R 语言简明教程

第二课、数据类型、结构和基础操作（第一部分）

张金龙
jinlongzhang01@gmail.com

2022-1-23

《R 语言简明教程》内容安排（I）

- 第一课、R 语言及其编程环境
- **第二课、数据类型、结构和基础操作**
- 第三课、用 graphics 包绘图
- 第四课、用 ggplot2 包绘图
- 第五课、tidydata 以及 tidyverse 系列程序包的使用
- 第六课、基础统计方法的实现
- 第七课、编写 R 函数
- 第八课、Rmarkdown 以及可重复性数据分析
- 第九课、git 与版本控制

《R 语言简明教程》内容安排（II）

- 第十课、R 程序包的编写、检查与发表
- 第十一课、生物多样性分析常用程序包
- 第十二课、空间数据处理和地图绘制
- 第十三课、系统发育研究的常用程序包
- 第十四课、课程总结

内容提要

1

R 中的对象

R 中的对象

R 中一切皆对象 `object`。

数字、字符、因子、函数、运算符，各种要处理的数据类型，在 R 中均为对象。

四则运算

- +: 加
- -: 减
- *: 乘
- /: 除

```
(2+5)*(15-4.1) # 注意用 () 确定运算的优先次序
```

```
## [1] 76.3
```

赋值

- `<-`: 尽量使用，将右侧的值传递给左侧的对象。
- `=`: 尽量在设定函数参数时使用
- `->`: 尽量不要用

  赋值符号前后都要有一个英文空格

为变量取名

1. 严格区分大小写
2. 不要用中文
3. 只用英文字母、数字、下划线
4. 每个单词有明确的意义，不要用 `a`、`b`、`x`、`y` 等抽象的名字，以增加代码的可读性
5. `i,j,k,m,n` 等，默认为循环中使用的局域变量（只在循环中起作用）。

2

元素的基本类型

元素的基本类型

- **数值型 Numeric**: 用户不必区分整数和小数，R 会自动判断，如 `12`, `5000`, `3.1415926`
- **字符串型 Character**: 如 `"Guangzhou"`
- **因子型 Factor**: 如某种实验处理，增加光照，不增加光照，就属于因子类型，用 `factor()` 或者 `as.factor()` 生成
- **逻辑型 Logical**: 如导师是否同意报销差旅费，是 `TRUE` 或否 `FALSE`，简写为 T/F。
- **复数型 Complex**: 较少用到。

怎样查询数据的类型?

```
class()
```

其他类型 NaN, NA, NULL

主要用于编写函数:

- NA: 缺失值, is.na()
- NULL: 不存在, is.null()
- NaN: Not a number 非数值
- Inf: 正无穷

```
dat1 <- NA
is.na(dat1)
```

```
## [1] TRUE
```

```
is.null(dat1)
```

```
## [1] FALSE
```

3

基本数据类型

基本数据类型

以上基本元素按照一定规律组合，就构成了 R 中的基本数据类型:

- **向量** `vector`: 元素按照一定顺序组合，就构成了向量。注意，单独一个元素也是向量。
- **矩阵** `matrix`：同一种类型数据按照行和列排列。
- **数组** `array`: 同一种类型数据，按照二维或者多个维度排列，高维数组在 R 中偶尔会用到。
- **数据框** `data.frame`: 当多个类型不同，但是长度相同的向量按照列合并，类似于常见的数据记录表。
- **列表** `list`: 向量、矩阵、数据框均可放入列表（list）中。

4

向量 vector

向量

```
aaa <- c("China", "Japan", "Korea", "Vietnam")
is.character(aaa)
```

```
## [1] TRUE
```

```
bbb <- c(1,2,3,4,5,6,6,7,7,7,8)
is.numeric(bbb)
```

```
## [1] TRUE
```

```
class(bbb)
```

```
## [1] "numeric"
```

向量的基本操作 I

向量中的元素必须是相同类型的，如果不相同，一般会自动转换为字符串类型或者逻辑型。

- `c()`: 生成向量，合并向量
- `length()`: 向量的元素个数，或称为长度
- `paste()`: 合并字符串向量
- `rep()`: 某一个或者几个元素重复指定的次数
- `cbind()`: 将等长的 `vector` 合并成 `data.frame`
- `head()`: 显示前几个值
- `tail()`: 显示后几个值

向量的基本操作 I 举例

```
aaa <- c("a", "b", "c", 1)
length(aaa)
```

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

```
paste(aaa, collapse = "-")
```

```
## [1] "a-b-c-1"
```

```
rep(aaa, c(1,2,1,3))
```

```
## [1] "a" "b" "b" "c" "1" "1" "1"
```

数值向量的基本操作 II

- `mean()`: 算数平均值
- `min()`: 最小值
- `max()`: 最大值
- `sd()`: 求标准差
- `var()`: 求方差
- `sum()`: 求和
- `table()`: 统计相同值出现的次数
- `summary()`: 求数值向量的最大值、最小值、中位数等
- `boxplot()`: 绘制箱线图
- `hist()`: 绘制频度直方图

向量的基本操作举例

```
ddd <- c(-1, 1,1,2,4,4,4,5,5,2,1,6,10)
boxplot(ddd)
```

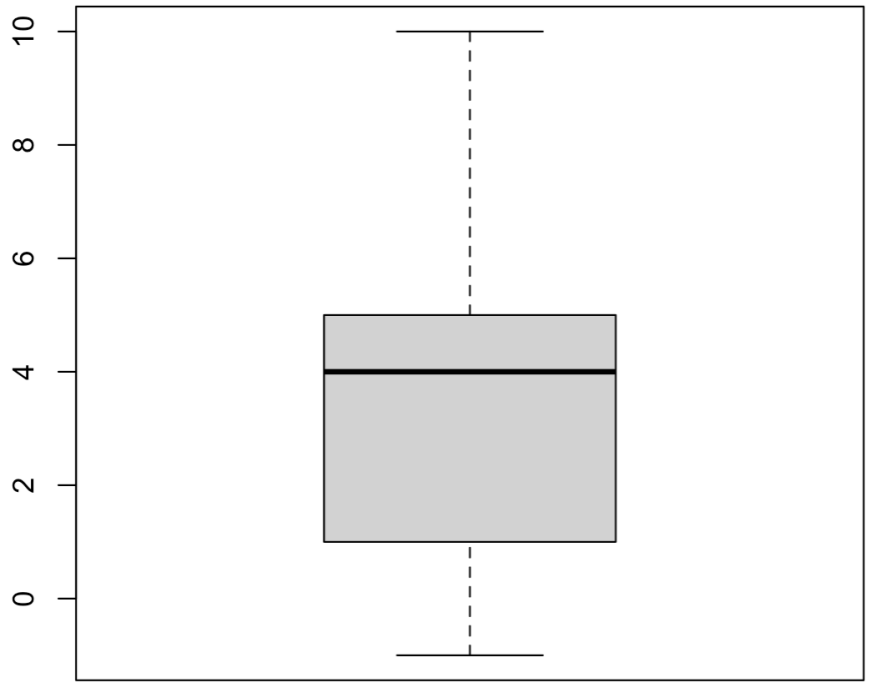

图 1: boxplot 输出结果

向量的基本操作举例

R 是**向量化**的语言。

```
mean(ddd)
```

```
## [1] 3.384615
```

```
median(ddd)
```

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

```
ddd * 2 # 每一个都乘以 2
```

```
##  [1] -2  2  2  4  8  8  8 10 10  4  2 12 20
```

```
sum(ddd)
```

```
## [1] 44
```

5

矩阵 matrix

矩阵 (matrix) 与数组 (array)

- 如果类型相同的一系列元素，按照一定的行和列排列，就称为矩阵。
- 如果除了行和列，如果还有更高的维度，就称为数组，其实向量和矩阵都是数组的特殊情况。

矩阵举例

新西兰奥克兰的 Maunga Whau 火山数据

```
data(volcano)
class(volcano)
image(volcano)
```

矩阵举例

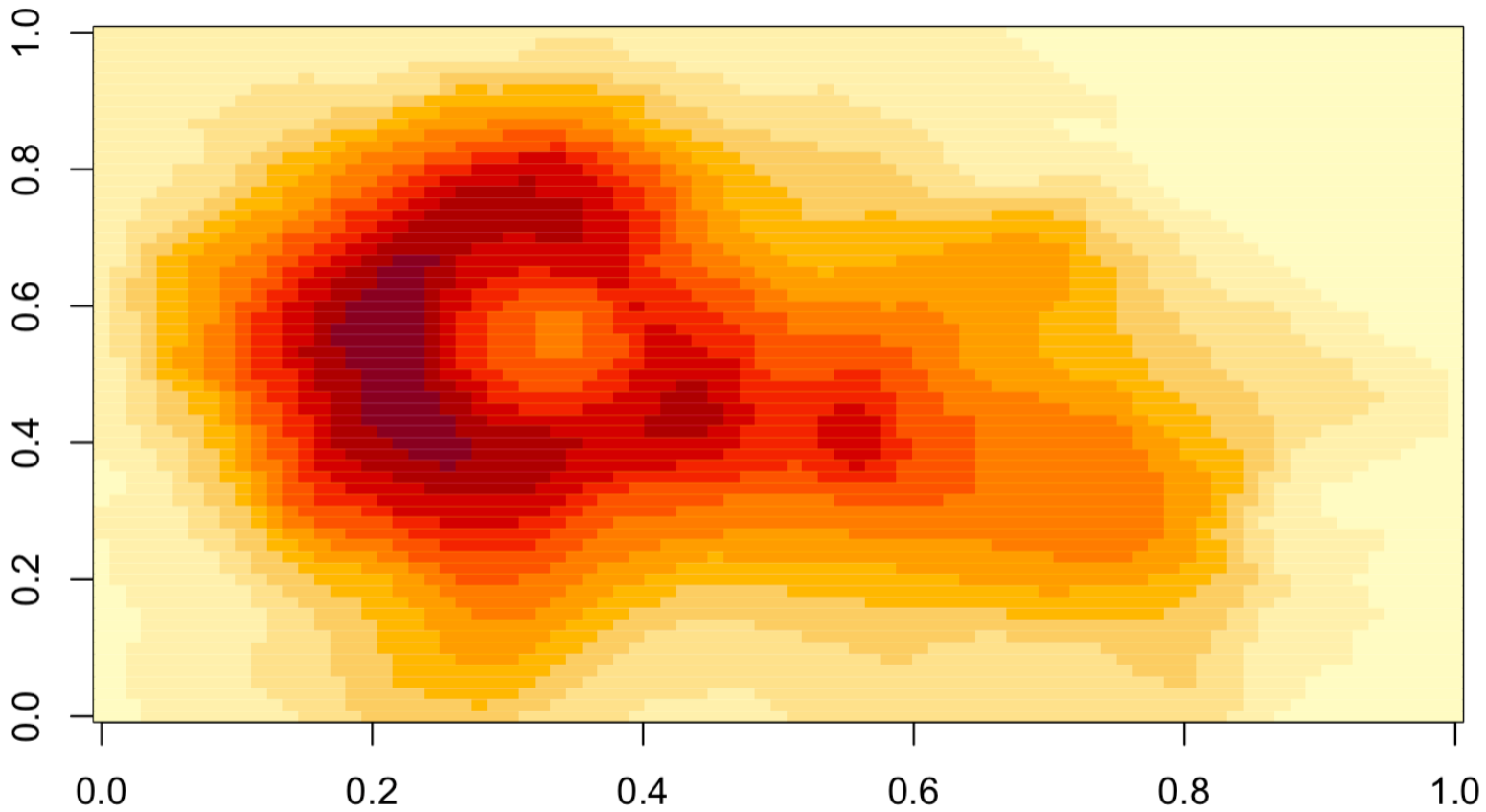

图 2: 新西兰奥克兰 Maunga Whau 火山数据

volcano 数据以矩阵的格式保存

类型为数值型 (numeric)

```
volcano[1:5,2:6]  # 数据集的 1 到 5 行, 2 到 6 列
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]  100  101  101  101  101
## [2,]  101  102  102  102  102
## [3,]  102  103  103  103  103
## [4,]  103  104  104  104  104
## [5,]  104  105  105  105  105
```

如何生成矩阵?

1. 从数据文件 (csv, txt, Excel 等) 读取
2. 从向量生成:

定义 matrix 常用有两种方式:

- `dim()` 查看 matrix 的行列数，也可以用来给向量指定行列，生成 matrix
- `matrix()` 生成 matrix

生成矩阵举例

```
dat <- 1:20
dim(dat) <- c(4, 5)
dat
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
```

先顺序填充第一列，再填充第二列，以此类推

生成矩阵的第三种方式

孔乙己：“茴香豆的茴字有四样写法，你知道么？”

生成 2 行 5 列的矩阵：

```
x <- 1:10
attr(x,"dim") <- c(2, 5)
```

查看行数和列数

查看行、列的数目

- nrow() 查看行数
- ncol() 查看列数

```
dim(dat)
```

```
## [1] 4 5
```

```
nrow(dat)
```

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

```
ncol(dat)
```

```
## [1] 5
```

转置：行列的转换

```
t(dat) # 转置，行转换为列，列转换为行
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
## [5,]   17   18   19   20
```

转置后，先顺序填充第一行，再填充第二行，以此类推

6

数据框 data.frame

数据框

矩阵的数据只能是同一类型。

但如果现有某班的成绩表，各列的数据类型不同怎么办?

解决方案：数据框

生成数据框

- `data.frame()`: 生成 data.frame
- `as.data.frame()`: 转换为 data.frame
- `cbind()`: 将某一向量添加到现有的数据框中

```
library(vegan)
data(dune.env)
head(dune.env, 3)
```

```
##    A1 Moisture Management       Use Manure
## 1 2.8        1         SF Haypastu      4
## 2 3.5        1         BF Haypastu      2
## 3 4.3        2         SF Haypastu      4
```

数据框的属性和操作

数据框的很多函数与矩阵“通用”。

- `nrow()`：查看行数
- `ncol()`：查看列数
- `rownames()`：行名
- `colnames()`：列名提取某一列，用 `$` 加列名即可
- `cbind()`: 为已有数据框增加一列
- `rbind()`: 将两个列名相同的数据框连接

数据框的属性和操作

注意：**在数据框中，各列的长度（也就是元素的个数）一定相等**，缺失值对应的是 NA，不是 NULL。

```
dim(dune.env) # 数据框有多少行，多少列
```

```
## [1] 20  5
```

```
colnames(dune.env) # 列名
```

```
## [1] "A1"          "Moisture"    "Management" "Use"          "Ma
```

```
dune.env$Moisture # 提取名为 Moisture 的列
```

```
##  [1] 1 1 2 2 1 1 1 5 4 2 1 4 5 5 5 5 2 1 5 5
## Levels: 1 < 2 < 4 < 5
```

生成数据框

```
records <- data.frame(c("John","Fred","George",
                        "Tonny","Daisy","Jane"),
                      c(100,85,60,72,90,95),
                      c("M", "M", "M", "M",
                        "F", "F"))
colnames(records) <- c("name", "score", "gender")
head(records, 3)
```

```
##     name score gender
## 1   John   100      M
## 2   Fred    85      M
## 3 George    60      M
```

数据框的一些基本操作

```
records[,1]    # 提取第一列
```

```
## [1] "John"   "Fred"   "George" "Tonny"  "Daisy"  "Jane"
```

```
records[1,]    # 提取第一行
```

```
##   name score gender
## 1 John   100      M
```

```
records[2:3,] # 提取第二行到第三行
```

```
##      name score gender
## 2    Fred    85      M
## 3 George    60      M
```

数据框的一些基本操作

```
rownames(records) # 行名
```

```
## [1] "1" "2" "3" "4" "5" "6"
```

```
colnames(records) # 列名
```

```
## [1] "name"   "score"  "gender"
```

提取数据框的列

```
records[,1]       # 第一列
```

```
## [1] "John"   "Fred"   "George" "Tonny"  "Daisy"  "Jane"
```

```
records[,2:3]     # 第 2 到 3 列
```

```
##   score gender
## 1   100      M
## 2    85      M
## 3    60      M
## 4    72      M
## 5    90      F
## 6    95      F
```

提取数据框的列

```
records$name      # 名为 name 的列
```

```
## [1] "John"   "Fred"   "George" "Tonny"  "Daisy"  "Jane"
```

```
records[,"name"] # 名为 name 的列
```

```
## [1] "John"   "Fred"   "George" "Tonny"  "Daisy"  "Jane"
```

选取部分数据

成绩表中女生的成绩

```
subset(records, gender == "F")
```

```
##    name score gender
## 5 Daisy    90      F
## 6  Jane    95      F
```

90 分及以上的部分

```
subset(records, score >= 90)
```

```
##    name score gender
## 1  John   100      M
## 5 Daisy    90      F
## 6  Jane    95      F
```

更改数据框

增加一个名为 attend 的列

```
cbind(records, attend = rep(TRUE, nrow(records)))
```

```
##      name score gender attend
## 1    John   100      M   TRUE
## 2    Fred    85      M   TRUE
## 3  George    60      M   TRUE
## 4   Tonny    72      M   TRUE
## 5   Daisy    90      F   TRUE
## 6    Jane    95      F   TRUE
```

更改数据框

John 成绩改为 NA

```
records[records$name == "John","score"] <- NA
records
```

```
##      name score gender
## 1    John    NA      M
## 2    Fred    85      M
## 3  George    60      M
## 4   Tonny    72      M
## 5   Daisy    90      F
## 6    Jane    95      F
```

7

列表 list

列表

此列表非购物清单的列表。

列表的组件可以是向量、矩阵、数组、数据框以及**列表**的任意一种。

R 的列表中，使用 `$` 提取某一个已经命名的组件。

生成列表

将 dune.env 数据中的 A1 列和 Use 列提取出来，放入 list

```
library(vegan)
data(dune.env)
ddd <- list(dune.env$A1, dune.env$Use)
```

生成列表，同时命名各元素

```
ddd2 <- list(A1 = dune.env$A1, Use = dune.env$Use)
```

列表内容的提取

提取列表的第一个元素

```
ddd2[[1]]
```

```
##  [1]  2.8  3.5  4.3  4.2  6.3  4.3  2.8  4.2  3.7  3.3  3.5
## [16]  5.7  4.0  4.6  3.7  3.5
```

列表内容的提取

从列表提取名为 A1 的元素

```
ddd2$A1
```

```
##  [1]  2.8  3.5  4.3  4.2  6.3  4.3  2.8  4.2  3.7  3.3  3.5
## [16]  5.7  4.0  4.6  3.7  3.5
```

8

下标、逻辑运算和数据提取

下标 index 与按照下标提取 subscripting

主要是用 `[]` 来提取

下标是一个向量，用来提取基本数据类型中的某一些符合条件的值。

- 对于向量 `vector`, 使用 `[]`
- 对于矩阵 `matrix`, 使用 `[,]`, 注意逗号，逗号前表示按行，逗号后表示按列
- 对于数据框 `data.frame`, 使用 `[,]`，同 matrix
- 对于列表 `list`，使用 `[[]]`，里面放数字

下标 index 与按照下标提取 subscripting

可以放置整数型或者逻辑型两种向量:

1. 放置 1，2，3，4 等，表示要提取元素的位置
2. 放置 TRUE、FALSE 等，表示是否提取对应的元素

```
letters[1:6]
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

```
ind <- letters %in% c("a", "c", "f")
letters[ind]
```

```
## [1] "a" "c" "f"
```

比较

比较数值的大小，结果为逻辑向量 (logical)。

- `==` 判断数值是否相等，是为 `TRUE`, 否为 `FALSE`，简写为 T,F
- `>`: 判断左侧值是否大于右侧值
- `<`: 判断左侧值是否小于右侧值
- `>=`：判断左侧值是否大于或等于右侧值
- `<=`：判断左侧值是否小于或等于右侧值
- `!=`：判断左侧值是否不等于右侧值
- `%in%`: 判断某些向量的值是否出现在另一个向量中

9

数据读取和保存

数据读取

```
read.csv()      # 读取 csv 文件
openxlsx::readxlsx() # 读取 xlsx 文件
readxl::read_xlsx()  # 读取 xlsx 文件
readr::read_csv()  # 读取 csv 文件, 读取为 tibble
load() # 载入 Rdata 文件
```

数据的保存

```
write.csv()                # 保存为 csv 文件，逗号间隔
readr::write_csv()         # 保存为 csv 文件，逗号间隔
openxlsx::write.xlsx()     # 保存为 Excel 文件
write.table()              # 保存为 txt 文件，制表符间隔
save.image()               # 保存为 Rdata 文件
```

内容回顾

1. R 是一种统计绘图的计算机语言
2. 工作空间、程序包、数据读取
3. 元素类型（数值型、字符型、逻辑型、因子型、复数型）
4. 数据的基本结构类型（向量、数组、矩阵、数据框、列表）

内容提要

1. 对象的下标（index）与按照下标提取（subscripting）
2. 逻辑向量以及逻辑运算
3. 条件控制 `if/else`
4. `for` 和 `while` 循环
5. `apply` 家族
6. 运算符的优先次序
7. 字符串的匹配、拆分和替换
8. 数据的去重复、排序、合并
9. R 工作空间的保存

1

对象的下标（index）与按照下标提取（subscripting）

对象的下标（index）

下标一般是数值或者逻辑类型的向量，用来提取基本数据类型中符合条件的值。

- 对于向量 `vector`, 下标放入 `[]`
- 对于矩阵 `matrix`, 下标放入 `[,]`。放在逗号前表示按行，逗号后表示按列
- 对于数据框 `data.frame`, 使用 `[,]`
- 对于列表 `list`，下标放入 `[[]]` 中

按照下标提取（subscripting）

1. 放置 1，2，3，4 等，表示要提取元素的位置。如果放数字，都是从 1 开始。
2. 放置 TRUE、FALSE 等，表示是否提取对应的元素，TRUE 则提取，FALSE 不提取

```
aaa <- c("a", "C", "F")
ind <-  aaa %in% letters
ind
```

```
## [1]  TRUE FALSE FALSE
```

```
aaa[ind] # 找出哪个字母在 letters 向量中
```

```
## [1] "a"
```

2

逻辑向量以及逻辑运算

生成逻辑向量：比较

比较数值的大小，结果为**逻辑向量**，由 TRUE 或者 FALSE 组成。

- == 判断数值是否相等，是为 TRUE, 否为 FALSE
- >: 判断左侧值是否大于右侧值
- <: 判断左侧值是否小于右侧值
- >= : 判断左侧值是否大于或等于右侧值
- <= : 判断左侧值是否小于或等于右侧值
- != : 判断左侧值是否不等于右侧值
- %in%: 判断某些向量的值是否出现在另一个向量中

举例

```
1 > 0
```

```
## [1] TRUE
```

```
1 == 1
```

```
## [1] TRUE
```

```
c(1, 2, 3) > 2
```

```
## [1] FALSE FALSE  TRUE
```

```
1 %in% c(1, 2, 3)
```

```
## [1] TRUE
```

生成逻辑向量：数据类型判断 is.xxx

判断数据结构是否为指定的类型，返回结果为 TRUE 或者 FALSE

```
apropos("is.")
is.character()  # 是否为字符串?
is.data.frame() # 是否为 data.frame?
is.matrix()     # 是否为矩阵?
is.vector()     # 是否为向量?
is.list()       # 是否为列表?
is.logical()    # 是否为逻辑类型?
is.na()         # 是否为缺失值 NA?
```

NA 类型的判断

注意，缺失值，即 NA 类型的数据，不能直接比较，否则结果仍然是 NA。要判断一个值是否为 NA，需要用 is.na()。

```
test <- c(runif(3), NA)
test == NA
```

```
## [1] NA NA NA NA
```

```
is.na(test)
```

```
## [1] FALSE FALSE FALSE  TRUE
```

类型转换 as.xxx

转换为所需要的类型:

```
apropos("as.")
as.character()   # 转换为字符串
as.data.frame()  # 转换为 data.frame
as.matrix()      # 转换为 matrix
as.vector()      # 转换为 vector
as.list()        # 转换为 list
as.logical()     # 转换为逻辑类型
......
```

数值类型转换为逻辑类型

数值类型，容易转换为逻辑类型，默认非零为 TRUE，零为 FALSE

```
dat <- c(2,3,5,7,0,9,0)
as.logical(dat)
```

```
## [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
```

逻辑向量的运算（与、或、非）

若现有相互独立的事件 A 和 B，以下为常见的逻辑运算：

- A 不发生，记为“非 A”（`!A`）
- B 不发生，记为“非 B”（`!B`）
- A、B 同时发生，“A 与 B”（`A&B`）
- A、B 至少有一个发生，记为“A 或 B” (`A|B`)

逻辑运算（取反，“非”）

逻辑运算的结果，还是逻辑向量

- !: **取反**，表示”不是”
  - !TRUE 结果为 FALSE
  - !FALSE 结果为 TRUE

```
had_breakfast <- TRUE
!had_breakfast
```

```
## [1] FALSE
```

```
like_bread <- c(TRUE, TRUE, FALSE, TRUE, TRUE)
res_not <- !like_bread
res_not
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE
```

逻辑运算（“与”）

- & 或者 &&: 与, 表示同时发生，& 两侧的事件必须同时为真，结果才为真 TRUE
  - TRUE & TRUE 结果为 TRUE
  - TRUE & FALSE 结果为 FALSE
  - FALSE & FALSE 结果为 FALSE

```
like_bread <- c(TRUE, TRUE, FALSE, TRUE, TRUE) # 喜欢面包
like_egg   <- c(TRUE, TRUE, FALSE, TRUE, FALSE) # 喜欢鸡蛋
res_and <- like_bread & like_egg # 两种都喜欢
res_and
```

```
## [1]  TRUE  TRUE FALSE  TRUE FALSE
```

逻辑运算（“或”）

- | 或者 ||: **或**，| 两侧只要一个条件满足，返回结果就是真（TRUE）
  - TRUE|TRUE, 结果为 TRUE
  - TRUE|FALSE, 结果为 TRUE
  - FALSE|FALSE, 结果为 FALSE

```
like_bread <- c(TRUE, TRUE, FALSE, TRUE, TRUE) # 喜欢面包
like_egg   <- c(TRUE, TRUE, FALSE, TRUE, FALSE) # 喜欢鸡蛋
res_or <- like_bread | like_egg # 喜欢面包或者鸡蛋任意一款
res_or
```

```
## [1]  TRUE  TRUE FALSE  TRUE  TRUE
```

&、&&、|、|| 的区别

- & 和 | 支持向量化运算，返回值为一个向量
- && 和 || 只支持向量的第一个元素，返回值为一个逻辑值 TRUE 或者 FALSE
- 如果编写函数，一般用 && 或者 ||

all() 和 any()

- all: 判断是否全部元素为真
- any: 判断是否任意一个元素为真

```
aaa <- c(T, T, T, T, F, F, F, T)
# T 是 TRUE 的缩写, F 是 FALSE 的缩写
all(aaa)
```

```
## [1] FALSE
```

```
any(aaa)
```

```
## [1] TRUE
```

用逻辑向量提取部分数据 (subset)

使用下标，或者 subset() 函数

```
library(vegan)
data(dune.env)
subset(dune.env, Use == "Haypastu")[1:3,]
```

```
##    A1 Moisture Management       Use Manure
## 1 2.8        1         SF Haypastu      4
## 2 3.5        1         BF Haypastu      2
## 3 4.3        2         SF Haypastu      4
```

用逻辑向量提取部分数据，用 [] 提取

用逻辑向量提取

```
dune.env[dune.env$A1 > 6,]
```

```
##       A1 Moisture Management       Use Manure
## 5    6.3        1         HF Hayfield      2
## 14   9.3        5         NM  Pasture      0
## 15 11.5        5         NM Haypastu      0
```

用逻辑向量提取部分数据 (示逻辑运算)

A1 > 3 同时 Use == "Haypastu"

```
dune.env[dune.env$A1 > 3 & dune.env$Use == "Haypastu",]
```

```
##       A1 Moisture Management       Use Manure
## 2    3.5        1         BF Haypastu      2
## 3    4.3        2         SF Haypastu      4
## 4    4.2        2         SF Haypastu      4
## 6    4.3        1         HF Haypastu      2
## 12   5.8        4         SF Haypastu      2
## 13   6.0        5         SF Haypastu      3
## 15  11.5        5         NM Haypastu      0
```

随机抽取部分数据

`sample()` 实现（伪）随机抽取部分数据

```
sample(x, size, replace = FALSE, prob = NULL)
```

`replace = TRUE` 实现”有放回抽样”，一般用于生成 bootstrap 样本。

随机抽样举例

```
set.seed(133)
xxx <- sample(1:20+30, 10, replace = TRUE)
table(xxx) # 计算每个数字出现的次数
```

```
## xxx
## 32 34 36 38 39 43 47
##  1  2  1  1  1  2  2
```

set.seed() 设定随机数种子，以保证每次生成的随机数序列是一致的。

3

条件控制 if/else

流程控制 if()

```
if(逻辑值) {
    # 如果逻辑值为 TRUE
    # 语句或者语句块 1
  } else {
    # 如果逻辑值为 FALSE
    # 语句或者语句块 2
  }
```

if 后的括号内只允许一个逻辑值，要么是 TRUE，要么是 FALSE

程序员买包子

程序员的老婆给程序员打电话：“下班后买十个包子，如果看到卖西瓜的，买一个。”

下班后，程序员**买了一个包子**回家了。

普通人理解

第一步，先买十个包子；
第二步，如果没见到卖西瓜的，什么都不做。
　　如果见到卖西瓜的，买一个西瓜。

程序员为什么只买了一个包子回家?

```
if(见到卖西瓜的){
   买一个包子
}

if(没见到卖西瓜的)
   买十个包子
}
```

流程控制 if() 和 else

if() 控制条件是否执行，如果为 TRUE，后面紧跟的语句就会执行

```
homework = TRUE
if(homework){
  print("Do homework") # 是，条件满足
} else {
  print("Go shopping") # 否，条件不满足
}
```

```
## [1] "Do homework"
```

以下两种写法有什么不同?

```r
homework = TRUE

## 写法 1
if(homework) print("Do homework!") else print("Go shopping!")

## 写法 2
if(homework){
  print("Do homework") # 是, 条件满足
} else {
  print("Go shopping") # 否, 条件不满足
}
```

什么时候用 `if()` 和 `else`

- 如果条件是可能变化的，而且不同情况要执行不同的语句，就要用 `if`
- 如果 `if()` 中已经包括要满足的条件，不用考虑其他情况，就不用 `else`
- 如果要考虑所有的情况，就要用 `else`

什么时候用 if()

```
had_breakfast <- NA
had_bread <- TRUE # 吃过面包
had_egg <- FALSE # 吃过鸡蛋
had_milk <- FALSE # 喝过牛奶

if(had_bread){
  had_breakfast <- TRUE
}
if(had_egg){
  had_breakfast <- TRUE
}
if(had_milk){
  had_breakfast <- TRUE
}

print(had_breakfast)
```

什么时候用 else?

```
had_breakfast <- NA
had_bread <- TRUE # 吃过面包
had_egg   <- FALSE # 吃过鸡蛋
had_milk   <- FALSE # 喝过牛奶

if(had_bread|had_egg|had_milk){
  print("I have had breakfirst!")
  had_breakfast <- TRUE
}else{
  print("I haven't had breakfirst yet!")
  had_breakfast <- FALSE
}
```

```
## [1] "I have had breakfirst!"
```

```
print(had_breakfast)
```

什么时候用花括号?

- 如果只执行 `if` 后的一条语句，花括号可有可无（optional）。
- 如果要执行多行语句，必须要用花括号 `{}` 括起来。
- 任何情况，都建议加 `{}`，将要执行的语句与其余部分分隔开。

ifelse 语句

ifelse 是 if(){}else{} 的简化版，常用于语句中，但在一定程度上降低了程序的可读性

```
x <- 3:-1
sqrt(x)# 复数不能计算平方根，因此返回 NaN
```

```
## Warning in sqrt(x): NaNs produced
```

```
## [1] 1.732051 1.414214 1.000000 0.000000      NaN
```

```
sqrt(ifelse(x >= 0, x, NA))# 先将 <0 的转换为 NA，返回结果为 NA
```

```
## [1] 1.732051 1.414214 1.000000 0.000000       NA
```

if 与多个逻辑值

注意：if 里面只能放一个 TRUE 或者 FALSE，如果放多个，只会按照第一个 TRUE 或者 FALSE 执行，而且有 warning 出现。

如果要满足多个 TRUE 怎么办?

答案：

```
all()
any()
```

all 与 any 的使用

```
students <- c("Jinlong", "Jianping", "Xiaoming",
              "Zhigang", "Zhiqiang")
attend <- c(FALSE, TRUE, TRUE, TRUE, TRUE)

if(all(attend)){
  print(" 同学全部到齐! ")
}

if(any(!attend)){
  print(paste(paste(students[!attend],
      collapse = ", "), " 逃课了"))
}
```

```
## [1] "Jinlong 逃课了"
```

4

for 和 while 循环

for 循环

`for(i in X 向量){要执行的语句}`

功能：保证花括号 `{}` 中的内容重复一定的次数。

- “X 向量”一般为 `1:length()`、`1:nrow()` 或向量本身。

for 循环的“变量”

- 循环一般用字母 `i`，`j`，`k`，`m`，`n`，这些字母，一般控制要执行语句的次数，也常作为花括号内变量的下标。
- R 中，循环一般”较慢”，因此要尽量减少循环的使用 (其实也可能循环比较难写)，
- 多用向量计算，或者 `tidyverse` 的 `dplyr` 包、`data.table` 包或者 `apply` 家族的函数操作数据。

什么时候用 for 循环?

- 批量绘图，如森林监测样地，绘制每个物种的分布图
- 部分数据处理，需要提取数据的某一部分，重复提取指定的次数，如随机抽样
- 批处理多个文件、数据集等，每个文件、数据集执行类似的操作，例如将植物照片移动到对应的科、属文件夹中。

for 循环举例

```r
aaa <- LETTERS[1:5]

for(i in 1:3){ # 重复打印 aaa 3 次
  print(aaa)
}

for(i in 1:length(aaa)){ # 每次 i 都在变化
  print(i)
}

for(i in 1:length(aaa)){ # i 可以作为 aaa 的下标
  print(aaa[i])
}
```

for 循环举例

i 可以遍历向量的每一个元素

```
for(i in c("A","B","C")){
  print(i)
}
```

```
## [1] "A"
## [1] "B"
## [1] "C"
```

for 循环举例

循环结束之后，i 的值不再改变

```
aaa <- LETTERS[1:3]

for(i in aaa){
  print(i) # 每次循环 i 的值
}
```

```
## [1] "A"
## [1] "B"
## [1] "C"
```

```
print(i) # 循环之后，i 的值
```

```
## [1] "C"
```

for 循环举例

i 常作为下标使用

```
aaa <- LETTERS[1:5]

for(i in 1:length(aaa)){
  print(aaa[i])
}
```

```
## [1] "A"
## [1] "B"
## [1] "C"
## [1] "D"
## [1] "E"
```

for 循环举例：print 放在 for 的“作用域”里面

可以在循环之前，先创建一个空白向量，每次循环，增加该次循环运算出的值。

```
aaa <- LETTERS[1:4]
res <- c()
# print(i)
for(i in 1:length(aaa)){
  res <- c(res, aaa[i])
  print(res)
}
```

```
## [1] "A"
## [1] "A" "B"
## [1] "A" "B" "C"
## [1] "A" "B" "C" "D"
```

```
print(res)
```

for 循环 + if 判断举例

随机生成 3 个数字，如果 <0.5，打印”小于 0.5”；如果 >=0.5，打印”大于或等于 0.5”

```
set.seed(123)
aaa <- runif(3) # 生成 3 个 0-1 之间的随机数
print(aaa)
```

```
## [1] 0.2875775 0.7883051 0.4089769
```

for 循环 + if 判断举例

```
for (i in 1:length(aaa)){
   if(aaa[i] < 0.5){
      print(paste(round(aaa[i], 3),
        "is less than 0.5"))
   } else {
      print(paste(round(aaa[i], 3),
        "is greater than or equal to 0.5"))
   }
}
```

```
## [1] "0.288 is less than 0.5"
## [1] "0.788 is greater than or equal to 0.5"
## [1] "0.409 is less than 0.5"
```

for 循环举例

1 到 10，如果是奇数，打印” 奇数”，如果是偶数，打印” 偶数”

写法一

```
res <- character() # 先创建一个空白向量以保存结果
for(i in 1:10){
  if(i %% 2 == 0){
    res <- c(res, " 偶")
  } else {
    res <- c(res, " 奇")
  }
}
res
```

```
##  [1] "奇" "偶" "奇" "偶" "奇" "偶" "奇" "偶" "奇" "偶"
```

for 循环举例

写法二

```
dat <- 1:10
res <- rep(NA, length(dat))

for(i in 1:length(dat)){
  if(dat[i] %% 2 == 0){
    res[i] <- " 偶"
  } else {
    res[i] <- " 奇"
  }
}
print(res)
```

```
##  [1] " 奇" " 偶" " 奇" " 偶" " 奇" " 偶" " 奇" " 偶" " 奇" " 偶"
```

for 循环是否真的有必要?

R 是一种向量化的语言

```
dat <- 1:10
res <- rep(NA, length(dat))
res[dat%%2 == 0] <- " 偶"
res[dat%%2 == 1] <- " 奇"
res
```

```
##  [1] " 奇" " 偶" " 奇" " 偶" " 奇" " 偶" " 奇" " 偶" " 奇" " 偶"
```

while 循环

while 循环用于不知道要进行多少次循环时，相比 for 循环，while 循环有时候更加简洁，但也更加抽象 (所以 while 比 for 要少用)：

```
res = 20
while(res > 10){
   res = res - 2
   print(res)
}
```

```
## [1] 18
## [1] 16
## [1] 14
## [1] 12
## [1] 10
```

switch 分支选择

绝大部分可以用 `if(){} else{}` 代替，但 `switch` 一般更简洁。

```
x <- " 面包"
unit <- switch(x,
       面包 = " 块",
       鸡蛋 = " 个",
       牛奶 = " 杯")
unit
```

```
## [1] "块"
```

5

apply 家族

apply 函数家族

apply 可减少循环的使用，如计算矩阵每一列的和，每一列的中位数等。

```
library(vegan)
data(BCI) # BCI 样地数据，行为样方名，列为物种名，值表示个体数
head(apply(BCI,MARGIN = 1, sum )) # 每个样方的个体数
```

```
##   1   2   3   4   5   6
## 448 435 463 508 505 412
```

```
head(apply(BCI,MARGIN = 2, sum ), 4) # 每个种的个体数
```

```
##    Abarema.macradenia Vachellia.melanoceras Acalypha.divers
##                     1                     3
## Acalypha.macrostachya
##                     1
```

已知成绩单，求男生和女生的平均分

```
records <- data.frame(c("John","Fred","George","Tonny",
                        "Daisy","Jane"),
                      c(100,85,60,72,90,95),
                      c("M", "M", "M", "M", "F", "F"))
colnames(records) <- c("name", "score", "gender")
head(records, 3)
```

```
##      name score gender
## 1    John   100      M
## 2    Fred    85      M
## 3 George    60      M
```

tapply 按照 gender 求平均成绩

```
# 各因子每个水平下运行 sum 函数
tapply(records$score, records$gender, mean)
```

```
##     F     M
## 92.50 79.25
```

6

运算符的优先次序

运算符的优先次序

类似于四则运算优先次序

- ()[]：最高
- ^：平方、立方、幂其次
- *，/，+，-，%%，%/%：再次
- ==，!=：低
- |，&：最低

任何不清楚优先等级的，都需要用 () 控制

```
(2 + 3)^2 - (5 + 12)/9 == 20
```

```
## [1] FALSE
```

7

字符串的匹配、拆分和替换

操作字符串的函数

- `nchar()`: 字符串中的字符数
- `cat()`: 连接字符串，并输送到设备中
- `paste()/paste0()`: 合并字符串
- `substr()`: 按照下标提取字符串的一部分
- `substring()`: 按照下标提取字符串的一部分
- `strsplit()`: 按照匹配规则拆分字符串

操作字符串的函数

- `gsub()`: 替换
- `grepl()`: 对字符串元素，匹配返回 TRUE，不匹配返回 FALSE
- `grep()`: 返回匹配字符串向量的”下标”
- `toupper()`: 全部转换为大写
- `tolower()`: 全部转换为小写

操作字符串的函数

```
nchar("USA")
```

```
## [1] 3
```

```
paste("Group", 1:3)
```

```
## [1] "Group 1" "Group 2" "Group 3"
```

```
paste0("Group", 1:3)
```

```
## [1] "Group1" "Group2" "Group3"
```

操作字符串的函数

```
substr("ABCDEFG", start = 1, stop = 3)
```

```
## [1] "ABC"
```

gsub 替换

现有四名参赛选手，分别来自不同班级，-之前是姓名，-之后是所属班级，问一共有几个班？每组里面有多少人？

```
aaa <- c("Zhang Jin-Class01", "Wang Tao-Class01",
         "Zhao Lin-Class02", "Li Shuo-Class03")
# 替换掉所有字母以及-
res <- gsub("[a-zA-Z-]", "", aaa)
res
```

```
## [1] " 01" " 01" " 02" " 03"
```

```
print(paste0("Number of classes: ",length(unique(res))))
```

```
## [1] "Number of classes: 3"
```

```
table(res) # 每个班级重复的次数
```

```
## res
```

grepl 匹配字段

找出姓 Zhang 同学相关的数据

```
aaa <- c("Zhang Jin-Class01", "Wang Tao-Class01",
         "Zhao Lin-Class02", "Li Shuo-Class03")
with_zhang <- grepl("Zhang", aaa)
with_zhang
```

```
## [1]  TRUE FALSE FALSE FALSE
```

```
aaa[with_zhang]
```

```
## [1] "Zhang Jin-Class01"
```

字符串处理举例

百家姓前几名的拼音中，各字母出现的频率

```
family_names <- c("Zhao", "Qian", "Sun", "Li",
    "Zhou", "Wu", "Zheng", "Wang")
all_chars <- paste(family_names, collapse = "")
sort(table(strsplit(all_chars, split = "")))
```

```
##
## e L Q S g i o W a h u Z n
## 1 1 1 1 2 2 2 2 3 3 3 3 4
```

正则表达式 (Regular Expression)

```
?regex
```

- 正则表达式用于字符串的查找与替换。
- 能够按照一定规则匹配所需要的字符。
- 正则表达式中区分大小写。

在 Perl、Python、Ruby、JavaScript 等语言中都已实现。

常用的特殊符号

```
\\n \n # 换行
\\t \t # 制表符
\\s \s # 所有空格
\\d \d # 所有数字
\\D \D # 所有非数字
\\w \w # 任意单词内的字符
\\b \b # 单词边的字符
```

常用正则表达式

```
[:digit:] # 数字
[:alpha:] # 字母
[:lower:] # 小写字母
[:upper:] # 大写字母
[:alnum:] # 数字和字母
[:punct:] # 标点符号
... ...
```

进一步学习正则表达式：stringr 包

常用字符串操作，用 `stringr` 包更为方便。

下载 `stringr` 包的 cheatsheet：

https://www.rstudio.com/resources/cheatsheets/

8

数据的去重复、排序、合并

去重复 unique

- 用于向量，去除重复元素
- 用于数据框，删除重复的行

```
b1 <- sample(1:20, 20, replace = TRUE)
table(b1)
```

```
## b1
##  3  4  5  7  8  9 10 11 14 17 18 19 20
##  2  1  2  1  1  2  2  1  3  1  1  2  1
```

```
unique(b1)
```

```
##  [1] 14  3 10 18 11  5 20 19  9  8  7  4 17
```

数据的排序: sort

直接排序

```
library(vegan)
data(dune.env)
sort(dune.env$A1) # 直接对向量排序
```

```
##  [1]  2.8  2.8  3.3  3.5  3.5  3.5  3.7  3.7  4.0  4.2  4.2
## [16]  5.8  6.0  6.3  9.3 11.5
```

数据的排序：order

生成排序用的下标

```
# 按照 A1 的值排序，order 返回的是向量出现的 index
dune.env[order(dune.env$A1),]
```

```
##       A1 Moisture Management       Use Manure
## 1    2.8        1         SF Haypastu      4
## 7    2.8        1         HF  Pasture      3
## 10   3.3        2         BF Hayfield      1
## 2    3.5        1         BF Haypastu      2
## 11   3.5        1         BF  Pasture      1
## 20   3.5        5         NM Hayfield      0
## 9    3.7        4         HF Hayfield      1
## 19   3.7        5         NM Hayfield      0
## 17   4.0        2         NM Hayfield      0
## 4    4.2        2         SF Haypastu      4
## 8    4.2        5         HF  Pasture      3
```

数据的合并：merge

merge，用来合并两个 data.frame()

```
records <- data.frame(c("John","Fred","George","Tonny",
                        "Daisy","Jane"),
                      c(100,85,60,72,90,95),
                      c("M", "M", "M", "M", "F", "F"))
colnames(records) <- c("name", "score", "gender")
head(records, 3)
```

```
##      name score gender
## 1    John   100      M
## 2    Fred    85      M
## 3 George    60      M
```

数据的合并：merge

attend_dat 数据

```
attend_dat <- data.frame(name = c("Jane",   "George", "Fred",
head(attend_dat)
```

```
##      name attend
## 1    Jane   TRUE
## 2 George   TRUE
## 3    Fred  FALSE
## 4  Tonny   TRUE
## 5  Daisy  FALSE
```

数据的合并：merge

```
merge(x = records, y = attend_dat, by = "name", all = TRUE)
```

```
##     name score gender attend
## 1  Daisy    90      F  FALSE
## 2   Fred    85      M  FALSE
## 3 George    60      M   TRUE
## 4   Jane    95      F   TRUE
## 5   John   100      M     NA
## 6  Tonny    72      M   TRUE
```

9

R 工作空间的保存

将 R 工作空间保存为 RData 文件

- `save()` 除了将 R 工作空间内特定的对象保存为.RData 文件，
- `save.image()` 将当前工作空间内所有对象 (working space) 保存为.RData 文件。

```
x <- stats::runif(20)
y <- list(a = 1, b = TRUE, c = "oops")
save(x, y, file = "xy.RData")
```

读取 RData 文件

- RData 文件是二进制文件，可以存储 R 工作空间的所有对象。
- 一般用 `load()` 读取。

```
?load
```

save() 函数要多用

最简单，最常用的存储结果的方式就是 *save()*，这个函数在工作目录下生成.*Rdata* 的文件，下回你直接打开这个文件，用 *ls()* 就能直接看到有哪些对象了，方便又简单，效率高，又不容易出错……

以后凡是让我看数据的，一律是要这种.*Rdata* 的文件，别给我 *csv* 了。

———— 赖江山

内容回顾

1. 下标、逻辑运算和数据提取
2. 类型判断和转换：`is` 和 `as`
3. 条件控制 `if/else`
4. `for` 和 `while` 循环
5. `apply` 家族
6. 运算符的优先次序
7. 字符串的匹配、拆分和替换
8. 数据的去重复与排序
9. 工作空间的保存