FDU 数理统计 1. R 语言

1. 创建数据集

1.1 数据结构

数据框 (data frame) 是 R 中用于存储数据的一种结构:
行表示**观测** (observation)，**列**表示**变量** (variable).
它是我们用来存储数据集的主要数据结构.

1.1.1 向量

向量是用于存储数值型、字符型、逻辑型数据的一维数组，由 `c()` 函数创建.

- **数值型 (numeric):**
  - 双精度型 (double): `numeric_vector <- c(1.1, 2.2, 3.3, 4.4)`
  - 整数型 (integer): `integer_vector <- c(1L, 2L, 3L, 4L, 5L)`
  - 复数型 (complex): `complex_vector <- c(1+2i, 3+4i, 5+6i)`
- **字符型 (character):**
  `character_vector <- c("Hello", "World", "R Language")` (使用单引号也可以)
- **逻辑型 (logical):** `logical_vector <- c(TRUE, FALSE)`

同一向量中无法混杂不同类型的数据，
若出现混杂，则 `c()` 会将低级别的类型强制转换为高级别的类型，从高到低为:

1. 列表型 (list)
2. 字符型 (character)
3. 复数型 (complex)
4. 双精度型 (double)
5. 整型 (integer)
6. 逻辑型 (logical)

如何访问向量中的元素:

```r
# 创建向量
v <- c(1, 2, 3, 4, 5)

# 使用下标访问
element_third <- v[3]
# 得到 3

# 使用负下标排除
v_without_second <- v[-2]
# 得到 1, 3, 4, 5

# 使用逻辑向量访问
v_odd_index <- v[c(TRUE, FALSE, TRUE, FALSE, TRUE)]
# 得到 1, 3, 5

# 使用序列号访问
v_first_to_third <- v[1:3]
# 得到 1, 2, 3
```

```r
# 使用which()函数
v_greater_than_3 <- v[which(v > 3)]
# 得到 4, 5

# 使用名称访问
names(v) <- c("a", "b", "c", "d", "e")
v_named_c <- v["c"]
# 得到 3
```

1.1.2 矩阵

矩阵是一个二维数组，所有元素具有相同的数据类型，可通过 `matrix()` 函数创建.
`matrix(vector, nrow, ncol, byrow, dimnames)`

- `vector`: 矩阵元素构成的向量
- `nrow` 和 `ncol` 用于指定行、列维数
- `byrow` 指定按行填充 (TRUE) 或按列填充 (FALSE)，默认按列填充
- `dimnames` 包含了可选的、以字符型向量表示的行名、列名 = list(rownames, colnames)

示例:

```r
cells = c(1:4)
rnames = c("R1", "R2")
cnames = c("C1", "C2")
mymatrix = matrix(cells, nrow = 2, ncol = 2, byrow = FALSE, dimnames =
list(rnames, cnames))
print(mymatrix)
```

我们创建了一个 2×2 含行、列名的矩阵，并按列进行填充.
结果:

```
    C1 C2
R1  1  3
R2  2  4
```

如何访问矩阵中的元素:

```r
# 创建一个 4x5 矩阵, 元素按列填充
m <- matrix(1:20, nrow = 4, ncol = 5)

# 访问单个元素, 例如 (2, 4) 元素
element_2_4 <- m[2, 4]

# 访问一行或一列, 例如第3行、第3列
elements_row_3 <- m[3, ]
elements_col_3 <- m[, 3]

# 访问多个特定元素, 例如第1和第3行的第2和第5列的元素
specific_elements <- m[c(1, 3), c(2, 5)]

# 使用逻辑索引访问元素
elements_greater_than_15 <- m[m > 15]

# 使用名称访问元素
```

```r
# 为矩阵的行和列命名
rownames(m) <- c("Row1", "Row2", "Row3", "Row4")
colnames(m) <- c("Col1", "Col2", "Col3", "Col4", "Col5")

# 使用行名和列名访问第2行第3列的元素
element_named <- m["Row2", "Col3"]
```

1.1.3 数组

数组是矩阵的推广，其所有元素具有相同的数据类型，可通过 `array()` 函数创建.
`array(vector, dimensions, dimnames)`

- `vector`: 数组元素构成的向量
- `dimensions`: 一个指定数组维数的数值型向量
- `dimnames`: 一个包含了可选的、各维度名称标签的列表

示例:

```r
dim1 = c("A1", "A2")
dim2 = c("B1", "B2", "B3")
dim3 = c("C1", "C2")
myarray = array(1:12, c(2, 3, 2), dimnames = list(dim1, dim2, dim3))
print(myarray)
```

结果:

```
, , C1

   B1 B2 B3
A1  1  3  5
A2  2  4  6

, , C2

   B1 B2 B3
A1  7  9 11
A2  8 10 12
```

1.1.4 数据框

数据框的概念较矩阵来说更为一般，
它允许我们将多种类型的数据按列组织在一起，每列可以有自己的数据类型，可由 `data.frame()` 创建.
`data.frame(col1, col2, ..., row.names, check.rows, )`

- `col`: 每个向量将成为数据框的一列. 向量的长度必须相同，或者能够被循环扩展到最长的向量的长度.
- `row.names`: 一个可选参数，用于指定行的名称，如果省略，行名默认为连续的整数.
- `check.rows`: 一个逻辑值，如果设置为 `TRUE`，R 会检查所有输入向量的长度是否相等.
- `check.names`: 逻辑值，用于指示是否检查列名的有效性，并在必要时修改列名以使其符合 R 的命名规则. 默认为 `TRUE`.
- `stringsAsFactors`: 用于指示字符向量是否应该被转换为因子 (factor)，默认值为 `FALSE`，意味着字符向量默认不转换为因子.

示例:

```r
df <- data.frame(
    ID = 1:4,
    Name = c("Alice", "Bob", "Charlie", "David"),
    Score = c(90, 85, 88, 95),
    Passed = c(TRUE, TRUE, TRUE, TRUE)
)
print(df)
```

结果:

```
    ID  Name    Score   Passed
1   1   Alice   90      TRUE
2   2   Bob     85      TRUE
3   3   Charlie 88      TRUE
4   4   David   95      TRUE
```

这与我们通常设想的数据集的形态较为接近.

如何访问数据框中的元素:
(以刚刚创建的数据框 `df` 为例)

```r
# 1. 使用列名访问列
# 选择单列, 返回一个向量
scores <- df$Score
# 选择多列, 使用c()函数, 返回一个新的数据框
subset_df <- df[c("Name", "Score")]

# 2. 使用索引 [行, 列] 访问元素
# 选择第二行的Score列元素
specific_element <- df[2, "Score"]
# 选择第1到第3行的所有列
rows_subset <- df[1:3, ]
# 选择Name和Score列的所有行
cols_subset <- df[, c("Name", "Score")]

# 3. 使用逻辑索引
# 选择Score大于88的所有行
high_scores <- df[df$Score > 88, ]

# 4. 使用subset()函数
# 使用subset()选择Score大于88的行
high_scores_subset <- subset(df, Score > 88)

# 5. 使用dplyr包的选择函数
# 安装并加载dplyr包
# install.packages("dplyr")
library(dplyr)
# 使用select()函数选择列
selected_cols <- select(df, Name, Score)
# 使用filter()函数基于条件选择行
filtered_rows <- filter(df, Score > 88)
```

在每个变量名前都键入一次数据框名 `df` 可能会令人生厌,
我们可以联合使用 `attach()` 和 `detach()` 或单独使用 `with()` 来简化代码.

- ① `attach()` & `detach()`
  `attach(df)` 可将数据框 `df` 添加到 R 的搜索路径中，
  而 `detach(df)` 可将数据框 `df` 从搜索路径中移除.
  (实际上 `detach(df)` 是可以省略的，但它应当被例行地放入代码，这是一个好的编程习惯)

  不幸的是，当存在名称相同的对象时，这种方法的局限性就很明显了.
  假设数据框 `df` 中有名为 `a` 的变量，
  但如果在我们用 `attach()` 将 `df` 添加到搜索路径之前，环境已经有另一个名为 `a` 的变量了，
  那么已有的 `a` 作为原始变量将取得优先权，
  于是数据框 `df` 中的变量 `a` 会被屏蔽 (masked).

- ② `with()`
  我们可以在 `with()` 中指定数据框 `df`，这样就无须担心名称冲突了.

```r
# 创建数据框
df <- data.frame(
    ID = 1:4,
    Name = c("Alice", "Bob", "Charlie", "David"),
    Score = c(90, 85, 88, 95),
    Passed = c(TRUE, TRUE, TRUE, TRUE)
)

# 1. 使用 with() 函数更新分数，并将结果存储在新变量中
df <- with(df, {
    Score <- Score + 5  # 将每个人的分数增加 5 分
    return(df)  # 返回更新后的数据框
})

# 2. 使用 with() 函数计算平均分数
average_score <- with(df, mean(Score))

# 3. 使用 with() 函数进行条件筛选
passed_students <- with(df, df[Passed == TRUE, ])
```

  然而 `with()` 的局限性在于，赋值只在 `with()` 的括号内生效.

```
>   with(df, {
        average_score <- with(df, mean(Score))
        average_score
    })
>   [1] 89.5
>   average_score
>   Error: object 'average_score' not found
```

  如果我们需要创建在 `with()` 结构以外存在的对象，
  使用**特殊赋值符** `<<-` 代替**标准赋值符** `<-` 即可，
  它可将 `with()` 中的对象保存到 `with()` 以外的全局环境中.

```
>   with(df, {
        average_score <<- with(df, mean(Score))
        average_score
    })
>   [1] 89.5
>   average_score
>   [1] 89.5
```

1.1.5 因子

类别变量 (categorical variables)，又称**因子** (factor)，可归结为三类:

- **名义型类别变量 (nominal):**
  例如糖尿病类型 `Diabetes(Type1, Type2)`，其中 `Type1` 和 `Type2` 无顺序关系;
- **有序型类别变量 (ordinal):**
  例如病情 `Status(poor, improved, excellent)`;

函数 `factor()` 以一个整数向量的形式存储类别值，
有多少个互不相同的类别，就有多少个整数值，其取值范围为 $[1, \ldots, k]$
同时将一个由字符串 (原始值) 构成的内部向量映射到这些整数上.

- 对于无序因子 `diabetes <- factor(c("Type1", "Type2", "Type1"))`
  它将被存储为 `(1, 2, 1)`，并在内部将其关联为:

$$\begin{cases} 1 = \text{Type1} \\ 2 = \text{Type2} \end{cases}$$

- 对于有序因子，我们通过设置 `order = TRUE` 和 `levels` 选项来定义.
  例如对于有序因子 `status`:

  ```
  status <- c("Poor", "Improved", "Excellent", "Poor")
  status <- factor(
      status,
      order = TRUE,
      levels = c("Poor", "Improved", "Excellent")
  )
  ```

  各水平的赋值将为:

$$\begin{cases} 1 = \text{Poor} \\ 2 = \text{Improved} \\ 3 = \text{Excellent} \end{cases}$$

  于是 `status` 被存储为 `(1, 2, 3, 1)`.
  若不设置 `levels` 选项，则字符型向量的因子水平默认依字母顺序创建，即:

$$\begin{cases} 1 = \text{Excellent} \\ 2 = \text{Improved} \\ 3 = \text{Poor} \end{cases}$$

1.1.6 列表

列表是对象的有序集合，它允许我们整合若干 (可能无关) 的对象，
其中的对象可以是任何数据结构.
我们使用 `list()` 创建列表:
`mylist <- list(name1 = object1, name2 = object2, ···)`
示例:

```
g <- "My First List"
h <- c(25, 26, 18, 39)
j <- matrix(1:10, nrow=5)
k <- c("one", "two", "three")
mylist <- list(title=g, age=h, j, k)
mylist
```

结果:

```
$title
[1] "My First List"

$age
[1] 25 26 18 39

[[3]]
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10

[[4]]
[1] "one"   "two"   "three"
```

调用:

`mylist[[2]]` 和 `mylist[["ages"]]` 调用的都是 `mylist` 的第 2 列 (即名为 `age` 的列)

```
> mylist[[2]]
[1] 25 26 18 39
> mylist[["age"]]
[1] 25 26 18 39
```

提醒程序员注意的一些事项

经验丰富的程序员通常会发现R语言的某些方面不太寻常。以下是这门语言中你需要了解的一些特性。

- 对象名称中的句点（.）没有特殊意义。但美元符号（$）却有着和其他语言中的句点类似的含义，即指定一个对象中的某些部分。例如，A$x是指数据框A中的变量x。
- R不提供多行注释或块注释功能。你必须以#作为多行注释每行的开始。出于调试目的，你也可以把想让解释器忽略的代码放到语句if(FALSE){...}中。将FALSE改为TRUE即允许这块代码执行。
- 将一个值赋给某个向量、矩阵、数组或列表中一个不存在的元素时，R将自动扩展这个数据结构以容纳新值。举例来说，考虑以下代码：

```
> x <- c(8, 6, 4)
> x[7] <- 10
> x
[1]  8  6  4 NA NA NA 10
```

通过赋值，向量x由三个元素扩展到了七个元素。

x <- x[1:3]会重新将其缩减回三个元素。

- R中没有标量。标量以单元素向量的形式出现。
- R中的下标不从0开始，而从1开始。在上述向量中，x[1]的值为8。
- 变量无法被声明。它们在首次被赋值时生成。

要了解更多，参阅John Cook的优秀博文“R programming for those coming from other languages”（www.johndcook.com/Rlanguagefor_programmers.html）。

那些正在寻找编码风格指南的程序员不妨看看“Google’R Style Guide”[1]（http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html。

- R programming for those coming from other languages

- Google's R Style Guide

1.2 实用函数