Functional Programming with R (Answers Included)

Miles D. Williams
Stephen Mullins

08/14/2019

Now that you've gained some familiarity with loops, let's add just another layer of complexity. This won't hurt too badly, we promise...

`R` is a functional programming language. In short, that means that you can create your own functions in `R` that will do basically whatever you tell them to do. (Word to the wise, they will also do whatever stupid thing you tell them to do)

This might seem like an odd feature. Why write a new function? `R` already comes with so many nice built-in functions; adding our own may appear silly.

But, consider for a moment the value of the `for` loop. It allows you to automate an iterative task. It takes what would otherwise require 1,000 lines of code and condenses it to a mere handful. Functional programming allows you to do much the same thing. In fact, you can write functions that perform loops for you, without having to rewrite a loop each time you want to do an iterative task.

A Simple `coin_toss` Function

Before getting too wild and crazy, let's start with a very simple coin toss function. This function will be somewhat redundant, but it's sole purpose is only to demonstrate the basic logic of functional programming.

We begin by specifying a `coin_toss` function as follows:

```
coin_toss = function(tosses, bias = 0.5){
  # ...
}
```

We specify an object `coin_toss` where we use the assignment operator, `=`, followed by the `function( ... )` command. Within the `function` command we enter `tosses` and `bias = 0.5`. This tells R that we want to create two commands in our `coin_toss` function, (`tosses` and `bias`) and that we want to specify a default value of 0.5 for the `bias` command.

Next we need to fill the `{ ... }` with content, which will allow the function to operate:

```
coin_toss = function(tosses, bias = 0.5){
  output = rbinom(n = tosses, prob = bias, size = 1)
  value = rep("heads", len = length(output))
  value[output == 0] = "tails"
  return(value)
}
```

In the above, we've returned to our old friend `rbinom`. In our function, we've told R that whatever integer value we input to `tosses` will be the number of times we generate a value with `rbinom`. Additionally, we've told R that the value we assign to `bias` is the probability of choosing 1 with `rbinom` (0.5 by default). The output from `rbinom` is assigned to an object `output` within the function. `output` will not be saved to the global environment but will, in essence, only exist within the function itself. With `output` defined as a vector of 0s and 1s, we then create a new object called `value` which we first specify as a vector where each element is the character string `"heads"`. We then use the `[ ]` operator to specify that for each instance

where `output` equals 0, we want `value` to be `tails`. Finally, we tell R to return the vector of `"heads"` and `"tails"` contained in the object `value`.

With our new function in hand, let's take it for a test run. First, notice the error that emerges if we run `coin_toss` without specifying a number of tosses:

```
try(coin_toss())
```

```
## Error in rbinom(n = tosses, prob = bias, size = 1) :
##   argument "tosses" is missing, with no default
```

This occurs because there is no default value for tosses, as the error message indicates. We must tell the function the number of times we wish to toss the coin:

```
coin_toss(tosses = 5)
```

```
## [1] "heads" "heads" "tails" "tails" "heads"
```

Though we do not need to enter a value for `bias`, since by default the function will specify the bias as 0.5, we can override this value when we call the function:

```
coin_toss(tosses = 5, bias = .99)
```

```
## [1] "heads" "heads" "heads" "heads" "heads"
```

A More Complicated Example

The above `coin_toss` function was fairly trivial. The function saves us from rewriting only three lines of code. The real value of functional programming can be more readily seen when the routine we would like to run is more complicated and verbose. As a more complicated example, consider a new `coin_toss` function that, in addition to the previous function, lets us specify a certain number of heads at which point we would like the function to terminate. Further, we want this function to print out a message alerting us to this termination if it occurs.

The below code specifies this new routine. It's details have been annotated in the code itself:

```
coin_toss = function(
  tosses = 1,  # Set default number of tosses
  bias = 0.5,  # Set default bias of coin (values may range from 0 to 1)
  stop_if=NULL # By default, the function does not stop once a predetermined
               # number of heads have been tossed.
  ){
  # Make empty vector.
  x = 0

  # Run this vector through a loop per the number of tosses specified by the user.
  for(i in 1:tosses){

    # If we don't care about reaching a certain number of heads:
    if(is.null(stop_if)){
      x[i] = rbinom(n = 1, size = 1, prob = bias)

    # If we do care about reaching a certain number of heads:
    } else {
      x[i] = rbinom(n = 1, size = 1, prob = bias)
      if(sum(x[1:i])==stop_if){
        print(paste("You have tossed",stop_if, "heads!")) # return message
```

```
        break # break the loop
      }
    }
  }

  # Set 1 values to read as "heads" and 0s to read as "tails."
  value = rep("heads", len = length(x))
  value[x == 0] = "tails"

  # Return the vector of heads and tails
  return(value)
}

# Run default:
coin_toss()
```

```
## [1] "heads"
```

```
# Run 5 times:
coin_toss(tosses = 5)
```

```
## [1] "tails" "tails" "heads" "heads" "tails"
```

```
# Run 200 times, or until we get 5 heads:
coin_toss(tosses = 200, stop_if = 5)
```

```
## [1] "You have tossed 5 heads!"
```

```
##  [1] "heads" "heads" "tails" "tails" "tails" "tails" "tails" "heads"
##  [9] "heads" "heads"
```

This function is several times more complex than the previous, which (hopefully) serves to demonstrate the value of functional programming as an addendum to loops. Just as loops simplify our life by automating iterative tasks, functional programming simplifies our life by automating lengthy and complex routines. As a final example, and as an excuse to introduce another function (`sample`), consider the following `dice_roll` function:

```
dice_roll = function(sides = 6, rolls = 1, dice = 1, snake_eyes = FALSE){
  value = 0
  if(snake_eyes == FALSE){
    for(i in 1:rolls){
      value[i] = sum(sample(1:sides, replace = T, size = dice))
    }
  } else {
    for(i in 1:rolls){
      value[i] = sum(sample(1:sides, replace = T, size = dice))
      if(sum(value[i])==dice){
          print(paste("Snake eyes in", i,"rolls!"))
          break
      }
    }
  }
  return(value)
}
```

The above function will by default roll a six-sided die one time. It may, however, be used to a roll a die with any number of sides, for as many rolls as desired. The user may also specify that more than one die may be

cast per roll. Further, the option `snake_eyes = TRUE` will tell the function to terminate if you roll a series of straight 1s. If you get "snake eyes," a message is returned to alert you.

Here's an example of the function in action:

```
dice_roll(rolls = 1000, dice = 2, snake_eyes = T)
```

```
## [1] "Snake eyes in 63 rolls!"
```

```
##  [1] 10  6  6  8  6  6 11  7  9  8  6 12  6  7  4 12 10  7  4 11  7 11  6
## [24]  8  4  4 12 11  5  4  7  7  8  5  8  8  8  6  8  8  7  8  7  4 10  5
## [47]  6  7  9  6  3  8  9  7 10  6  6  5  9  6 11  5  2
```

At the heart of `dice_roll` is the `sample` function. `sample` allows us to sample from the elements of a vector, either with or without replacement. Consider the following demonstration. Say we have a vector `x = 1:10`. By default, if we enter `x` into `sample` we will get the following output:

```
x = 1:10
sample(x)
```

```
##  [1] 10  2  7  8  6  5  3  9  4  1
```

Imagine our vector `x` is like pieces of paper, 10 in total, each of which we have numbered. `sample` is like a hat. In this instance, what we have done is put our 10 pieces of paper into this hat, shaken them up, and pulled them out, one by one, until we've removed all pieces. That is, we have sampled from the pieces of paper without replacement.

Now, suppose we wish to sample with replacement. We can tell `sample` to do this with the commands `replace = T` and `size = 10`, the former of which indicates that we want to resample from `x` with replacement and the latter of which indicates we want to resample 10 times:

```
sample(x, size = 10, replace = T)
```

```
##  [1] 8 9 1 8 7 6 5 8 4 7
```

In a way, `sample` is much like `rnorm` and `rbinom`. The main difference is that, rather than sample from a theoretical distribution (whose parameters we may specify), we sample from a unique set of predetermined values. We may sample from a character vector:

```
sample(c("heads","tails"), replace = T, size = 10)
```

```
##  [1] "tails" "heads" "tails" "tails" "heads" "tails" "tails" "heads"
##  [9] "tails" "heads"
```

Or we may use it to sample with replacement from, say, a random variable:

```
sample(rnorm(n = 10, mean = 5), replace = T, size = 10)
```

```
##  [1] 5.291626 5.291626 5.291626 4.174793 5.063456 6.142989 5.063456
##  [8] 5.828094 4.591055 5.063456
```

The principle of resampling with replacement goes by another name, "bootstrapping." Though we do not cover this procedure here, it will be useful to have some exposure to resampling with replacement going forward.

Questions

1. Using `sample` create a simpler version of the `dice_roll` function that merely lets you specify the number of sides, the number of dice, and the number of rolls. Set the default values for this function to `sides = 4`, `dice = 2`, and `rolls = 10`. Test this function to see if it works.

```
# The answer:
dice_roll = function(sides = 4, dice = 2, rolls = 10){
  output = 0
  for(i in 1:rolls){
    output[i] = sum(sample(1:sides, size = dice, replace = T))
  }
  return(output)
}
dice_roll()
```

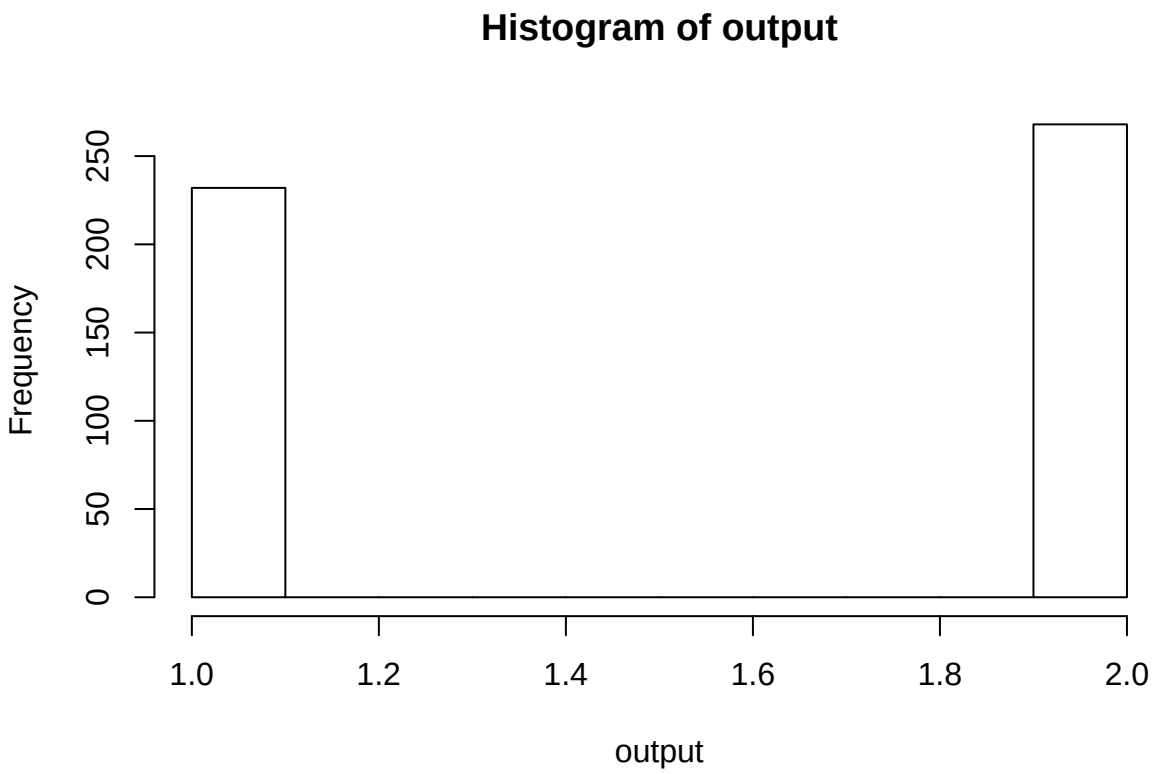
```
##  [1] 4 4 6 4 8 4 5 8 5 8
```

2. Update this new `dice_roll` function. Rather than return the raw output from the function, have the function return a histogram of the output.

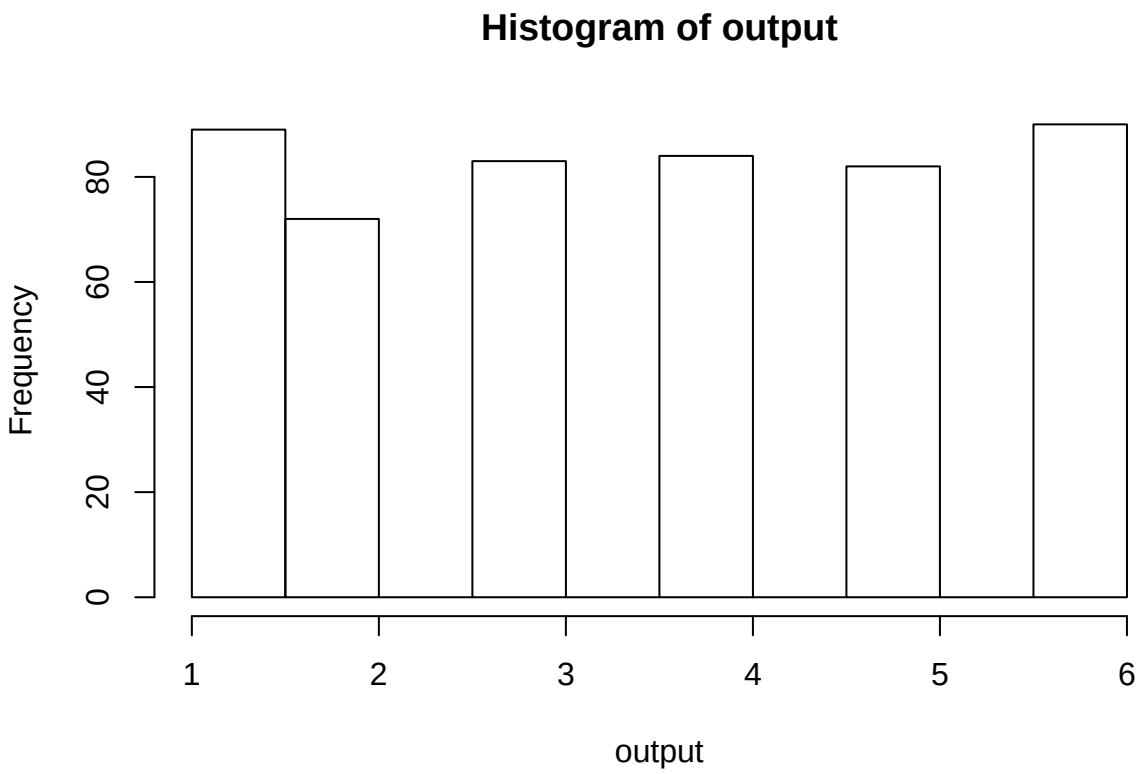
```
dice_roll = function(sides = 4, dice = 2, rolls = 10){
  output = 0
  for(i in 1:rolls){
    output[i] = sum(sample(1:sides, size = dice, replace = T))
  }
  return(hist(output))
}
```

3. Use the updated `dice_roll` function to plot the distribution of values obtained by rolling 1, 2 sided die, 500 times. Play around with the parameters and describe how the distribution of results changes as we increase the sides and the number of dice.

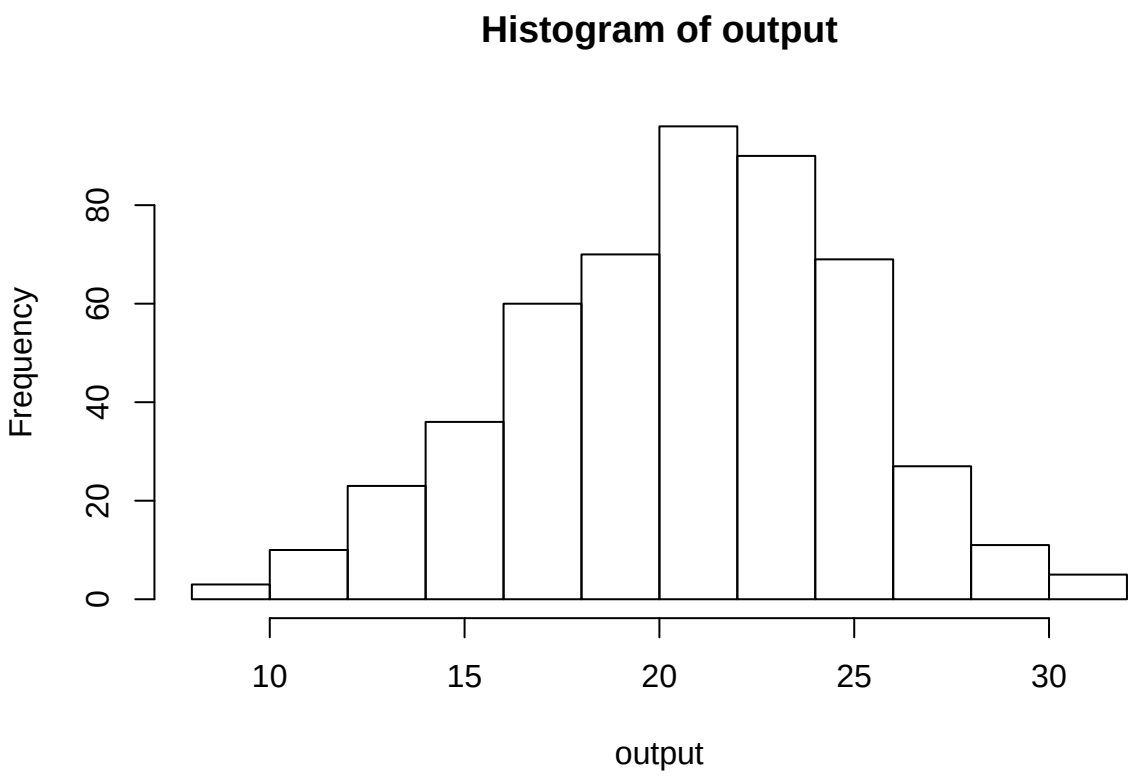
```
dice_roll(sides = 2, dice = 1, rolls = 500)
```




```
dice_roll(sides = 6, dice = 1, rolls = 500)
```




```
dice_roll(sides = 6, dice = 6, rolls = 500)
```


Histogram of output
Frequency
0
20
40
60
80
10
15
20
25
30
output