Homework 6 - Drew Kearny - Due November 17

Name: Drew Kearny netID: 231005935  
Collaborated with:

Your homework **must be submitted in Word or PDF format, created by calling “Knit Word” or “Knit PDF” from RStudio on your R Markdown document.**  
Submission in other formats may receive a grade of 0\*\*. Your responses must be supported by both textual explanations and the code you generate to produce your result. Note that all R code used to produce your results must be shown in your knitted file.

**Reminder** do not include excessive output and use the head() function where appropriate and do not evaluate code chunks with the browser() function.

## Q1 Browser

Below is a function add.up.inv.powers() that computes , via a for() loop, for some value of , specified in the first argument. The second argument is verbose; if this is TRUE (the default is FALSE), then the function prints out the current summand to the console, as a roman numeral. A short demo is given below. You’ll use add.up.inv.powers() and roman.cat() to do a bit of exploration with browser() in the next several questions. Remember to use eval = FALSE for chunks that call browser.

add.up.inv.powers = function(n, verbose=FALSE) {  
 x = 0  
 for (i in 1:n) {  
 x = x + i^(1/i)  
 if (verbose) roman.cat(i)  
 }  
 if (verbose) cat("\n")  
 return(x)  
}  
  
roman.cat = function(num) {  
 roman.num = as.roman(num)  
 roman.str = as.character(roman.num)  
 cat(roman.str, "... ")  
}  
  
add.up.inv.powers(n=3, verbose=FALSE)

## [1] 3.856463

add.up.inv.powers(n=5, verbose=FALSE)

## [1] 6.650406

add.up.inv.powers(n=10, verbose=FALSE)

## [1] 13.15116

1. Copy and paste the definition of add.up.inv.powers() below, into an R code chunk that will *not* be evaluated when you knit (hence the eval=FALSE). You’ll use this as a working ground for the code that you’ll run in your console. Place a call to browser() inside add.up.inv.powers(), in between the line x = 0 and the for() loop. Then update this function definition in your console (i.e., just run the code block that defines add.up.inv.powers()), and call the function in the console with n=5 and the default value of verbose.

add.up.inv.powers = function(n, verbose=FALSE) {  
 x = 0  
 browser()  
 for (i in 1:n) {  
 x = x + i^(1/i)  
 if (verbose) roman.cat(i)  
 }  
 if (verbose) cat("\n")  
 return(x)  
}  
#call add.up.inv.powers(n=5)

Answer the following questions, exploring what you can do in browser mode.

1. How do you display the value of the variable n defined in the add.up.inv.powers() function? (Recall that typing “n” just gives you the next line.)

You can display it by typing print(n)

1. How do you exit the browser mode prematurely, before the last line is reached?

Type ‘Q’ to exit browser

1. Suppose you were to run in the browser mode a call like cool.new.num = add.up.inv.powers(n=5) in the console; if you ran the browser to completion (i.e., stepped through all the lines of the function until a return), would the variable cool.new.num be defined in your console?

yes the variable ‘cool.new.num’ will be defined in the console

1. What happens if you were to save the output again in a different variable name, but you didn’t run the browser to completion, i.e., you exited prematurely?

The variable will either not be defined or will retain its previous value if already defined because the function did not complete its execution so there was no return value to be assigned to the variable.

1. Can you define new variables while in browser mode?

Yes you can define new variables while in browser mode.

1. Can you redefine existing variables in the browser? What happens, for example, if you were to redefine x the moment you entered the browser mode?

You can redefine existing variables in browser mode. And in this case if I redefined x right after entering browser mode, this new value of x will be used in subsequent computations within the function.

1. What happens if you change the location of the call to browser() within the definition of the function add.up.inv.powers()?

If you change the location of the browser() call within the function, browser() will activate at that new point in the function which changes where you start debugging within the function’s execution.

1. Typing the “f” key in browser mode, as soon as you enter a for() loop, will skip to the end of the loop. Try this a few times. What happens if you type “f” after say a few iterations of the loop? What happens if you type “f” right before the loop?

Typing “f” in browser mode skips to the end of the current loop. If you press “f” after a few iterations it will skip the remaining iterations of the loop and if you press “f” right before the loop starts it will skip the entire loop.

1. Lastly, typing the “s” key in browser mode will put you into an even more in-depth mode, call it “follow-the-rabit-hole” mode, where you step into each function being evaluated, and enter browser mode for that function. Try this, and describe what you find. Do you step into roman.cat()? Do you step into functions that are built-in? How far down the rabbit hole do you go?

Typing s brought up a more detailed debugger where it first brought me into the for loop and then to the x= x + i^(1/i) part of the for loop and then into the if statement: if (verbose) roman.cat(i), and then I could go into ‘roman.cat()’ and if I typed a number into the argument it would print the roman numeral followed by 3 dots. You can basically go as deep as anything that is nested and defined. You can step into built-in functions as long as they are coded in R.

## Q2 Debugging

In the following questions, you will debug each of the functions included in the corresponding questions so that they produce the correct results. After correcting those problems, you need to show the debugged function and display the result of the testing code based on your debugged function (change the test chunk option to eval = TRUE). Also, you have to explain in words what changes you made and why. For convenience, the testing code are also provided in the accompanying code chunks. We are using the testthat package for the tests. Make sure to install the package. Remember to use eval = FALSE for chunks that call browser.

1. my.dgamma.log() generates the log of the probability density of the Gamma distribution. Use ?lgamma to understand what it does. Note that this is exactly equal to the function dgamma(..., log=TRUE) so use this to verify your answers. (Hint: Tests can pass with 2 changes)  
   **The function to be debugged:**

I Changed (shape-1)*xx to (shape-1)*log(xx). This corrects the logarithmic transformation of the x ^ (shape−1) term in the Gamma PDF. And then I just reordered the way the function was written because it makes more sense for what the formula actually is.

my.dgamma.log <- function(xx, shape, rate) {  
 (shape - 1) \* log(xx) - rate \* xx + shape \* log(rate) - lgamma(shape)  
}

my.dgamma.log <- function(xx, shape, rate) {  
 browser()  
 log\_density = shape\*log(rate) - lgamma(shape) + (shape-1)\*xx - rate\*xx  
 return(log\_density)  
}

**The testing code:**

library(testthat)

## Warning: package 'testthat' was built under R version 4.2.2

test\_that("my.dgamma.log matches dgamma() with log = TRUE", {  
 expect\_equal(my.dgamma.log(seq(0.2, 5, by = 0.2), shape = 0.2, rate = 5),  
 dgamma(seq(0.2, 5, by = 0.2), shape = 0.2, rate = 5, log = TRUE))  
 expect\_equal(my.dgamma.log (seq(0.4, 10, by=0.4), shape=0.2, rate=5),  
 dgamma(seq(0.4, 10, by=0.4), shape=0.2, rate=5, log=TRUE))  
 expect\_equal(my.dgamma.log(seq(1, 10, by=0.2), shape=2, rate=6),   
 dgamma(seq(1, 10, by=0.2), shape=2, rate=6, log=TRUE))  
})

## Test passed 😸

1. my.dnorm.log() generates the log of the probability density of the normal distribution. Note that this is exactly equal to the function dnorm(..., log=TRUE) so use this to verify your answers.  
   **The function to be debugged:** (Hint: Tests can pass with 1 change)

I changed -1/2*log(2*pi*sd) to -1/2*log(2*pi*sd^2) because the sd needed to be squared to represent the variance. I also changed (xx-mean)^2/sd/2 to (xx-mean)^2/(2 \* sd^2) because the denominator was written incorrectly to what the formula should be.

my.dnorm.log <- function(xx, mean, sd) {  
 -1/2\*log(2\*pi\*sd^2) - (xx-mean)^2/(2 \* sd^2)  
}

my.dnorm.log <- function(xx, mean, sd) {  
 browser()  
 log\_density = -1/2\*log(2\*pi\*sd) - (xx-mean)^2/sd/2  
 return(log\_density)  
}

**The testing code:**

test\_that("my.dnorm.log matches dnorm() with log = TRUE", {  
 expect\_equal(my.dnorm.log(seq(-3, 3, by=0.2), mean=0, sd=1),   
 dnorm(seq(-3, 3, by=0.2), mean=0, sd=1, log=TRUE))  
 expect\_equal(my.dnorm.log(seq(-3, 3, by=0.2), mean=-1, sd=2),   
 dnorm(seq(-3, 3, by=0.2), mean=-1, sd=2, log=TRUE))  
 expect\_equal(my.dnorm.log(seq(-3, 3, by=0.2), mean=-2, sd=5),   
 dnorm(seq(-3, 3, by=0.2), mean=-2, sd=5, log=TRUE))  
})

## Test passed 🎊

1. The Fibonacci sequence [<https://en.wikipedia.org/wiki/Fibonacci_number>] is better defined dynamically, and one attempt at this is in fibonacci(). This function is supposed to generate the th number in the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, …, which begins with 1, 1, and where every number after this is the sum of the previous two. (Hint: Tests can pass with 2 changes)

The original function seemed to not be able to properly handle scenarios where n equaled 1 or 2 so I added if (n <= 2) return(1) to handle this properly and then I also changed return(my.fib[i]) to return(my.fib[n]) because when n was 1 or 2 i was not defined so now the nth fibonacci number is returned.

fibonacci = function(n) {  
 if (n <= 2) return(1)   
 my.fib = c(1, 1)  
 for (i in 2:(n-1)) {  
 my.fib[i+1] = my.fib[i] + my.fib[i-1]  
 }  
 return(my.fib[n])  
}

fibonacci = function(n) {  
 browser()   
 my.fib = c(1,1)  
 for (i in 2:(n-1)) my.fib[i+1] = my.fib[i] + my.fib[i-1]   
 return(my.fib[i])  
}

**The testing code:**

test\_that("fibonacci functions gives correct output",{  
 expect\_equal(fibonacci(1), 1)  
 expect\_equal(fibonacci(2), 1)  
 expect\_equal(fibonacci(3), 2)  
 expect\_equal(fibonacci(5), 5)  
 expect\_equal(fibonacci(9), 34)  
})

## Test passed 😀

1. Use browser() to find and fix bugs in the functions sentence.flipper() and word.flipper() below. The first function is supposed to take a sentence, i.e., a single string composed of words separated by spaces, and flip each of the words (meaning reverse the order of their characters); the second function is used by the first, to take a single word and flip it (reverse the order of the characters). Describe what bugs you found, how you found them, and what you did to fix them. Once this is done, your function should be producing outputs on the test cases below that pass.

By using browser() I stepped through the functions and inspected the state of variables at each step to identify where the functions deviated from the expected behavior and then I checked the values of str.words, rev.words, and chars at different steps to could see how the functions were incorrectly handling the input strings. In sentence.flipper I found that the line rev.words = lapply(str, word.flipper) was incorrect and changed it to lapply(str.words[[1]], word.flipper) because the lapply function should apply word.flipper to each word in the sentence, but str is the entire sentence not the individual words. In word.flipper I found that the line chars.flipped = rev(chars) was incorrect because strsplit returns a list and chars is a list so we should reverse the first element of this list so I changed it to chars.flipped = rev(chars[[1]]).

sentence.flipper = function(str) {  
 str.words = strsplit(str, split=" ")   
 rev.words = lapply(str.words[[1]], word.flipper)   
 str.flipped = paste(rev.words, collapse=" ")  
 return(str.flipped)  
}  
  
word.flipper = function(str) {  
 chars = strsplit(str, split="")  
 chars.flipped = rev(chars[[1]])  
 str.flipped = paste(chars.flipped, collapse="")  
 return(str.flipped)  
}

sentence.flipper = function(str) {  
 browser()  
 str.words = strsplit(str, split=" ")   
 rev.words = lapply(str, word.flipper)   
 str.flipped = paste(rev.words, collapse=" ")  
 return(str.flipped)  
}

word.flipper = function(str) {  
 browser()  
 chars = strsplit(str, split="")  
 chars.flipped = rev(chars)  
 str.flipped = paste(chars.flipped, collapse="")  
 return(str.flipped)  
}

**The testing code:**

test\_that("sentence flipper works on toy examples", {  
 expect\_equal(sentence.flipper("the quick brown fox jumped over the lazy dog"),   
 "eht kciuq nworb xof depmuj revo eht yzal god")  
 expect\_equal(sentence.flipper("to be or no to be that is the question") ,   
 "ot eb ro on ot eb taht si eht noitseuq")  
})

## Test passed 😀

1. Extend the function sentence.flipper() so that it is vectorized, i.e., if the input str is a vector of strings, then this function should return a vector where each element is a string that is flipped in accordance with the description above. Hint: there is certainly more than one way to modify sentence.flipper() so that it works over vectors. But look out for a simple strategy—you already know that sentence.flipper() works over single strings, so now just do something to apply this strategy over each element of a vector. Once this is done, your function should be producing outputs on the test cases below that pass.

sentence.flipper <- function(str) {  
   
 if (!is.vector(str) || length(str) == 1) {  
 str.words = strsplit(str, split=" ")  
 rev.words = lapply(str.words[[1]], word.flipper)  
 return(paste(rev.words, collapse=" "))  
 }  
 sapply(str, function(s) {  
 str.words = strsplit(s, split=" ")  
 rev.words = lapply(str.words[[1]], word.flipper)  
 paste(rev.words, collapse=" ")  
 })  
   
}

**The testing code:**

test\_that("sentence flipper is vectorized", {  
 expect\_equivalent(sentence.flipper(c("hello darkness my old friend",  
 "i've come to speak with you again")),   
 c("olleh ssenkrad ym dlo dneirf",   
 "ev'i emoc ot kaeps htiw uoy niaga"))  
   
 expect\_equivalent(sentence.flipper(c("never gonna give you up",  
 "never gonna let you down",  
 "never gonna run around and desert you")),   
 c("reven annog evig uoy pu",  
 "reven annog tel uoy nwod",  
 "reven annog nur dnuora dna tresed uoy"))  
})

## Test passed 🌈

## Q3 Testing

Now for a bit of unit testing. Download and install the testthat package if you haven’t already. We’ll use the test\_that() function. It works as follows. Each call we make to test\_that() has two arguments: the first is message that describes what we are testing, and the second is a block of code that evaluates to TRUE or FALSE. Typically the block of code will use an expect\_\*() function, as seen in the debugging examples, in the last line. The structure is thus:

test\_that("Message specifying what we're testing", {  
 some code can be here  
 some more code can be here  
 expect\_\*(more code goes here)   
})

If the output of your code is TRUE (the test passes), then the call to test\_that() will show a message the test passed; if the output of your code is FALSE (the test fails), then we’ll get an error message signifying this.

1. Finish writing the following unit tests for add.up.inv.powers() as indicated by the message.

library(testthat)

Redefining the Function

add.up.inv.powers = function(n, verbose=FALSE) {  
 x = 0  
 for (i in 1:n) {  
 x = x + i^(1/i)  
 if (verbose) roman.cat(i)  
 }  
 if (verbose) cat("\n")  
 return(x)  
}  
  
roman.cat = function(num) {  
 roman.num = as.roman(num)  
 roman.str = as.character(roman.num)  
 cat(roman.str, "... ")  
}

1. Test the output for a single case, n=3, by writing manually the formula from 2 for n=3 inside the test code.

test\_that("add.up.inv.powers() works for n=3", {  
 formula = 1^1 + 2^(1/2) + 3^(1/3)  
 expect\_equal(add.up.inv.powers(3), formula)  
})

## Test passed 🎉

1. Test the output errors for input n="c".

test\_that("add.up.inv.powers() fails for non-integer n", {  
 error = "NA/NaN argument"  
 expect\_error(suppressWarnings(add.up.inv.powers("c")), error)  
})

## Test passed 😀