# Laboratory 6 – Functions Roshambo Battle Royale!

Objectives:

Hopefully after last week’s lab you have a much better appreciation for breaking down and solving a more complex problem programmatically. R is also really good at performing simple tasks and solving small problems, but doing it many times over and over. This week you’ll explore how to make this easier through modularizing and re-using code. We will examine rather simple functions, and learn how these self-contained chunks of code can interact and be re-used to create rather complicated functionality but without writing much code yourself.

### Review and Introduction:

Previously, in lecture we learned about the syntax and structure of functions in R. Functions take arguments, perform a sequence of operations, sometimes calling other functions (or even itself, if it’s recursive), and often return something useful to us. This is the core of a *functional* programming language like R. R is a *functional* language since everything in the R environment is either a data object or a function (hence the term, *functional*… honest, I don’t make this stuff up).

So this week, you’re going to learn a bit more about programming functions through the process of creating the most sophisticated Rock-Paper-Scissors player ever! (Okay, probably not, but you will definitely be able to do better than break even!) In order to understand how to do that, you will be employing almost everything you’ve learned up until now, including a generous dose of code interpretation.

Even though this is meant to be a fun lab, I hope you realize that the practice you get from breaking down the code I provide, incorporating elements from existing code and thinking about the problem like a programmer are all very useful. All of these things should improve your ability to think about problems abstractly, conceive solutions and figure out how to program those solutions concretely in the R environment.

### The Problem

The rock-paper-scissors (RPS) game is an extremely simple concept. Two players, choose one of three options, each of which will “win” against only one of the possible three outcomes, losing to another and drawing with itself. So if two players are making choices between “rock,” “paper,” and “scissors” completely at random, over time they have the neutral expectation of winning, losing and drawing exactly 1/3 of the time.

An online game released by the New York Times showed to the world, however, that an RPS artificial intelligence (AI), a bot, can consistently beat human opponents. Why? The reason lies in that humans are predictable, we almost never can play completely randomly, and over time a bot with enough sophistication can detect patterns in our play and exploit those patterns. It is impossible to gain an advantage against a player playing completely randomly.

So, if playing randomly is the optimal strategy, what is the point of an RPS programming competition? The answer is that a completely random player will only expect to win about 1/3 of the time, with a win:loss ratio of about 1.0 (since you’re drawing 1/3 of time as well). And while it’s trivial to program a bot that plays randomly, against weaker opponents with predictable play you should be able to do much better than a win:loss ratio of 1.0 by exploiting that weakness. But in exploiting that weakness you yourself cannot play randomly, and therefore leave your play open to exploitation itself.

Therefore, identifying the pattern and the predictability of the play of your opponent based on previous turns and outcomes is a solution well-suited to computers. And today you will create your own RPS AI. You will try to maximize your outcomes against a suite of competitors who play with a variety of strategies, and hopefully come out with a bot capable of challenging your classmates for the title of supreme RPS champion!

### The Anatomy of a Function: The RPS Bot

Let’s begin by looking at the files in the supplied /src folder. You should see seven bots that I’ve provided for you. These will be your initial competitors. There is an 8th bot called “player\_skeleton.r”, let’s open it and check out the anatomy of an RPS player bot.

Save off “player\_skeleton.r” to a new file, give it the same name as the name you want your player to have (along with the “.r” extension of course). And begin your programming by naming your function with the same name as your file.

You should notice that the only runnable code in the file is a function definition. How many arguments does it have? How many of those arguments are optional? Why are these arguments optional? What happens when we run the function?

You should work through this code, and continue on to the comments below the original player\_skeleton() function declaration. Run the supplied functions and start to piece together what these functions and the provided source code does. This is very much what you will do as we move forward, learning and figuring out how to use analytical functions supplied in both the base R and loaded libraries. So pay attention to the process that we use to sort it out.

## Questions

**If you construct a bot that can generate a total win:loss ratio after competing against all of the bots of greater than 1.9, you will receive full credit for this week’s lab. If you don’t, I will base your score on a percentage of how close you came to the target of 1.9.**

**Next week we will discuss our bots together to compare our strategies and see who has the highest win:loss ratio! The winner will get to choose their position within the presentation order for their final project. But most importantly, you will earn glory and honor as well as the admiration of your classmates!**

**You must submit your player’s code, as an “.r” file attached to your Canvas submission. I will expect it to run without errors when supplied to the throwdown() function, verifying over 1000 rounds that it does indeed score a win/loss ratio greater than 1.9.**

**[Total Points: 100]**