HW: Week 8

36-350 – Statistical Computing

Week 8 – Fall 2020

Name: Kimberly Zhang

Andrew ID: kyz

You must submit **your own** lab as a PDF file on Gradescope.

If you do not have a `GitHub` account, you should sign up for one before proceeding.

If you have not installed and configured `Git`, you should do that before proceeding.

Question 1

(10 points)

Notes 8A (4-6)

Show us that you have a `GitHub` account. Create a repository on `GitHub` called "36-350". (Utilize the checklist in `Notes_8A`.) Then edit the code below so that we see the contents of `README.md`. To get the correct URL, do the following: go to your `GitHub` repo, click on `36-350` and then again on `README.md`, and click on the "Raw" button. Copy and paste the URL to the raw `README.md` file into the call to `readLines()` below.

```
readLines("https://raw.githubusercontent.com/kyz128/36-350/main/README.md")
```

```
## [1] "# 36-350"                  "CMU Statistical Computing"
```

Question 2

(10 points)

Notes 8A (5,7-8)

Show us that you have `Git` installed on your computer. Utilize the checklist in `Notes_8A` to create a new project within `RStudio` that is tied to your "36-350" repo on `GitHub`. Then follow the listed steps in `Notes_8A` to create a new `R Script` (and *not* an `R Markdown` file) in which you put `print("It was a dark and stormy night.")`. Save this file (call it `dark_and_stormy.R`) to your local "36-350" repo. Stage the file, commit the file (and add a commit message), and push the file to `GitHub`. (If when you try to commit you see an error referring to an `index.lock` file, try to commit again... I've seen such an error when trying to commit files on my machine and it appears to be a random occurrence.) Follow the steps that you followed in Q1 to find the URL to the raw file for `dark_and_stormy.R` and copy and paste that URL below in the call to `source_url()`. If everything works, "It was a dark and stormy night." should appear, along with a hash code that you can safely ignore.

```
if ( require("devtools") == FALSE ) {
  install.packages("devtools",repos="https://cloud.r-project.org")
  library(devtools)
}
```

```
## Loading required package: devtools
```

```
## Loading required package: usethis
```

```
source_url("https://raw.githubusercontent.com/kyz128/36-350/main/dark_and_stormy.R")
```

```
## SHA-1 hash of file is 80536db0829e00e85e68fb1f5085858fd311ed6c
```

```
## [1] "It was a dark and stormy night."
```

Question 3

(10 points)

Notes 8A (8-9)

Following the instructions in `Notes_8A`, create a new branch both on `GitHub` and in your local project. Call it `new_branch`. Once you have done this, alter `dark_and_stormy.R` so that it prints "It was a dark and stormy night; the rain fell in torrents." Stage, commit, and push as you would have done in Q2. Source the `main` branch file as you did in Q2, and also source the `new_branch` file. If the output of the first does not include "the rain fell in torrents," while the output of the second one does include that phrase, you're good.

```
source_url("https://raw.githubusercontent.com/kyz128/36-350/main/dark_and_stormy.R")
```

```
## SHA-1 hash of file is 80536db0829e00e85e68fb1f5085858fd311ed6c
```

```
## [1] "It was a dark and stormy night."
```

```
source_url("https://raw.githubusercontent.com/kyz128/36-350/new_branch/dark_and_stormy.R")
```

```
## SHA-1 hash of file is eb7745d2c0833afc02d7217e2cb0d1366514fd3b
```

```
## [1] "It was a dark and stormy night; the rain fell in torrents."
```

Question 4

(10 points)

Not in Notes 8A

Another way to create a code branch is to "fork" a repository. Two words that you will hear when working with `GitHub` repos are "fork" and "clone." The former refers to creating a new version of a remote repo in your own account and then building upon it, with little or no intention to try to merge your changes back to the remote repo. For instance, perhaps someone created a code in 2016 that they are done with, but you want to use it and edit it. Fork the repo the code is in, and play with your copy in perpetuity. On the other hand, if you goal is to edit a main code base as part of collaborative development, you would want to clone a repo, make changes, and submit a so-called "pull request" to the owner of the main code base. (This is a request for that owner to "pull" your changes into the code that he or she is hosting.)

In this exercise, you are going to fork the `36-350-Fork` repository in my (i.e., `pefreeman`'s) account. Point your browser to `https://github.com/pefreeman/36-350-Fork` and click on the `Fork` button at top right. When/if you are prompted as to where you want the forked repository to go, indicate that you want it to go to *your* `GitHub` account. Once you've done that, source via `source_url()` the `check_mark.R` file in the repo. The path to the file should include *your account name*, and not mine! (Note that when you run the code chunk, you may see a "spinning circle," with the ultimate output not shown. This is OK... the file should still knit with the correct output.)

```
source_url("https://raw.githubusercontent.com/kyz128/36-350-Fork/main/check_mark.R")
```

```
## SHA-1 hash of file is 8b1dfa670156668f1b5fb809e7a5221d224b21b2
```

```
## -----*
## ----*-
## ---*--
## *-*---
## -*----
```

Note that at this point, you will have a forked `36-350-Fork` repo, as well as a `36-350` repo that has two branches. Leave the repo and branches alone, so that you can always generate the output you need to answer Q3 and Q4. However, *after you have turned in your homework*, you can do the following on `GitHub` if you choose to (however, see #3):

1. Regarding the branches: from either the `main` or `new_branch` branch, you can click on the button "Compare & pull request". On a new page, you should see the words "Able to merge. These branches can be automatically merged." This is because the comparison shows that all one has to do to merge the files is add a clause. If the differences were more complex (because, e.g., you deleted "It was a dark and stormy night" and tried to merge something else, like "Hello world; the rain fell in torrents."), `GitHub` would not be certain how to proceed and would ask you how to do so.
2. Regarding the branches: click on "Create pull request." After this is done, you will find yourself in a state where `GitHub` says you can safely delete the new branch.
3. Regarding the branches: unfortunately, deleting the new branch in `RStudio` is a bit more complex, involving having to go into the `RStudio` terminal and type command-line `Git` commands. Avoiding this is another reason just to leave the branches alone.
4. Regarding the fork: you can delete `36-350-Fork`. On `GitHub`, this involves going to "Settings," scrolling down to the "Danger Zone" (no lie, that's what it is called), and clicking on "Delete this repository."

As far as your `36-350` repo is concerned: I would suggest that you leave the branches alone, and when the course is completely done (and not before, because we might revisit this repo in the future), just delete the `36-350` repo from both your computer and from `GitHub`. (If you want to. Having them sitting around does not hurt anyone... it just depends on whether you care that there are extraneous repositories in your `GitHub` account.) On `GitHub`, follow the instructions given in point (4) above. On your computer, it is as simple as removing the directory with the repo (i.e., the `36-350` directory and all its sub-directories).

In the following questions, utilize this base code by copying and pasting it into your code chunks, then adding material. (`error=TRUE` will cause `R Markdown` to keep knitting even if you throw an exception.)

```
f = function(x) {
  toupper(x)
}
```

Question 5

(5 points)

Notes 8B (2)

Add an appropriate warning, but don't change what is returned. Call the function is such a way that the warning message is observed. Was the final output from the function what you expected it to be?

```
f = function(x) {
  if (typeof(x) != "character"){warning("Input not of character type")}
  toupper(x)
}
f(5)
```

```
## Warning in f(5): Input not of character type
```

```
## [1] "5"
```

```
I was actually expecting an error but it seems that R
implicitly converted the int to a char.
```

Question 6

(5 points)

Notes 8B (2)

Change the warning in Q5 to an error. Call the function is such a way that the error message is observed.

```
f = function(x) {
  if (typeof(x) != "character"){stop("Input not of character type")}
  toupper(x)
}
f(5)
```

```
## Error in f(5): Input not of character type
```

Question 7

(5 points)

Notes 8B (2)

Keep your code from Q6, but add a message at the beginning saying what the function is supposed to do when called properly. Run the function two times, once with improper input, and a second time with proper input (e.g., "a") but in conjunction with `suppressMessages()`, so the message is not observed.

```
f = function(x) {
  message("Function converts x to uppercase when called properly")
  if (typeof(x) != "character"){
    stop("Input not of character type")}
  toupper(x)
}
f(2)
```

```
## Function converts x to uppercase when called properly
```

```
## Error in f(2): Input not of character type
```

```
suppressMessages(f("a"))
```

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

In the following questions, utilize this base code by copying and pasting it into your code chunks, then adding material.

```
f = function(x) {
  log(x)
  print("hello")
}
```

Question 8

(5 points)

Notes 8B (7)

In the code chunk below, first run the code as it is defined above with a character input. Confirm that an error is generated and that you don't see the word "hello". Then, below that function call, redefine the function `f()` so that it includes a call to `try()` at an appropriate place. Then call your updated function with a character input. If you do not see the word "hello" printed, something has gone wrong.

```
f = function(x) {
  log(x)
  print("hello")
}
f("a")
```

```
## Error in log(x): non-numeric argument to mathematical function
```

```
f = function(x) {
  try(log(x))
  print("hello")
}
f("a")
```

```
## Error in log(x) : non-numeric argument to mathematical function
## [1] "hello"
```

Question 9

(5 points)

Notes 8B (8-9)

Alter your function in Q8, replacing the `try()` with `tryCatch()` (and deleting the `print("hello")`). Catch what you feel are the relevant conditions here, and deal with them appropriately (by informing the user what went wrong). (For instance... do you think `log()` issues informative messages? Or not?) Call the function with a character, call it again with a negative number, and call it one last time with 0 as input.

```
f = function(x) {
  tryCatch({
    if (x < 0) stop("Negative input; out of natural log's domain")
    if (x == 0) warning("Input is 0, output should be indeterminate")
    log(x)},
    error = function(c) print(c$message),
    warning = function(c) print(c$message))
  print("hello")
}
f("a")
```

```
## [1] "non-numeric argument to mathematical function"
## [1] "hello"
```

```
f(-5)
```

```
## [1] "Negative input; out of natural log's domain"
## [1] "hello"
```

```
f(0)
```

```
## [1] "Input is 0, output should be indeterminate"
## [1] "hello"
```

Question 10

(5 points)

Notes 8B (8-9)

Repeat Q9, but add a `finally` argument to `tryCatch()` that prints (or more elegantly, if you want, cats) the value of x. Repeat the testing calls made in Q9.

```
f = function(x) {
  tryCatch({
    if (x < 0) stop("Negative input; out of natural log's domain")
    if (x == 0) warning("Input is 0, output should be indeterminate")
    log(x)},
    error = function(c) print(c$message),
    warning = function(c) print(c$message),
    finally = {cat(cat("Input is ", x), "\n")})
  print("hello")
}
f("a")
```

```
## [1] "non-numeric argument to mathematical function"
## Input is  a
## [1] "hello"
```

```
f(-5)
```

```
## [1] "Negative input; out of natural log's domain"
## Input is  -5
## [1] "hello"
```

```
f(0)
```

```
## [1] "Input is 0, output should be indeterminate"
## Input is  0
## [1] "hello"
```

Question 11

(15 points)

Notes 8C (4-7)

You are given the following code that is meant to convert one single-character string into a numeric value: "a" maps to 0, "b" maps to 0.693, etc. In theory, it should work with upper- and lower-case letters, and should throw an exception if non-character input is provided, and should throw an exception if more than one string is input, and should issue a warning and only use the first character if the number of characters in the string is greater than 1.

```
f = function(letter) {
  return(log(which(letters==letter)))
}
```

(Yeah, whoever gave this to you is a bit lazy.) As a dutiful member of the team, your first responsibility is to write a series of tests utilizing functions in the `testthat` that will determine whether or not this code is operating correctly, given the stated expectation of how it is to perform. Below, write at least five different test function calls, at least two of which should fail. (Don't use any one `testthat` function, like `expect_equal()`, more than twice.) They can include tests that you know will fail, based on some future expectation: for instance, you can test whether a certain input leads to a thrown exception (that test would fail currently). (Or whether that same input yields an error.) Then, when you improve the code in Q12, you can improve it in such a way that your now-known-to-fail test will pass the next time.

```
if ( require("testthat") == FALSE ) {
  install.packages("testthat",repos="https://cloud.r-project.org")
  library(testthat)
}
```

```
## Loading required package: testthat
```

```
##
## Attaching package: 'testthat'
```

```
## The following object is masked from 'package:devtools':
##
##     test_file
```

```
test_that(desc = "Test for string of length greater than 1",
          expect_warning(f("aa")))
```

```
## Error: Test failed: 'Test for string of length greater than 1'
## * `f("aa")` did not produce any warnings.
```

```
test_that(desc = "Test uppercase input", expect_equal(f("A"), 0))
```

```
## Error: Test failed: 'Test uppercase input'
## * f("A") not equal to 0.
## Lengths differ: 0 is not 1
```

```
test_that(desc = "Test lowercase input", expect_equal(f("a"), 0))
test_that(desc = "Test non-char input", expect_error(f(5)))
```

```
## Error: Test failed: 'Test non-char input'
## * `f(5)` did not throw an error.
```

```
test_that(desc = "Test input more than 1 string",
          expect_error(f(c("a", "b"))))
```

```
## Error: Test failed: 'Test input more than 1 string'
## * `f(c("a", "b"))` did not throw an error.
```

Question 12

(15 points)

Notes 8C (4-7)

Rewrite `f()` below in such a way that all your tests of Q11 pass. (And demonstrate that your tests pass!)

```
f = function(letter) {
  if ( typeof(letter) != "character" ) stop("Input type must be character.")
  if ( length(letter) != 1 ) stop("Input must be a string.")
  if ( nchar(letter) > 1 ) {
   letter = letter[1]
   warning("More than 1 letter. Only using the first letter of input.")
  }
  return(log(which(letters==tolower(letter))))
}
test_that(desc = "Test for string of length greater than 1",
          expect_warning(f("aa")))
test_that(desc = "Test uppercase input", expect_equal(f("A"), 0))
test_that(desc = "Test lowercase input", expect_equal(f("a"), 0))
test_that(desc = "Test non-char input", expect_error(f(5)))
test_that(desc = "Test input more than 1 string",
          expect_error(f(c("a", "b"))))
```