

# R 语言基础

---

王敏杰

2021 年 12 月 22 日

四川师范大学

# 本节课的内容

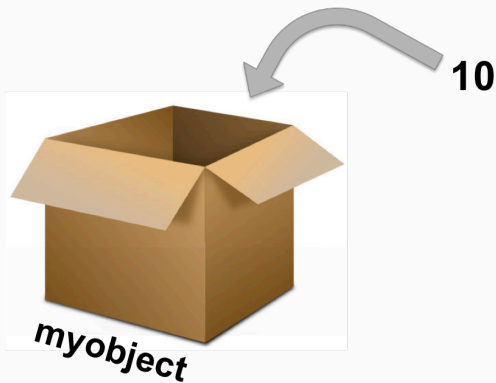
- 对象
- 向量
- 数据结构
- 运算符及向量运算
- 函数
- 子集选取

# 对象

---

# 对象

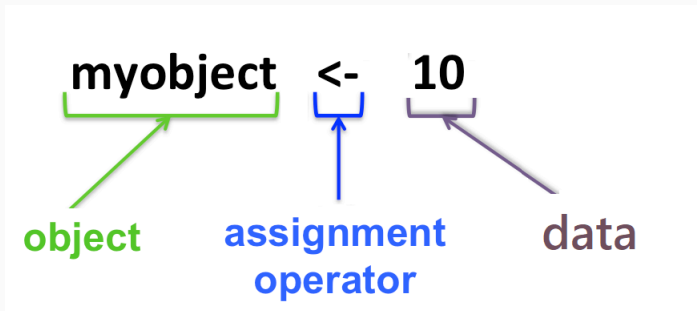
R 语言里一切都是对象 (object)。



可以装数据、装函数。

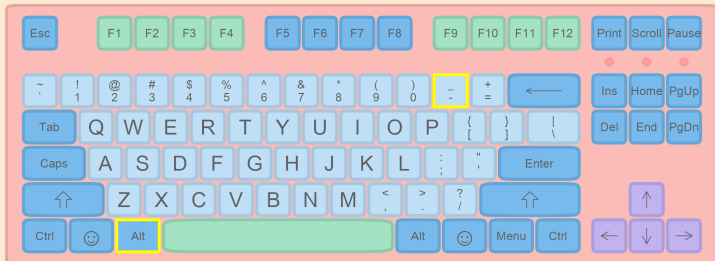
# 对象的创建

对象的创建与使用。首先确定一个对象名，然后使用赋值操作符 `<-`，将数据赋值给它。



# 赋值符号

在 Rstudio 中同时按下 Alt 和 -，就可以产生赋值箭头



# 对象的创建与使用

我们给这个盒子取名 `x`，然后把 10 这个数放入盒子。

```
x <- 10
```

当键入 `x` 然后回车，就打印出 `x` 的值。

```
x  
#> [1] 10
```

## 对象重新赋值

当我们再把 6 这个数放入盒子 `x` 后

```
x <- 6
```

此时 `x` 就被更新了，变成了最新的数值

```
x
```

```
#> [1] 6
```



# 变量命名规则

- 变量名必须以字母、数字、下划线 \_ 和句点 . 组成
- 开头不能是数字
- 大小写敏感, y 和 Y 是两个不同的变量名
- 不能有空格, 可以用下划线代替空格, 比如

```
my_age <- 30
```

# 变量名的可读性

测量男生的身高

- 一般的变量名

```
x <- 175
```

- 不错变量名

```
height <- 175
```

- 最佳的变量名

```
boy_height_cm <- 175
```

# 对象属性

```
x <- 6
```

所有 R 对象都有其属性，最重要的两个属性：

- 类型
- 长度

```
typeof(x)
```

```
#> [1] "double"
```

```
length(x)
```

```
#> [1] 1
```

# 向量

---

## 盒子可以多装点数据？

前面，我们把 6 这个数放入盒子 x，

```
x <- 6
```

现在，我们想多装一些数据（有顺序、好取出），比如 3,4,5,6,7

```
x <- 3, 4, 5, 6, 7 # 这样可以吗？
```

## 冰糖葫芦

我们小时候吃的冰糖葫芦，中间用一根木棒把水果串起来，有先后顺序，而且方便取出。



# 向量就像冰糖葫芦

对应到 R 语言里，用 `c()` 函数实现类似结构



```
x <- c( 3,  
        4,  
        5,  
        6,  
        7  
      )
```

vector



```
x <- c(3, 4, 5, 6, 7)
```

```
x
```

```
#> [1] 3 4 5 6 7
```

# 向量就像冰糖葫芦

```
x <- c(3, 4, 5, 6, 7)
```

我们观察到 `c()` 函数构造向量的几个要求

- 这里的 `c` 就是 combine 或 concatenate 的意思
- 它要求元素之间用英文的逗号分隔
- 且元素的数据类型是统一的，比如这里都是数值



## 聚合成新向量

c() 函数还可以把两个向量聚合成一个新的向量。

```
low      <- c(1, 2, 3)
high     <- c(4, 5, 6)
sequence <- c(low, high)
sequence
#> [1] 1 2 3 4 5 6
```

## 命名向量 (named vector)

向量元素可以有自己的名字

```
x <- c('a' = 5, 'b' = 6, 'c' = 7, 'd' = 8)
```

```
x
```

```
#> a b c d
```

```
#> 5 6 7 8
```

或者

```
x <- c(5, 6, 7, 8)
```

```
names(x) <- c('a', 'b', 'c', 'd')
```

```
x
```

```
#> a b c d
```

```
#> 5 6 7 8
```

## 单个值的向量，就可以偷懒

我们再回头看看之前的

```
x <- 6
```

它实际上就是

```
x <- c(6)
```

即长度为 1 的向量，相当于，只有一个草莓的糖葫芦。在我看来，`x <- 6` 是 `x <- c(6)` 偷懒的写法。

# 向量的属性

我们再来看看向量的两个基本属性

```
x <- c(3, 4, 5, 6, 7)
```

- 类型

```
typeof(x)
```

```
#> [1] "double"
```

- 长度

```
length(x)
```

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

# 数值型向量

向量的元素都是数值类型，因此也叫数值型向量。

数值型的向量，有 integer 和 double 两种：

```
x <- c(1L, 5L, 2L, 3L)      # 整数型
x <- c(1.5, -0.5, 2, 3)      # 双精度类型，常用写法
x <- c(3e+06, 1.23e2)        # 双精度类型，科学计数法
```

## 数值型向量，偷懒方法 1

但如果向量元素很多，用手工一个个去输入，实际运用中不现实。在特定情况下，有几种偷懒方法：

- `seq()` 函数可以生成等差数列，`from` 参数指定数列的起始值，`to` 参数指定数列的终止值，`by` 参数指定数值的间距：

```
s1 <- seq(from = 0, to = 10, by = 0.5)
```

```
s1
```

```
#> [1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0
```

```
#> [16] 7.5 8.0 8.5 9.0 9.5 10.0
```

## 数值型向量，偷懒方法 2

- `rep()` 是 repeat (重复) 的意思，可以用于产生重复出现的数字序列：
  - `times` 指定要生成的个数
  - `each` 指定每个元素重复的次数

```
s2 <- rep(x = c(0, 1), times = 3)
```

```
s2
```

```
#> [1] 0 1 0 1 0 1
```

```
s3 <- rep(x = c(0, 1), each = 3)
```

```
s3
```

```
#> [1] 0 0 0 1 1 1
```

## 数值型向量，偷懒方法 3

- $m:n$ ，如果单纯是要生成数值间距为 1 的数列，用  $m:n$  更快捷，它产生从  $m$  到  $n$  的间距为 1 的数列

```
# Colon operator (with by = 1):
```

```
s4 <- 0:10
```

```
s4
```

```
#> [1] 0 1 2 3 4 5 6 7 8 9 10
```

```
s5 <- 10:1
```

```
s5
```

```
#> [1] 10 9 8 7 6 5 4 3 2 1
```



## 字符串型向量

字符串 (String) 数据类型, 实际上就是文本类型, 必须用单引号或者是双引号包含

```
x <- c("a", "b", "c")
```

```
x <- c('Alice', 'Bob', 'Charlie', 'Dave')
```

```
x <- c("hello", "baby", "I love you!")
```

需要注意的是, x1 是字符串型向量, x2 是数值型向量

```
x1 <- c("1", "2", "3")
```

```
x2 <- c(1, 2, 3)
```

# 逻辑型向量

逻辑型常称为布尔型 (Boolean):

- 常量值只有 TRUE 和 FALSE。
- TRUE 和 FALSE 在 R 语言中是保留词汇

```
x <- c(TRUE, TRUE, FALSE, FALSE)
```

```
x <- c(T, T, F, F)      # 与上等价，但不推荐
```

以下两者不要混淆

```
x <- c(TRUE, FALSE)      # 逻辑型
```

```
x <- c("TRUE", "FALSE")  # 字符串型
```

# 因子型向量

因子型可以看作是字符串向量的增强版，它是带有层级（Levels）的字符串向量。

比如，这里四个季节的名称，他们构成一个向量。

```
four_seasons <- c("spring", "summer",  
                  "autumn", "winter")
```

我们使用 `factor()` 函数将其转换成因子型向量

```
four_seasons_factor <- factor(four_seasons)  
four_seasons_factor  
#> [1] spring summer autumn winter  
#> Levels: autumn spring summer winter
```

# 因子型向量

查看因子型向量的时候，同时也显示层级信息





- 默认的情况，它是按照字符串首字母的顺序排序

```
four_seasons_factor <- factor(four_seasons)
four_seasons_factor
#> [1] spring summer autumn winter
#> Levels: autumn spring summer winter
```

- 也可以指定顺序，比如按照我对四个季节的喜欢排序

```
four_seasons_factor <- factor(
  four_seasons,
  levels = c("summer", "winter", "spring", "autumn")
)
four_seasons_factor
#> [1] spring summer autumn winter
#> Levels: summer winter spring autumn
```

# 小结

Vector	Type	Examples
	numeric	<pre>&gt; c(1, 0.5, 3, 7) [1] 1.0 0.5 3.0 7.0</pre>
	character	<pre>&gt; c("Alice", "love", "30", "dog") [1] "Alice" "love"  "30"   "dog"</pre>
	logical	<pre>&gt; c(TRUE, FALSE, TRUE, FALSE) [1] TRUE FALSE TRUE FALSE</pre>
	factor	<pre>&gt; factor(c("a", "c", "c", "b")) [1] a c c b Levels: a b c</pre>

# 数据结构

---

# 向量

向量，是 R 语言最基础的数据结构。



vector



## 更多的数据结构

前面介绍了向量，它是 R 语言中最基础的数据结构，我们还会遇到其它数据结构

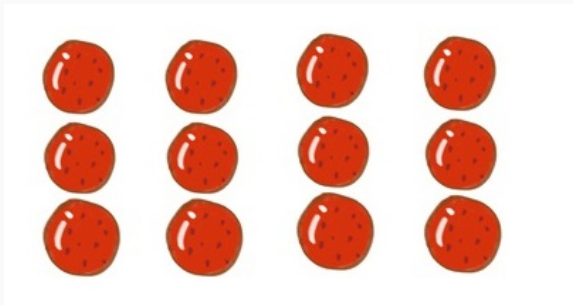
- 矩阵
- 列表
- 数据框

这些数据结构都可以看作由向量衍生出来的。



# 矩阵

矩阵可以存储行 (row) 和列 (column) 二维的数据



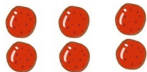
它实际上是向量的另一种表现形式。

# 矩阵

矩阵可以用 `matrix()` 函数创建，第一位参数的位置是用于创建矩阵的向量。比如下面把向量 `c(2, 4, 3, 1, 5, 7)` 转换成 2 行 3 列的矩阵

```
m <- matrix(  
  c(2, 4, 3, 1, 5, 7),  
  nrow = 2,  
  ncol = 3  
)  
m  
#>      [,1] [,2] [,3]  
#> [1,]    2    3    5  
#> [2,]    4    1    7
```

# 矩阵



```
matrix(  
  c(2, 4, 3, 1, 5, 7),  
  nrow = 2,  
  ncol = 3  
)
```

Matrix

2	3	5
4	1	7

# 矩阵的属性

- 类型

```
class(m)  
#> [1] "matrix" "array"
```

- 长度

```
length(m)  
#> [1] 6
```

- 维度

```
dim(m)  
#> [1] 2 3
```

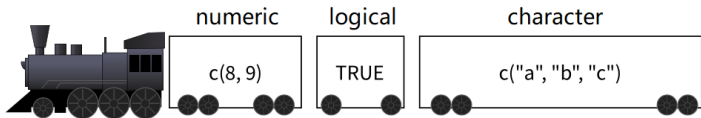
# 矩阵

- 向量是一个竖着的糖葫芦，在转换成矩阵的时候，也是先竖着排，第一列竖着的方法排满后，就排第二列，这是默认的情形。
- 如果想改变这一传统习惯，也可以增加一个语句 `byrow = TRUE`，这条语句让向量先横着排，排完第一行，再排第二行。

```
matrix(  
  c(2, 4, 3, 1, 5, 7), nrow = 2, byrow = TRUE  
)  
#>      [,1] [,2] [,3]  
#> [1,]    2    4    3  
#> [2,]    1    5    7
```

# 列表

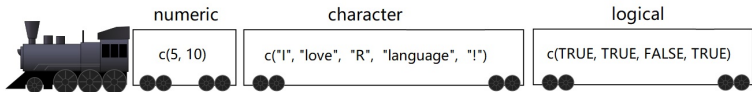
想象有一个小火车，小火车的每节车厢是独立的，因此每节车厢装的东西可以不一样。这种结构，装载数据的能力很强大，称之为列表（list）。



## 创建列表

```
list1 <- list(  
  a = c(5, 10),  
  b = c("I", "love", "R", "language", "!"),  
  c = c(TRUE, TRUE, FALSE, TRUE)  
)  
list1  
#> $a  
#> [1] 5 10  
#>  
#> $b  
#> [1] "I" "love" "R" "language" "  
#>  
#> $c  
#> [1] TRUE TRUE FALSE TRUE
```

# 创建列表



```
list(  
  a = c(5, 10),  
  b = c("I", "love", "R", "language", "!"),  
  c = c(TRUE, TRUE, FALSE, TRUE)  
)
```

List

5	"I"	TRUE
10	"love"	TRUE
	"R"	FALSE
	"language"	TRUE
	"!"	

numeric    character    logical



# 列表

`list()` 函数创建列表 Vs. `c()` 函数创建向量：

- **相同点：** 元素之间用逗号分开。
- **不同点：**
  - 向量的元素是单个值；列表的元素可以是更复杂的结构，可以是向量、矩阵或者列表。
  - 向量要求每个元素的数据类型必须相同，要么都是数值型，要么都是字符型；而列表的元素允许不同的数据类型。

# 列表的属性

- 类型

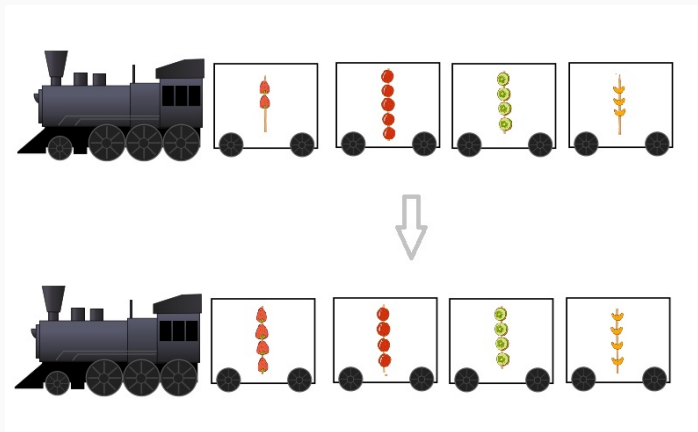
```
class(list1)  
#> [1] "list"
```

- 长度

```
length(list1)  
#> [1] 3
```

# 数据框

列表是一个小火车，如果每节车厢装的都是向量而且等长，那么这种特殊形式的列表就变成了数据框 (data frame)



换句话说，数据框是一种特殊的列表

# 创建数据框

我们可以使用 `data.frame()` 函数构建

```
df <- data.frame(  
  name      = c("Alice", "Bob", "Carl", "Dave"),  
  age       = c(23, 34, 23, 25),  
  marriage  = c(TRUE, FALSE, TRUE, FALSE),  
  color     = c("red", "blue", "orange", "purple")  
)  
df  
#>      name age marriage  color  
#> 1 Alice  23      TRUE    red  
#> 2  Bob   34     FALSE   blue  
#> 3 Carl  23      TRUE  orange  
#> 4 Dave  25     FALSE  purple
```

# 数据框就是我们经常用的 excel 表格

```
data.frame(  
  name = c("Alice", "Bob", "Carl", "Dave"),  
  age = c(23, 34, 23, 25),  
  marriage = c(TRUE, FALSE, TRUE, FALSE),  
  color = c("red", "blue", "orange", "purple")  
)
```

data frame

"Alice"	23	TRUE	"red"
"Bob"	34	FALSE	"blue"
"Carl"	23	TRUE	"orange"
"Dave"	25	FALSE	"purple"

character   numeric   logical   character

	A	B	C	D
1	name	age	marriage	color
2	Alice	23	TRUE	red
3	Bob	34	FALSE	blue
4	Carl	23	TRUE	orange
5	Dave	25	FALSE	purple

由于数据框融合了向量、列表和矩阵的特性，所以在数据科学的统计建模和可视化中运用非常广泛。

# 数据框的属性

- 类型

```
class(df)
#> [1] "data.frame"
```

- 维度

```
nrow(df)
#> [1] 4
ncol(df)
#> [1] 4
```

# 数据结构

R 对象的数据结构 (向量、矩阵、列表和数据框), 总结如下

Vector

"R"
"S"
"T"

character

Matrix

"1"	"R"	"TRUE"
"2"	"S"	"FALSE"
"3"	"T"	"TRUE"

character

List

1	"R"	TRUE
---	-----	------

numeric   character   logical

data frame

1	"R"	TRUE
2	"S"	FALSE
3	"T"	TRUE

numeric   character   logical