Homework 1, TSCI 5050

Alex F. Bokov, [bokov@uthscsa.edu](mailto:bokov@uthscsa.edu)

Student Name: Michael Nipper

This homework is graded pass/fail and cooperation is encouraged. Bring the homework (hard-copy) to class, but you can take it back with you and hand it in during the following class if you need more time.

Part 1.

The point of these exercises is to get you to appreciate patterns in code even when you can't fully rely on prior knowledge. Without running this code or looking anything up, see how far you can get writing down in the white space what you think each of the following blocks of code is intended to do.

1.1

riddle <- function (object = nm, nm)

{

names(object) <- nm

object

}

Initial: declares a function named "riddle" with input of nm. "names" is a function which does something with "object".

Final: takes two vectors and sets the second one to be the names attribute of the first, then prints it. Requires them to be of the same length.

1.2

conundrum <- function (len)

{

out <- c(1,1)

for (ii in 3:len){

out <- c(out,out[ii-1] + out[ii-2])

}

out

}

Initial: declares a function named "conundrum" with input of len. The function prints something, using data from dataset "c".

Final: takes a numerical input and creates a vector containing many numbers from the Fibonacci sequence, and then prints that vector

What do you think “<-” does?

Initial: Sets what's left of "<-" using information from the right of "<-"

What does “function” do?

Initial: Declares a function, with the parentheses inclosing arguments

1.3 This one is more challenging. But it is not one monolithic blob of gibberish, right?

mystery <- function (object, ..., digits = max(3L, getOption("digits") - 3L))

{

if (is.factor(object))

return(summary.factor(object, ...))

else if (is.matrix(object))

return(summary.matrix(object, digits = digits, ...))

value <- if (is.logical(object))

c(Mode = "logical", {

tb <- table(object, exclude = NULL)

if (!is.null(n <- dimnames(tb)[[1L]]) && any(iN <- is.na(n))) dimnames(tb)[[1L]][iN] <- "NA's"

tb

})

else if (is.numeric(object)) {

nas <- is.na(object)

object <- object[!nas]

qq <- stats::quantile(object)

qq <- signif(c(qq[1L:3L], mean(object), qq[4L:5L]), digits)

names(qq) <- c("Min.", "1st Qu.", "Median", "Mean", "3rd Qu.", "Max.")

if (any(nas))

c(qq, `NA's` = sum(nas))

else qq

}

else if (is.recursive(object) && !is.language(object) && (n <- length(object))) {

sumry <- array("", c(n, 3L), list(names(object), c("Length", "Class", "Mode")))

ll <- numeric(n)

for (i in 1L:n) {

ii <- object[[i]]

ll[i] <- length(ii)

cls <- oldClass(ii)

sumry[i, 2L] <- if (length(cls))

cls[1L]

else "-none-"

sumry[i, 3L] <- mode(ii)

}

sumry[, 1L] <- format(as.integer(ll))

sumry

}

else c(Length = length(object), Class = class(object), Mode = mode(object))

class(value) <- c("summaryDefault", "table")

value

}

Initial: declares function "mystery" with inputs, "object", a wildcard input, and "digits". Identifies whether "object" has certain qualities, and, if so, returns information about "object" related to the type of data that it determined "object" contains.

Final: does operations on the input based on what kind of input it is. Using this function on a numerical vector will give a 5 point summary of the distribution of elements of that vector and its mean. Given a matrix, it will give you the same output, but for each column of the matrix. When given a factor, it will give you a readout of its contents. When given a set of logical values, it will add the occurrences of each possible state and print them. When given a list it will print the kind and length of each element of the list. When given a vector of character elements, it will tell you the input was a "character" vector and give you its length; this seems to be the default behavior for any input type it recognizes as being R valid, but is not specifically accounted for.

I do not understand the distinction between 'class' and 'mode' in the element readout.

What is the purpose of the parentheses immediately after the “function”?

Initial: They contain the arguments of the function declared.

What is the purpose of “{” and “}”? How about “[” and “]”?

Initial: "{" and "}" mark where blocks of code begin and end. "[" and "]" seem to indicate specific parts of a list or table.

Final: I was right about "{" and "}". "[" and "]" enclose subscripts, that is, they indicate a specific element or elements within a vector or list. This can be used for extraction or replacement.

Part 2.

Please read the following sections in <http://www.burns-stat.com/pages/Spoetry/Spoetry.pdf> (S Poetry by Patrick J. Burns, 1998). If anything you read inspires you to update your answers in Part 1, please do so.

The basics.

1.1 – 1.3 (pp 1-13)

1.6 (pp 16-17)

1.8 – 1.9 (pp 18-19)

Now we get into Zen stuff. Once you get the point he's trying to make in each section, just skim until the next section.

2.1 – 2.4 (pp 23-30)

Some useful functions.

4.3, first part (pp 81- 83, the rest is optional but the abbreviate command is kind of cool)

We'll skip 4.4, but remind me to mention the simple and almost-universal way to get data into R.

4.8 (pp 110 – 113)

We'll skip 4.9, but remind me to mention the main graphics commands to know about.

Of course your code will always be perfect the first time you write it ;-) but maybe some of the following material can be helpful in troubleshooting your colleagues' work:

6.3 (pp 138 – 139, the rest is optional; if you do keep reading note that inspect is not a command in modern R, instead look at the help file for debug)

6.5 – 6.6 (pp 149 - 151)

Part 3.

Download RStudio from <http://www.rstudio.com/products/rstudio/download/> and install it if you haven't already. Try running the code from Part 1 and see what it does. Do you remember from section 6.3 (above) and from typing ?debug how to step through the code one command at a time? If there are any expressions in the Part 1 code that are still a mystery to you, underline just the parts you think are blocking your understanding, see how narrowly you can bound the parts you don't get.

Part 4

Think of a small but annoying data-related problem you deal with in real life and see how far you can get in RStudio writing a function to solve this problem. Save the function to hand in with the homework.

diceprob <- function(target, dice1=6, dice2=6){

if ((target<1) | (dice1<1) | (dice2<1)){

print("Target and dice values must be at least 1")

return(NULL)

}

dice <- c(dice1, dice2)

dicepos <- matrix(0, nrow=dice[1], ncol=dice[2])

for (i in 1:dice[1])

dicepos[i, 1:dice[2]] <- (i+(1:dice[2]))

posvector <- as.vector(dicepos)

possort <- order(posvector)

possorted <- rev(posvector[possort])

targetfreq = 0

for (i in 1:length(possorted)){

if (possorted[i] == target)

targetfreq = targetfreq+1

if (possorted[i] < target){

answer <- i-1

break

}

}

atleast <- answer/length(possorted)\*100

targetprob <- targetfreq/length(possorted)\*100

lessthan <- (length(possorted)-answer)/length(possorted)\*100

solutions <- c (atleast, targetprob, lessthan)

names(solutions) <- c("At least", "Exactly", "Less Than")

solutions

}

This function takes 1-3 integers. The probability that the first number is produced by the sum of the values of two fair dice with numbers of sides given by the second and third argument is given. The probability that the first argument will be at least produced by the sum is also stated, along with the probability that the first argument would not at least produced by the sum.