Programming Topics II: Writing Functions

UNIVERSITY OF
SAN FRANCISCO

Abbie M Popa

BSDS 100 - Intro to Data Science with R

Lab Solutions

Data located at:

https://raw.githubusercontent.com/abbiepopa/BSDS100/master/Data/titanic.csv

`titanic.csv`

1. Using a `for()` loop and an `if()` conditional, recode the entries in the `Survived` variable with `"Survived"` and `"Perished"` into a new column `survived_text`
2. Using the `if()` command and loop, create a new variable of type `ordered factor` in the data frame called `ageClass`, and map `Age` to: `"Minor"` if less than 18 yrs; 18 yrs $\leq$ `"Adult"` $\leq$ 65 yrs; and `"Senior"` if older than 65 yrs
3. Using a `switch()` statement, identify each passenger class, `Pclass`, as either `"First Class"`, `"Business Class"` or `"Economy"`, and print the results to the console

Outline

- What is a function?
- Writing your own functions
- Using functionals

Functions

- We have already used many functions, including `mean()`, `seq()`, `c()`, `data.frame()`, and `ggplot()`
- Just about anything we have used that is followed by parantheses `()` is a function
- Functions tell `R` to perform a specific task

Functions

- Each function has several parts
  1. A name (e.g., `mean`)
  2. A list of arguments often called `formals()`, these can be mandatory (e.g., `mean()` needs a vector of numbers to find the mean of, or optional (e.g., `mean()` can take the argument `na.rm = T`)
  3. A `body()`, which is the code a function executes

Functions

- Recall, we can find a list of a functions arguments with `?`
- Any argument in the documentation followed by `=` carries a default value, that means if the user doesn't fill in that argument, the function can still run using the default (i.e., the argument is optional)
- We can also see the code inside the function by calling it's name without the `()`

Writing a function

- We can write our own functions using assignment and the word "function"

```
name_of_func <- function(arguments_of_function){
body_of_function }
```

- To the left of the function we give it a name, inside the parentheses following the word "function" we list the arguments, and inside the curly brackets we write code
- A simple example

```
my_square <- function(x){
x^2 }
```

Writing a function

- In R whatever the last line of the body to run is what the function will "return", or report to the user
- **Exception!** If the last line of a function body is an assignment the function will return nothing, be sure to avoid this so your function returns something

Writing a function

- Functions are useful because the allow you to repeat the same set of commands without copy and pasting code, which can be error prone and time consuming
- Functions can contain much more complex code than we are showing here, including calling other functions, looping, or if/then statements

Writing a function - Example

- Let's write a function that takes a numeric vector, removes any numbers lower than 1 or higher than 26, then returns the letter of the alphabet that matches each number in the resulting vector

Writing a function - Example

- So far, we have shown functions that take one argument, but functions can also take more than one argument
- This function returns the sum of two numbers a user passes it

```
sum_two <- function(a, b){
a + b }
```

Writing a function - Practice

- Arrange the follow lines to make a function named `subtract_two` that takes two numbers and subtracts the smaller number from the larger number (or returns zero if they are equal)
  - `0`
  - `b - a`
  - `a - b`
  - `subtract_two <- function(a,b){`
  - `if(a == b){`
  - `} else if(a < b){`
  - `} else if(b < a){`
  - `}`
  - `}`
- Test your new function on a couple values to make sure it works!

Writing a function - Practice

- Write a function named `square_df` that takes a vector and returns a data frame with two columns where
  - the first column is named "original" and is the original vector
  - the second column is named "squared" and is each of the elements of the original vector squared
- Test your function on a couple vectors to make sure it works!

Writing a Function - Scoping

- We *could* write a function in the following manner:

```
x <- 1

return_one_two <- function(){

y <- 2

c(x, y)}
```

- Notice how the `x` is written outside of the function even though it is used inside the function?
- But then, what would happen if we forgot to define `x`?

Writing a Function - Scoping

- In this case we would say `x` is **globally** scoped while `y` is **locally** scoped
- We would also say `x` is defined in the **global environment** while `y` is defined in the **local environment**
- Function should be written with all variables locally scoped or passed as arguments to avoid errors
- How would we fix the previous function?

Scoping Practice

- We write a function which returns a number (chosen by the user) divided by 3, we initially write the function like so:

```
 user_chosen_number <- 10
three <- 3
num_divide_three <- function(){
user_chosen_number / three }
```

- Rewrite the function so that the user passes their chosen number as an argument and three is locally scoped

Formal Arguments of a Function

- It is important to distinguish between the formal and actual arguments of a function
- **Formal arguments** are a property of the function

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

`x`	An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.
`trim`	the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
`na.rm`	a logical value indicating whether `NA` values should be stripped before the computation proceeds.
`...`	further arguments passed to or from other methods.

Calling Arguments of a Function

- It is important to distinguish between the formal and actual arguments of a function
- **Actual or calling arguments** can vary each time you call a function

```
> mean(x = 1:10)
[1] 5.5

> mean(x = 99:999)
[1] 549
```

- In the above examples, the calling arguments are `1:10` and `99:999` respectively

Default Arguments

- It is also possible to define default values for your function
- The default values will be used if the user doesn't specify a value

Default Arguments

```
# w/o default values
myFunc_10 <- function(a, b) {
  c(a, b)
}

> myFunc_10()
Error in myFunc_10() : argument "a" is missing, with no default

# with default values
myFunc_11 <- function(a = 1, b = 2) {
  c(a, b)
}

> myFunc_11()
[1] 1 2
```

Default Arguments

Function arguments in R can be defined in terms of other arguments

```
myFunc_12 <- function(a = 1, b = a * 2) {
  c(a, b)
}

> myFunc_12()
[1] 1 2

> myFunc_12(111)
[1] 111 222

> myFunc_12(99, 100)
[1]  99 100
```

Return Values

The last expression evaluated in a function becomes the return value

```
myFunc_18 <- function(xyz) {
  if (xyz < 10) {
    0
  } else {
    10
  }
}

> myFunc_18(5)
[1] 0

> myFunc_18(10)
[1] 10
```

To `return()` or not to `return()`

- The last expression evaluated in a function is the return value
- You can always wrap the final expression in `return()` if you choose
- Using `return()` makes the code very slightly slower
- In simplistic functions, `R` programmers will typically omit `return()`
- In longer, more complicated functions, `return()` is often used when it makes the code easier to read

To `return()` or not to `return()`

```
# simple function, does not require a return()

myFunc_15 <- function(x){
  x + 10
}



# a more complex function benefits visually from having return()
#   but does not require return()

myFunc_18 <- function(xyz) {
  if (xyz < 10) {
    return(0)
  } else {
    return(10)
  }
}
```

Functionals

- In addition to saving you time copying and pasting code, functions can be used with "functionals" to avoid loops
- Because loops (for, while, and repeat) are slow in R avoiding them is useful
- Also, functionals can make your code more readable

Functionals

The `apply()` family of functionals are often used in lieu of *for* loops, coming in a variety of flavors (not exhaustive)

Functional	Input	Output
`apply()`	Array/Matrix	Vector/Array
`lapply()`	Vector/List	List
`sapply()`	Vector/List	List
`vapply()`	Vector/List	Vector

There is also a functional `do.call` which is very flexible and can take and output most types. We will focus on the two most commonly used functionals, `apply()` and `lapply()`

`apply()`

- `apply()` is useful for applying the same function to every row or every column of a matrix
- e.g., `apply(test_matrix, 1, mean)` would find the mean of each row in test_matrix

`apply()`

- What if we want to include `na.rm = T` in our `apply()`?
- Just adding it doesn't work
- Adding it with the placeholder doesn't work
- What should we do?

apply()

- First option, make a custom function, say, `na_mean()` that does what we want
- Second option, make an anonymous or lambda function

```
apply(test_matrix, 1, function(x){mean(x, na.rm = T)})
```

- The anonymous function doesn't have a name, and can't be called again later, but this can be a useful method if you don't anticipate reusing the function

`apply()` practice

- Use `apply()` to get the standard deviation, removing `NA`'s, from the test matrix
- You can either define a new function or use an anonymous function

`lapply()`

- `lapply()` takes and returns a list
- This is useful to us because `R` considers data frames to be a type of list!
- Consider the following:

`lapply()` example

- Assume you are given the following data frame

```
> myDataFrame_01
   A  B C   D   E G
1  1  6 1   5 -99 1
2 10  4 4 -99   9 3
3  7  9 5   4   1 4
4  2  9 3   8   6 8
5  1 10 5   9   8 6
6  6  2 1   3   8 5
```

- Your objective is to replace all of the `-99`s with `NA`s

`lapply()` example

- You could—but shouldn't—iterate through each column manually, e.g.

```
myDataFrame_01$A[myDataFrame_01$A == -99] <- NA
myDataFrame_01$B[myDataFrame_01$B == -99] <- NA
...
myDataFrame_01$F[myDataFrame_01$F == -99] <- NA
```

Problems with Brute-Force Approaches

1. It's easy to make copy-paste mistakes
2. It will take you a really long time
3. If you need to change the code later, you will have to change many lines of code rather than just a few

`lapply()` example

Let's write a function with the objective of replacing all `-99`s in a single column with `NA`s

```
fix99s_byCol <- function(myCol) {
  myCol[myCol == -99] <- NA
}
```

- Will the code above work as intended? Hint: **no**. Why not?

`lapply()` example

Let's write a function with the objective of replacing all `-99`s in a single column with `NA`s

```
fix99s_byCol <- function(myCol) {
  myCol[myCol == -99] <- NA
}
```

- Will the code above work as intended? Hint: **no**. Why not?

`lapply()` example

The following **does** work as intended:

```
fix99s_byCol <- function(myCol) {
  myCol[myCol == -99] <- NA
  myCol
}

myDataFrame_01$A <- fix99s_byCol(myDataFrame_01$A)
...
myDataFrame_01$F <- fix99s_byCol(myDataFrame_01$F)
```

- This reduces but doesn't eliminate the potential for errors
- There is no gain in efficiency (repetitive code is still required)
- We can instead use an `lapply()`

lapply() example

```
fix99s_byCol <- function(myCol) {
  myCol[myCol == -99] <- NA
  myCol
}

> myDataFrame_02 <- lapply(myDataFrame_01, fix99s_byCol)

> str(myDataFrame_02)
List of 6
 $ A: num [1:6] 1 10 7 2 1 6
 $ B: num [1:6] 6 4 9 9 10 2
 $ C: num [1:6] 1 4 5 3 5 1
 $ D: num [1:6] 5 NA 4 8 9 3
 $ E: num [1:6] NA 9 1 6 8 8
 $ F: num [1:6] 1 3 4 8 6 5
```

- This almost worked...but not quite

`lapply()` example

Here are two ways to correct the previous function call so that it returns a data frame

1.
```
> myDataFrame_03 <-
as.data.frame(lapply(myDataFrame_01,
fix99s_byCol))
```

2.
```
> myDataFrame_01[] <- lapply(myDataFrame_01,
fix99s_byCol)
```

Note, option 2 only works when replacing the old data frame with the new values, not for making a brand new data frame

`lapply()` practice

- First, make a function that takes a column and changes all values 4 or smaller to 0 and all values 5 or larger to 10 by filling in the blanks below:

```
big_small <- function(myCol){
myCol[myCol < ___] <- ___
myCol[myCol > ___] <- ___
___ }
```

- Then, use your new function and `lapply()` to change myDataFrame_01 into myDataFrame_0_10, where all the numbers have been transformed to 0s and 10s

End of Class

- On Nov 20 we will cover presentation tips for the final and do a function practice lab
- No class or Office Hours Nov 22
- Nov 27 we will do a data science wrap-up activity, final project work-time, and course evals
- There is an optional assignment on canvas named "Extra Review." If there are any topics you would like me to review on Nov 27, please submit them there. (This assignment is not required or graded.)