《R 与 tidyverse——数据分析入门》练习题答案

石天熠

2019-07-31

Contents

Chapter 1

R 和 RStudio 介绍和安装教程

Chapter 2

R 中的数据，逻辑，和函数

2.1 基础

2.1.1 向量取子集和逻辑运算

2.1.1.1 问题

```
x <- c(3, 4, 6, 1, NA, 8, 2, 5, NA, 9, 7)
```

`x[-c(1, 3)]`, `x[(length(x)-3):length(x)]`, `x[x < 5]`, `x[!(x < 5)]` 的计算结果分别是？如何得到（不包含 NA 的）所有小于 5 的值的向量？

2.1.1.2 答案

加上“且不为 NA”的条件即可：

```
x[x < 5 & !is.na(x)]
```

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

2.1.2 转换年份到世纪

2.1.2.1 问题

写一个名为 `as.century()` 的函数，把存储着年份的向量，比如 `years <- c(2014, 1990, 1398, 1290, 1880, 2001)`，转换成对应的世纪（注意，19XX 年是 20 世纪），像这样：

```
as.century(c(2014, 1990, 1398, 1290, 1880, 2001))
```

```
#> [1] 21 20 14 13 19 21
```

2.1.2.2 答案

```
as.century <- function(x) x%/%100 + 1
```

2.1.3 分割时间为时和分

2.1.3.1 问题

写名为 hour(), minute() 的函数，使得：

```
times <- c(0512, 0719, 2358, 0501)
hour(times)
```

```
#> [1]  5  7 23  5
```

```
minute(times)
```

```
#> [1] 12 19 58  1
```

2.1.3.2 答案

分别为求除以 100 的整数商和余数。

```
hour <- function(x) x%/%100
minute <- function(x) x%%100
```

2.1.4 转换年份到世纪

2.1.4.1 问题

写一个名为 as.century() 的函数，把存储着年份的向量，比如 years <- c(2014, 1990, 1398, 1290, 1880, 2001)，转换成对应的世纪（注意，19XX 年是 20 世纪），像这样：

```
years_1 <- c(2014, 1990, 1398, 1290, 1880, 2000)
as.century(years_1)
```

```
#> [1] 21 20 14 13 19 21
```

2.1.4.2 答案

```
as.century <- function(years) {
  centuries <- floor(years/100+1)
  return(centuries)
}
```

2.1.5 斐波那契数列

斐波那契数列是指 $F = [1, 1, 2, 3, 5, 8, ...]$[1]，其中：

[1]也有 $F_0 = 0, F_1 = 1$ 的说法，但是为了方便我们不用这个定义。

- $F_1 = 1,\ F_2 = 1$
- 从 F_3 开始，$F_i = F_{i-2} + F_{i-1}$

2.1.5.1 问题

创建一个函数名为 `fibon()` 的函数，使得 `fibon(i)`：

- 当 $i \in \mathbb{Z}^+$ 时，返回向量 $[F_1, F_2, \ldots, F_i]$
- 当 $i \notin \mathbb{Z}^+$ 时，返回" 请输入一个正整数作为 `fibon()`的参数。"[2]

提示：

- 虽然在 R 中整数用 1L, 2L 等表示，用户在被指示 “输入整数” 的时候很有可能输入的是 2 而不是 2L. 2 是否等于 2L? 如果是，如何利用它检测输入的是否是整数?（2 和 2L 都要被判定为 “是整数”）
- 斐波那契数列前两位是定义，从第三位开始才是计算得出的。

使用例：

```
fibon(10); fibon(-5)
```

```
#>  [1]  1  1  2  3  5  8 13 21 34 55
```

```
#> [1] "请输入一个正整数。"
```

2.1.5.2 答案

首先，因为 $F_1 = 1,\ F_2 = 1$ 是定义，所以在函数中创建一个向量，名为 `F`，存储前两项：

```
fibon <- function(){
  F <- c(1, 1)
}
```

从 F_3 开始，$F_i = F_{i-2} + F_{i-1}$，翻译成 R 代码就是：

```
F[i] <- F[i-2] + F[i-1]
```

把它放进 while 循环里：

```
fibon <- function(len = 10){
  F <- c(1, 1)
  i = 3 ## 从 3 开始
  while (i <= len) { ## 到指定的数值结束
    F[i] <- F[i-2] + F[i-1] ## 每次计算并加入第 i 个元素
  }
}
```

最后加入正整数的判别，然后整合一下，完成：

```
fibon <- function (len = 10) {
  if (len == as.integer(len) & len > 0) { #
    F <- c(1, 1) ## 前两项需要定义
    i <- 3 ## 从第三项开始计算
    while (i <= len) {
      F[i] <- F[i-2] + F[i-1]
      i <- i+1 ## R 中不可以使用 `i += 1`或者 `i++`
```

[2] 虽然正规的做法是制造一个错误/警告

```
    }
    return(F[1:len])
  } else {
    return(" 请输入一个正整数。")
  }
}
```

2.2 进阶

2.2.1 函数的使用

2.2.1.1 问题

`seq(0, 20, 5)`, `seq(by = 5, 0, 20,` 和 `seq(by = 5, 0, y = 30, 20)` 的结果分别是什么?

2.2.1.2 答案

有命名的实际参数优先，未命名的实际参数按照形式参数定义时的顺序。若有命名实参的名字在形参中无匹配，将被放进`...` 这个大箩筐。

形式参数定义时的顺序，可以通过帮助文档（`?seq`）获知。它是 `from`, `to`, `by`, `length.out`, `along.with`, `....`

`seq(0, 20, 5)` 的三个实参都未命名。因此按照形参的顺序，即 `seq(from = 0, to = 20, by = 5)`.

`seq(by = 5, 0, 20` 有一个命名实参（`by = 5`），剩余未命名的 `0, 20` 还是按顺序排，因此还是 `seq(from = 0, to = 20, by = 5)`.

`seq(by = 5, 0, y = 30, 20)` 多出了一个 `y`, 它被放进`...`, 还是剩余未命名的 `0, 20`, 因此仍然是 `seq(from = 0, to = 20, by = 5)`.

2.2.2 创建一个有序数列

分别用 `sapply()`, `rep()`, 和 `rapply()` 创建这样一个数列（向量）:

$$x = (1 \times 1 \times 1, 1 \times 1 \times 2 \ldots, 40 \times 50 \times 59, 40 \times 50 \times 60)$$

2.2.2.1 基于 sapply()

先创建 `m, N` 相乘的矩阵，再创建 `l` 乘这个矩阵形成的三维数组。最后化简为向量。

```
MN <- sapply(M, function(m){m*N})
LMN <- sapply(L, function(l){l*MN})
result <- as.vector(LMN)
```

或

```
MN <- sapply(M, "*", N)
LMN <- sapply(L, "*", MN)
result <- as.vector(LMN)
```

2.2.2.2 基于 rep()

```
MN <- rep(N, length(M)) * rep(M, each = length(N))
result <- rep(MN, length(L)) * rep(L, each = length(MN))
```

或（基于 recycling rule, 回收规则）

```
MN <- N * rep(M, each = length(N))
result <- MN * rep(L, each = length(MN))
```

2.2.2.3 基于 rapply()

```
MN <- rapply(as.list(M), "*", N)
result <- rapply(as.list(L), "*", MN)
```

2.2.3 质数表

创建一个 pr() 函数，使 pr(i) 得到 $(2, 3, 5, 7, 11, ...n)$, 其中 i 为大于或等于 3 的整数，n 为小于 i 的最大质数。

```
pr(100)
```

```
#>  [1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83
#> [24] 89 97
```

你能想到几种方法？哪一种更快？为什么？

2.2.3.1 使用 for 循环

```
pr.for <- function(i){
  pnums <- 2L
  for (num in 3:i) {
    isPrime <- TRUE
    for (pnum in pnums) {
      if (pnum > sqrt(num)) {
        break
      } else if (num %% pnum == 0) {
        isPrime <- FALSE
        break
      }
    }
    if (isPrime) {
      pnums <- c(pnums, num)
    }
  }
  return(pnums)
}
```

2.2.3.2 使用 while 循环

```
pr.while <- function(i){
  pnums <- 2L
  for (num in 3:i) {
    isPrime <- TRUE
    j = 1
    while (pnums[j] <= sqrt(num)){
      if (num %% pnums[j] == 0) {
        isPrime <- FALSE
        break
      }
      j = j+1
    }
    if (isPrime) {
      pnums <- c(pnums, num)  ## common mistake
    }
  }
  return(pnums)
}
```

2.2.3.3 使用 any

```
pr.any <- function(i){
  pnums <- 2L
  for (num in 3:i) {
    isPrime <- !any(num %% pnums[pnums <= sqrt(num)] == 0)
    if (isPrime) {
      pnums <- c(pnums, num)
    }
  }
  return(pnums)
}
```

2.2.3.4 使用 all

```
pr.all <- function(i){
  pnums <- 2L
  for (num in 3:i) {
    isPrime <- all(num %% pnums[pnums <= sqrt(num)] != 0)
    if (isPrime) {
      pnums <- c(pnums, num)
    }
  }
  return(pnums)
}
```

2.2.3.5 使用 '%in%

```
pr.in <- function(i){
  pnums <- 2L
  for (num in 3:i) {
    isPrime <- TRUE
    test <- pnums[pnums<=floor(sqrt(num))]
    isPrime <- !(0 %in% (num %% test))
    if (isPrime) {
      pnums <- c(pnums, num)
    }
  }
  return(pnums)
}
```

2.2.3.6 使用 lapply()

```
pr.lapply <- function(i){
  pnums <- 2
  lapply(3:i, function(x){
    test <- pnums[pnums <= floor(sqrt(x))]
    if (all(x%%test != 0)) pnums <<- c(pnums, x)
  })
  return(pnums)
}
```

2.2.3.7 使用 purrr 中的 map()

```
library(tidyverse)
```

```
pr.map <- function(i){
  pnums <- 2L
  map(3:i, ~{
    test <- pnums[pnums<=floor(sqrt(.))]
    if (all(.%%test != 0)) pnums <<- c(pnums, .)
    })
  return(pnums)
}
```

```
pr.map1 <- function(i){
  pnums <- 2L
  map(3:i, ~{
    test <- pnums[pnums<=floor(sqrt(.))]
    if (!(0 %in% (.%%test))) pnums <<- c(pnums, .)
  })
  return(pnums)
}
```

```
pr.map2 <- function(i){
  pnums <- 2
  map(3:i, ~{
    test <- pnums[pnums<=floor(sqrt(.))]
    if (is.na(match(0, .%%test))) pnums <<- c(pnums, .)
  })
  return(pnums)
}
```

2.2.3.8 整合

```
pr.funcs.names <- c('pr.for', 'pr.while', 'pr.any', 'pr.all', 'pr.in', 'pr.lapply', 'pr.map', 'pr.map
pr.funcs <- lapply(pr.funcs.names, match.fun)
names(pr.funcs) <- pr.funcs.names
```

2.2.3.9 小检验

```
t(sapply(pr.funcs, function(x) x(100)))
```

```
#>           [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
#> pr.for       2    3    5    7   11   13   17   19   23    29    31    37
#> pr.while     2    3    5    7   11   13   17   19   23    29    31    37
#> pr.any       2    3    5    7   11   13   17   19   23    29    31    37
#> pr.all       2    3    5    7   11   13   17   19   23    29    31    37
#> pr.in        2    3    5    7   11   13   17   19   23    29    31    37
#> pr.lapply    2    3    5    7   11   13   17   19   23    29    31    37
#> pr.map       2    3    5    7   11   13   17   19   23    29    31    37
#> pr.map1      2    3    5    7   11   13   17   19   23    29    31    37
#> pr.map2      2    3    5    7   11   13   17   19   23    29    31    37
#>           [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22]
#> pr.for       41    43    47    53    59    61    67    71    73    79
#> pr.while     41    43    47    53    59    61    67    71    73    79
#> pr.any       41    43    47    53    59    61    67    71    73    79
#> pr.all       41    43    47    53    59    61    67    71    73    79
#> pr.in        41    43    47    53    59    61    67    71    73    79
#> pr.lapply    41    43    47    53    59    61    67    71    73    79
#> pr.map       41    43    47    53    59    61    67    71    73    79
#> pr.map1      41    43    47    53    59    61    67    71    73    79
#> pr.map2      41    43    47    53    59    61    67    71    73    79
#>           [,23] [,24] [,25]
#> pr.for       83    89    97
#> pr.while     83    89    97
#> pr.any       83    89    97
#> pr.all       83    89    97
#> pr.in        83    89    97
#> pr.lapply    83    89    97
#> pr.map       83    89    97
#> pr.map1      83    89    97
#> pr.map2      83    89    97
```

2.2.3.10 小测速

```
n1 <- 100000
map(pr.funcs, ~ .(n1))
```

2.2.3.11 速度绘图

我们可以给各函数寻找不同数量的质数的速度绘图:

```
lims <- seq(10000,100000,5000)

timeData <- tibble(func = character(0), lim = integer(0), time = double(0))

for(lim in lims) {
  for(pr in pr.funcs.names){
    time <- system.time(match.fun(pr)(lim))[3]
    timeData <- add_row(timeData,
                        func = pr,
                        lim = lim,
                        time = time)
  }
}
```

执行上面的代码会花很多时间，因此不妨把这宝贵的数据记录下来，方便日后使用（可以跳过这一步）:

```
readr::write_csv(timeData, "src/prime_list_timeData.csv")
timeData <- read_csv("src/prime_list_timeData.csv")
```

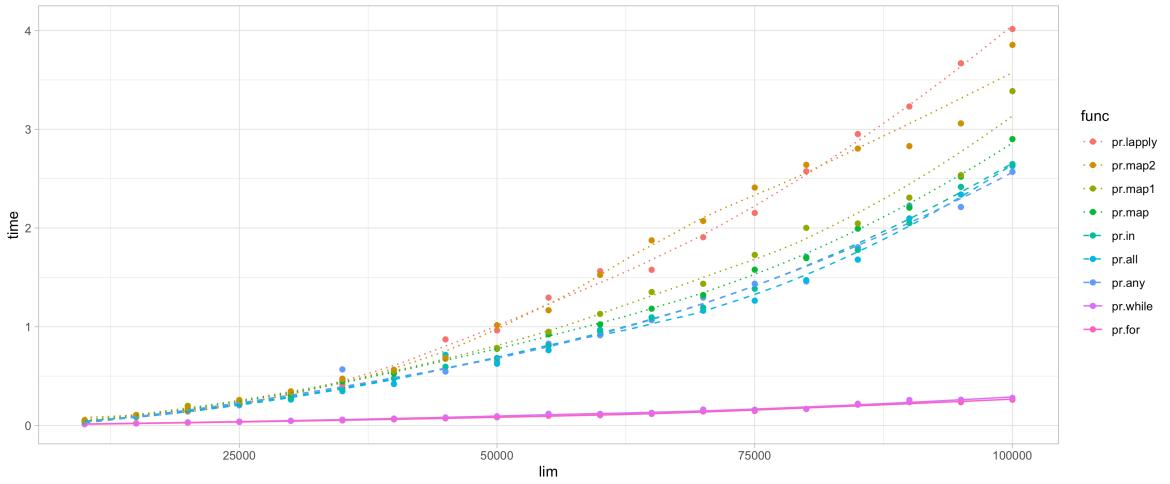
以下是绘图代码，前面两个步骤是为了看图例的时候更方便。最后一个步骤是保存 png，可以跳过。

```
timeData1 <- mutate(timeData, func = fct_reorder(func, time, .fun = function(x) -max(x)))

types <- c(rep("dotted", 4), rep("dashed", 3), rep("solid", 2))

ggplot(data = timeData1, mapping = aes(x=lim, y = time, color = func))+
  geom_point()+
  geom_smooth(se=FALSE, method = 'loess', formula = 'y ~ x', size = 0.5, aes(linetype = func))+
  theme_light()+
  scale_linetype_manual(values = types)

ggsave("prime_funcs.png", height = 5, width = 12, dpi = 300)
```



可以看到，用实线绘制的 `for` 和 `while` 循环最快，虚线代表的`%in%`, `all`, `any` 其次，而点线代表的 `apply()` 族函数最慢。

寻找质数本质上是一个迭代计算，无法用向量化的方法去完成，强行使用 `apply()` 族函数反而会加重负担。

2.2.4 判断是否是质数

写一个函数，判断一个数是否是质数。

```
is.prime <- function(n) n == 2L || all(n %% 2L:ceiling(sqrt(n)) != 0)
```

2.3 挑战

2.3.1 伪·OOP

2.3.1.1 问题

使用且仅使用 `function()`, `c()`, `list()`, `paste()`, `print()` 函数，`<-`, `$`, `==` 符号，和 `if`，实现这样的效果:

`Pigeon()`, `Turtle()`, `Cat()` 分别创建一只鸽子，一只乌龟和一只猫（即产生一个 list，各自的元素展示如下）:

```
Guoguo <- Pigeon("Guoguo")
Felix <- Cat("Felix", "TRUE")
Kazuya <- Turtle("Kazuya")

str(Guoguo)
```

```
#> List of 5
#>  $ name         : chr "Guoguo"
#>  $ common_name  : chr "pigeon"
#>  $ binomial_name: chr "Columba livia"
#>  $ speak        : chr "coo"
#>  $ greet        :function (time = "not_specified")
#>   ..- attr(*, "srcref")= 'srcref' int [1:8] 6 12 12 3 12 3 6 12
#>   .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7ff051c41100>
```

```
str(Kazuya)
```

```
#> List of 5
#>  $ name         : chr "Kazuya"
#>  $ common_name  : chr "turtle"
#>  $ binomial_name: chr "Trachemys scripta elegans"
#>  $ speak        : logi NA
#>  $ greet        :function (time = "not_specified")
#>   ..- attr(*, "srcref")= 'srcref' int [1:8] 6 12 12 3 12 3 6 12
#>   .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7ff051c41100>
```

```
str(Felix)
```

```
#> List of 6
#>  $ name         : chr "Felix"
#>  $ common_name  : chr "cat"
#>  $ binomial_name: chr "Felis catus"
#>  $ speak        : chr "meow"
#>  $ greet        :function (time = "not_specified")
#>   ..- attr(*, "srcref")= 'srcref' int [1:8] 6 12 12 3 12 3 6 12
#>   .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7ff051c41100>
#>  $ sterilized   : chr "TRUE"
```

注意, 鸽子, 乌龟和猫都有名字 (`name`), 通称 (`common_name`), 学名 (`binomial_name`), 和打招呼函数 (`greet()`); 此外, 乌龟没有叫声 `speak()`, 猫额外地有绝育 `sterilized` 信息。可以这样查看信息和使用打招呼函数:

```
Felix$binomial_name
```

```
#> [1] "Felis catus"
```

```
Kazuya$greet("afternoon")
```

```
#> [1] "Good afternoon, I'm a turtle and my name is Kazuya"
```

其中 `greet()` 的参数如果是 `morning`, `afternoon` 或 `evening`, 则返回`"Good < 时间段 > ..."`, 否则返回`"Hi ..."`.

此外, 另写两个仅对这些宠物使用的函数 `binomial_name()` 和 `greet()`, 使之能够这样使用:

```
binomial_name(Kazuya)
```

```
#> [1] "Trachemys scripta elegans"
```

```
greet(Guoguo)
```

```
#> [1] "Hi, I'm a pigeon and my name is Guoguo"
```

你可能需要的额外信息:

鸽子, 乌龟和猫的学名分别为 Columba livia, Trachemys scripta elegans, Felis catus.

`paste()` 函数把多个字符串拼接成一个, 其中参数 `sep` 指定连接符号, 默认为空格:

```
x <- "world"
paste("Hello", x, "Bye", x, sep = "---")
```

```
#> [1] "Hello---world---Bye---world"
```

2.3.1.2 答案

首先，把 Cat() 函数做出来，这个应该不难看懂：

```
Cat <- function(name){
  name = name
  binomial_name <- "Felis catus"
  speak <- "Meow"
  greet <- function(time = "not_specified"){
    intro <- paste("my name is", name)
    if(time == "morning") print(paste("Good morning,", intro))
    if(time == "afternoon") print(paste("Good afternoon,", intro))
    if(time == "evening") print(paste("Good evening,", intro))
    if(time == "not_specified") print(paste("Hi,", intro))
  }
  list(name = name, binomial_name = binomial_name, speak = speak, greet = greet)
}
```

你可以对 Turtle() 和 Pigeon() 做同样的事情，但是这样会产生很多重复的代码。每当有重复的代码发生时，作为一个正经的编程语言，一定有方法去消除重复。这三类动物都是宠物，因此我们可以把共同的代码写进一个 Pet() 函数中，再定义不同种类的宠物。这是 OOP 的 **inheritance** 和 **polymorphism** 的实现：

```
Pet <- function(name = NA, common_name = NA, binomial_name = NA, speak = NA){
  name <- name
  common_name <- common_name
  binomial_name <- binomial_name
  speak <- speak
  greet <- function(time = "not_specified"){
    intro <- paste("I'm a", common_name, "and my name is", name)
    if(time == "morning") print(paste("Good morning,", intro))
    if(time == "afternoon") print(paste("Good afternoon,", intro))
    if(time == "evening") print(paste("Good evening,", intro))
    if(time == "not_specified") print(paste("Hi,", intro))
  }
  list(name = name, common_name = common_name, binomial_name = binomial_name, speak = speak, greet =
}

Pigeon <- function(name = NA){
  PetAaM <- Pet(name, "pigeon", "Columba livia", "coo")
}

Turtle <- function(name = NA){
  Pet(name, "turtle", "Trachemys scripta elegans") ## 实现 inheritance ## 龟没有叫声
}

Cat <- function(name = NA, sterilized = NA){
  sterilized <- sterilized ## 猫可能绝育 ## 新增 attribute, 实现了广义的 polymorphism
  PetAaM <- Pet(name, "cat", "Felis catus", "meow")
  CatOnlyAaM <- list(sterilized = sterilized)
  c(PetAaM, CatOnlyAaM)
}

## 实现了 Python 语境中的 polymorphism
```

```
greet <- function(pet, time = "not_specified"){
  pet$greet(time)
}

binomial_name <- function(pet){
  pet$binomial_name
}
```

搞定！没有 `class`，没有 `self`，没有 `__init__`，*it just works.*