Lab 01 - Control Structures

- Game Functions
    - STEP 1 - CREATE A NEW GAME
    - STEP 2 - CONTESTANT SELECTS A DOOR
    - STEP 3 - HOST OPENS GOAT DOOR
    - STEP 4 - CHANGE DOORS
    - STEP 5 - DETERMINE IF CONTENSTANT HAS WON
    - TESTING YOUR CODE
- Challenge Questions
    - PART 01:
    - PART 02:
    - UNIT TESTS
    - Formatting Results for Inspection
- Submission Instructions

For this lab you will design solution to the Monty Hall Problem (http://www.montyhallproblem.com/).

When the mathematical solution is beyond your reach, a good tractable way to approach these types of analytical problems is to write a program to solve it for you. In this case, we can create a simulation that allows us to play the game thousands of times, and record the results for the two strategies – staying with the initial door or switching (we already know that switching is superior).

You will need to design a virtual game by creating a function for each step of the game. The virtual game will be played by running the functions in order in a script (which we will use next week to create a simulation to test strategies in the game).

1. One function that sets up the game (three doors, one car, two goats).
2. One function that selects a door for your first guess.
3. One function that reveals a goat for the host.
4. One function that makes your final pick (depending upon if you intend to stay or switch).
5. One function that decides if you win the car or not.

The five functions form the basic steps of the virtual game.

For your homework, submit your knitted markdown file with the following:

- Your five functions, each in its own code chunk.
- Psuedo-code that explains the logic of each function.
- Below each include another chunk that tests the function using appropriate arguments and input data.

Note that you will need to think carefully about input arguments and return values for each function. Some function do not need arguments.

The first function that creates a new game, for example, does not require any additional information other than a new game is required. It will randomly assign the two goats and one car to the three doors in the game and return the game set-up.

For the final function, however, in order to determine if the player has won we need information about the door they have selected and the game set-up in order to evaluate whether they win the car or the goat.

Unit Tests: (*some helpful vocabulary*)

A **UNIT TEST** is code that is written to check if your code is running properly.

If you are building new functions or conducting complex analysis **it is never sufficient to check that the code runs without producing an error.** You need to check that the code is producing the correct or expected results.

This requires actually knowing what a right answer looks like.

Unit tests typically test the code using multiple examples of input data that is well-understood and the answers are known.

It will also include **edge cases**, scenarios where the data may contain extreme or unexpected values or data types.

Game Functions

Here is some code to get you started.

STEP 1 - CREATE A NEW

GAME

This function sets up a new game.

Psuedocode

```
# step 1: create a vector of 3 doors: 2 goa
ts and 1 car
# step 2: randomize the position of the car
for a new game
# step 3: return the new game vector

# note that no external information is need
ed
# so no arguments are passed to the vector
```

```
create_game <- function( )
{
    a.game <- sample( x=c("goat","goat","ca
        r"), size=3, replace=F )
    return( a.game )
}
```

Test of Function:

Note that when functions utilize randomization it is helpful to test functions multiple times to make sure randomization is working properly. In this case we are randomizing the position of the car in each new game.

```
# try three times to see randomization
create_game()
create_game()
create_game()
```

STEP 2 - CONTESTANT SELECTS A DOOR

The contestant makes their first selection. Write a function to select one door at random.

Psuedocode

```
# step 1: create a vector of doors numbered
1,2,3
# step 2: randomly select ONE of the doors
# step 3: return the selection

# since the contestant will not know the po
sition
# of the car when they select a door we do
not
# need to share information about the game
set-up
# before the selection is made
```

```
select_door <- function( )
{

  doors <- c(1,2,3)
  a.pick <- # YOUR CODE HERE...
  return( a.pick )  # number between 1 and
        3

}
```

```
# test the function
select_door()
select_door()
select_door()
```

STEP 3 - HOST OPENS GOAT DOOR

Note to call this function you need information from previous functions.

The host will always open a door with a goat behind it. But it can't be a door the contestant has already selected. So it must be a door that is not a car and not a current contestant selection.

Note that if the contestant selects the car on the first guess the host can open either door, but if the contestant selects a goat the host only has one option.

Psuedocode

```
# describe your logic here
```

```
open_goat_door <- function( game, a.pick )
{

   # YOUR CODE HERE...

   return( opened.door ) # number between 1
        and 3

}
```

```
# test it

this.game <- create_game()

this.game

my.initial.pick <- select_door()

my.initial.pick

open_goat_door( this.game, my.initial.pick
        )
```

STEP 4 - CHANGE DOORS

The contestant is given the option to change from their initial selection to the other door that is still closed. The function will represent the game-playing strategy as the argument **stay=TRUE** or **stay=FALSE**.

Psuedocode

```
# describe your logic here
```

```
change_door <- function( stay=T, opened.doo
        r, a.pick )
{

   # YOUR CODE HERE...

   return( final.pick )  # number between 1
        and 3

}
```

```
# test it

opened.door <- open_goat_door( this.game, m
        y.initial.pick )

change_door( stay=T,
             opened.door=opened.door,
             a.pick=my.initial.pick )
change_door( stay=F,
             opened.door=opened.door,
             a.pick=my.initial.pick )

my.final.pick <- change_door( stay=F,
                              opened.door=o
        pened.door,
                              a.pick=my.ini
        tial.pick )

this.game
my.initial.pick
my.final.pick
```

STEP 5 - DETERMINE IF CONTENSTANT HAS WON

Psuedocode

```
# describe your logic here
```

```
determine_winner <- function( final.pick, g
        ame )
{

   if( ...YOUR CODE HERE... )
   {
      return( "WIN" )
   }
   if( ...YOUR CODE HERE... )
   {
      return( "LOSE" )
   }

}
```

```
# test code

this.game
my.initial.pick

my.final.pick <- change_door( stay=T,
                              opened.door=o
        pened.door,
                              a.pick=my.ini
        tial.pick )

determine_winner( final.pick=my.final.pick,
                  game=this.game )

my.final.pick <- change_door( stay=F,
                              opened.door=o
        pened.door,
                              a.pick=my.ini
        tial.pick )

determine_winner( final.pick=my.final.pick,
                  game=this.game )
```

TESTING YOUR CODE

Use the following code to test your game and print results.

Add the argument echo=F to your R chunk for a more readable print-out.

```{r, echo=F}

# your game "recipe"
this.game <- create_game()
my.initial.pick <- select_door()
opened.goat.door <- open_goat_door( this.ga
        me, my.initial.pick )

# save results for both strategies for the
        game
my.final.pick.stay <- change_door( stay=T,
                                   opened.d
        oor=opened.goat.door,
                                   a.pick=m
        y.initial.pick )
my.final.pick.switch <- change_door( stay=
        F,
                                     opene
        d.door=opened.goat.door,
                                     a.pick
        =my.initial.pick )

# print game details and if you won

# if you stayed:
paste0( "GAME SETUP" )
this.game
paste0( "My initial selection: ", my.initia
        l.pick )
paste0( "The opened goat door: ", opened.go
        at.door )
paste0( "My final selection: ", my.final.pi
        ck.stay )
paste0( "GAME OUTCOME:" )
determine_winner( final.pick=my.final.pick.
        stay,
                  game=this.game )

# if you switched:
paste0( "GAME SETUP" )
this.game
paste0( "My initial selection: ", my.initia
        l.pick )
paste0( "The opened goat door: ", opened.go
        at.door )
paste0( "My final selection: ", my.final.pi
        ck.switch )
```

```
paste0( "GAME OUTCOME:" )
determine_winner( final.pick=my.final.pick.
        switch,
                  game=this.game )

```

Challenge Questions

Try these out. If you can figure out a solution include it with your submission and post it to YellowDig to get bonus admiration points from your classmates.

PART 01:

Let's change the rules a little to make outcomes more interesting. Create a board with 5 doors and 2 cars. After the contestant makes an initial selection the host will open one car door and one goat door. If the contestant decides to switch they then have to select from the two remaining doors.

How does this new board change the pay-off from the game? Is switching still the best strategy?

PART 02:

We are building functions to play a game in a static world. There are always three doors, one car, and two goats.

What happens if we are in a dynamic world?

The game can have three or more doors (in a game with two doors there would be no switching so there is no strategy to study).

And we can also have one or more cars up to N-2 (N being the number of doors, there always need to be at least two goats so that the host can open a goat door, even if the contestant selected a goat in the first round).

How would you change the code to build this game?

Note that in the first game the user would only have to specify their strategy (stay or switch). Here the user would have to specify the game size, the number of cars, and their strategy. So game size and number of cars would be added as arguments to the **build_game()** function.

UNIT TESTS

Each chunk includes some testing code afterwards to demonstrate proper use.

Add unit tests as well. Instead of randomly selecting the game set-up and initial selection, assign these values so that you know the current state of the game.

With this information you can also determine the outcome under each strategy (stay and switch). So **given any game set-up you can evaluate if your functions produce the correct answer**.

Create variables that stores the correct answers and use them to evaluate your code:

```
# UNIT TEST A

this.game <- c("goat","car","goat")
my.initial.pick <- 2

A.STAY <- "WIN"
A.SWITCH <- "LOSE"

my.final.pick <- change_door( stay=T,
                              opened.door=o
        pened.door,
                              a.pick=my.ini
        tial.pick )

outcome.stay <- determine_winner( final.pic
        k=my.final.pick,
                  game=this.game )

my.final.pick <- change_door( stay=F,
                              opened.door=o
        pened.door,
                              a.pick=my.ini
        tial.pick )

outcome.switch <- determine_winner( final.p
        ick=my.final.pick,
                  game=this.game )

case.stay <- outcome.stay == A.STAY
case.switch <- outcome.switch == A.SWITCH

passes <- all( case.stay & case.switch )
```

Define several unit tests to inspect the efficacy of your code.

Include several in your file after Step 5.

```
**Unit Test A:**

Game Setup: **`r paste0( toupper(this.gam
        e), collapse=" " )`**
Initial selection: **`r my.initial.pick`**
Passes unit test:  **`r passes`**
```

Formatting Results for Inspection

Try using the following code to present your results in a more visual manner.

NOTE, the code itself might not make much sense yet but you will see why it's easier to inspect game results to see if your code is working when you are using a visual layout of game results.

Copy and paste the first part (IF CONTESTANT STAYS chunk) directly into your RMD document - it is regular markdown text with inline code.

```
### IF CONTESTANT STAYS

Game Setup: **`r paste0( toupper(this.gam
        e), collapse=" " )`**
Initial selection: **`r my.initial.pick`**
The opened goat door: **`r opened.goat.door
        `**
Final door selection: **`r my.final.pick.st
        ay`**
Game outcome: **`r game.outcome.stay`**
```

Copy and paste the following R chunk below that.

```
```{r, echo=F}
this.game[ this.game == "car" ] <- "car "
this.game <- toupper( this.game )

first.pick <- c("        "," "," "
        ")
first.pick[ my.initial.pick ] <- "1st Pick"
open.door <- c("        "," "," "
        ")
open.door[ opened.goat.door ] <- " Opened "
final <- c("        "," "," "
        ")
final[ my.final.pick.stay ] <- " Final  "

win.lose <- final
win.lose[ my.final.pick.stay ] <- paste0(
        "! ", game.outcome.stay, " !" )

outcome <- paste0( rep( paste0( "! ", game.
        outcome.stay, " !" ), 5 ), collapse
        ="" )

paste0( "  | ", this.game[1], " |     | ",
        this.game[2], " |     | ", this.gam
        e[3], " |  " )
paste0( "  ", first.pick[1], "     ", firs
        t.pick[2], "     ", first.pick[3],
        "  " )
paste0( "  ", open.door[1], "     ", open.d
        oor[2], "     ", open.door[3], "  "
        )
paste0( "  ", final[1], "     ", final[2],
        "     ", final[3], "  " )
paste0( "  ", win.lose[1], "     ", win.los
        e[2], "     ", win.lose[3], "  " )

```
```

Repeat these steps with the second part (IF CONTESTANT SWITCHES):

```
### IF CONTESTANT SWITCHES

Game Setup: **`r paste0( toupper(this.gam
        e), collapse=" " )`**
Initial selection: **`r my.initial.pick`**
The opened goat door: **`r opened.goat.door
        `**
Final door selection: **`r my.final.pick.sw
        itch`**
Game outcome: **`r game.outcome.switch`**
```

```{r, echo=F}

this.game[ this.game == "car" ] <- "car "
this.game <- toupper( this.game )
first.pick <- c("        ","        ","
        ")
first.pick[ my.initial.pick ] <- "1st Pick"
open.door <- c("        ","        ","
        ")
open.door[ opened.goat.door ] <- " Opened "
final <- c("        ","        ","
        ")
final[ my.final.pick.switch ] <- " Final  "
outcome <- paste0( rep( paste0( " ", game.o
        utcome.switch, "! " ), 5 ), collaps
        e="" )

win.lose <- final
win.lose[ my.final.pick.switch ] <- paste0(
        "! ", game.outcome.switch, " !" )

paste0( "  | ", this.game[1], " |     | ",
        this.game[2], " |     | ", this.gam
        e[3], " |  " )
paste0( "  ", first.pick[1], "     ", firs
        t.pick[2], "     ", first.pick[3],
        "  " )
paste0( "  ", open.door[1], "     ", open.d
        oor[2], "     ", open.door[3], "  "
        )
paste0( "  ", final[1], "     ", final[2],
        "     ", final[3], "  " )
paste0( "  ", win.lose[1], "     ", win.los
        e[2], "     ", win.lose[3], "  " )


```

Submission Instructions

When you have completed your assignment, knit your RMD file to generate your rendered HTML file. Platforms like BlackBoard and Canvas often disallow you from submitting HTML files when there is embedded computer code, so create a zipped folder with both the RMD and HTML files.

Login to Canvas at http://canvas.asu.edu (http://canvas.asu.edu) and navigate to the assignments tab in the course repository. Upload your zipped folder to the appropriate lab submission link.

Remember to:

- name your files according to the convention: **Lab-##-LastName.Rmd**
- show your solution, include your code.
- do not print excessive output (like a full data set).
- follow appropriate style guidelines (spaces between arguments, etc.).

See Google's R Style Guide (https://google.github.io/styleguide/Rguide.xml) for examples.

Markdown Trouble?

If you are having problems with your RMD file, visit the **RMD File Styles and Knitting Tips** (https://ds4ps.org/cpp-526-spr-2020/labs/r-markdown-files.html) manual.

Notes on Knitting

Note that when you knit a file, it starts from a blank slate. You might have packages loaded or datasets active on your local machine, so you can run code chunks fine. But when you knit you might get errors that functions cannot be located or datasets don't exist. Be sure that you have included chunks to load these in your RMD file.

Your RMD file will not knit if you have errors in your code. If you get stuck on a question, just add `eval=F` to the code chunk and it will be ignored when you knit your file. That way I can give you credit for attempting the question and provide guidance on fixing the problem.

Data Science for Public Service

(https://ds4ps.org/)

STUDENT POLICIES AND FEDERAL COPYRIGHT LAWS ©