Writing functions

So far we have relied on the built-in functionality of R to carry out our analyses. In the next several lectures, we cover:

- How to write your own functions
- How to use control flow
- Debugging your code when something goes wrong
- The meaning of environments and variable scope
- Timing and writing efficient code

Steps In Writing a Function

Explain: Describe the task in words

Concrete: Write code for a specific example

Abstract: Identify the variables and decide if they are required or have defaults

Encapsulate: Wrap the code into a function where the parameters are the general variables

Test: Check the function works as expected with your original data AND try the function on test cases with other data

Writing your own functions in R

The code we have been writing so far in R has been

- made up of a sequence of commands, one after another
- specific to the particular dataset we are working with.

Functions allow us to

- organize our code into tasks
- reuse the same code on different datasets by making the data an *argument* to the function.

Function Name

Assign Function
to this Name

Function Signature:
Required arguments
Default arguments

```
calcRainSize = function (x, traceAmt = 0) {
  mean(x[x> traceAmt])
}
```

Function Body
Between { }

Typically we assign the function to a particular name. This should describe what the function does. Using a VERB in the name is a good idea.

```
function ( arguments ) body
```

The keyword **`function`** just tells R that you want to create a function.

Recall that the *parameters* to a function are its inputs, which may have default values.

```
> args(median)
function (x, na.rm = FALSE)
```

Here, if we do not explicitly specify `na.rm` when we call `median`, it will be assigned the default value of `FALSE`.

A few notes on specifying the arguments:

When you're writing your own function, it's good practice to put the most important arguments first. Often these will not have default values.

This allows the user of your function to easily specify the arguments by position, eg.

```
calcRainSize(xvec)
```

rather than

```
calcRainSize(x = xvec)
```

Next we have the *body* of the function, which typically consists of expressions surrounded by curly brackets. Think of these as performing some operations on the input values given by the arguments.

```
{
    expression 1
    expression 2
    return(value)
}
```

The `return` expression hands control back to the caller of the function and returns a given **`value`**

If the function returns more than one thing, this is done using a named list, for example:

```
return(list(total = sum(x), avg = mean(x)))
```

In the absence of a return expression, a function will return the *last* evaluated expression. This is particularly common if the function is short.

That is the case for our simple function:

```
calcRainSize = function(x) mean(x[x>0])
```

Here we don't need brackets `{}`, since there is only one expression in the function.

A return expression anywhere in the function will cause the function to return control to the user *immediately*, without evaluating the rest of the function.

Considerations when writing a function:

- What will the function do?
- What should we call it? (Relate the name to what it does)
- What will be the arguments?
- Which arguments have default values and what are they?
- What (if anything) should the function return?

Anonymous functions

Apply `calcRainSize` to `rain`

```
sapply(rain, calcRainSize)
```

We don't actually have to go through the hassle of writing a function definition.

We can use an anonymous function:

```
sapply(rain, function(station) {
 mean(station[station > 0])
 }
)
```

Multiple Inputs and Apply

We can specify the value for `traceAmt` as an additional argument to sapply

```
sapply(rain, calcRainSize,
         traceAmt = 5)
```

What if we want to specify a different value of `traceAmt` for each weather station?

```
mapply(calcRainSize, rain,
         traceAmt = c(0, 1, 5, 10, 0))
```

Multiple Inputs and Apply

Recall another version of function was:

```
calcRainSize =
 function(x, tA = 0, sumFun = mean)
{
  sumFun(x[x > tA])
}
```

What if we want to specify a different value of `traceAmt` for each weather station and use the median function?

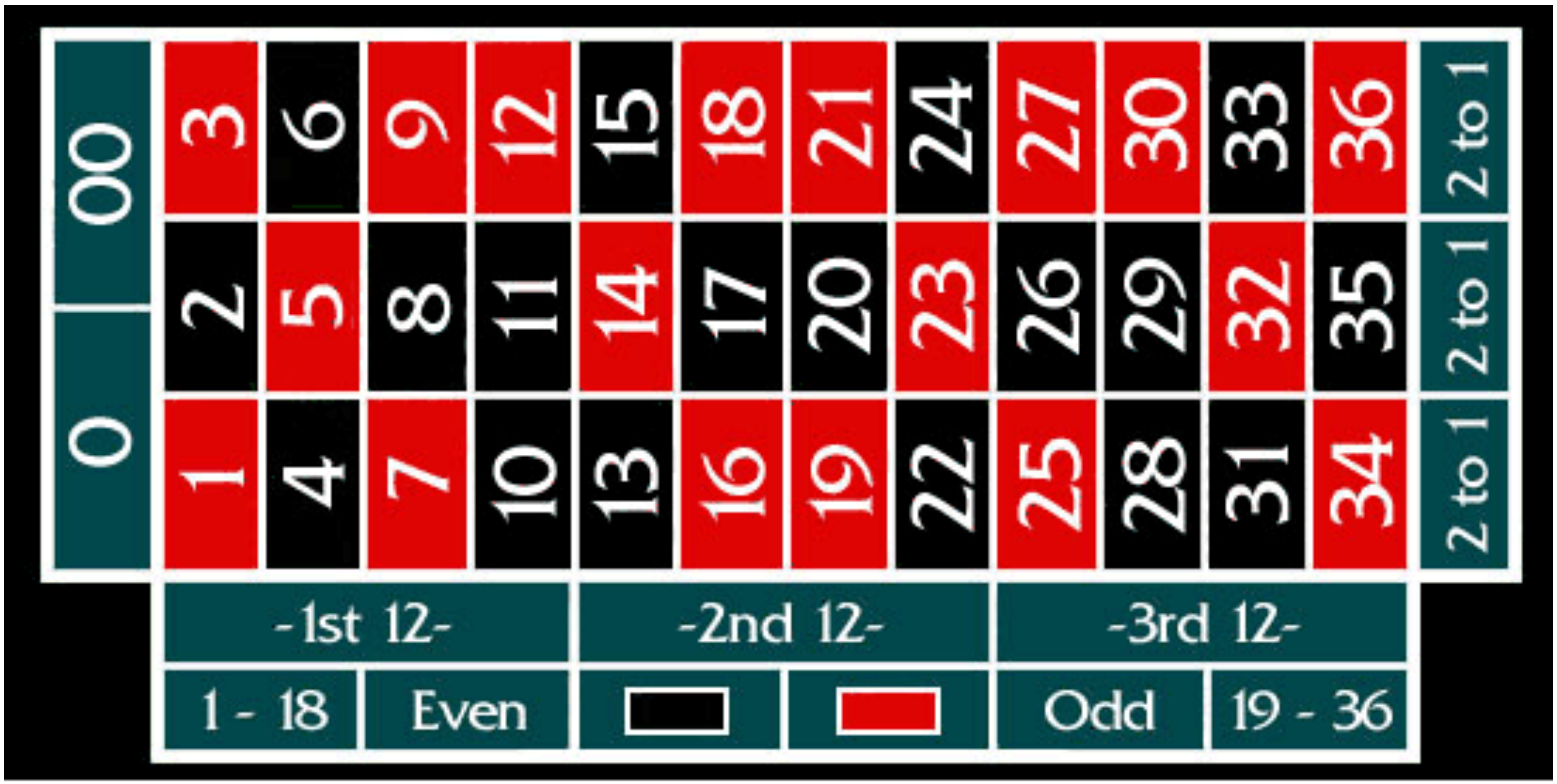
```
mapply(calcRainSize, rain,
       traceAmt = c(0, 1, 5, 10, 0),
       MoreArgs = list(sumFun = median))
```

Roulette Wheel Study

This function used for a *Monte Carlo study*: Use a computer to simulate random variables (a random experiment). Use the simulations to learn about properties of the random process

The Wheel

The Table


00
0
3
6
9
12
15
18
21
24
27
30
33
36
2
5
8
11
14
17
20
23
26
29
32
35
1
4
7
10
13
16
19
22
25
28
31
34
2 to 1
2 to 1
2 to 1
-1st 12-
-2nd 12-
-3rd 12-
1 - 18
Even
Odd
19 - 36

Outside Bets

- Red or Black
- Even or Odd
- Low or High

Even Odds (PAY 1 to 1)
Place \$1 bet,
If win keep your \$1 and get \$1 more
If lose, house gets your \$1

- 1st, 2nd, or 3rd dozen
- 1st, 2nd, or 3rd column

PAY 2 to 1)
Place \$1 bet,
If win keep your \$1 and get \$2 more
If lose, house gets your \$1

Inside Bets

• Straight up (single number)	Pays 35 to 1
• Split (2 numbers)	Pays 17 to 1
• Street (3 numbers)	Pays 11 to 1
• Corner (4 numbers)	Pays 8 to 1

How do winnings from
100 bets of $1 on Red
compare to
1 bet of $100 one Red?

Let’s write a function that can simulate these two situations –

100 $1 bets

1 $100 bet

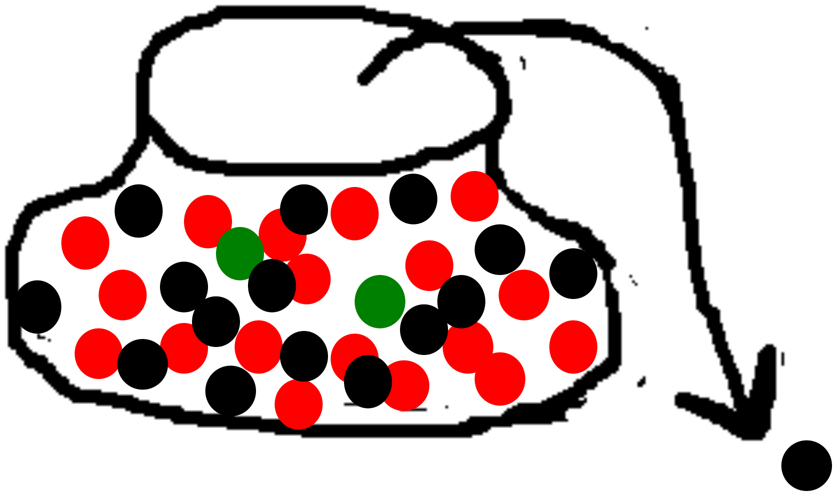
Explain

- Simulate the spins from a roulette wheel and track the winnings for a specified number of bets for a specified bet size

Explain

Spins from a roulette wheel behave like draws from an urn

18 Red Balls
18 Black Balls
2 Green Balls



Draw 1 ball
Win if it's Red
Replace ball in urn
Ready to play again

Code an example

```
wheel = rep(c("red”,"black”,"green"),
            c(18, 18, 2))

spins = sample(wheel, 100,
               replace = TRUE)

winnings = numeric(100)
winnings[spins == "red"] = 1
winnings[spins != "red"] = -1

totalWinnings = sum(winnings)
```

Can we work with 1 and -1 instead of red, black and green?

Simplify Code

```
wheel = rep(c(1, -1), c(18, 20))

wins = sample(wheel, 100,
              replace = TRUE)

totalWinnings = sum(wins)
```

How would this code change for the other betting scenario?

Code for Other Scenario

```
wheel = rep(c(1, -1), c(18, 20))

wins = sample(wheel, 1,
              replace = TRUE)

totalWinnings = sum(wins * 100)
```

How do these 2 scenarios compare?
To answer this question we want to evaluate our code many times

Generalize

What are the inputs?

A. Number of spins
B. Size of bet
C. Number of +1s and -1s
D. A & B
E. A, B, & C

Generalize

What are reasonable default values?

A. Number of spins
B. Size of bet
C. Number of +1s and -1s

1 spin
$1 bet
18 +1s and 20 -1s

Encapsulate

```
betRed =
   function(numBets,betAmt = 1) {

   wheel = rep(c(1, -1), c(20, 18))
   wins = sample(wheel, numBets,
                 replace = TRUE)

   totWinnings = sum(wins * betAmt)
   return(totWinnings)
}
```

Try out

```
> betRed(1, 100)
[1] -100
> betRed(1, 100)
[1] 100
> betRed(1, 100)
[1] 100
> betRed(1, 100)
[1] -100
> betRed(1, 100)
[1] -100
> betRed(1, 100)
[1] -100
> betRed(1, 100)
[1] -100
```

```
> betRed(100, 1)
[1] 8
> betRed(100, 1)
[1] -8
> betRed(100, 1)
[1] 10
> betRed(100, 1)
[1] -8
> betRed(100, 1)
[1] -20
> betRed(100, 1)
[1] -2
> betRed(100, 1)
[1] -4
```

The winnings look different, but if we could play the game over and over again, how would they compare?

Call function many times

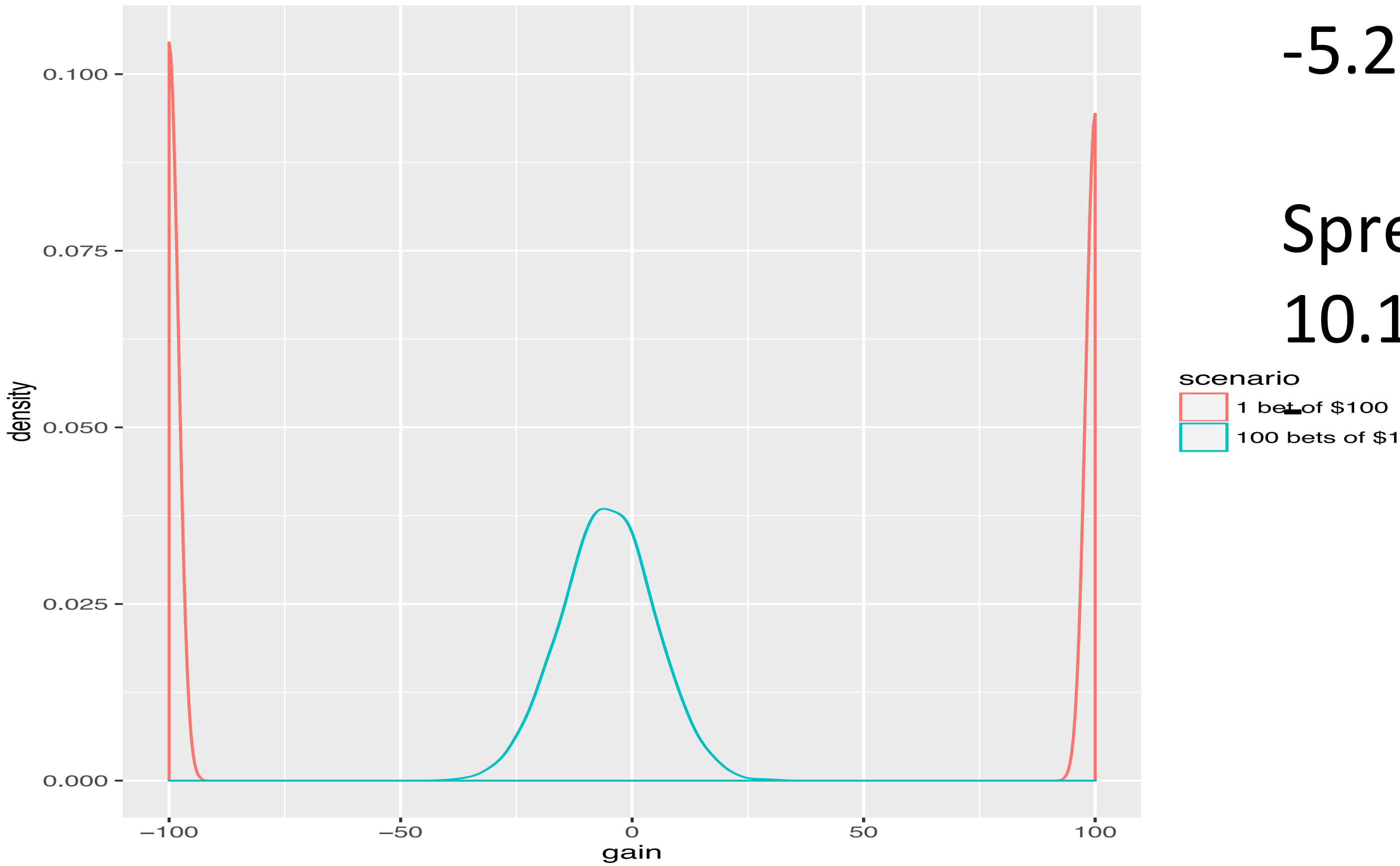
```
red100B.1D = sapply(rep(100, 10000),
                    betRed, betAmt = 1)

red1B.100D = sapply(rep(1, 10000),
                    betRed, betAmt = 100)
```

Notice that the calls to betRed do not depend on the input – shortcut

```
replicate(10000, betRed(100, 1))
```

What next?

We have the results from 10000 roulette spins for each scenario.



Mean

-5.23 and -5.06

Spread

10.1 and 99.9