Functional Abstraction and Probabilistic Nontransitivity

Doug Koerber

11/28/2017

```
suppress <- function(x) {
  suppressMessages(suppressWarnings(x))
}
suppress(library(ggplot2))
suppress(library(dplyr))
```

Tip:

In the code chunk above, you can see the `suppress` function that I define. What this function does is suppress the messages and warnings of any function that it is called upon. I did this to prevent unnecessary output from taking up excessive space in this post.

Introduction

Functional abstraction involves suppressing the details of computations in order to make a computing process easier to understand. Functional abstraction is something that we haven't explicitly covered this semester, but we have been unknowingly abstracting away since we learned about creating functions. Rather than give many basic examples of how functional abstraction works, I will instead use it to cover an interesting topic in statistics: probabilistic nontransitivity.

What is Transitivity?

Before we can explore what nontransitivity is, we must define what it means to be *transitive*. Perhaps the easiest way to understand this concept is by thinking about values on a 1-dimensional number line. Let's pick three positive integers; 5, 72, and 113. Transitivity claims that because 113 is *greater than* 72, and because 72 is *greater than* 5, then it must follow that 113 is *greater than* 5. All that we care about in transitivity is that because *a* is greater than *b*, and *b* is greater than *c*, then we know *a* to be greater than *c*.

What is Nontransitivity?

Nontransitivity can be a somewhat confusing concept at first. I believe the easiest way to understand this is by thinking about a game of rock, paper, scissors (RPS). In RPS, there is not a single strategy that always wins a match - think about it! *Paper* beats *rock*, and *rock* beats *scissors*, but *paper* does not beat *scissors*. Therefore no matter which of the three options I pick, there is always the possibility that someone else will beat me. Compare this to a game in which you play against one other person and you both pick from our list of numbers in the previous paragraph (5, 72, and 113) and attempt to select the highest number. 113 beats 72, and 72 beats 5, therefore 113 beats 5. In this scenario, you would always pick 113 because you will always either win or tie - never lose. To introduce functional abstraction and also give a more complicated example of probabilistic nontransitivity, I will now walk through an example that uses four 6-sided dice.

Our First 2 Dice

In the code chunk below, you can see that we define two dice - our first, and most basic, form of functional abstraction. We assign the faces of these dice as a simple numeric vector of length 6, and we are now able to refer to these dice as their assigned name rather than the `combine` functions they represent. `die1` has the number 3 on all sides, and `die2` has four 4s and two 0s.

```
die1 <- c(rep(3, 6))
die2 <- c(rep(4, 4), rep(0, 2))
```

Abstracting Our Method for Rolling Dice

Now that we have our dice, we need to create an abstraction for rolling our dice. This will make is easier to re-roll as we introduce new dice, and comparing any of our dice will be much quicker. Our function takes three inputs; the primary die, the alternate die, and the number of rolls we want to make. The function `roll_dice` will first set the seed in order to maintain reproducibility. Next, it creates a vanilla data.frame that will be altered as dice are rolled in throughout the function. After that, a list of the two dice that are passed into the function as parameters will be created - this is used later on when abstracting away the graph we will create. **Note:** the special operator `<<-` used in this statement will assign the variable `rolled_dice` *outside* of the function. Without this operator, we would lose our `rolled_dice` list as soon as the function has ended. We then roll the dice `n` number of times, which depends on the final parameter of our `roll_dice` function, and save the results of each roll in our `rolls` data.frame. The `rolls` data.frame is then returned by the function.

```
roll_dice <- function(prm_die, alt_die, n) {
  set.seed(518)

  rolls <- data.frame(index = as.numeric(1:n),
                      results = as.numeric(1:n)
                      )

  rolled_dice <<- c(substitute(prm_die), substitute(alt_die))

  for (i in 1:n) {
    player_1 <- sample(prm_die, 1)
    player_2 <- sample(alt_die, 1)
    if (player_1 > player_2) {
      result = 1
    } else {
      result = 0
    }
    rolls[i, 'results'] = result
  }
  return(rolls)
}
```

Abstracting Our Method for Calculating Winrate

The `roll_dice` functions merely returns a data.frame with 2 columns; one that contains the cumulative number of rolls, and another that contains a dummy variable that represents whether or not the `prm_die` won or lost. However, we need more information than this to determine if one die has a higher probability of winning against the other. To attain this information, and make it easily reproducible, we are now going to abstract away the process for calculating the winrate of the dice. Our function `dice_stats` only takes in one argument; a data.frame that contains the information output of `roll_dice`. `dice_stats` will then calculate the cumulative sum of the results for our rolls, the winrate of the primary die, and the winrate of the alternate die.

```
dice_stats <- function(rolls) {
  rolls <- rolls %>%
               mutate(prm_cusum = cumsum(results),
                      winrate = (prm_cusum / index),
                      winrate2 = (1 - winrate)
                     )
  return(rolls)
}
```

Abstracting Our Method for Graphing Winrates

At this point, we have all of the data we need. All that is left before we have a well-abstracted and easily reproducible process is the abstraction of how our data is graphed. The function `graph_probs` will take in the output of `dice_stats` and generate a beautiful graph of the winrates of both dice using `ggplot2`. Functional abstraction allows us to contain the desired graphical parameters and various `geoms` in one simple function call, rather than having to copy and paste the code for our graph over and over.

```
graph_probs <- function(dat) {
  prm_die <- as.character(rolled_dice[[1]])
  alt_die <- as.character(rolled_dice[[2]])
  ggplot(data = dat, aes(x = index)) +
    geom_line(aes(y = winrate, col = prm_die), show.legend = TRUE) +
    geom_line(aes(y = winrate2, col = alt_die), show.legend = TRUE) +
    scale_y_continuous(limits = c(0, 1)) +
    labs(y = 'Winrate', x = 'Number of Rolls', color = 'Dice',
         title = paste0('Winrates Between ', prm_die, ' and ', alt_die))
}
```

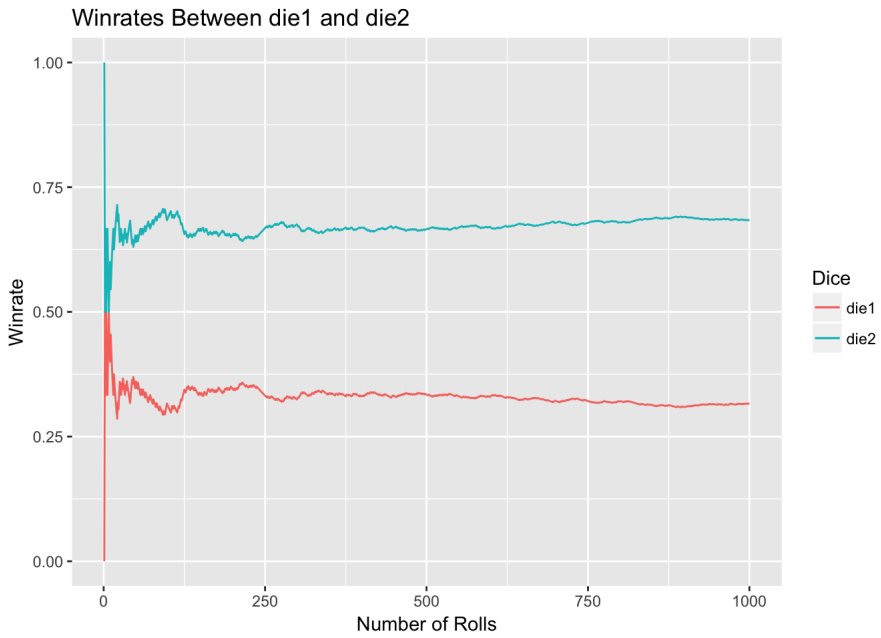
Proving Nontransitivity

Now that we have completed all of our functional abstractions (creating dice, rolling dice, calculating useful roll statistics, and generating a graph), we can use them to easily show how certain dice can be nontransitive.

`die1` vs `die2`

`die1` has 3s on all sides, and `die2` has 4s on four sides and 0s on two sides. Therefore we know that if `die2` lands on 4, it will win, no matter what. If `die2` lands on 0, we will lose, no matter what. This leads us to conclude that `die2` will win 66.7% of the time, and lose 33.3% of the time. To examine this, we will use our functional abstractions in the code chunk below.

```
my_rolls <- roll_dice(die1, die2, 1000)
test <- dice_stats(my_rolls)
graph_probs(test)
```



The graph above shows that `die2` does in fact have a higher probability of winning when played against `die1`.

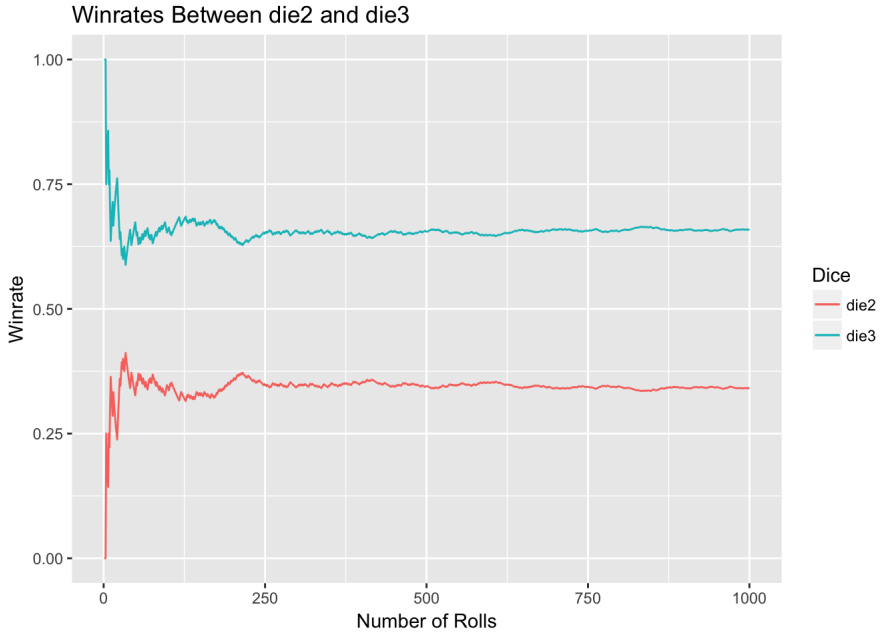
`die2` vs `die3`

We are now going to introduce another die, called `die3`, and compare it with `die2`.

```
die3 <- c(rep(5, 3), rep(1, 3))
```

`die2` has 4s on four sides and 0s on two sides, and `die3` has 5s on three sides and 1s on three sides. Which die has a higher winrate when played against the other? Once again, we will use just three lines of code to run this test.

```
my_rolls <- roll_dice(die2, die3, 1000)
test <- dice_stats(my_rolls)
graph_probs(test)
```



In the graph above, we can see that `die3` has a higher probability of winning than `die2`.

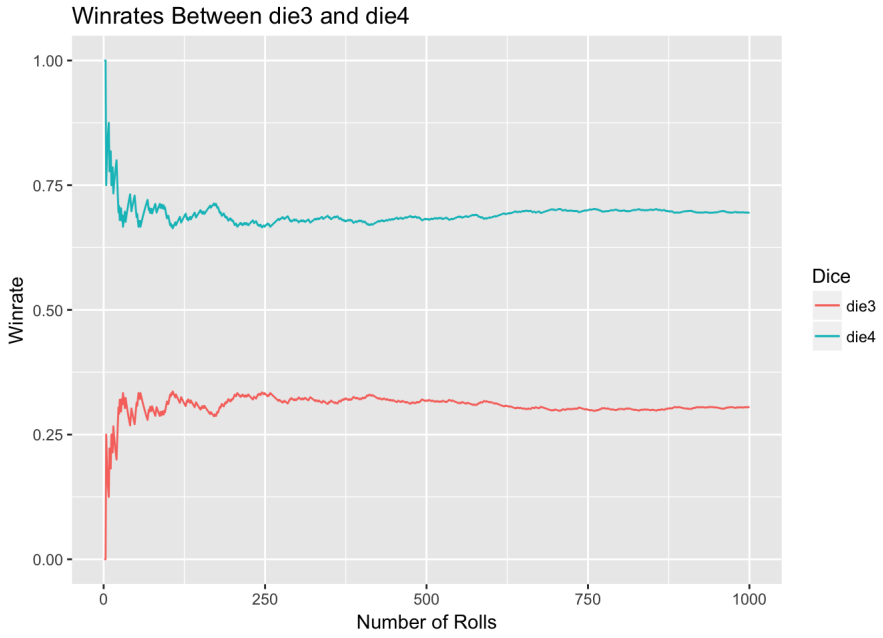
`die3` vs `die4`

At this point, we know that `die3` is has a higher probability of winning than `die2`, and that `die2` as a higher probability of winning than `die1`. We could stop here and test for transitivity by testing that `die3` has a higher probability of winning than `die1`, but let's add one more die to show how the example scales to more than just three options. We are now going to introduce our final die, called `die4`, and compare it with `die3`.

```
die4 <- c(rep(6, 2), rep(2, 4))
```

`die3` has 5s on three sides and 1s on three sides, and `die4` has 6s on two sides and 2s on four sides. Which die has a higher winrate when played against the other? Once again, we will use just three lines of code to run this test.

```
my_rolls <- roll_dice(die3, die4, 1000)
test <- dice_stats(my_rolls)
graph_probs(test)
```



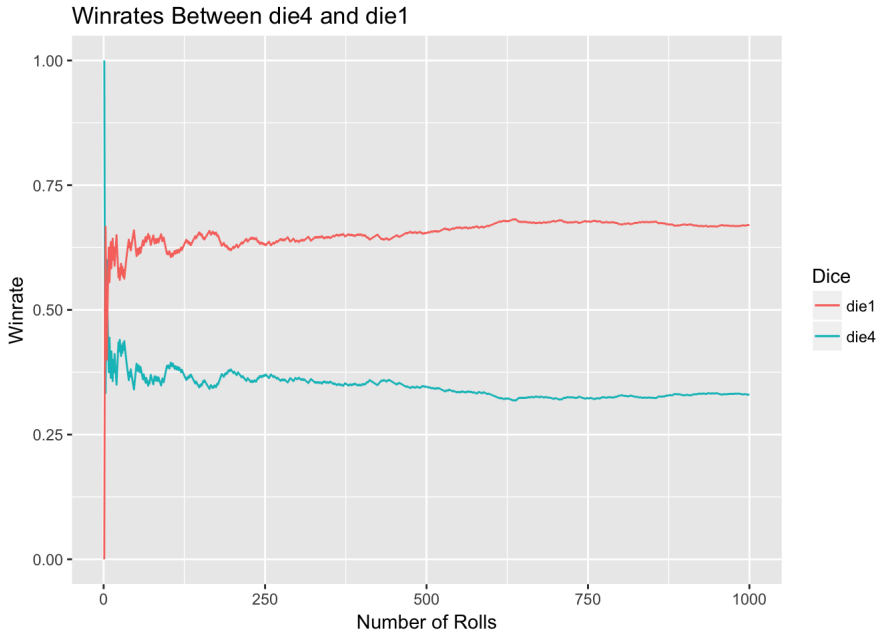
In the graph above, we can see that `die4` has a higher probability of winning than `die3`.

Discovering Nontransitivity

Here is what we have discovered in each independent test so far: `die4` > `die3` > `die2` > `die1`, or `die4` beats `die3`, which beats `die2`, which beats `die1`.

Is this conclusion *really* correct? Let's test the *transitive* conclusion that because `die4` beats `die3`, and because `die3` beats `die2`, and because `die2` beats `die1`, then `die4` must beat `die1`.

```
my_rolls <- roll_dice(die4, die1, 1000)
test <- dice_stats(my_rolls)
graph_probs(test)
```



We have discovered nontransitivity among these four dice! The graph above shows that `die4` *does not* have a higher probability of winning when played against `die1`.

Further Testing Thanks to Abstraction

Because we have abstracted away our testing procedures, it is easy for us to run as many transitivity tests as we want. New die can be created in a single line of code, and we can test for winrates using only 3 lines of code. We could even abstract our three `roll_dice`, `dice_stats`, and `graph_probs` functions into one *master* abstraction, but this is not entirely necessary. In fact, one might consider this to be over-abstraction because this would not allow us to keep track of the roll data that we calculate in the steps to producing a winrate graph.

Take Home Message

My hope with this post is that you now feel more comfortable creating your own functions, using those functions to reduce the amount of code you have to type, increasing report reproducibility with your functions, and removing unnecessary details from certain processes by using functional abstraction. I hope this example, although maybe not so glamorous, provided insight into why one might find it useful to define their own functions, and how a function can be more than just `rescale` - you can make graphs easy to create, simulate random processes, and much, much more.

References

1. https://en.wikipedia.org/wiki/Nontransitive_dice
2. https://stat.ethz.ch/R-manual/R-devel/library/base/html/assignOps.html
3. https://www.r-bloggers.com/closures-in-r-a-useful-abstraction/
4. http://singingbanana.com/dice/article.htm
5. https://en.wikipedia.org/wiki/Transitive_relation
6. https://www.programiz.com/r-programming/environment-scope
7. https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf