



Fundamentals of Function in R

 Evans_G02504972

 2023-11-07

The main reasons to write an R function and to create it is to avoid copying and pasting repetitive codes many times. Example shown below from the three data sets

```
dataSet1 <- data.frame(x = 1:5, y = 1:5)
dataSet2 <- data.frame(x = 1:5, y = 1:5)
dataSet3 <- data.frame(x = 1:5, y = 1:5)
#drop last column of data set 1, and only keep last two rows
```

```
dataSet1 %>% select(-2)%>% slice_head(n = 2)
```

```
##      x
## 1 1
## 2 2
```

```
dataSet2 %>% select(-2)%>% slice_head(n = 2)
```

```
##      x
## 1 1
## 2 2
```

```
dataSet3 %>% select(-2)%>% slice_head(n = 2)
```

```
##      x
## 1 1
## 2 2
```

```
dataSet1 <- data.frame(x = 1:5, y = 1:5)
dataSet2 <- data.frame(x = 1:5, y = 1:5)
dataSet3 <- data.frame(x = 1:5, y = 1:5)
```

```
dataSet1 %>% select(-2)%>% slice_head(n = 2) %>% arrange(x)
```

```
##    x
##  1  1
##  2  2
```

```
dataSet2 %>% select(-2)%>% slice_head(n = 2) %>% arrange(x)
```

```
##    x
##  1  1
##  2  2
```

```
dataSet3 %>% select(-1)%>% slice_head(n = 3) %>% arrange(y)
```

```
##    y
##  1  1
##  2  2
##  3  3
```

```
#create tables to drop columns of dataset, and only keep some rows
drop_cols_keep_rows <- function(x, drop_cols, keep_rows){
  return(x %>% select(-drop_cols) %>% slice_head(n = keep_rows) %>% ar
}
```

```
drop_cols_keep_rows(dataSet1,drop_cols = 2, keep_rows = 2)
```

```
##    x
##  1  1
##  2  2
```

```
drop_cols_keep_rows(dataSet1,drop_cols = 2, keep_rows = 2)
```

```
##    x
##  1  1
##  2  2
```

```
drop_cols_keep_rows(dataSet1,drop_cols = 1, keep_rows = 3)
```

```
##    y  
## 1 1  
## 2 2  
## 3 3
```

lets start by loading necessary packages for this activity

```
library(tidyverse)  
library(stringr)
```

The general structure of a function in R is displayed below

```
function_name <- function(arg1,arg2,arg3,.....) {  
  #code is excecuted and  
  #depends on the values supplied to the arguments  
  #This codes can be multiple lines  
  return(someObjects)  
}
```

First, lets create a function called mySum()

```
# Creating a function called mySum with 2 required arguments  
mySum <- function(number1, number2) {  
  return(number1 + number2)  
}
```

```
# Defaulting to order of arguments in function  
mySum(5, 3)
```

```
## [1] 8
```

```
# Manually specifying arguments  
mySum(number1 = 5, number2 = 3)
```

```
## [1] 8
```

Create the mySum() function, and use it to calculate the sum of 13 and 1989.

```
mySum(13,1989)
```

```
## [1] 2002
```

```
mySum(NA,1989)
```

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

Try 1.

```
#create function that adds two numbers potentially with NA values
myNewSum <- function(number1, number2, remove.na = TRUE) {
  if(remove.na) {
    # if remove na is true, replaces NA values with 0's
    calcSum <- ifelse(is.na(number1), 0, number1) +
      ifelse(is.na(number2), 0, number2)
  } else {
    #
    calcSum <- number1 + number2
  }
  return(calcSum)
}
myNewSum(13,1989)
```

```
## [1] 2002
```

```
myNewSum(NA,1989)
```

```
## [1] 1989
```

```
myNewSum(13,NA, remove.na = FALSE)
```

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

Read the source code for each of the following three functions, determine the purpose of each function, and propose better function names.

Try 2

```
#create function thats check if a string has a specified prefix  
check_prefix <- function(string, prefix) {  
  str_sub(string, 1, nchar(prefix)) == prefix  
}
```

```
check_prefix("provide",prefix="pro")
```

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

```
check_prefix("provide",prefix="anti")
```

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

```
check_prefix("provide",prefix="vide")
```

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

```
#Create a function that drops last element of a vector  
drop_last <- function(x) {  
  if (length(x) <= 1) return(NULL) else {  
    return(x[-length(x)])  
  }  
}
```

```
#create a function to calculate harmonic mean
f3 <- function(x) {
  1 / (mean(1 / x))
}
harmonic_mean <- function(x) {
  1 / (mean(1 / x))
}

harmonic_mean(1:50)
```

```
## [1] 11.11307
```

```
mean(1:50)
```

```
## [1] 25.5
```

Create a new function to calculate the geometric mean of a vector of numbers which is equal to the nth root of the product of n numbers.

```
# Define the geometric_mean function
geometric_mean <- function(x) {
  exp(mean(log(x)))
}
# Calculate the geometric mean of a vector
vector <- c(2, 4, 8, 16,32)
geometric_mean(vector)
```

```
## [1] 8
```

```
geometric_mean <- function(x){
  return(exp(mean(log(x))))
}
geometric_mean(1:50)
```

```
## [1] 19.48325
```

#condirional execution lets create our own custom square-root function

```
# Function to calculate square root
calcSqrt <- function(val) {
  if (val >= 0) {
    return(val^0.5)
  } else {
    warning("val is negative but must be >= 0!")
    return(NaN)
  }
}
```

```
calcSqrt(4)
```

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

```
calcSqrt(-9)
```

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

The hierarchy in the ways to inform a user more about executing code. 1. 'Messages' The least serious, provide information to the user, but does not halt execution of the code, and typically is for information purpose, not necessarily due to code being used properly.

2. 'Warnings'

Somewhere in between the seriousness of a message and an error. does not halt execution of the code, and typically is for information purpose, could be due to code being used improperly.

```
# Function to calculate square root
calcSqrt <- function(val) {
  if (val >= 0) {
    return(val^0.5)
  } else {
    warning("val is negative but must be >= 0!")
    return(NaN)
  }
}
```

```
calcSqrt(4)
```

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

```
calcSqrt(-9)
```

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

```
# Function to calculate square root
calcSqrt <- function(val) {
  if (val >= 0) {
    return(val^0.5)
  } else {
    stop("val is negative but must be >= 0!")
    return(NaN)
  }
}
```

```
calcSqrt(4)
```

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

```
#calcSqrt(-9)
```

```
if (condition1) { # do that } else if (condition2) { # do something else } else { # }
```

```
foodType <- function(food) {
  if (food %in% c("apple", "orange", "banana")) {
    return("fruit")
  } else if (food %in% c("broccoli", "asparagus")) {
    return("vegetable")
  } else {
    return("other")
  }
}
```

```
foodType("Reese's Puffs")
```

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

```
foodType("apple")
```



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

```
foodType("Apple")
```

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

```
foodType <- function(food) {  
  food <- stringr::str_to_lower(food)  
  if (food %in% c("apple", "orange", "banana")) {  
    return("fruit")  
  } else if (food %in% c("broccoli", "asparagus")) {  
    return("vegetable")  
  } else {  
    return("other")  
  }  
}
```

```
foodType("Reese's Puffs")
```

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

```
foodType("apple")
```

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

```
foodType("Apple")
```

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

Whole number function

Create a function called `is.whole.number()`, which checks if a given value is a whole number (e.g., 0, 1, 2, ...) or not.

solution 1

```
is.whole.number <- function(val) {  
  if (val > as.integer(val)) {  
    return(TRUE)  
  } else {  
    return(FALSE)  
  }  
}  
is.whole.number(0)
```

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

solution 4

```
is.whole.number <- function(val) {  
  if (val >= 0 & as.integer(val)) {  
    return(TRUE)  
  } else {  
    return(FALSE)  
  }  
}  
is.whole.number(4)
```

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

solution 2

```
is.whole.number <- function(val) {  
  if (val >= 0 & val == floor(val)) {  
    return(TRUE)  
  } else {  
    return(FALSE)  
  }  
}  
is.whole.number(4)
```

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

```
is.whole.number(0)
```

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

```
is.whole.number(-1)
```

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

```
is.whole.number(1.01)
```

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

solution 3

```
is.whole.number <- function(val) {  
  if (val >= 0 & val == round(val)) {  
    return(TRUE)  
  } else {  
    return(FALSE)  
  }  
}  
is.whole.number(4)
```

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

```
is.whole.number(0)
```

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

```
is.whole.number(-1)
```

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

```
is.whole.number(1.01)
```

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

The ^ operator raises a number to a specified power in R. Create a function that raises a number to a specified power, and returns a warning and an NaN value if the number is negative and the power is not a whole number.

```
#create function to raise a number to a specific power
```

```
raise_power <- function(val, power){  
  if(is.whole.number(power)){  
    return(val^power)  
  }  
  else{  
    warning("power is not a whole number, returning NAN")  
    return(NaN)  
  }  
}  
raise_power(val=4, power=2)
```

```
## [1] 16
```

```
raise_power(val=-9, power=2)
```

```
## [1] 81
```

```
raise_power(val=3, power=1/3)
```

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

```
raise_power(val=-3, power=1/3)
```

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

The 'ifelse' function

```
#ifelse for a single value
school <- "GVSU"
mascot <- ifelse(school == "GVSU", "Lakers", "Bulldogs")
#vectorized checking of equality
school <- c("GVSU", "FSU")
mascot <- ifelse(school == "GVSU", "Lakers", "Bulldogs")

#Nested ifelse statement
many_school <- c("GVSU", "FSU", "SVSU")
many_mascot <- case_when(
  many_school == "GVSU" ~ "Lakers",
  many_school == "FSU" ~ "Bulldogs",
  many_school == "SVSU" ~ "Cardinals",
  TRUE ~ "OTHER"
)
```

The function `myNewSum()`, created below, has two required arguments: `number1` and `number2`, and one optional argument, `remove.na`. This function uses the `ifelse()` function, which has 3 main arguments itself. Look at the help file for `ifelse()` to see descriptions of its arguments and examples of its use.

```
# Creating a function called mySum with 2 required arguments, 1 option
myNewSum <- function(number1, number2, remove.na = TRUE) {
  if(remove.na) {
    calcSum <- ifelse(is.na(number1), 0, number1) +
      ifelse(is.na(number2), 0, number2)
  } else {
    calcSum <- number1 + number2
  }
  return(calcSum)
}
myNewSum(13, 1989)
```

```
## [1] 2002
```

```
myNewSum(NA, 1989)
```

```
## [1] 1989
```

logical operator

```
# The 'or' operator  
(3 < 5) || (-1 > 0)
```

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

```
# The 'and' operator  
(3 < 5) && (-1 > 0)
```

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

```
# Hadley would frown at this  
# The 'or' operator  
(3 < 5) || (-1 > 0)
```

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

```
# The 'and' operator  
(3 < 5) && (-1 > 0)
```

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

Create a function that takes a numeric vector as input, and returns a single boolean value indicating whether or

not the vector has any negative values.

```
#function to check for negative values in a vector  
any_negative <- function(numbers){  
  return(any(numbers < 0))  
}
```

```
any_negative(c(1,2,3,0))
```

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

```
any_negative(c(1,2,3))
```

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

```
any_negative(c(1,2,3,-1))
```

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

```
has_negative <- function(num_vec){  
  if(any(num_vec < 0)){  
    return(TRUE)  
  
  } else {  
    return(FALSE)  
  }  
}  
has_negative(2)
```

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

```
has_negative(-1)
```

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

Does the code `(1:4 < 4) & (1:4 > 1)` produce the same result as `(1:4 < 4) && (1:4 > 1)`? Which code achieves what appears to be its intended purpose?

```
#Looks at all values in both vectors 👍  
(1:4 < 4) & (1:4 > 1)
```

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

```
#Only look at 1st value in each vector 👎  
#(1:4 < 4) && (1:4 > 1)
```

Bonus (optional)

Write a function to check if a number is even or odd that returns a vector containing the values “even”, “odd”, or “not an integer” depending on its input.

Illustration

For positive numbers, the Modulus is the remainder of the division of two numbers. For example, 10 divided by 3 is 1, so 10 modulus 3 is 1. The modulus operator in R is implemented using `%%` as below:

10 mod 3

```
10 %% 3 ## [1] 1 # 10 mod 2 10 %% 2 ## [1] 0 # 10 mod 5 10 %% 5 ## [1] 0 # 12 mod 5 12 %% 5 ##  
[1] 2
```

solution

#function that takes a number as input and returns output whether it

```
check_even_odd <- function(number) {  
  if (is.numeric(number) && number %% 2 == 0) {  
    return("even")  
  } else if (is.numeric(number)) {  
    return("odd")  
  }  
}
```

Test the function with various numbers

```
check_even_odd(10)
```

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

```
check_even_odd(7)
```

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

```
check_even_odd(2.5)
```

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