

R 语言编程：基于 tidyverse

第 07 讲 自定义函数

张敬信

2022 年 2 月 10 日

哈尔滨商业大学

- 编程中的函数，是用来实现某个功能，其一般形式为：

(返回值 1, ..., 返回值 m) = 函数名 (输入 1, ..., 输入 n)

- 你只要把输入给它，它就能在内部进行相应处理，把你想要的返回值给你。
- 这些输入和返回值，在函数定义时，都要有固定的类型（模具）限制，叫作形参（形式上的参数）；在函数调用时，必须给它对应类型的具体数值，才能真正的去做处理，这叫作实参（实际的参数）。
- 定义函数就好比创造一个模具，调用函数就好比用模具批量生成产品。
- 使用函数最大的好处，就是将实现某个功能，封装成模具，从而可以反复使用。这就避免了写大量重复的代码，程序的可读性也大大加强。

一. 自定义函数

1. 如何自定义函数

- R 中自定义函数的一般格式为：

```
函数名 = function(输入 1, ..., 输入 n) {  
  函数体  
  return(返回值)  
}
```

注意，return 并不是必需的，默认函数体最后一行的值作为返回值。

案例：自定义函数实现把百分制分数转化为五级制分数的功能

第一步，分析输入和输出，设计函数外形

- 输入有几个，分别是什么，适合用什么数据类型存放；
- 输出有几个，分别是什么，适合用什么数据类型存放。

本问题，输入有 1 个：百分制分数，数值型；输出有 1 个：五级制分数，字符串

- 然后就可以设计自定义函数的外形：

```
Score_Conv = function(score) {  
# 实现将一个百分制分数转化为五级分数  
# 输入参数：score 为数值型，百分制分数  
# 返回值：res 为字符串型，五级分数  
...  
}
```

第二步，梳理功能的实现过程

- 即前言中讲到的如何自己写代码：“分解问题 + 实例梳理 + 翻译及调试”。
- 拿一组本例（只有一个）具体的形参的值作为输入，比如 76 分，分析怎么到达对应的五级分数“良”。这依赖于对五级分数界限的选取，选定之后做分支判断即可实现。
- 复杂的功能，就需要更耐心的梳理和思考甚至借助一些算法，当然也离不开逐代码片段的调试。
- 拿一组具体的形参值作为输入，通过逐步调试，得到正确的返回值结果，这一步骤非常关键和有必要。

```
score = 76
if(score >= 90) {
  res = " 优"
} else if(score >= 80) {
  res = " 良"
} else if(score >= 70) {
  res = " 中"
} else if(score >= 60) {
  res = " 及格"
} else {
  res = " 不及格"
}
res
#> [1] " 中"
```

第三步，将第二步的代码封装到函数体

就是原样作为函数体放入函数，原来的变量赋值语句不需要了，只需要形参。

```
Score_Conv = function(score) {  
    if(score >= 90) {  
        res = " 优"  
    } else if(score >= 80) {  
        res = " 良"  
    } else if(score >= 70) {  
        res = " 中"  
    } else if(score >= 60) {  
        res = " 及格"  
    } else {  
        res = " 不及格"  
    }  
    res  
}
```

2. 调用函数

要调用自定义函数，必须要先加载到当前变量窗口（内存），有两种方法：

- 需要选中并执行函数代码，或者
- 将函数保存为同名的 `Score_Conv.R` 文件，注意勾选“Source on save”再保存，然后执行 `source("Score_Conv.R", encoding="UTF-8")`

然后就可以调用函数：

```
Score_Conv(76)
```

```
#> [1] " 中"
```


关于函数传递参数

- 要调用一个函数，比如 $f(x, y)$ ，首先要清楚其形参 x, y 所要求的类型，假设 x 要求是数值向量， y 要求是单个逻辑值。
- 那么，要调用该函数，首先需要准备好与形参类型相符的实参（同名异名均可）

```
a = c(3.56, 2.1)
```

```
b = FALSE
```

```
f(a, b)      # 同直接给值：f(c(3.56,2.1), FALSE)
```

- 调用函数时若不指定参数名，则默认是根据位置关联形参，即以 $x = a$, $y = b$ 的方式进入函数体；若指定参数名，则根据参数名关联形参，位置不再重要

```
f(y = b, x = a)      # 效果同上
```

3. 向量化改进

改进自定义函数，使其能接受向量输入，即输入多个百分制分数，能一下都转化为五级分数。

方法一：修改自定义函数

- 将输入参数设计为数值向量，函数体也要相应的修改，借助循环依次处理向量中的每个元素，就相当于再套一层 `for` 循环。

```
Score_Conv2 = function(score) {  
  n = length(score)  
  res = vector("character", n)  
  for(i in 1:n) {  
    if(score[i] >= 90) {  
      res[i] = "  优"  
    } else if(score[i] >= 80) {  
      res[i] = "  良"  
    } else if(score[i] >= 70) {  
      res[i] = "  中"  
    } else if(score[i] >= 60) {  
      res[i] = "  及格"  
    }  
  }  
}
```

```
    } else {  
      res[i] = " 不及格"  
    }  
  }  
  res  
}
```

测试函数

```
scores = c(35, 67, 100)  
Score_Conv2(scores)
```

```
#> [1] "不及格" "及格"   "优"
```

方法二：借助 apply 族或 map 系列函数

- apply 族或 map 系列函数，可实现依次“应用”某函数，到序列的每个元素上。
- 也就是说，不需要修改原函数，直接就能实现向量化操作：

```
scores = c(35, 67, 100)
map_chr(scores, Score_Conv)
#> [1] " 不及格" " 及格"   " 优"
```

4. 处理多个返回值

- R 中若自定义函数需要有多个返回值，则将多个返回值打包成一个列表（或数据框），再返回。
- 例如，自定义函数，实现计算一个数值向量的均值和标准差：

```
MeanStd = function(x) {  
  mu = mean(x)  
  std = sqrt(sum((x-mu)^2) / (length(x)-1))  
  list(mu = mu, std = std)  
}
```

```
# 测试函数
```

```
x = c(2, 6, 4, 9, 12)
```

```
MeanStd(x)
```

```
#> $mu
```

```
#> [1] 6.6
```

```
#>
```

```
#> $std
```

```
#> [1] 3.97
```

5. 默认参数值

- 有时候需要为输入参数设置默认值。比如，标准差的计算公式有两种形式，一种是总体标准差除以 n ，一种是样本标准差除以 $n - 1$ 。
- 此时，没有必要写两个版本的函数，只需要再增加一个指示参数，将用的多的版本设为默认。

```
MeanStd2 = function(x, type = 1) {  
  mu = mean(x)  
  n = length(x)  
  if(type == 1) {  
    std = sqrt(sum((x - mu) ^ 2) / (n - 1))  
  } else {  
    std = sqrt(sum((x - mu) ^ 2) / n)  
  }  
  list(mu = mu, std = std)  
}
```


测试函数

```
x = c(2, 6, 4, 9, 12)
```

```
MeanStd2(x)
```

```
# 同 MeanStd(x, 1)
```

```
#> $mu
```

```
#> [1] 6.6
```

```
#>
```

```
#> $std
```

```
#> [1] 3.97
```

```
MeanStd2(x, 2)
```

```
#> $mu
```

```
#> [1] 6.6
```

```
#>
```

```
#> $std
```

```
#> [1] 3.56
```

- 用 `type = 1` 来指示表意并不明确，可以用表意更明确的字符串来指示，这就需要用到 `switch()`，让不同的指示值 = 相应的代码块¹。

```
MeanStd3 = function(x, type = "sample") {  
  mu = mean(x)  
  n = length(x)  
  switch(type,  
    "sample" = {  
    std = sqrt(sum((x - mu) ^ 2) / (n - 1))  
    },  
    "population" = {  
    std = sqrt(sum((x - mu) ^ 2) / n)  
    })  
  list(mu = mu, std = std)  
}
```

¹因为代码块往往是多行，需要用大括号括起来，注意分支与分支之间的逗号不能少。

6. “...” 参数

- 一般函数参数只接受一个对象，比如对两个数加和的函数，给它 3 个数加和就会报错

```
my_sum = function(x, y) {  
    sum(x, y)  
}  
my_sum(1, 2)  
my_sum(1, 2, 3)      # 会报错
```

解决办法：用特殊的... 参数，它可以接受任意多个对象，并打包为一个列表传递它们：

```
dots_sum = function(...) {  
  sum(...)  
}  
dots_sum(1)  
#> [1] 1  
dots_sum(1, 2, 3, 4, 5)  
#> [1] 15
```

注：很多 R 自带函数都在用... 这样传递参数。

二. R 自带函数

- 除了自定义函数，还可以使用现成的函数：
 - 来自 base R: 可直接使用
 - 来自各个扩展包: 需加载包, 或加上包名前缀: `包名::函数名()`
- 这些函数的使用, 可以通过? 函数名 查阅其帮助, 以及查阅包主页的 Reference manual 和 Vignettes (若有)。

1. 基本数学函数

<code>round(x, digits)</code>	# IEEE 754 标准的四舍五入, 保留 <code>n</code> 位小数
<code>signif(x, digits)</code>	# 四舍五入, 保留 <code>n</code> 位有效数字
<code>ceiling(x)</code>	# 向上取整, 例如 <code>ceiling(pi)</code> 为 4
<code>floor(x)</code>	# 向下取整, 例如 <code>floor(pi)</code> 为 3
<code>sign(x)</code>	# 符号函数
<code>abs(x)</code>	# 取绝对值
<code>sqrt(x)</code>	# 求平方根
<code>exp(x)</code>	# e 的 x 次幂

<code>log(x, base)</code>	# 对 x 取以... 为底的对数, 默认以 e 为底
<code>log2(x)</code>	# 对 x 取以 2 为底的对数
<code>log10(x)</code>	# 对 x 取以 10 为底的对数
<code>Re(z)</code>	# 返回复数 z 的实部
<code>Im(z)</code>	# 返回复数 z 的虚部
<code>Mod(z)</code>	# 求复数 z 的模
<code>Arg(z)</code>	# 求复数 z 的辐角
<code>Conj(z)</code>	# 求复数 z 的共轭复数

2. 三角函数与双曲函数

<code>sin(x)</code>	# 正弦函数
<code>cos(x)</code>	# 余弦函数
<code>tan(x)</code>	# 正切函数
<code>asin(x)</code>	# 反正弦函数
<code>acos(x)</code>	# 反余弦函数
<code>atan(x)</code>	# 反正切函数
<code>sinh(x)</code>	# 双曲正弦函数
<code>cosh(x)</code>	# 双曲余弦函数
<code>tanh(x)</code>	# 双曲正切函数
<code>asinh(x)</code>	# 反双曲正弦函数
<code>acosh(x)</code>	# 反双曲余弦函数
<code>atanh(x)</code>	# 反双曲正切函数

3. 矩阵函数

<code>nrow(A)</code>	# 返回矩阵 A 的行数
<code>ncol(A)</code>	# 返回矩阵 A 的列数
<code>dim(A)</code>	# 返回矩阵 x 的维数 (几行 × 几列)
<code>colSums(A)</code>	# 对矩阵 A 的各列求和
<code>rowSums(A)</code>	# 对矩阵 A 的各行求和
<code>colMeans(A)</code>	# 对矩阵 A 的各列求均值
<code>rowMeans(A)</code>	# 对矩阵 A 的各行求均值
<code>t(A)</code>	# 对矩阵 A 转置
<code>det(A)</code>	# 计算方阵 A 的行列式
<code>crossprod(A, B)</code>	# 计算矩阵 A 与 B 的内积, <code>t(A) %*% B</code>
<code>outer(A, B)</code>	# 计算矩阵的外积 (叉积), <code>A %o% B</code>
# 取矩阵对角线元素, 或根据向量生成对角矩阵	
<code>diag(x)</code>	
<code>diag(n)</code>	# 生成 n 阶单位矩阵

```
solve(A)           # 求逆矩阵（要求矩阵可逆）
solve(A, B)        # 解线性方程组  $AX=B$ 
# 求矩阵 A 的广义逆（Moore-Penrose 逆），MASS 包
ginv(A)
eigen()            # 返回矩阵的特征值与特征向量（列）
kronecker(A, B)    # 计算矩阵 A 与 B 的 Kronecker 积
svd(A)             # 对矩阵 A 做奇异值分解， $A=UDV'$ 
# 对矩阵 A 做 QR 分解： $A=QR$ ，Q 为酉矩阵，R 为阶梯形矩阵
qr(A)
# 对正定矩阵 A 做 Choleski 分解， $A=P'P$ ，P 为上三角矩阵
chol(A)
A[upper.tri(A)]    # 提取矩阵 A 的上三角矩阵
A[lower.tri(A)]    # 提取矩阵 A 的下三角矩阵
```

4. 概率函数

```
factorial(n)      # 计算 n 的阶乘
choose(n, k)      # 计算组合数
gamma(x)          # Gamma 函数
beta(a, b)        # beta 函数
combn(x, m)       # 生成 x 中任取 m 个元的所有组合, x 为向量或整数
```

```
combn(4, 2)
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,]    1    1    1    2    2    3
#> [2,]    2    3    4    3    4    4
combn(c(" 甲", " 乙", " 丙", " 丁"), 2)
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,] " 甲" " 甲" " 甲" " 乙" " 乙" " 丙"
#> [2,] " 乙" " 丙" " 丁" " 丙" " 丁" " 丁"
```

- R 中常用的概率函数有密度函数、分布函数、分位数函数、生成随机数函数，其前缀表示分别为：
 - d = 密度函数 (density)
 - p = 分布函数 (distribution)
 - q = 分位数函数 (quantile)
 - r = 生成随机数 (random)
- 上述 4 个字母 + 分布缩写，就构成通常的概率函数，例如：

```
dnorm(3, 0, 2)      # 正态分布 N(0, 4) 在 3 处的密度值
#> [1] 0.0648

pnorm(1:3, 1, 2)    # N(1,4) 分布在 1,2,3 处的分布函数值
#> [1] 0.500 0.691 0.841
```

命中率为 0.02，独立射击 400 次，至少击中两次的概率

```
1 - sum(dbinom(0:1, 400, 0.02))
```

```
#> [1] 0.997
```

```
pnorm(2, 1, 2) - pnorm(0, 1, 2) #  $X \sim N(1, 4)$ , 求  $P\{0 < X \leq 2\}$ 
```

```
#> [1] 0.383
```

```
qnorm(1-0.025, 0, 1) #  $N(0, 1)$  的上 0.025 分位数
```

```
#> [1] 1.96
```

- 自然界中的随机现象是真正随机发生不可重现的，计算机中模拟随机现象，包括生成随机数、随机抽样并不是真正的随机，而是可以重现的。通过设置为相同的起始种子值就可以重现，故称为“伪随机”。

```
set.seed(123)           # 设置随机种子，以重现随机结果
rnorm(5, 0, 1)          # 生成 5 个服从  $N(0,1)$  分布的随机数
#> [1] -0.5605 -0.2302  1.5587  0.0705  0.1293
```

表 1: 常用概率分布及缩写

分布名称	缩写	参数及默认值
二项分布	binom	size, prob
多项分布	multinom	size, prob
负二项分布	nbinom	size, prob
几何分布	geom	prob
超几何分布	hyper	m, n, k
泊松分布	pois	lambda
均匀分布	unif	min=0, max=1
指数分布	exp	rate=1
正态分布	norm	prob
对数正态分布	lnorm	meanlog=0, stdlog=1

表 2: 常用概率分布及缩写 (续表)

分布名称	缩写	参数及默认值
t 分布	t	df
卡方分布	chisq	df
F 分布	f	df1, df2
Wilcoxon 符号秩分布	signrank	n
Wilcoxon 秩和分布	wilcox	m, n
柯西分布	cauchy	location=0, scale=1
Logistic 分布	logis	location=0, scale=1
Weibull 分布	weibull	shape, scale=1
Gamma 分布	gamma	shape, scale=1
Beta 分布	beta	shape1, shape2

- `sample(x, size, replace=FALSE, prob)`: 从向量中重复或非重复地随机抽样

```
set.seed(123)
```

```
# 模拟抛 10 次硬币
```

```
sample(c(" 正", " 反"), 10, replace=TRUE)
```

```
#> [1] " 正" " 正" " 正" " 反" " 正" " 反" " 反" " 反" " 正"
```

```
# 随机生成 1~10 的某排列
```

```
sample(1:10, 10, replace=FALSE)
```

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

5. 统计函数

<code>min(x)</code>	# 求最小值
<code>cummin(x)</code>	# 求累计最小值
<code>max(x)</code>	# 求最大值
<code>cummax(x)</code>	# 求累计最大值
<code>range(x)</code>	# 求 x 的范围：[最小值，最大值]（向量）
<code>sum(x)</code>	# 求和
<code>cumsum(x)</code>	# 求累计和
<code>prod(x)</code>	# 求积
<code>cumprod(x)</code>	# 求累计积
<code>mean(x)</code>	# 求平均值

```
median(x)           # 求中位数
quantile(x, pr)      # 求分位数, x 为数值向量, pr 为概率值
sd(x)               # 求标准差
var(x)              # 求方差
cov(x)              # 求协方差
cor(x)              # 求相关系数
scale(x, center=TRUE, scale=FALSE) # 对数据做中心化: 减去均值
scale(x, center=TRUE, scale=TRUE)  # 对数据做标准化
```

- 自定义归一化函数:

```
Rescale = function(x, type = "pos") {  
  rng = range(x, na.rm = TRUE)  # 计算最小值最大值  
  if(type == "pos") {  
    (x - rng[1]) / (rng[2] - rng[1])  
  } else {  
    (rng[2] - x) / (rng[2] - rng[1])  
  }  
}  
  
# 测试  
x = c(1, 2, 3, NA, 5)  
Rescale(x)  
#> [1] 0.00 0.25 0.50    NA 1.00  
Rescale(x, "neg")  
#> [1] 1.00 0.75 0.50    NA 0.00
```

6. 时间序列函数

- `lag(x, k=1, ...)`: 计算时间序列 x 的 k 阶滞后
- `diff(x, lag=1, difference=1, ...)`: 计算时间序列 x 的滞后为 `lag` 阶的 `difference` 阶差分
- Y_t 的 j 阶滞后为 Y_{t-j} :

```
x = ts(1:8, frequency = 4, start = 2015)
```

```
x
```

```
#>      Qtr1 Qtr2 Qtr3 Qtr4
```

```
#> 2015     1     2     3     4
```

```
#> 2016     5     6     7     8
```

```
stats::lag(x, 4)      # 避免被 dplyr::lag() 覆盖
```

```
#>      Qtr1 Qtr2 Qtr3 Qtr4
```

```
#> 2014     1     2     3     4
```

```
#> 2015     5     6     7     8
```

- Y_t 的一阶差分为 $\Delta Y_t = Y_t - Y_{t-1}$, 二阶差分为 $\Delta^2 Y_t = \Delta Y_t - \Delta Y_{t-1}, \dots$

```
x = c(1, 3, 6, 8, 10)
x
#> [1] 1 3 6 8 10
diff(x, differences = 1)
#> [1] 2 3 2 2
diff(x, differences = 2)
#> [1] 1 -1 0
diff(x, lag = 2, differences = 1)
#> [1] 5 5 4
```

7. 其他函数

```
unique(x, ...)      # 返回唯一值，即去掉重复元素或观测  
# 判断元素或观测是否重复（多余），返回逻辑值向量  
duplicated(x, ...)  
anyDuplicated(x, ...) # 返回重复元素或观测的索引  
rle(x)              # 统计向量中连续相同值的长度  
inverse.rle(x)      # rle() 的反向版本，x 为 list(lengths,
```

本篇主要参阅(张敬信, 2022), (Hadley Wickham, 2017), 模板感谢(黄湘云, 2021), (谢益辉, 2021).

参考文献

Hadley Wickham, G. G. (2017). *R for Data Science*. O' Reilly, 1 edition. ISBN 978-1491910399.

张敬信 (2022). *R 语言编程：基于 tidyverse*. 人民邮电出版社, 北京.

谢益辉 (2021). *rmarkdown: Dynamic Documents for R*.

黄湘云 (2021). *Github: R-Markdown-Template*.