

# STAT 408: Week 6

Simulation, Loops, and If/Else Statements

2/22/2022

## Simulation in R

# What is Simulation

A few questions about simulation:

1. What does statistical simulation mean to you?
2. Describe a setting where simulation can be used.

3/48

## Simulation of Roulette

Consider the casino game Roulette.



We can use simulation to evaluate gambling strategies.

4/48

# Roulette Simulation in R

```
RouletteSpin <- function(num.spins){  
  # function to simulate roulette spins  
  # ARGS: number of spins  
  # RETURNS: result of each spin  
  outcomes <- data.frame(number = c('00', '0', as.character(1:36)),  
                          color=c('green', 'green', 'red', 'black', 'red', 'black',  
                                'red', 'black', 'red', 'black', 'red', 'black',  
                                'black', 'red', 'black', 'red', 'black',  
                                'red', 'black', 'red', 'red', 'black',  
                                'red', 'black', 'red', 'black', 'red',  
                                'black', 'red', 'black', 'black', 'red',  
                                'black', 'red', 'black', 'red', 'black', 'red'))  
  return(outcomes[sample(38, num.spins, replace = TRUE),])  
}
```

5/48

```
kable(RouletteSpin(2), row.names=FALSE)
```

number	color
10	black
8	black

---

6/48

# Exercise: Probability of Red, Green, and Black

1. Calculate/derive the probability of landing on green, red, and black.
2. How can the `RouletteSpin()` function be used to compute or approximate these probabilities?

7/48

## Solution: Probability of Red, Green, and Black

In this situation, it is easy to compute the probabilities of each color analytically. However, consider simulating this process many times to estimate these probabilities.

```
num.sims <- 1000
spins <- RouletteSpin(num.sims)
p.red <- sum(spins[,2] == 'red') / num.sims
p.black <- sum(spins[,2] == 'black') / num.sims
p.green <- sum(spins[,2] == 'green') / num.sims
```

Analytically  $P[\text{red}] = \frac{18}{38} = 0.4737$ , this is estimated as 0.47. Similarly,  $P[\text{black}] = \frac{18}{38} = 0.4737$ , this is estimated as 0.472 and  $P[\text{green}] = \frac{2}{38} = 0.0526$ , this is estimated as 0.058

8/48

## Exercise: Simulation Questions – Part 2

Now what happens if we:

1. run the simulation again with the same number of trials?
2. run the simulation with more trials, say 1 million?

9/48

## Solution: Simulation Questions – Part 2

Run the simulation again with the same number of trials:

```
num.sims <- 1000
spins <- RouletteSpin(num.sims)
p.red <- sum(spins[,2] == 'red') / num.sims
p.black <- sum(spins[,2] == 'black') / num.sims
p.green <- sum(spins[,2] == 'green') / num.sims
```

The simulated results are different Analytically  $P[\text{red}] = \frac{18}{38} = 0.4737$ , this is estimated as 0.458. Similarly,  $P[\text{black}] = \frac{18}{38} = 0.4737$ , this is estimated as 0.488 and  $P[\text{green}] = \frac{2}{38} = 0.0526$ , this is estimated as 0.054

10/48

## Solution: Simulation Questions – Part 2

Run the simulation with more trials, say 1 million:

```
num.sims <- 1000000
spins <- RouletteSpin(num.sims)
p.red <- sum(spins[,2] == 'red') / num.sims
p.black <- sum(spins[,2] == 'black') / num.sims
p.green <- sum(spins[,2] == 'green') / num.sims
```

Analytically  $P[\textit{red}] = \frac{18}{38} = 0.4737$ , this is estimated as 0.4742. Similarly,  $P[\textit{black}] = \frac{18}{38} = 0.4737$ , this is estimated as 0.4727 and  $P[\textit{green}] = \frac{2}{38} = 0.0526$ , this is estimated as 0.0531

## Conditional Expressions in R

# Exercise: Conditions in R

We have touched on many of these before, but here are some examples of expressions (conditions) in R. Evaluate these expressions:

```
pi > 3 & pi < 3.5
c(1,3,5,7) %in% 1:3
1:3 %in% c(1,3,5,7)
rand.uniform <- runif(n = 1, min = 0, max = 1)
rand.uniform < .5
```

13/48

# Solutions: Conditions in R

```
pi > 3 & pi < 3.5

## [1] TRUE

c(1,3,5,7) %in% 1:3

## [1] TRUE TRUE FALSE FALSE

1:3 %in% c(1,3,5,7)

## [1] TRUE FALSE TRUE

rand.uniform <- runif(n = 1, min = 0, max = 1); rand.uniform

## [1] 0.6718267

rand.uniform < .5

## [1] FALSE
```

14/48

# Conditional Expression: If and Else

```
print(rand.uniform)
```

```
## [1] 0.6718267
```

Now what does this return?

```
if (rand.uniform < .5){  
  print('value less than 1/2')  
} else {  
  print('value greater than or equal to 1/2')  
}
```

15/48

# Conditional Expression: If and Else

```
print(rand.uniform)
```

```
## [1] 0.6718267
```

```
if (rand.uniform < .5){  
  print('value less than 1/2')  
} else {  
  print('value greater than or equal to 1/2')  
}
```

```
## [1] "value greater than or equal to 1/2"
```

16/48



# Conditional Expression: Vectorized?

Does this function accept a vector as an input?

```
rand.uniform2 <- runif(2)
print(rand.uniform2)
if (rand.uniform2 < .5){
  print('value less than 1/2')
} else {
  print('value greater than or equal to 1/2')
}
```

17/48

# Conditional Expression: Vectorized?

```
rand.uniform2 <- runif(2)
print(rand.uniform2)

## [1] 0.8362597 0.8058175

if (rand.uniform2 < .5){
  'value less than 1/2'
} else {
  'value greater than 1/2'
}

## [1] "value greater than 1/2"
```

18/48

# Conditional Expression: `ifelse()`

```
print(rand.uniform2)
```

```
## [1] 0.8362597 0.8058175
```

```
ifelse(rand.uniform2 < .5, 'less than 1/2',  
       'greater than 1/2')
```

```
## [1] "greater than 1/2" "greater than 1/2"
```

The `ifelse()` function is vectorized and generally preferred with a single set of if/else statements.

19/48

## Exercise: Conditional Expression

Write a conditional statement that takes a playing card with two arguments, number (A, 2,..., 10, J, Q, K) and suit (C, D, H, S), and prints **Yes** if the card is a red face card and **No** otherwise.

- 4 of clubs: `card.number <- '4'` and `card.suit <- 'C'`
- King of hearts: `card.number <- 'K'` and `card.suit <- 'H'`

Verify this works using the following inputs:

- `card.number <- 'J'` and `card.suit <- 'D'`
- `card.number <- 'Q'` and `card.suit <- 'S'`

20/48

# Solution: Conditional Expression

```
card.number <- 'J'  
card.suit <- 'D'  
ifelse(card.number %in% c('J', 'Q', 'K') &  
       card.suit %in% c('H', 'D'), 'Yes', 'No')
```

```
## [1] "Yes"
```

```
card.number <- 'Q'  
card.suit <- 'S'  
ifelse(card.number %in% c('J', 'Q', 'K') &  
       card.suit %in% c('H', 'D'), 'Yes', 'No')
```

```
## [1] "No"
```

## Loops

# for loops

When you want to do the same thing more than once:

```
output <- vector("numeric", 100) # Set up empty object
for(i in LOOP_OVER_THIS_SEQUENCE) {
  # Repeat this code on each item in the sequence
  # Store in output vector
}
```

23/48

# for loops

What will each of these two loops print?

```
rand.uniform2
```

```
## [1] 0.8362597 0.8058175
```

```
for (i in seq_along(rand.uniform2)){
  print(i)
}
```

and

```
for (i in rand.uniform2){
  print(i)
}
```

24/48

# for loops

We can loop through a sequence or a vector.

```
for (i in seq_along(rand.uniform2)){  
  print(i)  
}
```

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

```
for (i in rand.uniform2){  
  print(i)  
}
```

```
## [1] 0.8362597  
## [1] 0.8058175
```

25/48

# While loops

An alternative to for loops is to use the while statement.

```
set.seed(02012017)  
total.snow <- 0  
while (total.snow < 36){  
  print(paste('need more snow, only have',  
             total.snow, 'inches'))  
  total.snow <- total.snow + rpois(1,15)  
}  
print(paste('okay, we now have', total.snow, 'inches'))
```

26/48

## Exercise: Loops

Assume you plan to wager \$1 on red for ten roulette spins. If the outcome is red you win a dollar and otherwise you lose a dollar. Write a loop that simulates ten rolls and determines your net profit or loss.

*#hint: to get color from a single spin use*

```
RouletteSpin(1)[2]
```

```
##    color  
## 38    red
```

27/48

## Solution: Loops

Assume you plan to wager \$1 on red for ten roulette spins. If the outcome is red you win a dollar and otherwise you lose a dollar. Write a loop that simulates ten rolls and determines your net profit or loss.

```
profit <- 0  
for (i in 1:10){  
  ifelse(RouletteSpin(1)[2] == 'red',  
         profit <- profit + 1,  
         profit <- profit - 1 )  
}  
profit
```

```
## [1] 4
```

28/48

# Why not loops?

Some of you have seen or heard that loops in R should be avoided.

- **Why:** it has to do with how code is compiled in R. In simple terms, vectorized operations are much more efficient than loops.
- **How:** we have seen some solutions to this, explicitly using the apply class of functions. We can also write vectorized functions, consider the earlier roulette example where number of spins was an argument for the function.

## Monte Carlo Procedures

# Motivation for Monte Carlo procedures

Some probabilities can easily be calculated either intuitively or using pen and paper; however, as we have seen we can also simulate procedures to get an approximate answer.

Consider poker, where players are dealt a hand of five cards from a deck of 52 cards. What is the probability of being dealt a full house?

31/48

## Full House Probability Calculation

Could we analytically compute this probability? **Yes** Is it an easy calculation? not necessarily. So consider a (Monte Carlo) simulation.

```
DealPoker <- function(){  
  # Function returns a hand of 5 cards
```

```
}
```

32/48



# Full House Probability Calculation

Could we analytically compute this probability? Yes Is it an easy calculation, not necessarily. So consider a (Monte Carlo) simulation.

```
DealPoker <- function(){  
  # Function returns a hand of 5 cards  
  deck <- data.frame( suit = rep(c("H", "S", "C", "D"), each=13),  
    card = rep(c('A', 2:10, 'J', "Q", 'K'), 4) )  
  return(deck[sample(52, 5),])  
}
```

33/48

# Full House Probability Calculation

Next write another function to check if the hand is a full house.

```
IsFullHouse <- function(hand){  
  #determines whether a hand of 5 cards is a full house  
  #ARGS: data frame of 5 cards  
  #RETURNS: TRUE or FALSE  
  cards <- hand[,2]  
  num.unique <- length(unique(cards))  
  ifelse(num.unique == 2, return(TRUE), return(FALSE))  
}
```

Does this work? If not, what is the problem and how do we fix it?

34/48

# Full House Probability Calculation

```
IsFullHouse <- function(hand){  
  #determines whether a hand of 5 cards is a full house  
  #ARGS: data frame of 5 cards  
  #RETURNS: TRUE or FALSE  
  cards <- hand[,2]  
  num.unique <- length(unique(cards))  
  num.appearances <- aggregate(rep(1,5),  
                               list(cards),sum)  
  max.appearance <- max(num.appearances[,2])  
  ifelse(num.unique == 2 & max.appearance ==3,  
         return(TRUE), return(FALSE))  
}
```

35/48

# Full House Probability Calculation

```
num.sims <- 1e5  
sim.hands <- replicate(num.sims, DealPoker(), simplify=FALSE)  
results <- lapply(sim.hands, IsFullHouse)  
prob.full.house <- sum(unlist(results))/num.sims
```

Analytically the probability of getting a full house *can* be calculated as approximately 0.00144, with our simulation procedure we get 0.00141.

36/48

# Closing Thoughts on Monte Carlo

A few facts about Monte Carlo procedures:

- They return a random result due to randomness in the sampling procedures.
- The run-time (or number of iterations) is fixed and typically specified.
- Mathematically, Monte Carlo procedures are computing an integral or enumerating a sum.
- They take advantage of the law of large numbers.
- They were given the code name Monte Carlo in reference to the Monte Carlo Casino in Monaco.

## Advanced Debugging

# Overview

We can often fix bugs using the ideas sketched out previously and this becomes *easier* with more experience coding in R. Trial and error can be very effective and strategic use of `print()` function help to identify where bugs are occurring.

However, R does also have advanced tools to help with debugging code.

- `traceback()`
- “Rerun with debug”
- `browser()`

39/48

## traceback()

Consider the following code:

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
f(10)
```

```
## Error in "a" + d: non-numeric argument to binary operator
```

40/48

# traceback()

Consider the `traceback()` function. Which identifies which functions have been executed (along with the row number of the function).

```
> traceback()
```

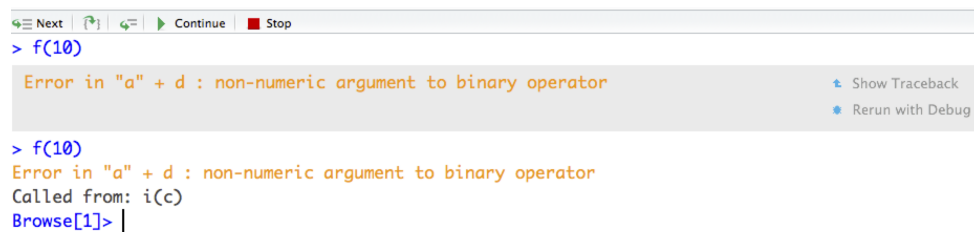
```
4: i(c) at #1  
3: h(b) at #1  
2: g(a) at #1  
1: f(10)
```

Note: due to the way that R Markdown is compiled, `traceback()` needs to be run directly in R, not R Markdown.

41/48

## Browsing on an error

Another option (in R Studio) is to browse on the error. This gives you an interactive way to move through the function calls to identify the problem of the location. This can also be called explicitly using `debug()`.



```
Next | Previous | Continue | Stop  
> f(10)  
Error in "a" + d : non-numeric argument to binary operator  
Show Traceback  
Rerun with Debug  
> f(10)  
Error in "a" + d : non-numeric argument to binary operator  
Called from: i(c)  
Browse[1]> |
```

42/48

# browser()

The browser function can also be used to interactively step through a function.

```
SS <- function(mu, x) {  
  browser()  
  d <- x - mu  
  d2 <- d^2  
  ss <- sum(d2)  
  ss  
}
```

43/48

## browser() step 1

The screenshot shows the RStudio interface with the following components:

- Source Editor:** Displays the function definition for `SS`. The `browser()` function call on line 2 is highlighted with a yellow background, indicating the current execution point.
- Console:** Shows the execution of the function `SS(2, 4)`. The output is `Called from: SS(2, 4)` and `Browse[1]>`, indicating that the browser has been invoked.
- Values Panel:** Shows the current values of the variables `mu` (2) and `x` (4).
- Traceback Panel:** Shows the call stack, with `SS(2, 4)` as the current function being executed.
- Documentation Panel:** Shows the documentation for the `lapply` function, which is the function being executed in the current step.

44/48

# browser() step 2

```
1- function(mu, x) {
2-   browser()
3-   d <- x - mu
4-   d2 <- d^2
5-   ss <- sum(d2)
6-   ss
7- }
```

```
> SS <- function(mu, x) {
+   browser()
+   d <- x - mu
+   d2 <- d^2
+   ss <- sum(d2)
+   ss
+ }
>
> SS(2, 4)
Called from: SS(2, 4)
Browse[1]>
debug at #3: d <- x - mu
Browse[2]> |
```

Values	
mu	2
x	4

Traceback  Show

- SS(2, 4)

R: Apply a Function over a List or Vector

Apply a Function over a List Vector

description

lapply returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.

lapply is a user-friendly version and wrapper of lapply by default returning a vector, matrix or, if lapply = "array", an array if appropriate, by using eapply2array(). apply(x, f, lapply = FALSE, USE.NAMES = FALSE) is the same as lapply(x, f).

lapply is similar to sapply, but has a pre-specified return value, so it can be safer (and sometimes faster) to use.

lapply is a wrapper for the common use of lapply for repeated evaluation of an expression that will usually involve random number generation.

lapply2array() is the utility called from lapply2array()

45/48

# browser() step 3

```
1- function(mu, x) {
2-   browser()
3-   d <- x - mu
4-   d2 <- d^2
5-   ss <- sum(d2)
6-   ss
7- }
```

```
> SS <- function(mu, x) {
+   browser()
+   d <- x - mu
+   d2 <- d^2
+   ss <- sum(d2)
+   ss
+ }
>
> SS(2, 4)
Called from: SS(2, 4)
Browse[1]>
debug at #3: d <- x - mu
Browse[2]>
debug at #4: d2 <- d^2
Browse[3]> |
```

Values	
d	2
mu	2
x	4

Traceback  Show

- SS(2, 4)

R: Apply a Function over a List or Vector

Apply a Function over a List Vector

description

lapply returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.

lapply is a user-friendly version and wrapper of lapply by default returning a vector, matrix or, if lapply = "array", an array if appropriate, by using eapply2array(). apply(x, f, lapply = FALSE, USE.NAMES = FALSE) is the same as lapply(x, f).

lapply is similar to sapply, but has a pre-specified return value, so it can be safer (and sometimes faster) to use.

lapply is a wrapper for the common use of lapply for repeated evaluation of an expression that will usually involve random number generation.

lapply2array() is the utility called from lapply2array()

46/48

# browser() step 4

```
Function: SS (GlobalEnv)
Debug location is approximate because the source is not available.
1- function(mu, x) {
2-   browser()
3-   d <- x - mu
4-   d2 <- d^2
5-   ss <- sum(d2)
6-   ss
7- }
```

Values

d	2
d2	4
mu	2
x	4

Traceback

- SS(2, 4)

Console

```
> SS(2, 4)
Called from: SS(2, 4)
Browse[1]>
debug at #3: d <- x - mu
Browse[2]>
debug at #4: d2 <- d^2
Browse[2]>
debug at #5: ss <- sum(d2)
Browse[2]> |
```

47/48

# browser() step 5

```
1- SS <- function(mu, x) {
2-   browser()
3-   d <- x - mu
4-   d2 <- d^2
5-   ss <- sum(d2)
6-   ss
7- }
8
9 SS(2, 4)
10
```

Functions

- function (a)
- function (b)
- function (c)
- function (d)
- function (in.val)
- function (mu, x)

Console

```
> SS(2, 4)
Called from: SS(2, 4)
Browse[1]>
debug at #3: d <- x - mu
Browse[2]>
debug at #4: d2 <- d^2
Browse[2]>
debug at #5: ss <- sum(d2)
Browse[2]>
debug at #6: ss
Browse[2]>
[1] 4
> |
```

48/48