“Second programming exercise if, else if, else syntax and logic”

Alan Berger Aug 25, 2020 minor edits Jan 18, 2021

version 1

Introduction

This is the second in a sequence of programming exercises intended to fill the gap between learning the correct syntax of basic R commands and the programming assignments in the R Programming course in the Johns Hopkins University Data Science Specialization on Coursera. In this sequence of exercises in “composing” an R function to carry out a particular task, the idea is to practice correct use of R constructs and built in functions (functions the “come with” the basic R installation), while learning how to “put together” a correct sequence of blocks of commands that will obtain the desired result.
Note these exercises are quite cumulative - one should do them in order.

In these exercises, there will be a statement of what your function should do (what are the input variables and what the function should return) and a sequence of “hints”. To get the most out of these exercises, try to write your function using as few hints as possible.
Note there are often several ways to write a function that will obtain the correct result. For these exercises the directions and hints may point toward a particular approach intended to practice particular constructs in R and a particular line of reasoning.
There may be an existing R function or package that will do what is stated for a given practice exercise, but here (unlike other aspects of the R Programming course) the point is to practice formulating a logical sequence of steps, with each step a section of code, to obtain a working function, not to find an existing solution or a quick solution using a more powerful R construct that is better addressed later on.

Motivation for this exercise

If statements are a basic construct for determining which commands or blocks of commands should be executed. The specific function for this exercise is described below.

Some if, else if, else templates

Below are templates for common if statement constructs. Note it is helpful to use indentation of lines of code and blank spaces between sections of code as illustrated below. The number of spaces of indentation is somewhat a personal choice, balancing making code easy to read by having separate blocks of code stand out by having more spaces in indentations, with not having too many R commands extend over more than 1 line (and in general, don’t go over 80 characters in a line). The motivation is that code that is easier to read is easier to proofread and spot bugs in.

```
if (condition1) short.code  # short.code is a short R command
# condition1 (and below, condition2, condition3) is a logical statement evaluating to TRUE or FALSE
# when condition1 is true, execute the short.code statement



if (condition1) {
    code1  # where code1 (and below, code2, code3, code4) stands for one or more lines of code
}
# when condition1 is true, execute code1
```

```
if (condition1) {
    code1
    } else {
    code2
}
# when condition1 is TRUE, then the line(s) of code1 are executed,
# otherwise the line(s) of code2 are executed
```

```
if (condition1) {
    code1
    } else if (condition2) {
    code2
}
# when condition1 is TRUE, code1 is executed
# when condition1 is FALSE, then code2 will be executed if condition2 is TRUE;
# when neither condition is TRUE, neither code1 nor code2 get executed, and
# R "proceeds" to the next line of code
```

```
# The full range of possibilities within one if, else if, else block are
# illustrated here
if (condition1) {
    code1
    } else if (condition2) {
    code2
    } else if (condition3) {
    code3
    } else {
    code4
}
# there can be any (reasonable) number of "else ifs"


# Another quick way to use a sequence of if tests when there are only
# a few cases is shown in this example, of getting the rgb (red, green, blue)
# color scale values when there are only a couple possible color names.
# This is, if nothing else, code that is easy to read/understand

rgbvec <- 0 # when, after the if statements, rgbvec is still 0, throw an error
if(color == "Magenta") rgbvec <- c(255, 0, 255)
if(color == "ForestGreen") rgbvec <- c(34, 139, 34)
if(color == "Cyan") rgbvec <- c(0, 255, 255)
# test that color matched one of the above choices
if(identical(rgbvec, 0)) stop("color did not match one of the choices")

# This is just an example, for "real" use for getting rgb colors one would want to have a data frame
# with this information (many color names and corresponding rgb values)
# and extract the rgb values for a given color from it.
# The color information above came from
# https://en.wikipedia.org/wiki/Web_colors
# side note: there are web sites for checking how a color figure would appear
# to people with various types of color blindness, for example
# https://www.color-blindness.com/coblis-color-blindness-simulator/
```

In many cases in a block of if, else if, else statements the conditions will be mutually exclusive (at most one of the conditions will be TRUE), and (when there is an else statement) one of the code blocks will certainly be executed. The range of values for which the conditions are TRUE can also be increasing or decreasing sets ("nested" sets) if one orders the sequence of if tests appropriately as in the exercise program below.

Instruction for the exercise function using an if, else if, else block

Constructing the function specified below is an exercise in using an if, else if, else block. Yes, one could do this by simply using the **signif** or **format** function, but the point here is to practice if logic.

We are given a p-value, denoted by pval, (a number between 0 and 1) and we want to convert it to a value having 3 significant digits (specifically, we want the corresponding character string). For example each of these values have 3 significant digits (for a number < 1, "leading 0's" to the right of the decimal point before one encounters a non-zero digit "don't count" toward the number of significant digits): 0.123, 0.0123, 0.00123, 0.000123, 0.0000123, 0.00000123

We would like to do this since for a p-value such as 0.012345678, in most circumstances digits beyond 0.0123 are just "clutter" since they would rarely matter, and also often wouldn't be justified due to limited accuracy of the data. An aside: one reason to keep many digits would be to check code by comparing results with an independent calculation or "known" value.

Background note (but not necessary for writing this function): p-values often come from some statistical test, such as on whether measured values from (independent random) samples from two groups indicate that the two groups have different group means (for what was being measured). The pval is then (for this example) the probability that one would have seen, just by random chance, a difference in group means as large in magnitude or larger than the difference one actually observed, if the "truth" was that there was no difference between the two groups (in what was being measured).

The bottom line is that for this exercise we want to convert pval to a character string having 3 significant digits.

Again, one could do this by simply using the **signif** or **format** function, but the point here is to practice if logic. So you are to use the **round** function which takes as input a number (or vector) and "rounds" the input value(s) to have only a specified number of digits appearing to the right of the decimal point.

The lines below illustrate the behavior of the **round** function, round(numeric.value, digits), which specifies the number of digits to the right of the decimal point in the returned value.

```
> round(1.123, digits = 3)
[1] 1.123
> round(0.123, digits = 3)
[1] 0.123
> round(0.666666667, digits = 3)
[1] 0.667
> round(0.0123, digits = 3)
[1] 0.012
> round(0.0123, digits = 4)
[1] 0.0123
> round(0.00123, digits = 5)
[1] 0.00123
> round(0.000123, digits = 6)
```

```
[1] 0.000123
> round(0.0000123, digits = 7)
[1] 1.23e-05
```

Your function, call it pval_To_3Sig_Digits should have the 1 argument pval (a number between 0 and 1, including 0 and 1) and should return a character string corresponding to pval rounded to 3 significant digits, using one if, else if, else block of code (that will have multiple else if statements within it), and the round function. If pval is < 0.00001, then return the character string “p $<$ 0.00001” The **as.character** function will convert a numeric value to the corresponding character string

Some hints follow, try programming your function using as few hints as possible.

Think about how to order the conditions in the if, else if, else statements. Start by treating smaller p-values first and end with larger p-values. Note more than 1 test within an if, else if, else block of code might be satisfied: in that case the “consequence” of the first test that is satisfied will be carried out and then control will pass to the first statement after the if, elseif, else block of code.

Larger hint: the beginning and end of your function should resemble

```
pval_To_3Sig_Digits <- function(pval) {
# the input pval is to be a number between 0 and 1
# use the round function and an if, else if, else block to
# return a character string corresponding to pval rounded to
# 3 significant digits

# check pvalue is a number between 0 and 1
if(!is.numeric(pval)) stop("pval is not numeric")
# check pval is between 0 and 1
if(pval < 0 || pval > 1) stop("pval not between 0 and 1")

# the start of the if, else if, else block

if (pval < 0.00001) {
    pval.string <- "p < 0.00001"

#   note if one has gotten to here, pval must be >= 0.00001 so
#   digits should be 7 be treat values like 0.0000123
#   How small does pval have to be to not get more than
#   3 significant digits from round(pval, digits = 7): this
#   determines the next else if condition

    } else if (pval < 0.0001) {
    pval.string <- as.character(round(pval, digits = 7))
#   if pval was >= 0.0001 then digits = 7 would in general get
#   4 significant digits
#   Hint: the code in the next else if block will use digits = 6,
#   what should the condition be in the next else if so that
#   digits = 6 does not give more than 3 significant digits


#   MORE CODE needs to be provided


    } else {
    pval.string <- as.character(round(pval, digits = 3))
}

return(pval.string)
}
```

One could also start testing if pval >= 0.001 and “work downward” using successively smaller values of pval; doing it this way, the else if tests will involve testing whether pval >= some value

A working version of pval_To_3Sig_Digits

```r
pval_To_3Sig_Digits <- function(pval) {
# the input pval is to be a number between 0 and 1
# use the round function and an if, else if, else block to
# return a character string corresponding to pval rounded to
# 3 significant digits

# check pvalue is a number between 0 and 1
if(!is.numeric(pval)) stop("pval is not numeric")
# check pval is between 0 and 1
if(pval < 0 || pval > 1) stop("pval not between 0 and 1")

# the if, else if, else block

if (pval < 0.00001) {
    pval.string <- "p < 0.00001"
    } else if (pval < 0.0001) {
    pval.string <- as.character(round(pval, digits = 7))
    } else if (pval < 0.001) {
    pval.string <- as.character(round(pval, digits = 6))
    } else if (pval < 0.01) {
    pval.string <- as.character(round(pval, digits = 5))
    } else if (pval < 0.1) {
    pval.string <- as.character(round(pval, digits = 4))
    } else {
    pval.string <- as.character(round(pval, digits = 3))
}

return(pval.string)
}
```

Do some test runs

```r
pval_To_3Sig_Digits(1.0)
```

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

```r
pval_To_3Sig_Digits(0.123456)
```

```
## [1] "0.123"
```

```r
pval_To_3Sig_Digits(0.0123456)
```

```
## [1] "0.0123"
```

```r
pval_To_3Sig_Digits(0.00123456)
```

```
## [1] "0.00123"
```

```r
pval_To_3Sig_Digits(0.000123456)
```

```
## [1] "0.000123"
```

```r
pval_To_3Sig_Digits(0.0000123456)
```

```
## [1] "1.23e-05"
```

```r
pval_To_3Sig_Digits(0.00001)
```

```
## [1] "1e-05"
```

```r
pval_To_3Sig_Digits(0.00000999)
```

```
## [1] "p < 0.00001"
```

```r
pval_To_3Sig_Digits(0)
```

```
## [1] "p < 0.00001"
```

```r
# a note just for future reference
# if one doesn't want scientific (exponential) notation
# one can use the options function to change
# "scipen" which is an R system variable (here an integer) governing
# when R will use scientific notation (for small or large numbers)
# options can change various R options for the current R session
getOption("scipen") # 0  so can change it back
```

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

```r
options("scipen" = 999) # don't do scientific (exponential) notation
pval_To_3Sig_Digits(0.0000123456)
```

```
## [1] "0.0000123"
```

```r
pval_To_3Sig_Digits(0.00001)
```

```
## [1] "0.00001"
```

```
options("scipen" = 0)  # reset to the default value we retrieved above
getOption("scipen")  # check it was reset
```

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

Hope this programming exercise was informative and good practice with writing a function with an if block.