Lab 11: Debugging and Testing

Statistical Computing, 36-350

Week of Tuesday November 15, 2022

Name:
Andrew ID:
Collaborated with:

This lab is to be done in class (completed outside of class time if need be). You can collaborate with your classmates, but you must identify their names above, and you must submit **your own** lab as an knitted PDF file on Gradescope, by Friday 9pm, this week.

This week's agenda: practicing debugging with `cat()`, `print()`, and `browser()`; simple testing.

Q1. Bug hunt practice

In this section of the lab, you will fix a bunch of buggy function definitions. Probably the easiest workflow is to define the function in your console, and then run the sample commands—they will either give errors or produce the wrong outputs. Using any combination of: reading the error messages, `traceback()`, and `cat()` or `print()`, you must find and fix the bugs. Sometimes it can also help to try multiple different inputs, i.e., try new function calls, rather than just looking at the sample calls given to you, in order to determine the bugs. You shouldn't show any of your debugging work in your final knitted answers—so, don't show calls to `traceback()`, and don't leave any `cat()` or `print()` calls in the final, fixed function. (You don't have to do anything yet, this was just to setup this section of the lab.)

- **1a.** Below is a function called `get.cols.with.ab.zeros()`, but it has a few bugs. A few sample matrices are given below in `mat`, `identity.mat`, along with some sample calls that give errors. After fixing the bugs, the calls to `get.cols.with.ab.zeros()` should produce the outputs as described in comments.

```
# Function: cols.with.ab.zeros, to retrieve columns of matrix that have between
#   a and b zeros, each
# Inputs:
# - my.mat: the original matrix
# - a: lower bound for number of zeros allowed; default is 0
# - b: upper bound for number of zeros allowed; default is Inf
# Output: the new matrix

cols.with.ab.zeros = function(my.mat, a=0, b=Inf) {
  zeros.per.column = colSums(mat != 0)
  i.to.keep = a <= zeros.per.column && zeros.per.column <= b
  return(my.mat[i.to.keep,])
}

mat = matrix(c(0,0,1,0,1,1,1,1,1), 3, 3)
identity.mat = diag(1, 3)
cols.with.ab.zeros(mat) # Should get back original matrix
```

```
## Warning in a <= zeros.per.column && zeros.per.column <= b:
## 'length(x) = 3 > 1' in coercion to 'logical(1)'

## Warning in a <= zeros.per.column && zeros.per.column <= b:
## 'length(x) = 3 > 1' in coercion to 'logical(1)'

##      [,1] [,2] [,3]
## [1,]    0    0    1
## [2,]    0    1    1
## [3,]    1    1    1
```

```
cols.with.ab.zeros(mat, a=1, b=2) # Should get back first 2 columns of mat
```

```
## Warning in a <= zeros.per.column && zeros.per.column <= b:
## 'length(x) = 3 > 1' in coercion to 'logical(1)'

## Warning in a <= zeros.per.column && zeros.per.column <= b:
## 'length(x) = 3 > 1' in coercion to 'logical(1)'

##      [,1] [,2] [,3]
## [1,]    0    0    1
## [2,]    0    1    1
## [3,]    1    1    1
```

```
cols.with.ab.zeros(mat, a=2, b=2) # Should get just 1st column of mat; note
```

```
## Warning in a <= zeros.per.column && zeros.per.column <= b:
## 'length(x) = 3 > 1' in coercion to 'logical(1)'

##      [,1] [,2] [,3]
```

```
  # this should still be a matrix though, and not a numeric vector!
cols.with.ab.zeros(identity.mat, a=2, b=2) # Should get back original matrix
```

```
## Warning in a <= zeros.per.column && zeros.per.column <= b:
## 'length(x) = 3 > 1' in coercion to 'logical(1)'

##      [,1] [,2] [,3]
```

- **1b.** Below is a function called `list.extractor()`, but it has a few bugs. A sample list is given below in `cool.list`, along with some sample calls that give errors. After fixing the bugs, the calls to `list.extractor()` should produce the outputs as described in comments.

```
# Function: list.extractor, to extract elements of a list
# Inputs:
# - my.list: the original list
# - i.to.keep: vector of indices, corresponding to elements of the list we
#   want to keep. Default is NULL, in which case this argument is ignored
# - i.to.remove: vector of indices, corresponding to elements of the list we
#   want to remove Default is NULL, in which case this argument is ignored.
#   NOTE: if both i.to.keep and i.to.remove are non-NULL, then the first
#   one should take precedence (i.e., we don't remove anything)
# Output: the new list

list.extractor = function(my.list, i.to.keep=NULL, i.to.remove=NULL) {
  if (i.to.keep!=NULL) {
    L = my.list[[i.to.keep]]
  }
  if (i.to.remove!=NULL) {
```

```
    L = my.list[[-i.to.remove]]
  }
  return(L)
}

cool.list = list(ints=1:10, lets=letters[1:8], fracs=1:7/7,
                 bools=sample(c(TRUE,FALSE), 5, replace=TRUE))
list.extractor(cool.list, i.to.keep=c(1,3)) # Should get list with ints, fracs
```

```
## Error in if (i.to.keep != NULL) {: argument is of length zero
```

```
list.extractor(cool.list, i.to.remove=4) # Should get list without bools
```

```
## Error in if (i.to.keep != NULL) {: argument is of length zero
```

```
list.extractor(cool.list, i.to.keep=2:4, i.to.remove=4) # Should get list with
```

```
## Error in if (i.to.keep != NULL) {: argument is of length zero
```

```
  # lets, fracs, and bools (the i.to.remove argument should be ignored)
```

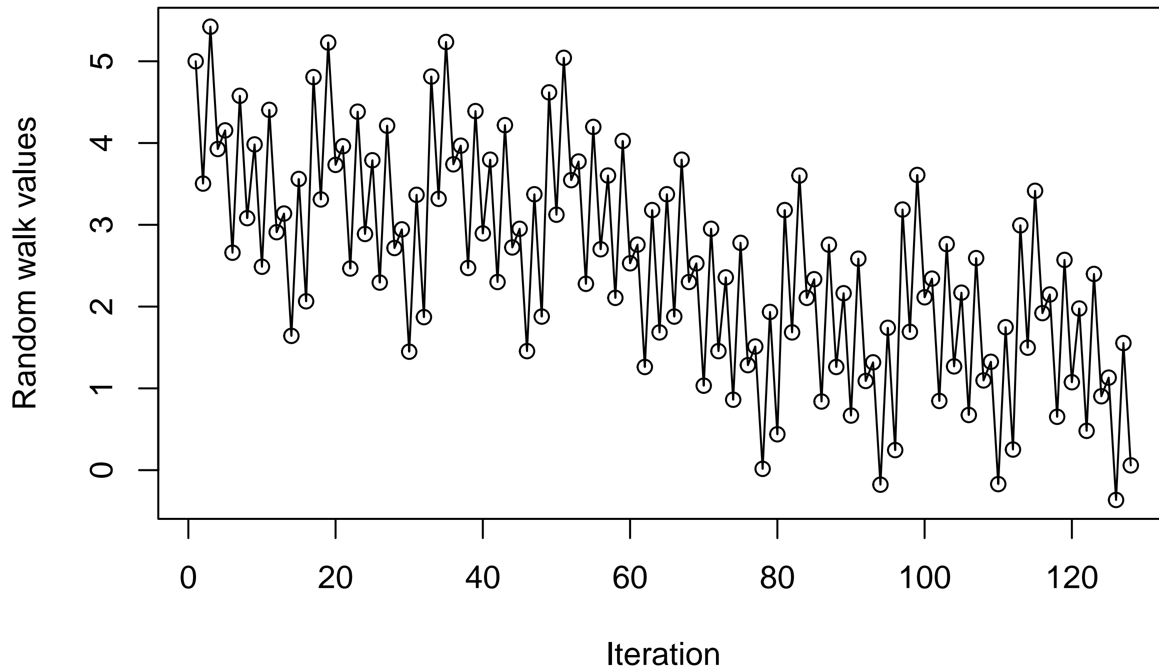
- **1c.** Below is a function called `random.walk()`, but it has a few bugs. Some sample calls are given below that produce errors. After fixing the bugs, the calls to `random.walk()` should produce the outputs as described in comment.

```
# Function: random.walk, to run a simple random walk over the reals, which
#   terminates when it reaches 0
# Inputs:
# - x.start: starting position. Default is 5
# - plot.walk: should the result be plotted? Default is TRUE
# - seed: integer seed to pass to set.seed(). Default is NULL, which means
#   effectively no seed is set
# Output: a list with elements x.vals, the values visited by the random walk,
#   and num.steps, the number of steps taken before termination

random.walk = function(x.start=5, plot.walk=TRUE, seed=NULL) {
  if (!is.null(seed)) set.seed(seed) # Set the seed, if we need to
  x.vals = x.start
  while (TRUE) {
    r = runif(1, -2, 1)
    if (tail(x.vals+r,1) <= 0) break
    else x.vals = c(x.vals, x.vals+r)
  }
  if (plot.walk <- TRUE)
    plot(x.vals, xlab="Iteration", ylab="Random walk values", type="o")
  return(x.vals=x.vals, num.steps=length(x.vals))
}

random.walk(x.start=5, seed=3)$num.steps # Should print 8 (this is how many
```

```
## Error in return(x.vals = x.vals, num.steps = length(x.vals)): multi-argument returns are not permitte
```

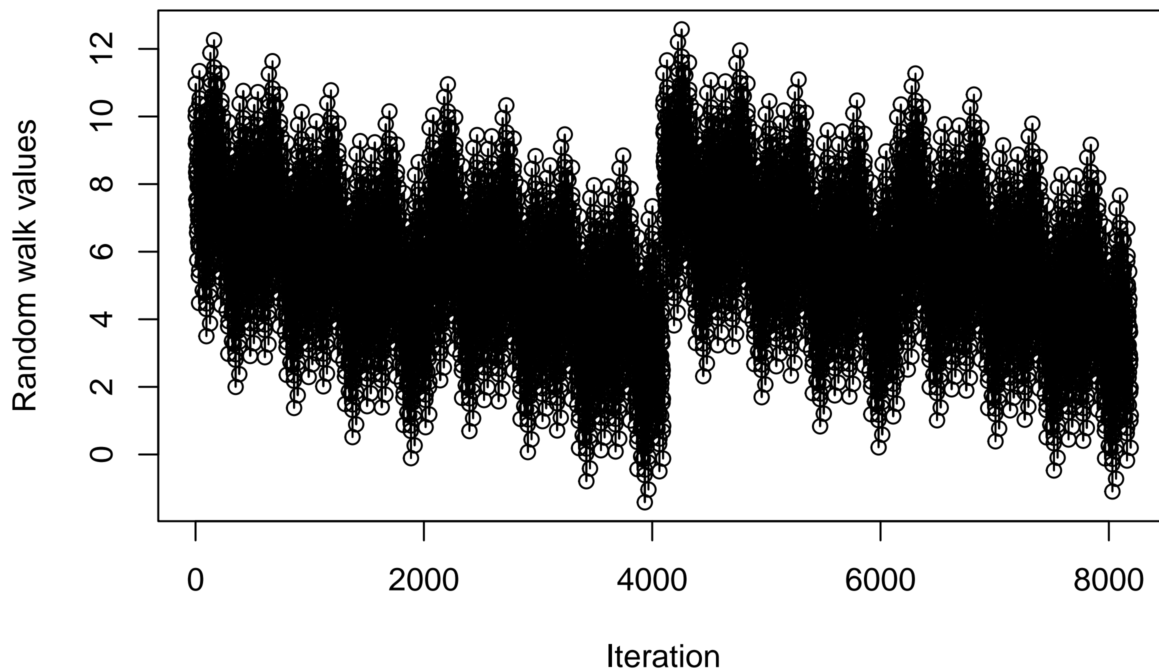



```
  # steps it took the random walk), and produce a plot
random.walk(x.start=10, seed=7)$num.steps # Should print 14 (this is how many
```

```
## Error in return(x.vals = x.vals, num.steps = length(x.vals)): multi-argument returns are not permitte
```

```
  # steps it took the random walk), and produce a plot
random.walk(x.start=10, plot.walk=FALSE, seed=7)$num.steps # Should print 14
```

```
## Error in return(x.vals = x.vals, num.steps = length(x.vals)): multi-argument returns are not permitte
```




```
  # (this is how many steps it took the random walk), and not produce a plot
```

Q2. Browsing practice

- **2a.** Below is a function `add.up.inv.powers()` that computes $1^1+2^{1/2}+\ldots+(n-1)^{1/(n-1)}+n^{1/n}$, via a `for()` loop, for some value of n, 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. But before this, for good vectorization practice, show that you can compute the same expression as done in `add.up.inv.powers()`, but without any explicit looping, i.e., just using vectorization and `sum()`. Check that you get the same answers for the demo inputs. Hint: for this, you can use `all.equal()`, to check for "very near" equality, since you may not get exact equality in all digits.

```
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, verb=FALSE)
```

```
## [1] 3.856463
```

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

```
## [1] 6.650406
```

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

```
## [1] 13.15116
```

- **2b.** Copy and paste the definition of `add.up.inv.powers()` below, into an R code chunk that should *not* be evaluated when you knit (hence the use of `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 `verbose=TRUE`.

  Now you'll enter the RStudio browser mode. First, just look around: you should see the "Console" panel (as always), the "Source Viewer" panel, the "Environment" panel, and the "Traceback" panel. (The console is arguably the most important but the others add nice graphical displays.) Hit the return key repeatedly (while your cursor is in the console) to step through the function line by line, until you get to the last line of the function. Once this last line is run, you'll immediately exit the browser mode. Try the whole process again a few times, each time looking at the various R Studio panels and familiarizing yourself with what they are displaying. Instead of hitting the return key, note that you can type "n" in the console to step to the next line. Note also that you can type in variable names in the console and hit enter, to see their current values (alternatively, the "Environment" panel will show you this too).

```
# YOUR CODE GOES HERE
```

- **2c.** Answer the following questions, exploring what you can do in browser mode. - 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.) - How do you exit the browser mode prematurely, before the last line is reached?
 - Suppose you were to run the browser with a call like `cool.new.num = add.up.inv.powers(n=5)` in the console; if you ran the browser to completion, would the variable `cool.new.num` be defined in your console? - 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? - Can you define new variables while in browser mode? - 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?
 - What happens if you change the location of the call to `browser()` within the definition of the function `add.up.inv.powers()`?

- **2d.** 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?
- **2e.** Typing the "c" key in browser mode will exit browser mode and continue on with normal evaluation. Try this too.
- **2f.** 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?

Q3. Browsing for bugs

- **3a.** Now that you've had good practice with it, use `browser()` to find and fix bugs in the function `fibonacci()` below. This function is supposed to generate the nth 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. 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 match those described in the comments.

```
fibonacci = function(n) {
  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])
}

fibonacci(1) # Should be 1
```

```
## Error in my.fib[i + 1] = my.fib[i] + my.fib[i - 1]: replacement has length zero
```

```
fibonacci(2) # Should be 1
```

```
## Error in my.fib[i + 1] = my.fib[i] + my.fib[i - 1]: replacement has length zero
```

```
fibonacci(3) # Should be 2
```

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

```
fibonacci(5) # Should be 5
```

```
## [1] 3
```

```
fibonacci(9) # Should be 34
```

```
## [1] 21
```

- **3b.** 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 match those described in the comments.

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

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

# Should be "eht kciuq nworb xof depmuj revo eht yzal god"
sentence.flipper("the quick brown fox jumped over the lazy dog")
```

```
## [1] "c(\"t\", \"h\", \"e\", \" \", \"q\", \"u\", \"i\", \"c\", \"k\", \"
\", \"b\", \"r\", \"o\", \"w\", \"n\", \" \", \"f\", \"o\", \"x\", \" \",
\"j\", \"u\", \"m\", \"p\", \"e\", \"d\", \" \", \"o\", \"v\", \"e\", \"r\",
\" \", \"t\", \"h\", \"e\", \" \", \"l\", \"a\", \"z\", \"y\", \" \", \"d\",
\"o\", \"g\")"
```

```
# Should be "ot eb ro on ot eb taht si eht noitseuq"
sentence.flipper("to be or no to be that is the question")
```

```
## [1] "c(\"t\", \"o\", \" \", \"b\", \"e\", \" \", \"o\", \"r\", \" \",
\"n\", \"o\", \" \", \"t\", \"o\", \" \", \"b\", \"e\", \" \", \"t\", \"h\",
\"a\", \"t\", \" \", \"i\", \"s\", \" \", \"t\", \"h\", \"e\", \" \", \"q\",
\"u\", \"e\", \"s\", \"t\", \"i\", \"o\", \"n\")"
```

- **3c.** 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 match those described in the comments.

```
# Redefine sentence.flipper() here

# Should be "olleh ssenkrad ym dlo dneirf",
#           "ev'i emoc ot kaeps htiw uoy niaga"
sentence.flipper(c("hello darkness my old friend",
                   "i've come to speak with you again"))
```

```
## [1] "c(\"h\", \"e\", \"l\", \"l\", \"o\", \" \", \"d\", \"a\", \"r\",
\"k\", \"n\", \"e\", \"s\", \"s\", \" \", \"m\", \"y\", \" \", \"o\", \"l\",
\"d\", \" \", \"f\", \"r\", \"i\", \"e\", \"n\", \"d\") c(\"i\", \"'\", \"v\",
\"e\", \" \", \"c\", \"o\", \"m\", \"e\", \" \", \"t\", \"o\", \" \", \"s\",
\"p\", \"e\", \"a\", \"k\", \" \", \"w\", \"i\", \"t\", \"h\", \" \", \"y\",
\"o\", \"u\", \" \", \"a\", \"g\", \"a\", \"i\", \"n\")"
```

```
# Should be "reven annog evig uoy pu",
#           "reven annog tel uoy nwod",
#           "reven annog nur dnuora dna tresed uoy"
sentence.flipper(c("never gonna give you up",
                   "never gonna let you down",
                   "never gonna run around and desert you"))
```

```
## [1] "c(\"n\", \"e\", \"v\", \"e\", \"r\", \" \", \"g\", \"o\", \"n\",
\"n\", \"a\", \" \", \"g\", \"i\", \"v\", \"e\", \" \", \"y\", \"o\", \"u\",
\" \", \"u\", \"p\") c(\"n\", \"e\", \"v\", \"e\", \"r\", \" \", \"g\", \"o\",
\"n\", \"n\", \"a\", \" \", \"l\", \"e\", \"t\", \" \", \"y\", \"o\", \"u\",
\" \", \"d\", \"o\", \"w\", \"n\") c(\"n\", \"e\", \"v\", \"e\", \"r\", \" \",
\"g\", \"o\", \"n\", \"n\", \"a\", \" \", \"r\", \"u\", \"n\", \" \", \"a\",
\"r\", \"o\", \"u\", \"n\", \"d\", \" \", \"a\", \"n\", \"d\", \" \", \"d\",
\"e\", \"s\", \"e\", \"r\", \"t\", \" \", \"y\", \"o\", \"u\")"
```

- **3d.** Define a function `sentence.scrambler()` that operates similarly to to `sentence.flipper()`, but which randomly scrambles the order of characters in each word, instead of deterministically reversing them. The function `sentence.scrambler()` should be vectorized, just like the current version of `sentence.flipper()`. Hint: you can use `browser()` at any point if you run into bugs in your development, or simply to see how your function is handling certain test inputs. Also, the implementation of `sentence.scrambler()` should be pretty similar to `sentence.flipper()`; really, you just need to replace `word.flipper()` by a suitable function. Once done, run `sentence.scrambler()` on the test string below to display the output.

```
# Define sentence.scrambler() here

sentence.scrambler(c("I have no theorems, well",
  "I do have theorems, but none of them are named Fienberg's Theorem",
  "Even if there were a Fienberg's Theorem, it probably wouldn't be important",
  "What's important is the attitude, for what statistics is",
  "and how it's recognized by other people outside of our field"))
```

```
## Error in sentence.scrambler(c("I have no theorems, well", "I do have theorems, but none of them are n
```

Q4. Testing practice

- **4a.** Download and install the `assertthat` package, if you haven't already. Using the `assert_that()`, add assertions to the start of your (final, bug-fixed) `random.walk()` function from from Q1c to ensure that the inputs being passed in are of the correct type. Demonstrate by example that these work and pass informative error messages by calling `random.walk()` with faulty inputs.

```
# YOUR CODE GOES HERE
```

- **4b.** Similarly, add assertions to `sentence.flipper()` from Q3c to ensure proper inputs, and then demonstrate via examples that these work as expected.

```
# YOUR CODE GOES HERE
```

- **4c.** 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 `expect_true()` or `expect_error()`, in the last line. The structure is thus:

```
test_that("Message specifying what we're testing", {
  code goes here
  code goes here
  expect_true(code goes here)
})
```

If the output of your code is TRUE (the test passes), then the call to `test_that()` will show nothing; if the output of your code is FALSE (the test fails), then we'll get an error message signifying this. Here is an example for checking that the function from Q2a works for `n=3`:

```
test_that("add.up.inv.powers() works for n=3", {
  res = add.up.inv.powers(n=3, verb=FALSE)
  expect_true(res==(1 + 2^(1/2) + 3^(1/3)))
})
```

And another example for checking that the function from Q2a fails for non-integer inputs:

```
test_that("add.up.inv.powers() fails for non-integer n", {
  expect_error(add.up.inv.powers(n="c", verb=FALSE))
})
```

Running these code chunks (once you have the `testthat` package installed), you'll find that neither of these calls to `test_that()` produced any messages, which means that tests executed as we expected.

Now implement your own unit tests for `sentence.flipper()` from Q3c. Write several tests, checking that it flips simple sentences as you'd expect, that it returns a vector of strings when it's passed a vector of strings, and that its output vector has the same length as the input vector.

```
# YOUR CODE GOES HERE
```

- **Challenge.** Write a sensible unit test for `sentence.scrambler()`. Because its behavior is random, you can't exactly predict what it should do on simple strings (like you could with `sentence.flipper()`), but that doesn't mean you can't write a sensible unit test. Hint: after scrambling the order of the letters in each word, think about what would be preserved, before and after scrambling . . .

```
# YOUR CODE GOES HERE
```