R Small Group: Class 5

Amy Allen & Dayne Filer

July 12, 2016

Using this document

- Code blocks and R code have a grey background (note, code nested in the text is not highlighted in the pdf version of this document but is a different font).
- # indicates a comment, and anything after a comment will not be evaluated in R
- The comments beginning with ## under the code in the grey code boxes are the output from the code directly above; any comments added by us will start with a single #
- While you can copy and paste code into R, you will learn faster if you type out the commands yourself.
- Read through the document after class. This is meant to be a reference, and ideally, you should be able to understand every line of code. If there is something you do not understand please email us with questions or ask in the following class (you're probably not the only one with the same question!).

Class 5 expectations

1. Know how to write and run a basic function in R
2. Understand function environments and how functions find things
3. Understand the "do not repeat yourself" (DRY) principle

Basic Functions

So far we've used a lot of functions that already exist in R. We can also add our own functions, which on of the great strengths of R. User-written functions take the following structure.

```
myfunction <- function(arg1, arg2, ... ){
  do some stuff
  return(object)
}
```

Let's make a simple function that just returns the arument that it is given.

```
first_function <- function (x){
  return(x)
}
```

Now if we hand a value to `first_function` it should return that value

```
first_function(9)
## [1] 9
```

This function isn't really very useful, so let's try make a function that squares a value and then adds one to it.

```
second_function <- function(x){
  ans <- x^2+1
  return(ans)
}
second_function(9)
## [1] 82
```

Function Environment

An environment is a place to store variables. So far when we've made assignments in R, they have been added as entries to the global environment.

Functions have their own environments. When a function is called a new environment is created. This new environment is called the evaluation environment. Functions also have an enclosing environment, which is the environment where the function was defined. For a functions defined in the workspace the enclosing environment is the global envrionment.

When a function is evaluated R looks through a series of environments for variables called. It first searches the evaluation environment and then the enclosing environment. This means that a global variable can be referenced inside a function. This principle is shown below by `third_function` which sums the argument `x` with the variable `a` from the global environment.

```
a <- 9
third_function <- function(x) {x+a}
third_function(11)
## [1] 20
```

The evaluation environment is populated with local variables as the evaluation of the function procedes. Let's take `second_function` for example. Within the function the variable `ans` is assigned. However, when we evaluate the function this variable does not show up in the global environment because it was assigned in the evaluation environment.

```
second_function(9)
## [1] 82
ls()
## [1] "a"               "first_function"  "second_function" "third_function"
```

As you can see when we list objects in the global environment we only get the global variable `a` assigned previously and the three functions that we have defined so far.

DRY principle

The don't repeat yourself (DRY) principle states that "every piece of knowledge must have a single, unambigous, authorative representation within a system" according to programmers Dave Thomas and Andy Hunt. This means that you want to define each variable once and only once, and you don't want to duplicate variables. Duplication can lead to bugs in your code that could easily be avoided.

Adding Complexity for Functions

Now we understand how R finds variables for functions we can start to add complexity to functions.

Suppose you got your apples and oranges from two different farmers. They both give you an inventory list, but one farmer measures his fruit in inches, and the other measures his fruit in centimeters. You want to convert the data so all of it is in centimeters.

The following code generates the inventory lists from each farmer

```
# Farmer 1 measures in inches
farmer1 <- list(type = sample(x = c("orange", "apple"),
                              size = 25,
                              replace = TRUE),
                wdth = rnorm(n = 25, mean = 6, sd = 2.5))
# Farmer 2 measures in centimeters
farmer2 <- list(type = sample(x = c("orange", "apple"),
                              size = 25,
                              replace = TRUE),
                wdth = rnorm(n = 25, mean = 15, sd = 6))
```

Now let's write a function that will convert centimeters to inches and try it out.

```
# write function to convert centimeters to inches
convert_cm_in <- function(data){
  new_data <- data*0.393701
  return(new_data)
}
# apply to the widths provided by farmer2
convert_cm_in(farmer2$wdth)
##  [1]  1.639297  4.530532  3.286089  3.507966  5.522106  7.235569  9.797999
##  [8]  4.078695  9.699004  3.170533  7.456512 11.926757  5.823404  4.323703
## [15]  5.887551 10.103355  3.215889  9.136605  9.046221  6.700333  5.921797
## [22]  4.829604  5.039710  7.436901 10.795921
# update the data from farmer2
farmer2$wdth <- convert_cm_in(farmer2$wdth)
```

Functions can be even more complex than this. For example, we can make the code that we wrote last week into a function to make it easy to apply to multiple datasets reproducibly. To change from a script to a function you must write the main body of the code in terms of the argument (in this case `data`) rather than a specific set of data. Note that this function will only run when applied to a list or data frame that has the entries `wdth` and `type`

```
fruit_machine <- function(data){
  # create results list
  res <- list (type = character(), pieces = numeric(),
               slice_width = numeric())
  ## For-loop iterating through the 1:(number of fruit)
  for (f in 1:length(data$type)) {
    f_type <- data$type[f] ## Get the fruit type from the list
    f_wdth <- data$wdth[f] ## Get the fruit width from the list
    n_pieces <- 1 ## Each fruit is initially 1 piece

    ## Peel the oranges
    if (f_type == "orange") {
      f_wdth <- f_wdth - 1/8
    }
```

```
    ## Divide the fruit
    while (f_wdth >= 1) {
      f_wdth <- f_wdth/2
      n_pieces <- n_pieces*2
    }

    res$type[f] <- f_type
    res$pieces[f] <- n_pieces
    res$slice_width[f] <- f_wdth
  }
  return(res)
}
```

Now let's apply this function to our two data sets `farmer1` and `farmer2`

```
results1 <- fruit_machine(farmer1)
results2 <- fruit_machine(farmer2)
```

Small Group Exercises

Write functions to do some basic math and statistics.

1. Write a function that solves the quadratic formula.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

   Use it to solve the following equations.

$$x^2 + x - 4 = 0$$
$$x^2 - 3x - 4 = 0$$
$$6x^2 + 11x - 35 = 0$$

2. Write a function that finds the standard deviation of a set of numbers. Use the `sd` function to check your results. Note: $\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}$