 2.5
```

```
split(x, interaction(f1, f2))
$`1.1`
[1] -0.6264538  0.1836433

$`2.1`
numeric(0)

$`1.2`
[1] -0.8356286  1.5952808

$`2.2`
numeric(0)

$`1.3`
[1] 0.3295078

$`2.3`
[1] -0.8204684

$`1.4`
numeric(0)

$`2.4`
[1] 0.4874291 0.7383247

$`1.5`
numeric(0)

$`2.5`
[1]  0.5757814 -0.3053884


# 使用 str 来查看 R 对象的内部结构
str(split(x, list(f1, f2, drop = TRUE)))
List of 10
 $ 1.1.TRUE: num [1:2] -0.626 0.184
 $ 2.1.TRUE: num(0)
 $ 1.2.TRUE: num [1:2] -0.836 1.595
 $ 2.2.TRUE: num(0)
 $ 1.3.TRUE: num 0.33
 $ 2.3.TRUE: num -0.82
 $ 1.4.TRUE: num(0)
 $ 2.4.TRUE: num [1:2] 0.487 0.738
 $ 1.5.TRUE: num(0)
```

```
 $ 2.5.TRUE: num [1:2] 0.576 -0.305
```

interaction 可能会创建出空的等级，当使用 split 函数时，并不一定同时使用 interaction 函数，可以直接传递一个包含两个因子的列表给 split，程序会自动调用 interction

```
x <- rnorm(10)
f1 <- gl(2, 5); f2 <- gl(5, 2); f1; f2
[1] 1 1 1 1 1 2 2 2 2 2
Levels: 1 2
[1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5

str(split(x, list(f1, f2)))
List of 10
 $ 1.1: num [1:2] -1.586 -0.258
 $ 2.1: num(0)
 $ 1.2: num [1:2] -0.198 1.039
 $ 2.2: num(0)
 $ 1.3: num 0.667
 $ 2.3: num -1.31
 $ 1.4: num(0)
 $ 2.4: num [1:2] -1.14 -0.749
 $ 1.5: num(0)
 $ 2.5: num [1:2] -2.11 -1.17
```

split 实际应用举例

例如有如下成绩的数据

成绩	班级	性别
10	1	男
20	1	男
30	2	男
40	2	女
50	3	女
60	3	女

可以根据班级和性别组合起来的因子对成绩进行分组. 班级*性别，班级有 3 个班，性别有 2 个，理论上会产生 6 个等级: 1.男, 1.女, 2.男, 2.女, 3.男, 3.女，但是 1.女和 3.男没有对应的数据，称为空等级. 实际中只能得到 4 个有效的因子，在程序处理时默认是按照理论上的因子进行分组的. 分组的结果为

1.男 10, 20
1.女 NA

2.男 30
2.女 40
3.男 NA
3.女 50, 60

可以在 split 中使用 drop = TRUE 来去除空的等级

```
x <- rnorm(10)

f1 <- gl(2, 5); f2 <- gl(5, 2); f1; f2
 [1] 1 1 1 1 1 2 2 2 2 2
Levels: 1 2
 [1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5

str(split(x, list(f1, f2), drop = T))
List of 6
 $ 1.1: num [1:2] 0.804 -0.408
 $ 1.2: num [1:2] 0.753 0.797
 $ 1.3: num -0.759
 $ 2.3: num -0.461
 $ 2.4: num [1:2] 1.904 -0.349
 $ 2.5: num [1:2] 0.219 1.145
```

Practices 4

- 对于如下 list, 分别使用 lapply 和 sapply 函数计算每个元素的均值, 以及每个元素 1/4 分位数, 中位数, 四分之三分位数

```{r, eval = F}
x <- list(a = 1:10, b = exp(-4:4), logic = c(T, F, F, T))
x
$a
 [1]  1  2  3  4  5  6  7  8  9 10

$b
[1]  0.01831564  0.04978707  0.13533528  0.36787944  1.00000000  2.71828
183
[7]  7.38905610 20.08553692 54.59815003

$logic
[1]  TRUE FALSE FALSE  TRUE
```

```
#使用匿名函数，把均值和分位数在一起计算
lapply(x, function(x) {c(mean = mean(x), quantile(x, probs = 1:3/4))})
$a
mean   25%   50%   75%
5.50  3.25  5.50  7.75

$b
       mean          25%          50%          75%
9.5958158  0.1353353  1.0000000  7.3890561

$logic
mean   25%   50%   75%
 0.5   0.0   0.5   1.0

sapply(x, function(x) {c(mean = mean(x), quantile(x, probs = 1:3/4))})
             a             b  logic
mean  5.50  9.5958158     0.5
25%   3.25  0.1353353     0.0
50%   5.50  1.0000000     0.5
75%   7.75  7.3890561     1.0
```

- 对于系统自带的鸢尾花 iris 数据，计算除了最后一列以外的所有行均值和列均值，使用 apply 函数

```
apply(iris[, -5], 1, mean, na.rm = T)
apply(iris[, -5], 2, mean, na.rm = T)
```

- 使用 mapply 函数生成如下对象，注意结合 unlist 函数

```
unlist(mapply(rep, letters[1:4], c(4:2, 5)))
```

- 对于系统自带的鸢尾花 iris 数据，计算每个种类(种类是第五列)萼片长度(第一列)的均值，计算均值的时候指明去掉 NA 值，使用 tapply 函数

```
tapply(iris$Sepal.Length, iris$Species, mean, na.rm = T)
```

- 同样是鸢尾花数据，计算每个种类所有尺寸(前四列)的均值，结果得到一个数据框

```
t1 <- as.data.frame(sapply(split(iris[, -5], iris$Species), colMeans, na.rm = T))
class(t1) #为数据框
```

回顾

对象　　vector　list　matrix　array　　data.frame　　factor
子集　　[]　[[]]　数值　　逻辑　　名字　---最常用
常用函数　　时间　　字符　　运算
控制结构　　if...else, for, while
函数定义　　function　匿名函数
循环函数　　lapply, sapply, apply, tapply, split

强制转换，自建循环逻辑-->向量化运算
工作中就是不断的学习新的函数，完成各种各样的功能.
其它函数，清洗函数，作图函数，建模函数.

常见数据清理

清洗　作图　建模

排序-sort

sort 的对象只能是向量

sort {base}　　R Documentation

Sorting or Ordering Vectors

Description
Sort (or *order*) a vector or factor (partially) into ascending or descending order. For ordering along more than one variable, e.g., for sorting data frames, see order.

Usage
```
sort(x, decreasing = FALSE, ...)

## Default S3 method:
sort(x, decreasing = FALSE, na.last = NA, ...)

sort.int(x, partial = NULL, na.last = NA, decreasing = FALSE,
         method = c("auto", "shell", "quick", "radix"), index.return = FALSE)
```

Arguments

x	for sort an R object with a class or a numeric, complex, character or logical vector. For sort.int, a numeric, complex, character or logical vector, or a factor.

decreasing	logical. Should the sort be increasing or decreasing? For the"radix" method, this can be a vector of length equal to the number of arguments in .... For the other methods, it must be length one. Not available for partial sorting.
...	arguments to be passed to or from methods or (for the default methods and objects without a class) to sort.int.
na.last	for controlling the treatment of NAs. If TRUE, missing values in the data are put last; if FALSE, they are put first; if NA, they are removed.
partial	NULL or a vector of indices for partial sorting.
method	character string specifying the algorithm used. Not available for partial sorting. Can be abbreviated.
index.return	logical indicating if the ordering index vector should be returned as well. Supported by method = = "radix" for anyna.last mode and data type, and the other methods whenna.last = NA (the default) and fully sorting non-factors.

```
x <- data.frame(var1 = sample(2:7), var2 = sample(6:11), var3 = sample(13:1
8)); x
  var1 var2 var3
1    7    7   18
2    6    9   14
3    4   11   16
4    3    8   15
5    2    6   13
6    5   10   17

set.seed(1)
x1 <- x[sample(1:5), ]; x1$var2[c(1, 3)] <- NA; x1
  var1 var2 var3
2    6   NA   14
5    2    6   13
4    3   NA   15
3    4   11   16
1    7    7   18

sort(x1$var1)
[1] 2 3 4 6 7
sort(x1$var1, decreasing = TRUE)
[1] 7 6 4 3 2
sort(x1$var2)
[1]  6  7 11
sort(x1$var2, na.last = TRUE)
[1]  6  7 11 NA NA
```

视频演示

```
x <- data.frame(v1 = sample(2:7), v2 = sample(6:11), v3 = sample(13:18)); x
  v1 v2 v3
1  2  8 16
2  4  7 15
3  6  6 18
4  5  9 13
5  7 11 17
6  3 10 14
x[5, 1] <- 2; x[1, 2] <- NA; x
  v1 v2 v3
1  2 NA 16
2  4  7 15
3  6  6 18
4  5  9 13
5  2 11 17
6  3 10 14

#以第一列 v1 为标准进行排序
sort(x$v1)
[1] 2 2 3 4 5 6
sort(x$v2)
[1]  6  7  9 10 11
```

默认会自动把 NA 值去掉，可以使用 na.last = TRUE 来把 NA 值放在末尾, na.last = FALSE 把 NA 值放在开头. na.last - NA 把 NA 值去掉.

```
sort(x$v2, na.last = TRUE)
[1]  6  7  9 10 11 NA

#默认以升序排列，decreasing = T 则为降序排列
sort(x$v2, decreasing = T)
[1] 11 10  9  7  6
```

sort 只能对单个向量排序，如果要根据一个向量对列表排序，就要使用 order

排序-order

根据一个向量对列表排序，返回元素从小到大排列的位置索引, order 排序在实际中应用更多.

```
order(c(10, 1, 7, 3))
[1] 2 4 3 1
```

视频演示

```
x <- data.frame(v1 = sample(2:7), v2 = sample(6:11), v3 = sample(13:18)); x
  v1 v2 v3
1  5  8 14
2  7  9 13
3  4 11 17
4  3  7 16
5  6 10 18
6  2  6 15
x[5, 1] <- 2; x[1, 2] <- NA; x
  v1 v2 v3
1  5 NA 14
2  7  9 13
3  4 11 17
4  3  7 16
5  2 10 18
6  2  6 15

#返回元素在原数据中从小到大的位置信息
order(x$v1)
[1] 5 6 4 3 1 2
order(x$v3)
[1] 2 1 6 4 3 5

#根据第一列中的元素对数据框进行升序排序, 如果第一列中的元素有重复值, 就根据原数据中重复值的排序方式进行排序.
x[order(x$v1), ]
  v1 v2 v3
5  2 10 18
6  2  6 15
4  3  7 16
3  4 11 17
1  5 NA 14
2  7  9 13

#根据第一列对 x 中的元素进行排序, 如果第 1 列中的数据有重复值, 重复值的行再根据第 2 列的数据进行排序
x[order(x$v1, x$v2), ]
  v1 v2 v3
6  2  6 15
5  2 10 18
4  3  7 16
```

```
3   4  11  17
1   5  NA  14
2   7   9  13
```

#order 默认是升序排序，想要降序排列要使用 decreasing = T，但只能对所有排序的列同时降序排序，不能对一列升序而另一列降序．想要对不同的列使用不同的排序方法，就要使用 arrange

```
x[order(x$v1, x$v2, decreasing = T), ]
  v1 v2 v3
2  7  9 13
1  5 NA 14
3  4 11 17
4  3  7 16
5  2 10 18
6  2  6 15
```

课件代码

```
x <- data.frame(var1 = sample(2:7), var2 = sample(6:11), var3 = sample(13:18)); x
  var1 var2 var3
1    2    8   17
2    3   11   16
3    6   10   13
4    7    7   15
5    4    6   18
6    5    9   14
set.seed(1)
x1 <- x[sample(1:5), ]; x1$var2[c(1, 3)] <- NA
x1
  var1 var2 var3
2    3   NA   16
5    4    6   18
4    7   NA   15
3    6   10   13
1    2    8   17

order(x1$var1)
[1] 5 1 2 4 3

x1[order(x1$var1), ]
  var1 var2 var3
1    2    8   17
2    3   NA   16
```

```
5      4      6     18
3      6     10     13
4      7     NA     15
```

```
x1[order(x1$var1,  x1$var3),  ]
   var1  var2  var3
1      2      8     17
2      3     NA     16
5      4      6     18
3      6     10     13
4      7     NA     15
```

排序-arrange

安装 plyr, dplyr 包

课件代码

```{r}
library(plyr)

x <- data.frame(var1 = sample(2:7), var2 = sample(6:11), var3 = sample(13:18));  x
   var1  var2  var3
1      4      9     15
2      2     10     18
3      6      7     13
4      7     11     16
5      5      6     14
6      3      8     17
x[3,  1]  <-  4;  x
   var1  var2  var3
1      4      9     15
2      2     10     18
3      4      7     13
4      7     11     16
5      5      6     14
6      3      8     17
arrange(x,  var1,  var3)
   var1  var2  var3
1      2     10     18
```

```
2     3     8    17
3     4     7    13
4     4     9    15
5     5     6    14
6     7    11    16
arrange(x, desc(var1), var3)
  var1 var2 var3
1     7    11    16
2     5     6    14
3     4     7    13
4     4     9    15
5     3     8    17
6     2    10    18
```

视频演示

```{r}
library(plyr)
x <- data.frame(v1 = sample(2:7), v2 = sample(6:11), v3 = sample(13:18)); x
  v1 v2 v3
1  2  9 14
2  7 10 16
3  4  8 18
4  5 11 13
5  3  7 17
6  6  6 15
x[5, 1] <- 2; x[1, 2] <- NA; x
  v1 v2 v3
1  2 NA 14
2  7 10 16
3  4  8 18
4  5 11 13
5  2  7 17
6  6  6 15

arrange(x, v1)
  v1 v2 v3
1  2 NA 14
2  2  7 17
3  4  8 18
4  5 11 13
5  6  6 15
6  7 10 16
```

```
arrange(x, v1, v2)
  v1 v2 v3
1  2  7 17
2  2 NA 14
3  4  8 18
4  5 11 13
5  6  6 15
6  7 10 16

#对某一列降序，其它列升序
arrange(x, desc(v1), v2)
  v1 v2 v3
1  7 10 16
2  6  6 15
3  5 11 13
4  4  8 18
5  2  7 17
6  2 NA 14
```

匹配数据 match

match {base} R Documentation

Value Matching

Description

match returns a vector of the positions of (first) matches of its first argument in its second.

%in% is a more intuitive interface as a binary operator, which returns a logical vector indicating if there is a match or not for its left operand.

Usage

match(x, table, nomatch = NA_integer_, incomparables = NULL)

x %in% table

Arguments

x	vector or NULL: the values to be matched. Long vectors are supported.
table	vector or NULL: the values to be matched against. Long vectors are not supported.
nomatch	the value to be returned in the case when no match is found. Note that it is coerced to integer.

incomparables	a vector of values that cannot be matched. Any value in xmatching a value in this vector is assigned the nomatchvalue. For historical reasons, FALSE is equivalent to NULL.

Details

%in% is currently defined as

"%in%" <- function(x, table) match(x, table, nomatch = 0) 0

- match 返回 x 中的各个元素在 table 中的位置，没有的话返回 nomatch 指定的值，默认为 NA

```
match(c(1, 3, 5, 9, 21), -10:10)
[1] 12 14 16 20 NA
c(1, 3, 5, 9, 21) %in% -10:10
[1]  TRUE  TRUE  TRUE  TRUE FALSE
```

which, which.max, which.min

- which 输入逻辑向量，返回真值所在的位置
- which.max 和 which.min 在向量中分别寻找最大值和最小值，返回对应的值所在的位置

```
which(c(TRUE, FALSE, TRUE, NA, FALSE, FALSE, TRUE))
[1] 1 3 7
which.max(c(1:4, 11, 0:5))
[1] 5
which.min(c(1:4, 11, 0:5))
[1] 6
```

学函数，注意输入的对象和参数. 第二要注意函数能实现的功能.

切割数据，分箱

分箱，离散化，即把连续变量转化为分类变量. 年龄，从 20-60，转化为分类变量，从 20-25, 25-35, 35-50, 50-60

split 是按因子进行分组.

分箱，又称为离散化.

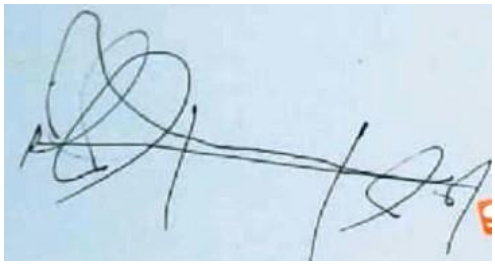
等宽分箱，根据数据分布的区间进行等宽分类，不改变数据大致的分布情况.

有一组年龄的数据 age <- c(22, 23, 25, 30, 45, 50)

想要把 age 分成 2 组, 20-35 是一组. 35-55 是一组，把符合的元素的数据都替换为组名，这样就把连续型变量转换为分类型变量了. 得到的结果是 age <- c("20-35", "20-35", "20-35", "20-35", "35-55", "35-55")，又可以把组名替换为 c(1, 1, 1, 1, 2, 2).

22-50 这些数据可以看成是右偏的分布.

c(1, 2, 3, 4, 5, 6, 7, 100)是一个右偏的分布, 共 9 个数据, 想要按照区间长度把数据平均分成 3 类, [1-34), [34, 67), [67, 100), 这种分类方法就是等宽分箱方法, 它不改变数据大致的分布情况, 得到的是 c(1, 1, 1, 1, 1, 1, 1, 3). 如果分布本身不均匀的话, 使用等宽分箱方法得到的数据每组中的数据也可能是极度不均匀的.



等深分箱, 根据数据的数量进行分类, c(1, 2, 3, 4, 5, 6, 7, 100), 8 个数据平均分成 3 份, 每份中的数据个数相等, [1-4), [4-7), [7-100)

也可以自定义分箱的方法.

分箱一定会造成信息的丢失. 箱子越少, 信息的丢失越多.
在业务需要减化处理的逻辑时使用分箱. 如不同年龄段信用卡的消费情况. 需要分箱时一定要考虑如何在满足业务需要的同时尽量减少信息的丢失.

等宽分箱函数: cut

cut {base}　　R Documentation
Convert Numeric to Factor

Description

cut divides the range of x into intervals and codes the values in x according to which interval they fall. The leftmost interval corresponds to level one, the next leftmost to level two and so on.

Usage

```
cut(x, ...)

## Default S3 method:
cut(x, breaks, labels = NULL,
    include.lowest = FALSE, right = TRUE, dig.lab = 3,
    ordered_result = FALSE, ...)
```

Arguments

x
a numeric vector which is to be converted to a factor by cutting.
breaks
either a numeric vector of two or more unique cut points or a single number (greater than or equal to 2) giving the number of intervals into which x is to be cut.
labels
labels for the levels of the resulting category. By default, labels are constructed using "(a, b]" interval notation. If labels = FALSE, simple integer codes are returned instead of a factor.
include.lowest
logical, indicating if an ‘x[i]’ equal to the lowest (or highest, for right = FALSE) ‘breaks’ value should be included.
right
logical, indicating if the intervals should be closed on the right (and open on the left) or vice versa.
dig.lab
integer which is used when labels are not given. It determines the number of digits used in formatting the break numbers.
ordered_result
logical: should the result be an ordered factor?
...
further arguments passed to or from other methods.

```{r}
# 根据数据分布的区间等分成 5 个箱子
cut(1:20, 5)
 [1] (0.981, 4.8] (0.981, 4.8] (0.981, 4.8] (0.981, 4.8] (4.8, 8.6]   (4.8, 8.6]
 [7] (4.8, 8.6]   (4.8, 8.6]   (8.6, 12.4]  (8.6, 12.4]  (8.6, 12.4]  (8.6, 12.4]
[13] (12.4, 16.2] (12.4, 16.2] (12.4, 16.2] (12.4, 16.2] (16.2, 20]   (16.2, 20]
[19] (16.2, 20]   (16.2, 20]
Levels: (0.981, 4.8] (4.8, 8.6] (8.6, 12.4] (12.4, 16.2] (16.2, 20]
```

箱子是左不包含右包含, 所以要从 0.981 开始. 3.8 左右的宽度. 把原来的数据使用箱子的区间这个字符型的变量来代替

```
#分箱的结果是因子.
x <- cut(1:20, 5); class(x)
[1] "factor"

# 自定义的标签来代替分组的区间.
x <- cut(1:20, 5, labels = c("g1", "g2", "g3", "g4", "g5")); x
 [1] g1 g1 g1 g1 g2 g2 g2 g2 g3 g3 g3 g3 g4 g4 g4 g4 g5 g5 g5 g5
Levels: g1 g2 g3 g4 g5
```

#程序自动分箱

```
cut(c(1:7, 100), 3)
[1] (0.901,34] (0.901,34] (0.901,34] (0.901,34] (0.901,34] (0.901,34]
[7] (0.901,34] (67,100]
Levels: (0.901,34] (34,67] (67,100]
```

#自定义分箱，定义出分割点. 要给出开头 0.9，结尾 100，和分割点 4, 30. 因为区间是左不包含右包含的，所以开头要略小于数据的最小值.

```
cut(c(1:7, 100), c(0.9, 4, 30, 100))
[1] (0.9,4]   (0.9,4]   (0.9,4]   (0.9,4]   (4,30]    (4,30]    (4,30]
[8] (30,100]
Levels: (0.9,4] (4,30] (30,100]
```

#可用在用户的等级划分

```
cut(c(1:7, 100), c(0.9, 4, 30, 100), labels = c("小学", "中学", "大学"))
[1] 小学 小学 小学 小学 中学 中学 中学 大学
Levels: 小学 中学 大学
```

```
cut(1:20, c(0.9, 5, 9, 13, 17, 20))
 [1] <NA>   (1,5]   (1,5]   (1,5]   (1,5]   (5,9]   (5,9]   (5,9]
 [9] (5,9]   (9,13]  (9,13]  (9,13]  (9,13]  (13,17] (13,17] (13,17]
[17] (13,17] (17,20] (17,20] (17,20]
Levels: (1,5] (5,9] (9,13] (13,17] (17,20]
```

```
cut(1:20, c(0.9, 5, 9, 13, 17, 20))
 [1] (0.9,5] (0.9,5] (0.9,5] (0.9,5] (0.9,5] (5,9]   (5,9]   (5,9]
 [9] (5,9]   (9,13]  (9,13]  (9,13]  (9,13]  (13,17] (13,17] (13,17]
[17] (13,17] (17,20] (17,20] (17,20]
Levels: (0.9,5] (5,9] (9,13] (13,17] (17,20]
```

```

## ## 等深分箱-Hmisc 包

?Hmisc::cut2

**cut2 {Hmisc} R Documentation**

Cut a Numeric Variable into Intervals

**Description**

Function like cut but left endpoints are inclusive and labels are of the form [lower, upper), except that last interval is [lower, upper]. If cuts are given, will by default make sure that cuts include entire range of x. Also, if cuts are not given, will cut x into quantile groups (g given) or groups with a
```

given minimum number of observations (m). Whereas cut creates a category object, cut2 creates a factor object.

Usage

cut2(x, cuts, m, g, levels.mean, digits, minmax = TRUE, oneval = TRUE, onlycuts = FALSE)

Arguments

x	numeric vector to classify into intervals
cuts	cut points
m	desired minimum number of observations in a group. The algorithm does not guarantee that all groups will have at least mobservations.
g	number of quantile groups
levels.mean	set to TRUE to make the new categorical vector have levels attribute that is the group means of x instead of interval endpoint labels
digits	number of significant digits to use in constructing levels. Default is 3 (5 if levels.mean=TRUE)
minmax	if cuts is specified but min(x)<min(cuts) ormax(x)>max(cuts), augments cuts to include min and max x
oneval	if an interval contains only one unique value, the interval will be labeled with the formatted version of that value instead of the interval endpoints, unless oneval=FALSE
onlycuts	set to TRUE to only return the vector of computed cuts. This consists of the interior values plus outer ranges.

视频演示

#cut2 的区间是左闭右开的区间，最后一个是闭合的区间. 这样就不用指定区间的最大值与最小值了.

```
cut2(c(1:7, 100), g = 3)
[1] [1,  4) [1,  4) [1,  4) [4,  7) [4,  7) [4,  7) [7,100] [7,100]
Levels: [1,  4) [4,  7) [7,100]
# 自定义分割点分箱
cut2(c(1:7, 100), cuts = c(1, 4, 30, 100))
[1] [  1,  4) [  1,  4) [  1,  4) [  4, 30) [  4, 30) [  4, 30) [  4, 30)
[8] [ 30,100]
Levels: [  1,  4) [  4, 30) [ 30,100]
```

cut 中指定分割点和指定分割成的箱子数目的参数都是 breaks, cut2 中对应的参数分别是 g 和 cuts

```{r}
Hmisc::cut2(1:20, g = 5)
 [1] [ 1, 5) [ 1, 5) [ 1, 5) [ 1, 5) [ 5, 9) [ 5, 9) [ 5, 9) [ 5, 9)
 [9] [ 9,13) [ 9,13) [ 9,13) [ 9,13) [13,17) [13,17) [13,17) [13,17)
[17] [17,20] [17,20] [17,20] [17,20]
Levels: [ 1, 5) [ 5, 9) [ 9,13) [13,17) [17,20]

Hmisc::cut2(1:20, cuts = c(1, 5, 9, 13, 17, 20))
 [1] [ 1, 5) [ 1, 5) [ 1, 5) [ 1, 5) [ 5, 9) [ 5, 9) [ 5, 9) [ 5, 9)
 [9] [ 9,13) [ 9,13) [ 9,13) [ 9,13) [13,17) [13,17) [13,17) [13,17)
[17] [17,20] [17,20] [17,20] [17,20]
Levels: [ 1, 5) [ 5, 9) [ 9,13) [13,17) [17,20]
```

重塑数据

重塑数据-熔化 reshape2::melt

?reshape2::melt

melt {reshape2} R Documentation

Convert an object into a molten data frame.

Description

This the generic melt function. See the following functions for the details about different data structures:

Usage

melt(data, ..., na.rm = FALSE, value.name = "value")

Arguments

data	Data set to melt
...	further arguments passed to or from other methods.
na.rm	Should NA values be removed from the data set? This will convert explicit missings to implicit missings.
value.name	name of variable used to store values

Details

melt.data.frame for data.frames

melt.array for arrays, matrices and tables

melt.list for lists

视频演示

```{r}
library(reshape2)

x <- data.frame(id = 1:4, class = rep(c("1 班", "2 班"), 2), math = c(93, 23, 88, 83), eng = c(32, 89, 87, 34), chn = c(69, 78, 98, 70)); x
  id class math eng chn
1  1   1 班   93  32  69
2  2   2 班   23  89  78
3  3   1 班   88  87  98
4  4   2 班   83  34  70

# melt 把成绩分散开，每一行是一个人一科的成绩. 相当于表内转置，把 math, eng, chn 三列的成绩放在一列中. "id"和"class"是标签，是用来进行重复的列, "math", "eng", "chn"是用来转置的列. 无论是多少列，每一列都会生成的两列，分别是原来列的变量名和变量值. 相当于把一个宽表变成一个长表.
x1 <- melt(x, id.vars = c("id", "class"), measure.vars = c("math", "eng", "chn"))
x1
   id class variable value
1   1   1 班     math    93
2   2   2 班     math    23
3   3   1 班     math    88
4   4   2 班     math    83
5   1   1 班      eng    32
6   2   2 班      eng    89
7   3   1 班      eng    87
8   4   2 班      eng    34
9   1   1 班      chn    69
10  2   2 班      chn    78
11  3   1 班      chn    98
12  4   2 班      chn    70

# 结果中进行重复的列只是在 id.vars 中定义的列，没有定义的列不会出现.
x1 <- melt(x, id.vars = c("id"), measure.vars = c("math", "eng", "chn"))
x1
   id variable value
1   1     math    93
2   2     math    23
3   3     math    88
4   4     math    83
5   1      eng    32
```

6 2 eng 89
7 3 eng 87
8 4 eng 34
9 1 chn 69
10 2 chn 78
11 3 chn 98
12 4 chn 70

```

## 重塑数据-铸造 reshape2::dcast

**cast {reshape2}  R Documentation**

Cast functions Cast a molten data frame into an array or data frame.

**Description**

Use acast or dcast depending on whether you want vector/matrix/array output or data frame output. Data frames can have at most two dimensions.

**Usage**

```
dcast(data, formula, fun.aggregate = NULL, ..., margins = NULL,
  subset = NULL, fill = NULL, drop = TRUE,
  value.var = guess_value(data))

acast(data, formula, fun.aggregate = NULL, ..., margins = NULL,
  subset = NULL, fill = NULL, drop = TRUE,
  value.var = guess_value(data))
```

**Arguments**

| | |
|---|---|
| **data** | molten data frame, see melt. |
| **formula** | casting formula, see details for specifics. |
| **fun.aggregate** | aggregation function needed if variables do not identify a single observation for each output cell. Defaults to length (with a message) if needed but not specified. |
| **...** | further arguments are passed to aggregating function |
| **margins** | vector of variable names (can include "grand\_col" and "grand\_row") to compute margins for, or TRUE to compute all margins . Any variables that can not be margined over will be silently dropped. |
```

subset	quoted expression used to subset data prior to reshaping, e.g. subset = .(variable=="length").
fill	value with which to fill in structural missings, defaults to value from applyingfun.aggregate to 0 length vector
drop	should missing combinations dropped or kept?
value.var	name of column which stores values, seeguess_value for default strategies to figure this out.

视频演示

```
library(reshape2)
x <- data.frame(id = 1:4, class = rep(c("1 班", "2 班"), 2), math = c(93, 23, 88, 83), eng = c(32, 89, 87, 34), chn = c(69, 78, 98, 70)); x
```

```
  id class math eng chn
1  1   1 班   93  32  69
2  2   2 班   23  89  78
3  3   1 班   88  87  98
4  4   2 班   83  34  70
```

```
x1 <- melt(x, id.vars = c("id", "class"), measure.vars = c("math", "eng", "chn"))
```

```
   id class variable value
1   1   1 班     math    93
2   2   2 班     math    23
3   3   1 班     math    88
4   4   2 班     math    83
5   1   1 班      eng    32
6   2   2 班      eng    89
7   3   1 班      eng    87
8   4   2 班      eng    34
9   1   1 班      chn    69
10  2   2 班      chn    78
11  3   1 班      chn    98
12  4   2 班      chn    70
```

```
# ~一般表示一个公式, ~左边一般是 y, 右边是 x.
#相当于从 x1 回到原来的 x. id 是一个分类变量, 以 id 来进行分类. 把 varibale 中的数值合并起来作为列标签. 取 value 中对应的值作为合并起来的列标签的值.
dcast(x1, id~variable, value.var = "value")
```

```
  id math eng chn
1  1   93  32  69
```

2 2 23 89 78
3 3 88 87 98
4 4 83 34 70

#以 class 来进行分类，以 variable 中的值合并起来为列标签，如果不定义对 value 中的值进行操作的函数，默认对 value 中的值进行个数的统计，统计的结果作为列标签的值. 使用 dcast 时，对于 value.var 中定义的列 value，如果 value 列中的每一类值对应的 variable 的值是唯一，如上面的 id，使用 dcast 得到的结果就是 melt 的原表，长表变宽表，此时 dcast 与 melt 互为逆操作. 如果 value 列中每一类值对应的 variable 的值是不唯一的，如上面的 class，使用 dcast 时就可以加上函数对 class 中不同的值对应的 value 值进行操作了，默认以 class 中的值对 value 中的值进行分类汇总.

```
dcast(x1, class~variable, value.var = "value")
```

```
  class math eng chn
1  1 班    2   2   2
2  2 班    2   2   2
```

dcast(x1, class~variable, value.var = "value", mean) #对一班二班的成绩求均值. 可以看成是分组汇总的一个方法. 可以看成是组内转置.

在某些分类汇总时，或归纳数据，或衍伸变量时，使用 melt 和 dcast 能够很简便的得到结果. melt 宽表变长表. 把每一列竖着排下来生成一个长表. dcast 可以看成是 melt 的逆操作，把长表变宽表

对数据集使用 melt 和 dcast

使用 head 查看 mtcars 数据集. head 返回数据或函数的第 1 部分. Returns the first or last parts of a vector, matrix, table, data frame or function. Since head() and tail() are generic functions, they may also have been extended to other classes.

```
head(mtcars, 4)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21	6	160	110	3.9	2.62	16.46	0	1	4	4
Mazda RX4 Wag	21	6	160	110	3.9	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1

从 mtcars 数据集中取出来所有的列名，赋值到 mtcars 中新的一列 carname 中.

```
mtcars$carname <- rownames(mtcars)
```

```
head(mtcars, 4)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	carname

Mazda RX4	21	6	160	110	3.9	2.62	16.46	0	1	4	4	Mazda RX4
Mazda RX4 Wag	21	6	160	110	3.9	2.875	17.02	0	1	4	4	Mazda RX4 Wag
Datsun 710	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1	Datsun 710
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1	Hornet 4 Drive

```
library(reshape2)
# carnames, gear, cyl 是用来进行重复的列, 把 mpg, hp 两列的值进行转置.
carMelt <- melt(mtcars, id.vars = c("carname", "gear", "cyl"),
                        measure.vars = c("mpg", "hp"))

mtcars$carname <- NULL

head(carMelt, n = 3)
          carname gear cyl variable value
1       Mazda RX4    4   6      mpg  21.0
2 Mazda RX4 Wag    4   6      mpg  21.0
3     Datsun 710    4   4      mpg  22.8

tail(carMelt, n = 3)
            carname gear cyl variable value
62   Ferrari Dino    5   6       hp   175
63 Maserati Bora    5   8       hp   335
64      Volvo 142E    4   4       hp   109


dcast(carMelt, cyl ~ variable, value.var = "value")
Aggregation function missing: defaulting to length
  cyl mpg hp
1   4  11 11
2   6   7  7
3   8  14 14

dcast(carMelt, cyl ~ variable, mean)
  cyl      mpg       hp
1   4 26.66364  82.63636
2   6 19.74286 122.28571
3   8 15.10000 209.21429

dcast(carMelt, cyl+gear ~ variable, mean)
  cyl gear    mpg      hp
1   4    3 21.500 97.0000
2   4    4 26.925 76.0000
3   4    5 28.200 102.0000
4   6    3 19.750 107.5000
```

```
5     6       4  19.750  116.5000
6     6       5  19.700  175.0000
7     8       3  15.050  194.1667
8     8       5  15.400  299.5000
```

##使用 SQL 语句汇总

在熟悉 SQL 语句并且数据量不大的情况下，可以使用常见的 SQL 语句汇总数据

- 加载包，读取文件

```
setwd("D:/18th/R_Data")
library(sqldf)
# sale <- read.csv("./data/sale.csv")
sale <- read.csv("sale.csv")
```

```
sqldf("select * from sale")
      year market   sale  profit
1   2010       东  33912    2641
2   2010       南  32246    2699
3   2010       西  34792    2574
4   2010       北  31884    2673
5   2011       东  31651    2437
6   2011       南  30572    2853
7   2011       西  34175    2877
8   2011       北  30555    2749
9   2012       东  31619    2106
10  2012       南  32443    3124
11  2012       西  32103    2593
12  2012       北  31744    2962
```

- 按照年份汇总

```
sqldf("select year, sum(sale) as sum_sale, sum(profit)
        as sum_profit from sale group by year")
```

```
   year sum_sale sum_profit
1  2010   132834      10587
2  2011   126953      10916
3  2012   127909      10785
```

- 汇总某列；定位低于全国销售平均水平地域

```
sqldf("select distinct market from sale")
  market
1     东
2     南
3     西
4     北
sqldf("select avg(sale) as all_avg_sale from sale")
  all_avg_sale
1        32308
sqldf("select market, avg(sale) as mavg_sale from sale
       group by market having mavg_sale<32308 order by mavg_sale")
  market mavg_sale
1     北  31394.33
2     南  31753.67
```

- where 引导的条件字节分组前处理，放在 group by 之前
- having 引导的条件字节一般跟在 group by 后面，相当于分组后的再处理.

```
sqldf("select market, sum(sale) as sum_sale from sale
      where year= 2011 group by market having sum_sale>3000
+        order by sum_sale desc")
  market sum_sale
1     西    34175
2     东    31651
3     南    30572
4     北    30555
sqldf("select market, avg(sale) as mavg_sale from sale
      group by market
      having mavg_sale<(select avg(sale) as all_avg_sale from sale)
      order by mavg_sale")
  market mavg_sale
1     北  31394.33
2     南  31753.67
```

纵向连接表，求交集，并集，差集

视频演示

```
library(sqldf)

x <- data.frame(id = 1:3, x1 = c("a", "b", "a")); x
```

```
  id x1
1  1  a
2  2  b
3  3  a
y <- data.frame(id = c(3, 4), y1 = c("a", "b")); y
  id y1
1  3  a
2  4  b

#求并集, 会自动把重复的行去掉
sqldf("select * from x union select * from y")
  id x1
1  1  a
2  2  b
3  3  a
4  4  b

- 求并集中如果使用 union all 则相当于 rbind, 区别在于 rbind 要求名字必须相同
sqldf("select * from x union all select * from y")
  id x1
1  1  a
2  2  b
3  3  a
4  3  a
5  4  b


#求交集.
sqldf("select * from x intersect select * from y")
  id x1
1  3  a

#求差集
sqldf("select * from x except select * from y")
  id x1
1  1  a
2  2  b

#使用 rbind 时 x, y 的列名必须要一致. 并且只能简单的放在一起. 不能求并交差集
colnames(y)[2] <- "x1"
rbind(x, y)
  id x1
1  1  a
2  2  b
```

3 3 a
4 3 a
5 4 b

课件代码

```
x <- data.frame(id = c(1, 1, 1, 2, 3, 4, 6), x1 = c("a", "a", "b", "c", "v", "e", "g"))
y <- data.frame(id = c(1, 2, 3, 3, 5), y1 = c("x", "y", "z", "v", "w"))

sqldf("select * from x union select * from y")
sqldf("select * from x INTERSECT select * from y")
sqldf("select * from x EXCEPT select * from y")
```

横向连接表

交叉连接

- 交叉连接 cross join, 笛卡尔乘积

```{r}
x <- data.frame(id = c(1, 2, 3), x1 = c("a", "b", "c")); x
  id x1
1  1  a
2  2  b
3  3  c
y <- data.frame(id = c(4, 3), y1 = c("d", "e")); y
  id y1
1  4  d
2  3  e

sqldf("select * from x, y")
  id x1 id y1
1  1  a  4  d
2  1  a  3  e
3  2  b  4  d
4  2  b  3  e
5  3  c  4  d
6  3  c  3  e
```

视频演示

```{r}
x <- data.frame(id = 1:3, x1 = c("a", "b", "a")); x
  id x1
1  1  a
2  2  b
3  3  a
y <- data.frame(id = c(3, 4), y1 = c("a", "b")); y
  id y1
1  3  a
2  4  b
sqldf("select * from x, y")
  id x1 id y1
1  1  a  3  a
2  1  a  4  b
3  2  b  3  a
4  2  b  4  b
5  3  a  3  a
6  3  a  4  b

```

内连接

merge {base} R Documentation

Merge Two Data Frames

Description

Merge two data frames by common columns or row names, or do other versions of database *join* operations.

Usage

```
merge(x, y, ...)

## Default S3 method:
merge(x, y, ...)

## S3 method for class 'data.frame'
merge(x, y, by = intersect(names(x), names(y)),
      by.x = by, by.y = by, all = FALSE, all.x = all, all.y = all,
      sort = TRUE, suffixes = c(".x",".y"),
      incomparables = NULL, ...)
```

Arguments

x, y	data frames, or objects to be coerced to one.

by, by.x, by.y	specifications of the columns used for merging. See 'Details'.
all	logical; all = L is shorthand for all.x = L and all.y = L, where L is either TRUE orFALSE.
all.x	logical; if TRUE, then extra rows will be added to the output, one for each row in x that has no matching row in y. These rows will have NAs in those columns that are usually filled with values from y. The default is FALSE, so that only rows with data from both x and y are included in the output.
all.y	logical; analogous to all.x.
sort	logical. Should the result be sorted on the bycolumns?
suffixes	a character vector of length 2 specifying the suffixes to be used for making unique the names of columns in the result which are not used for merging (appearing in by etc).
incomparables	values which cannot be matched. See match. This is intended to be used for merging on one column, so these are incomparable values of that column.
...	arguments to be passed to or from methods.

注意: 尽量使用 merge 和 dplyr 中的函数, 而不要使用 sql 语句, 有时候会出问题. sql 中的 right join 和 full join 在 sqldf 中不能使用.

在列名相同的情况下

```
x <- data.frame(id = 1:3, x1 = c("a", "b", "a")); x
  id x1
1  1  a
2  2  b
3  3  a

y <- data.frame(id = c(3, 4), y1 = c("a", "b")); y
  id y1
1  3  a
2  4  b

#按照 id 进行匹配, all = F 只查找相同名字的.
merge(x, y, by = "id", all = F)
  id x1 y1
1  3  a  a

library(dplyr, warn.conflicts = F)

# 使用 dplyr 包中的 inner_join 来实现
inner_join(x, y, by = "id")
  id x1 y1
1  3  a  a
```

```
sqldf("select * from x as a inner join y as b on a.id = b.id")
  id x1 id y1
1  3  a  3  a

sqldf("select * from x inner join y on x.id = y.id")
  id x1 id y1
1  3  a  3  a
```

merge() 前两个参数是要连接的表. 如果参数一致, 则 by = "id", 如果参数不一致, 则 by.x = "", by.y = "". all 参数, 全连接时 all = T, 内连接时 all = F, 左连接时是 all.x = T, 右连接时是 all.y = T. 默认为 all = F, 即默认是内连接

```
merge(x, y, by = "id", all = F)
  id x1 y1
1  3  a  a
merge(x, y, by = "id")
  id x1 y1
1  3  a  a
```

dplyr 中的 inner_join, left_join, right_join, full_join.

```
library(dplyr, warn.conflicts = F)

inner_join(x, y , by = "id")
  id x1 y1
1  3  a  a
```

```

#在列名不同的情况下

```
x <- data.frame(id = 1:3, x1 = c("a", "b", "a")); x
  id x1
1 1 a
2 2 b
3 3 a
y <- data.frame(id = c(3, 4), y1 = c("a", "b")); y
  id y1
1 3 a
2 4 b

colnames(x) <- c("id1", "x1")
colnames(y) <- c("id2", "y1")
```
```

```
sqldf("select * from x inner join y on x.id1 = y.id2")
  id1 x1 id2 y1
1   3  a   3  a

merge(x, y, by.x = "id1", by.y = "id2", all = F)
  id1 x1 y1
1   3  a  a

dplyr::inner_join(x, y, by = c("id1" = "id2"))
  id1 x1 y1
1   3  a  a

#dplyr 包中的 inner_join 来实现
library(dplyr, warn.conflicts = F)
inner_join(x, y, by = c("id1" = "id2"))
  id1 x1 y1
1   3  a  a
```

左连接

列名相同的情况

```
x <- data.frame(id = 1:3, x1 = c("a", "b", "a")); x
  id x1
1  1  a
2  2  b
3  3  a
y <- data.frame(id = c(3, 4), y1 = c("a", "b")); y
  id y1
1  3  a
2  4  b
merge(x, y, by = "id", all.x = TRUE)
  id x1   y1
1  1  a <NA>
2  2  b <NA>
3  3  a    a
left_join(x, y, by = "id")
  id x1   y1
1  1  a <NA>
2  2  b <NA>
3  3  a    a
sqldf("select * from x as a left join y as b on a.id = b.id")
```

```
  id x1 id   y1
1  1  a NA <NA>
2  2  b NA <NA>
3  3  a  3    a
```

#对于列名不同的情况下

```
x <- data.frame(id = 1:3, x1 = c("a", "b", "a")); x
  id x1
1  1  a
2  2  b
3  3  a
y <- data.frame(id = c(3, 4), y1 = c("a", "b")); y
  id y1
1  3  a
2  4  b

colnames(x) <- c("id1", "x1")
colnames(y) <- c("id2", "y1")

sqldf("select * from x left join y on x.id1 = y.id2")
  id1 x1 id2   y1
1   1  a  NA <NA>
2   2  b  NA <NA>
3   3  a   3    a

merge(x, y, by.x = "id1", by.y = "id2", all.x = T)
  id1 x1   y1
1   1  a <NA>
2   2  b <NA>
3   3  a    a

dplyr::left_join(x, y, by = c("id1" = "id2"))
  id1 x1   y1
1   1  a <NA>
2   2  b <NA>
3   3  a    a
```

右连接

```
x <- data.frame(id = 1:3, x1 = c("a", "b", "a")); x
  id x1
1  1  a
2  2  b
3  3  a
```

```
y <- data.frame(id = c(3, 4), y1 = c("a", "b")); y
  id y1
1  3  a
2  4  b

> merge(x, y, by = "id", all.y = TRUE)
  id   x1 y1
1  3    a  a
2  4 <NA>  b

> right_join(x, y, by = "id")
  id   x1 y1
1  3    a  a
2  4 <NA>  b
```

sqldf("select * from x as a right join y as b on a.id = b.id") 目前不支持右连接和全连接

全连接

```{r}
> x <- data.frame(id = 1:3, x1 = c("a", "b", "a")); x
  id x1
1  1  a
2  2  b
3  3  a
> y <- data.frame(id = c(3, 4), y1 = c("a", "b")); y
  id y1
1  3  a
2  4  b
> merge(x, y, by = "id", all = TRUE)
  id   x1   y1
1  1    a <NA>
2  2    b <NA>
3  3    a    a
4  4 <NA>    b
> full_join(x, y, by = "id")
  id   x1   y1
1  1    a <NA>
2  2    b <NA>
3  3    a    a
4  4 <NA>    b
```

##读入 Excel 文件

可以安装 xlsx 包, 但要配合 jre 才能使用
readxl 也是读 excel 文件的包

也有连 mysql 数据库的包, 读取 json, xml 文件的包, api 接口的包,

数据清洗时 %in%用的较多, 连续数据的离散化用的较多,

读取 excel 的最好方式是将文件另存为 csv 格式. 当 excel 文件过大时, 可安装常用包并且使用对应函数, 常用包有 xlsx 和 openxlsx 包等, xlsx 包安装较复杂, 大型文件容易报错, 且 xlsx 包需要预装 java, 但是 xlsx 包相对可选项比较多. openxlsx 是比较新的包, 读取大型文件相对比较稳定.
安装 xlsx 之前需要有两前提包，一个是 rJava，一个是 xlsxjars.

```
install.packages("xlsx", dependencies = TRUE, lib="C:/R/R-3.4.4/library")
```

安装以后顺序加载就可以了，初学者偶尔会遇到一个问题，就是关于 rJava，首先需要在本机安装 java 中的 jre，同时将 jre 的路径地址放入到环境变量 path 中。
我的电脑右键-属性-高级系统设置-环境变量-系统变量中找到 path 编辑添加即可

```{r}
library(openxlsx, quietly = T)
# data <- read.xlsx("./data/hsb2.xlsx", sheet = 1)
#也可以指定 sheet 的名称, 但只能使用英文, 不能使用中文
data <- read.xlsx("hsb2.xlsx", sheet = 1)

View(data)

detach("package:openxlsx")

library(xlsx)
data <- read.xlsx("hsb2.xlsx", sheetIndex = 1, colIndex = 2:10, rowIndex = 1:10)
```