Basic Data Structures in R

WISERCLUB 段孙蓬

按照个人要求的格式来创建含有研究信息的数据集是任何数据分析的第一步。当我们使用 R 进行数据分析时，我们首先需要了解的就是 R 里面的数据类型和数据结构。

1 数据类型

1.1 数据类型组成

在介绍 R 语言的基础数据结构之前，我们需要先了解数据的类型。R 语言中的五种基础的数据类型，它们分别是：

1. Logical，逻辑型：仅有 TRUE 和 FALSE 两个取值，用于逻辑判断；
2. Integer，整数型：所有整数；
3. Numerical/Double，数值型：包含了小数；
4. Character，字符型：表示文本内容；
5. Complex，复数型：复数表达，形如 $a + b * 0i$。

在数据分析时，我们很少会处理复数型数据，故本次讲解中不会涉及。

注意，在 R 语言中，数值型数据均为浮点型。请看以下示例：

```
sin(pi) == 0
```

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

1.2 如何判断数据类型？

我们经常通过 typeof(), is.integer() 等几个函数来了解变量的类型。前者返回数据类型，后者返回逻辑值。

```
x <- 2.5
typeof(x)
```

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

```
is.numeric(x)
```

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

```
is.double(x)
```

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

值得注意的是，如果我们需要得到一个 integer 变量，需要在这个整数后面加上 “L” 后缀，否则会识别为 double。

```
x <- 3L
is.integer(x)
```

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

```
is.double(x)
```

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

```
x <- 3
is.integer(x)
```

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

```
is.double(x)
```

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

有时明确区分 integer 和 double 是有必要的，尤其在大规模运算中可以减小的存储空间并提升运行速度。

除此之外，我们还会经常遇到的空值 (NULL) 和缺失值 (NA)，我们会在后面具体应用时讨论。

1.3 类型强制转换 (Coercion)

思考下面程序将会得到什么结果：

```
FALSE + 1 + TRUE

a = c(TRUE,1)
typeof(a)
```

实际上这里 FASLE 和 TRUE 被自动转换为了 0 和 1。不同的数据类型之间转换总体上符合下面的这个顺序：`NULL < logical < integer < numeric < character`。

当然你也可以自己实现类型间的强制转换。这些函数通常是以 `as.` 开头的，比如下面的程序：

```
x <- 1L
as.logical(x)
```

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

```
as.numeric(x)
```

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

```
as.character(x)
```

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

2 基础数据结构

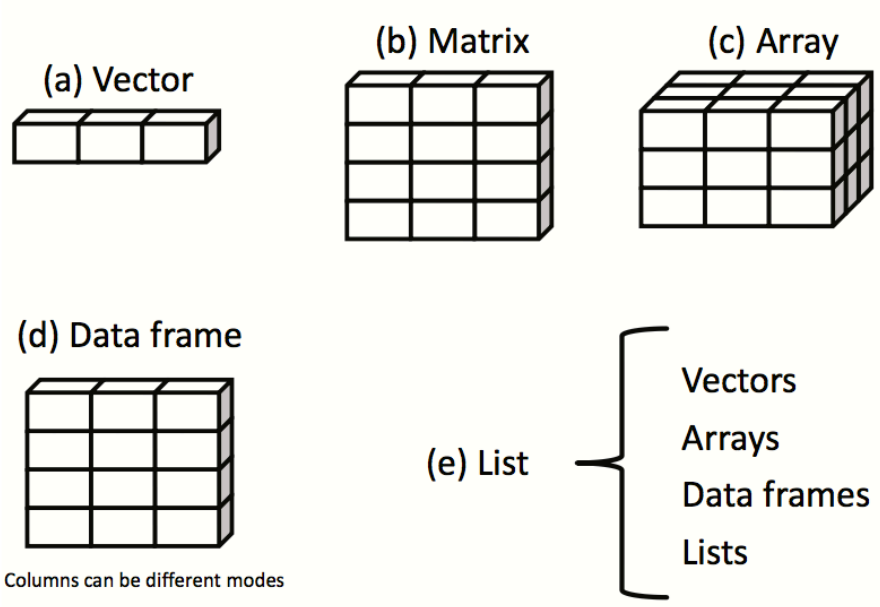

图 1: R 中的数据结构

2.1 Vector 向量

2.1.1 创建向量

Vector(向量) 在 R 语言中具有核心地位。建立一个向量通常用 `c()` 函数即可 (字母 c 表示 combine)，并且可以用 `length()` 查看它的长度:

```
vec <- c(2, 3)
vec
```

```
## [1] 2 3
```

```
length(vec)
```

```
## [1] 2
```

也有很多方法帮助我们快速生成有规律的数字序列，比较常用的是：和 seq() 和 rep()：

```
1:5
```

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

```
6:pi
```

```
## [1] 6 5 4
```

```
seq(from = 0, to = 10, by = 2)
```

```
## [1]  0  2  4  6  8 10
```

```
seq(from = 1, length.out = 5)
```

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

```
a = 1:12
seq(from = 0, to = 10, along.with = a)
```

```
## [1] 0.0000000 0.9090909 1.8181818 2.7272727 3.6363636 4.5454545
## [7] 5.4545455 6.3636364 7.2727273 8.1818182 9.0909091 10.0000000
```

```
rep(1:4, 2)
```

```
## [1] 1 2 3 4 1 2 3 4
```

```
rep(1:4, each = 2)
```

```
## [1] 1 1 2 2 3 3 4 4
```

```
rep(1:4, times = 2)
```

```
## [1] 1 2 3 4 1 2 3 4
```

```
rep(1:4,1:4)
```

```
##  [1] 1 2 2 3 3 3 4 4 4 4
```

同时我们也可以通过 names() 对向量的元素命名：

```
vec <- 1:5
names(vec) <- LETTERS[1:5] #LETTERS 是 R 内置的一个向量
vec
```

```
## A B C D E
## 1 2 3 4 5
```

Vector 要求其内的元素是同质的 (Homogeneous)，即所有元素必须是同一个数据类型，但并不必须是数值型 (Numeric) 的。如果向量中的元素类型不一致将会强制转换，应该注意和避免不恰当的强制转换。

```
x <- c("a", "c", "d")
typeof(c("a", 3))
```

```
## [1] "character"
```

最后 R 语言中不存在标量 (Scalar)。单个数字在 R 语言中视为长度为 1 的向量。

2.1.2 向量拼接

我们也可以对 Vector 做拼接，将几个向量串接为一个向量：

```
vec1 <- 1:5
vec2 <- c(-3, vec1, c(-3, -4))
vec2
```

```
## [1] -3  1  2  3  4  5 -3 -4
```

Vector 中的拼接是没有层级结构的，也就是它总是一个一维的向量，这一点和后面介绍的 List 存在区别。

2.1.3 向量运算

数值向量间可以进行 +，-，*，/等各类数学运算，但是这些运算和数学运算的定义并不完全一致。应该注意这些运算是 “点对点”(element-wise)的运算。

```
1:3 + 2:4
```

```
## [1] 3 5 7
```

```
1:3 - 2:4
```

```
## [1] -1 -1 -1
```

```
1:3 * 2:4
```

```
## [1]  2  6 12
```

```
1:3 / 2:4
```

```
## [1] 0.5000000 0.6666667 0.7500000
```

```
(1:3)^2
```

```
## [1] 1 4 9
```

另外应该非常注意的是，当参与计算的向量长度不等时，R 将会自动循环长度较短的向量，再进行运算。这个过程是没有提示的。

```
1 + 2:4 # equal to c(1+2, 1+3, 1+4)
```

```
## [1] 3 4 5
```

```
1:2 + 2:5 # equal to c(1+2, 2+3, 1+4, 2:5)
```

```
## [1] 3 5 5 7
```

2.1.4 向量元素访问

有很多种方法可以帮助我们选取一个向量中的特定元素。最简单的方法是通过下标选取。

```
vec <- c(4, 5, 3, 2, 7)
vec[2]
```

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

```
vec[3:4]
```

```
## [1] 3 2
```

```
vec[c(1, 3, 4)]
```

```
## [1] 4 3 2
```

```
vec[-2]
```

```
## [1] 4 3 2 7
```

另一种方式是通过逻辑向量进行筛选：

```
vec <- c(4, 5, 3, 2, 7)
vec[c(T, F, T, T, F)]
```

```
## [1] 4 3 2
```

```
vec[vec > 4]
```

```
## [1] 5 7
```

如果 Vector 有 names 的话，也可以通过 names 去索引：

```
vec <- c(4, 5, 3, 2, 7)
names(vec) <- LETTERS[1:5] #LETTERS 是 R 内置的一个向量
vec[c("B", "D")]
```

```
## B D
## 5 2
```

2.1.5 其他常用函数

unique(), table()

```
vec <- rep(LETTERS[1:5], each = 5)
unique(vec)
```

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

```
table(vec)
```

```
## vec
## A B C D E
## 5 5 5 5 5
```

rev(), order(), rank()

```
vec <- c(1, 3, 4, 2, 1, 2)
rev(vec)
```

```
## [1] 2 1 2 4 3 1
```

```
order(vec)
```

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

```
vec[order(vec)] # same as sort(vec)
```

```
## [1] 1 1 2 2 3 4
```

```
rank(vec)
```

```
## [1] 1.5 5.0 6.0 3.5 1.5 3.5
```

%in%

```
vec <- c(1, 3, 4, 2, 1, 2)
c(2, 1, 9) %in% vec
```

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

2.1.6 Factor

Factor 是一类比较特殊的向量。数据一般可以分为名义型、有序型或连续型变量。比如产品质量等级 A、B、C、D，学生性别“男”“女”。这类数据很常见，而 Factor 提供了一些列适当的处理方法。

```
vec <- c("Good", "Bad", "Bad", "Good", "Good")
fac <- factor(vec)
typeof(vec)
```

```
## [1] "character"
```

```
typeof(fac) # Factor 背后实际上是一组数字
```

```
## [1] "integer"
```

```
str(fac)
```

```
##  Factor w/ 2 levels "Bad","Good": 2 1 1 2 2
```

```
levels(fac)
```

```
## [1] "Bad"  "Good"
```

```
fac[2] <- "Normal" # 不可以赋予不在 levels 内的值
```

```
## Warning in `[<-.factor`(`*tmp*`, 2, value = "Normal"): invalid factor
## level, NA generated
```

```
fac
```

```
## [1] Good <NA> Bad  Good Good
## Levels: Bad Good
```

```
c(factor("a"), factor("b"))  #Factor 之间不可以拼接
```

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

请注意以下例子：

```
vec <- c(1,3,5,8,1,4)
fac <- factor(vec)
as.numeric(fac)
```

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

```
as.numeric(as.character(fac))
```

```
## [1] 1 3 5 8 1 4
```

2.2 Matrix 矩阵

2.2.1 创建一个矩阵

R 语言中的 Matrix（矩阵）和数学中的矩阵相似。比如我们创建一个 3×4 的矩阵：

```
mat1 <- matrix(1:12, nrow = 3, ncol = 4, byrow = TRUE)
mat2 <- matrix(1:12, nrow = 3, ncol = 4, byrow = FALSE)
mat3 <- matrix(1:12, nrow = 3, byrow = FALSE)

mat1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
```

```
mat2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
mat3
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

byrow 限制了给定向量的填充顺序是按行填充还是按列填充。在 matrix 函数中，byrow 的缺省值为 FALSE。

同样我们可以给 Matrix 的行（row）和列（column）命名：

```
mat <- matrix(1:12, nrow = 3, ncol = 4, byrow = TRUE,
              dimnames = list(c("r1", "r2", "r3"), c("c1", "c2", "c3", "c4")))

mat
```

```
##    c1 c2 c3 c4
## r1  1  2  3  4
## r2  5  6  7  8
## r3  9 10 11 12
```

```
rownames(mat) <- c("Row1","Row2", "Row3")
colnames(mat) <- c("Col1","Col2", "Col3", "Col4")

mat
```

```
##      Col1 Col2 Col3 Col4
## Row1    1    2    3    4
## Row2    5    6    7    8
## Row3    9   10   11   12
```

2.2.2 非数值型矩阵

和 Vector 相同，Matrix 要求其内的元素必须是同一个数据类型，但它可以不是 numeric 的。比如我们也可以建立 character 的矩阵，虽然它的很多数学计算无法实现。

```
mat <- matrix(LETTERS[1:12], nrow = 3, byrow = TRUE)
mat
```

```
##      [,1] [,2] [,3] [,4]
## [1,] "A"  "B"  "C"  "D"
## [2,] "E"  "F"  "G"  "H"
## [3,] "I"  "J"  "K"  "L"
```

```
typeof(mat)
```

```
## [1] "character"
```

2.2.3 矩阵运算

数量矩阵的数学计算和 Vector 一样，应该注意 `*`, `/`, `^` 等运算是“点对点”的运算。要实现数学中的矩阵乘积需要使用`%*%`。在 R 中，我们对矩阵求逆则需要用到 `solve` 函数。

```
mat1 <- matrix(1:12, nrow = 3, ncol = 4, byrow = TRUE)
mat2 <- matrix(1:12, nrow = 3, ncol = 4, byrow = FALSE)

mat1 + 2*mat2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    3   10   17   24
```

```
## [2,]    9   16   23   30
## [3,]   15   22   29   36
```

```
mat1 * mat2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    8   21   40
## [2,]   10   30   56   88
## [3,]   27   60   99  144
```

```
mat1 / mat2
```

```
##      [,1]     [,2]      [,3]      [,4]
## [1,]  1.0 0.500000 0.4285714 0.4000000
## [2,]  2.5 1.200000 0.8750000 0.7272727
## [3,]  3.0 1.666667 1.2222222 1.0000000
```

```
mat1^2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    9   16
## [2,]   25   36   49   64
## [3,]   81  100  121  144
```

```
mat1 %*% t(mat2)
```

```
##      [,1] [,2] [,3]
## [1,]   70   80   90
## [2,]  158  184  210
## [3,]  246  288  330
```

```
mat1 %*% c(1, 3, 2, 4)
```

```
##      [,1]
## [1,]   29
## [2,]   69
## [3,]  109
```

```
mat3 <- matrix(c(2,1,3,2), nrow = 2, ncol = 2)
solve(mat3)
```

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

Matrix 的拼接也是我们常用的操作之一：

```
rbind(mat1, mat2)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]    1    4    7   10
## [5,]    2    5    8   11
## [6,]    3    6    9   12
```

```
cbind(mat1, mat2)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    1    2    3    4    1    4    7   10
## [2,]    5    6    7    8    2    5    8   11
## [3,]    9   10   11   12    3    6    9   12
```

2.2.4 矩阵元素访问

Matrix 内的元素选取规则和 Vector 非常相似，只是此时我们需要同时指定行和列两个维度。比如：

```
mat1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
```

```
mat1[2, 3]
```

```
## [1] 7
```

```
mat1[1, ]
```

```
## [1] 1 2 3 4
```

```
mat1[, 2]
```

```
## [1]  2  6 10
```

```
mat1[, 1:3]
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    5    6    7
## [3,]    9   10   11
```

```
mat1[, c(2, 4)]
```

```
##      [,1] [,2]
## [1,]    2    4
## [2,]    6    8
## [3,]   10   12
```

```
mat1[, -2]
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    4
## [2,]    5    7    8
## [3,]    9   11   12
```

```
rownames(mat1) <- c("r1", "r2", "r3")
mat1["r1", ]
```

```
## [1] 1 2 3 4
```

同样，我们可以利用这种引用方式对特定元素直接进行修改：

```
mat1[2, 3] <- 999
mat1
```

```
##    [,1] [,2] [,3] [,4]
## r1    1    2    3    4
## r2    5    6  999    8
## r3    9   10   11   12
```

也许你会觉得奇怪，对于 Vector 的选取方式同样适用于 Matrix：

```
mat1[3]
```

```
## [1] 9
```

```
mat2[2:5]
```

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

这是因为 R 语言中的 Matrix 在本质上仍然是一个 Vector，只是它的表达方式和运算规则发生了改变。

2.3 Array 数组

2.3.1 创建一个数组

Array 是 Vector 和 Matrix 向高维度的一个非常自然的扩展。前两者分别是一维和二维数据，而 Array 可以表现任意有限维数据。

```
arr <- array(1:24, dim = c(2, 4, 3))
arr
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    9   11   13   15
## [2,]   10   12   14   16
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
## [1,]   17   19   21   23
## [2,]   18   20   22   24
```

同样我们可以对 Array 中的各个维度命名：

```
dimnames(arr) <- list(c("r1", "r2"), c("c1", "c2", "c3", "c4"), c("k1", "k2", "k3"))
arr
```

```
## , , k1
##
##    c1 c2 c3 c4
## r1  1  3  5  7
## r2  2  4  6  8
##
## , , k2
##
##    c1 c2 c3 c4
## r1  9 11 13 15
## r2 10 12 14 16
##
```

```
## , , k3
##
##    c1 c2 c3 c4
## r1 17 19 21 23
## r2 18 20 22 24
```

当然，也可以利用 dimnames 参数在创建 Array 时命名。

2.3.2 数组数据访问

Array 的数据选取方式也是 Vector 和 Matrix 数据选取方式的非常自然的扩展。

```
arr[2, 3, 1]
```

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

```
arr[, , 3]
```

```
##    c1 c2 c3 c4
## r1 17 19 21 23
## r2 18 20 22 24
```

```
arr[, 3, ]
```

```
##    k1 k2 k3
## r1  5 13 21
## r2  6 14 22
```

```
arr["r1", ,]
```

```
##    k1 k2 k3
## c1  1  9 17
## c2  3 11 19
## c3  5 13 21
## c4  7 15 23
```

```
arr[15:18]
```

```
## [1] 15 16 17 18
```

2.4 Data Frame

2.4.1 创建数据框

Data Frame 和一般的数据表相同。数据框中各列可以是不同类型的数据。数据框每列是一个变量，每行是一个观测。

```
Name <- c("John", "Carl", "Jane")
Math <- c(70, 90, 80)
Physics <- c(80, 75, 85)
Score <- data.frame(Name , Math , Physics)

Score
```

```
##   Name Math Physics
## 1 John   70      80
## 2 Carl   90      75
## 3 Jane   80      85
```

2.4.2 元素访问

Data Frame 的元素访问有多种形式，如下展示了类似 matrix 的元素访问。

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
df[c(1, 3), ]
```

```
##   x y z
## 1 1 3 a
## 3 3 1 c
```

```
df[c("x", "z")]
```

```
##   x z
## 1 1 a
## 2 2 b
## 3 3 c
```

```
df[, c("x", "z")]
```

```
##   x z
## 1 1 a
## 2 2 b
## 3 3 c
```

```
str(df["x"])
```

```
## 'data.frame':    3 obs. of  1 variable:
##  $ x: int  1 2 3
```

```
str(df[, "x"])
```

```
##  int [1:3] 1 2 3
```

此外，data frame 还可以采用 $ 调用的方法，例如：

```
df[df$x == 2,]
```

```
##   x y z
## 2 2 2 b
```

```
df$x
```

```
## [1] 1 2 3
```

对于维度较大的 data frame，使用形如 df$ 的形式会较繁琐，因此，我们可以使用 `attach`,`detach` 以及 `with` 函数。函数 `attach` 可将 data frame

添加到 R 的搜索路径中。R 在遇到一个变量名后，将检查搜索路径中的 data frame，以定位这个变量。函数 detach 则是将 data frame 从搜索路径中移除。

```
attach(mtcars)
summary(mtcars)
plot(mpg,disp)
plot(mpg,wt)
detach(mtcars)
```

当名称相同的对象不止一个时，此方法存在局限性。

```
mpg <- c(25,36,47)
attach(mtcars)
```

```
## The following object is masked _by_ .GlobalEnv:
##
##     mpg
```

```
plot(mpg,wt)
```

```
## Error in xy.coords(x, y, xlabel, ylabel, log): 'x' and 'y' lengths differ
```

```
detach(mtcars)
```

当然，我们还可以使用 with 函数。

```
with(mtcars,{
        summary(mpg,disp,wt)
        plot(mpg,disp)
        plot(mpg,wt)
})
```

函数 with 的局限在于，赋值仅在此函数括号内生效。

```
with(mtcars,{
        stats <- summary(mpg)
        stats
})
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   10.40   15.42   19.20   20.09   22.80   33.90
```

```
stats
```

```
## Error in eval(expr, envir, enclos): object 'stats' not found
```

如果你希望能调用 with 结构内的对象，使用 <<-特殊赋值符号即可。它可将对象保存到 with 之外的全局环境中。

```
with(mtcars,{
        stats <<- summary(mpg)
        stats
})
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   10.40   15.42   19.20   20.09   22.80   33.90
```

```
stats
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   10.40   15.42   19.20   20.09   22.80   33.90
```

2.4.3 快速了解你的数据

这些函数对于快速了解数据会有很多帮助：

```
class(iris)
```

```
## [1] "data.frame"
```

```r
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 ...
```

```r
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```r
tail(iris)
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 145          6.7         3.3          5.7         2.5 virginica
## 146          6.7         3.0          5.2         2.3 virginica
## 147          6.3         2.5          5.0         1.9 virginica
## 148          6.5         3.0          5.2         2.0 virginica
## 149          6.2         3.4          5.4         2.3 virginica
## 150          5.9         3.0          5.1         1.8 virginica
```

```r
tail(iris, n = 10)
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 141          6.7         3.1          5.6         2.4 virginica
## 142          6.9         3.1          5.1         2.3 virginica
## 143          5.8         2.7          5.1         1.9 virginica
```

```
## 144          6.8         3.2          5.9         2.3 virginica
## 145          6.7         3.3          5.7         2.5 virginica
## 146          6.7         3.0          5.2         2.3 virginica
## 147          6.3         2.5          5.0         1.9 virginica
## 148          6.5         3.0          5.2         2.0 virginica
## 149          6.2         3.4          5.4         2.3 virginica
## 150          5.9         3.0          5.1         1.8 virginica
```

```
nrow(iris)
```

```
## [1] 150
```

```
ncol(iris)
```

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

```
dim(iris)
```

```
## [1] 150   5
```

```
unique(iris$Species)
```

```
## [1] setosa     versicolor virginica
## Levels: setosa versicolor virginica
```

```
table(iris$Species)
```

```
##
##     setosa versicolor  virginica
##         50         50         50
```

2.5 List

2.5.1 创建一个列表

List（列表）是 R 语言中最灵活的数据结构，一个 List 里面可以包含各种不同的数据结构，甚至可以再嵌套一个 List。

一个简单的例子如下：

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
str(x)
```

```
## List of 4
##  $ : int [1:3] 1 2 3
##  $ : chr "a"
##  $ : logi [1:3] TRUE FALSE TRUE
##  $ : num [1:2] 2.3 5.9
```

list 对于我们管理一些有层次结构的数据很有帮助：

```
john <- list(Name = "John", Birthplace = "America", Hobby = c("Movies", "Soccer"))
paul <- list(Name = "Paul", Birthplace = "Autrialia", Hobby = c("Soccer"))
chris <- list(Name = "Chris", Birthplace = "England", Hobby = c("Basketball"))

members <- list(ClassA = list(John = john, Paul = paul), ClassB = list(Chris = chris
str(members)
```

```
## List of 2
##  $ ClassA:List of 2
##   ..$ John:List of 3
##   .. ..$ Name      : chr "John"
##   .. ..$ Birthplace: chr "America"
##   .. ..$ Hobby     : chr [1:2] "Movies" "Soccer"
##   ..$ Paul:List of 3
##   .. ..$ Name      : chr "Paul"
##   .. ..$ Birthplace: chr "Autrialia"
##   .. ..$ Hobby     : chr "Soccer"
##  $ ClassB:List of 1
##   ..$ Chris:List of 3
##   .. ..$ Name      : chr "Chris"
##   .. ..$ Birthplace: chr "England"
##   .. ..$ Hobby     : chr "Basketball"
```

2.5.2 列表元素访问

有几个不同的方式来访问 list 中的元素。

```
l1 <- list(a=1,b=2,c=3)
l1$a
```

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

```
l1["a"]
```

```
## $a
## [1] 1
```

```
l1[["a"]]
```

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

```
l1[c("a","b")]
```

```
## $a
## [1] 1
##
## $b
## [1] 2
```

```
names(l1) <- c("A","B","C")
```

另外，在 list 中利用 [[和 $ 访问元素时，R 允许元素名称部分匹配。

```
x <- list(aardvark = 1:5)
x$a
```

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

```
x[["a"]]
```

```
## NULL
```

```
x[["a", exact = FALSE]]
```

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

我们也可以通过 $ 随时添加新的元素到 list 中。

```
l <- list(a=c(1,2),b=c(2,3,4),c="hello")
l$d <- c(T, T, F)
str(l)
```

```
## List of 4
##  $ a: num [1:2] 1 2
##  $ b: num [1:3] 2 3 4
##  $ c: chr "hello"
##  $ d: logi [1:3] TRUE TRUE FALSE
```

我们还可以通过给 list 中的元素赋空值 (NULL) 来删除元素。

```
l$d <- NULL
str(l)
```

```
## List of 3
##  $ a: num [1:2] 1 2
##  $ b: num [1:3] 2 3 4
##  $ c: chr "hello"
```

实际上，随之你对 R 了解的不断深入，会惊奇的发现许多 R 的结果都是用 list 来表示的，并且可以直接提取出丰富的结果。

```
model <- lm(mpg ~ wt, data = mtcars)
str(model)
model$coefficients
```

3 练习

1. 请用向量运算的方法生成下列一组数：

$$2^1, 2^2, 2^3, 2^4, 2^5$$

2. 请用矩阵方法解下列方程组：

$$2x + 3y = 4$$

$$3x + 2y = 5$$

4 参考资料

以上我们介绍了 R 语言中最基础的数据类型和结构，但实际上 R 语言对于数据结构的设计是很精巧而复杂的。如果你想更深入的了解相关的内容，推荐阅读以下资料：

1. learnR, Basic Objects: http://renkun.me/learnR/basic-objects/index.html
2. RClub 2014, Data Structures: https://github.com/wise-r/R-Training/blob/master/Chapter02/2.data_structures.Rmd
3. RClub 2014, Subsetting: https://github.com/wise-r/R-Training/blob/master/Chapter03/3.1.subsetting.Rmd
4. Advanced R, Data Structures: http://adv-r.had.co.nz/Data-structures.html
5. Advanced R, Subsetting: http://adv-r.had.co.nz/Subsetting.html
6. Try R：http://tryr.codeschool.com/