Practice - Programming Basics

Stat 133, Fall 2017, Prof. Sanchez

The underlying purpose of the exercises in this document is to put in practice the main programming concepts that we've seen so far in the course:

- working with various types of data objects
- working with R "compound" expression
- writing functions
- using conditionals
- using loops

Keep in mind that this document is less *guided*. We expect that you feel more comfortable with R, and you should also have more freedom to use data objects, names of objects and functions, control flow structures, and plotting approaches.

Before you start ...

If you use an `Rmd` to write narrative and code for this practice, you must include a code chunk at the top of your file like the one in the following screen capture:

````
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE, error = TRUE)
```
````

By setting the global option `error = TRUE` you avoid the knitting process to be stopped in case an code chunk generates an error.

Since you willl be writing a couple of functions with `stop()` statements, it is essential that you set up `error = TRUE`, otherwise `"knitr"` will stop knitting your `Rmd` if it encounters an error.

1) Toggling Switches

A room contains 100 toggle switches, originally all turned off. These switches can be initialized in R with the following character vector `switches`:

```
num_switches <- 100
switches <- rep("off", num_switches)

head(switches)
```

```
## [1] "off" "off" "off" "off" "off" "off"
```

100 people enter the room in turn. The first person toggles every switch, the second toggles every second switch, the third every third switch, and so on, to the last person who toggles the last switch only. Write R code to find out, at the end of this toggling process, which switches are turned on.

In addition to using the vector `switches`, you can use any control flow structure: if-then-else statements, function `switch()`, `for` loops, `repeat` loops, or `while` loops.

Strategy: In order to solve this problem, I suggest starting with a smaller vector of switches, for instance: `switches <- rep("off", 3)`. Workout the toggling process and see what happens. Then consider a larger vector: `switches <- rep("off", 5)` and follow the toggling process. Once you get the right code, then generalize to 100 switches.

2) Distance Matrix of Letters

The following code generates a random matrix `distances` with arbitrary distance values among letters in English:

```
# random distance matrix
num_letters <- length(LETTERS)
set.seed(123)
values <- sample.int(num_letters)
distances <- values %*% t(values)
diag(distances) <- 0
dimnames(distances) <- list(LETTERS, LETTERS)
```

The first 5 rows and columns of `distances` are:

```
distances[1:5, 1:5]
```

```
##     A   B   C   D   E
## A   0 160  80 168 184
## B 160   0 200 420 460
## C  80 200   0 210 230
## D 168 420 210   0 483
## E 184 460 230 483   0
```

Consider the following character vector `vec <- c('E', 'D', 'A')`. The idea is to use the values in matrix `distances` to compute the total distance between the letters: that is from `E` to `D`, and then from `D` to `A`:

```
# (E to D) + (D to A)
483 + 168
```

```
## [1] 651
```

Hence, you can say that the word `'E' 'D' 'A'` has a value of 651.

a) Write a function `get_dist()` that takes two inputs:

- `distances` = the matrix of distance among letters.
- `ltrs` = a character vector of upper case letters.

The function must return a numeric value with the total distance. Also, include a stopping condition—via `stop()`—for when a value in `ltrs` does not match any capital letter. The error message should be `"Unrecognized character"`

Here's an example of how you should be able to invoke `get_dist()`:

```
vec <- c('E', 'D', 'A')
get_dist(distances, vec)
```

```
## [1] 651
```

And here's an example that should raise an error:

```
err <- c('E', 'D', ')')
get_dist(distances, err)
```

```
## Error in get_dist(distances, err): Unrecognized character
```

Test your function with the following character vectors:

- `cal <- c('C', 'A', 'L')`
- `stats <- c('S', 'T', 'A', 'T', 'S')`
- `oski <- c('O', 'S', 'K', 'I')`
- `zzz <- rep('Z', 3)`
- `lets <- LETTERS`
- a vector `first` with letters for your first name, e.g. `c('G', 'A', 'S', 'T', 'O', 'N')`
- a vector `last` for your last name, e.g. `c('S', 'A', 'N', 'C', 'H', 'E', 'Z')`

b) Assuming that you alredy created the objects listed above, now create an R list `strings` like this:

```
# use your own 'first' and 'last' objects
strings <- list(
  cal = cal,
  stats = stats,
  oski = oski,
  zzz = zzz,
```

```
  lets = lets,
  first = first,
  last = last
)
```

Write a `for()` loop to iterate over the elements in `strings`, and compute their distances. At each iteration, store the calculated distances in a list called `strings_dists`; this list should have the same names as `strings`.

Confirm that your list `strings_dists` looks like this (except for `first` and `last`):

```
strings_dists
```

```
## $cal
## [1] 136
##
## $stats
## [1] 990
##
## $oski
## [1] 834
##
## $zzz
## [1] 0
##
## $lets
## [1] 4800
##
## $first
## [1] 731
##
## $last
## [1] 1310
```

3) Vending Machine

The next challenge is to write a set of functions that mimic the buying process of snacks from a vending machine.

Think about the buying process:

1. You look at the available `products`
2. Decide what item you want
3. Look at the corresponding keys *letter* and *number*

Figure 1: a vending machine

4. Input enough money (coins and bills)
5. Enter item keys
6. Get item
7. Get change (if necessary)

Products

Consider a simple vending machine with 16 products:

- Clif bars: peanut-toffee, brownie, carrot-cake, oatmeal-raisin
- Chocolate candy bars: m&m's, kit-kat, hersheys, snickers
- Luna bars: blueberry, lemonzest, vanilla, cookies
- Nabisco cookies: chips-ahoy, oreo, ritz, nutter-butter

Clif bars cost $1.60, candy bars cost $1.80, Luna bars cost $1.50, and cookies cost $1.70.

To simulate the products of the vending machine, you have to create a data frame `products` (containing NO factors) like this one:

```
     brand            bar price letter number
1    cliff  peanut-toffee   1.6      A      1
2    cliff        brownie   1.6      B      1
3    cliff    carrot-cake   1.6      C      1
```

```
4    cliff oatmeal-raisin   1.6      D      1
5     mars            m&m   1.8      A      2
6     mars        kit-kat   1.8      B      2
7     mars       hersheys   1.8      C      2
8     mars       snickers   1.8      D      2
9     luna      blueberry   1.5      A      3
10    luna      lemonzest   1.5      B      3
11    luna        vanilla   1.5      C      3
12    luna        cookies   1.5      D      3
13 nabisco     chips-ahoy   1.7      A      4
14 nabisco           oreo   1.7      B      4
15 nabisco           ritz   1.7      C      4
16 nabisco  nutter-butter   1.7      D      4
```

Show Price

Write a function `show_price()` with three arguments: `products`, `letter`, and `number`. The function should return the price of the selected item with the name of the bar (i.e. a named numeric vector).

You should be able to call `show_price()` like this:

```
show_price(products, letter = 'A', number = 1)
```

```
## $bar
## [1] "peanut-toffee"
##
## $price
## [1] 1.6
```

Buy Item

Write a function `buy_item()` with four arguments:

- `products` = data frame of candy bars
- `letter` = a single character, any of `A`, `B`, `C` or `D` (default `A`)
- `number` = single number, any of 1, 2, 3 or 4 (default 1)
- `money` = numeric vector with input amount (default 0)

The default call of `buy_item()` is like this:

```
buy_item(products, letter = 'A', number = 1, money = 2)
```

```
## $bar
## [1] "peanut-toffee"
##
```

```
## $price
## [1] 1.6
##
## $money
## [1] 2
##
## $change
## [1] 0.4
```

When the amount is 0, the function should print a message `"not enough money"`

The argument `amount` is a numeric vector. This represents the input amount.

4) Distances in Euclidean Spaces

For a point $(x_1, x_2, \ldots, x_n)$ and a point $(y_1, y_2, \ldots, y_n)$, the Minkowski distance of order p (p-norm distance) is defined as:

$$\text{1-norm distance} = \sum_{i=1}^{n} |x_i - y_i|$$

$$\text{2-norm distance} = \left(\sum_{i=1}^{n} |x_i - y_i|^2 \right)^{1/2}$$

$$\text{p-norm distance} = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}$$

$$\text{infinity norm distance} = max(|x_1 - y_1|, |x_2 - y_2|, \ldots, |x_n - y_n|)$$

p need not be an integer, but it cannot be less than 1, because otherwise the triangle inequality does not hold.

https://en.wikipedia.org/wiki/Distance

Write a function `minkowski()` for the Minkowski distances, without using vectorized code. In other words, you have to use loops.

- The function should take three arguments:
  - `x` = numeric vector for one point
  - `y` = numeric vector for the other point
  - `p` = numeric value greater than 1

- Check that `x` and `y` have the same length, otherwise raise an error.
- Check that `p` is greater than 1, otherwise raise an error.

Test your function

```
point1 <- c(0, 0)
point2 <- c(1, 1)

# manhattan distance
minkowski(point1, point2, p = 1)
```

```
## [1] 2
```

```
# euclidean distance
minkowski(point1, point2, p = 2)
```

```
## [1] 1.414214
```

5) Two Given Points

Let p_1 and p_2 be two points with two coordinates: $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$.

The Eucidean distance d between two points can be calculated with the formula:

$$d = \sqrt{(x_2 - x_2)^2 + (y_2 - y_1)^2}$$

The midpoint of the line segment between p_1 and p_2 can be found as:

$$p = \left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2} \right)$$

The intercept a and the slope b of the line $y = a + bx$ connecting two points p_1 and p_2 can be found as:

$$b = \frac{y_2 - y_1}{x_2 - x_1}, \quad a = y_1 - bx_1$$

Distance

Write a function `find_distance()` that returns the distance between two given points. You should be able to call the function like this:

```
# coordinates for point-1 and point-2
p1 <- c(0, 0)
p2 <- c(1, 1)

find_distance(p1, p2)
```

Midpoint

Write a function `find_midpoint()` that returns the midpoint between two given points. You should be able to call the function like this:

```
p1 <- c(0, 0)
p2 <- c(1, 1)

find_midpoint(p1, p2)
```

Slope

Write a function `find_slope()` that returns the slope of the line connecting two given points. You should be able to call the function like this:

```
p1 <- c(0, 0)
p2 <- c(1, 1)

find_slope(p1, p2)
```

Intercept

Write a function `find_intercept()` that returns the intercept of the line connecting two given points. This function must internally use `find_slope()`

```
p1 <- c(0, 0)
p2 <- c(1, 1)

find_intercept(p1, p2)
```

Line

Write a function `find_line()`. This function must use `find_slope()` and `find_intercept()`. The output should be a list with two named elements: `"intercept"` and `"slope"`, Here is how you should be able to use `find_line()`:

```
p1 <- c(0, 0)
p2 <- c(1, 1)

eq <- find_line(p1, p2)
eq$intercept
eq$slope
```

Information about two given points

Once you have the functions `find_distance()`, `find_midpoint()`, and `find_line()`, write an overall function called `info_points()` that returns a list with the distance, the midpoint, and the line's slope and intercept terms. Here is how you should be able to use `info_points()`:

```
p1 <- c(-2, 4)
p2 <- c(1, 2)

results <- info_points(p1, p2)
results$distance
results$midpoint
results$intercept
results$slope
```