STAT 209: Lab 9

Code

Making Your Code More Modular

Goal

Learn to identify repetition in code that could be made more concise by writing a function or an iterative construct, and learn to write such things in R.

The Data

One of these days, we'll work with some different data, I promise.

Load the packages and data:

Code

A repetitive task

One of the questions we have been interested in when working with the baby names data is: "In what year did the name reach its peak in popularity?"

For the name Colin, for example, we can answer this question (more or less) with the following pipeline:

Code:

Code

```
## # A tibble: 1 x 2
##    year overall_percentage
##   <dbl>              <dbl>
## 1  2004              0.122
```

(I say more or less because in adding one half of the proportions within each sex, I'm implicitly assuming equal numbers of male and female births overall, which is not exactly correct, but it's not toooo far off.)

So, the name "Colin" has never been a more popular choice for new babies than it was in 2004, being given to (approximately) 0.12% of all babies, regardless of sex.

Writing a function

If I want to get the same result for a different name, say "Fred", I could just copy and paste the above code and change the name. But,

1. This is annoying
2. This makes my code harder to read
3. If I want to change something (for instance, I decide I want to correct the fact that I'm assuming equal numbers of male and female babies born per year), I have to go through and change it in every place.

Instead, I can write a function that captures the "template" for this calculation, and lets me instantiate that template with whatever specific input I want.

What are the inputs to this function? If I always want to return the single peak year, there's just one input: the name. So I can write:

Code:

Code

Now I can just run this function, plugging in whatever name I want, and I quickly get results. Here are some results for various members of my own family:

Code:

Code

```
## # A tibble: 1 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Colin  2004              0.122
```

Code

```
## # A tibble: 1 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Megan  1985              0.544
```

Code

```
## # A tibble: 1 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Bruce  1951              0.369
```

Code

```
## # A tibble: 1 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Mary   1880               3.63
```

Code

```
## # A tibble: 1 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Arlo   2017             0.0287
```

Code

```
## # A tibble: 1 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Esai   2002            0.00261
```

Function signatures

Most functions are designed to work with certain *kinds* of inputs. For example, `name_of_interest` in the above should be a quoted text string, not a number, not a data frame, etc. In some languages, when you write a function, you explicitly encode that your function *must* take a certain kind of input. In R, you don't do that; R is what's called a "dynamically typed" language, in which functions will accept whatever input you give them, and if what they do happens to work for that input (even if it's not something the author envisioned), it will do it; otherwise you'll get an error somewhere in the execution of the function.

As I'm sure you've seen, It can be difficult to track down what is causing an error in R, and so it is worth trying to avoid this sort of thing by including some documentation at the top of your function indicating what type of input you intend the function to be used with. The user of the function is free to violate that intention, but at least they go in with their eyes open.

In R, you can type `formals(my_function_name)` to see at least the *names* of the arguments to a function. For example: **Code**

Code

```
## $name_of_interest
```

we see that `most_popular_year` takes one argument, called `name_of_interest` .

For functions that are part of an R package, documentation is viewable with the `?function_name` syntax (or, equivalently, with `help(function_name)`).

Return values

The "value" of a function (the thing it returns, if, for example, you are assigning its result to a variable) is, by default, the return value of the last command executed by the function. In our function there is only one command (which consists of several component commands connected in a pipeline), and so the return value is the return value of the pipeline.

If we wanted to be more explicit, we could assign the result of the pipeline to a variable (we might call it `result`), and add the line `return(result)` at the end of our function.

It's a good idea to do this if your function contains more than one line, to make it clear which part of the function body is the return value. For one-liners (and maybe some very simple multi liners), it's a judgment call as to whether it makes it clearer to do this or not.

In "statically typed" languages, part of the signature of a function is the type of thing that it returns. In dynamically typed languages, the type of the return value could well depend on the types of the arguments provided. But, again, it is a good idea to document the *intended* return type.

Default arguments

Often times, we want to allow our functions to be flexible, by allowing the user to alter several aspects of what it does. We make our functions more flexible by adding more inputs, each of which constitutes a "degree of freedom" for our function. But if most use cases involve sensible defaults, it is cumbersome to force the user to input these defaults every time they use the function.

We can have the "best of both worlds" (flexibility without cumbersome function calls) by using *default argument values*.

For example, I could make my `most_popular_year` function more flexible by having the function return the most popular `n` years:

Code:

Code

As written, this function now *requires* the user to specify a number of years. The following will produce an error, since I haven't supplied the `num_years` argument.

Code:

Code

If we think that most often the user will just want to see the single most popular year, I can give that second argument a *default value* that makes the above work as before.

Function (Re-)definition:

Code

Some function calls:

Code

```
## # A tibble: 1 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Colin  2004              0.122
```

Code

```
## # A tibble: 5 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Colin  2004             0.122
## 2 Colin  2003             0.116
## 3 Colin  2005             0.107
## 4 Colin  2006             0.0882
## 5 Colin  2009             0.0862
```

Code

```
## # A tibble: 5 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Colin  2004             0.122
## 2 Colin  2003             0.116
## 3 Colin  2005             0.107
## 4 Colin  2006             0.0882
## 5 Colin  2009             0.0862
```

Function scope

You might have noticed that in our function we hardcoded the dataset to be `babynames` . If we had tried to call this function without having run `library(babynames)` above, we'd get an error, since `babynames` would not then be defined. If you "undo" the `library()` command and then try to call the function, R will complain.

Code:

Code

(Let’s make sure to bring back the `babynames` library for later)

Code:

Code

How does R know where to look for definitions of things that are referenced in a function? A complete answer would involve a lot of caveats, but for the most part, R will first look inside the function for a definition (at its arguments, and at anything that is created within the function itself), and if it doesn’t find anything, it will look in the “global” environment (that is, at stuff that was defined or loaded into the environment by previous assignments or calls).

In theory we could have made a `dataset` argument to our `most_popular_years()` function so that it didn’t depend on something defined in the global environment:

Code:

Code

Notice that we still have hardcoded variable names here, so this function will only work if the dataset we provide has the right columns, but this can be useful if we are going to work with (say) different subsets of a dataset that we obtain by `filter()` ing:

Code

```
## # A tibble: 1 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Mary   1922               2.90
```

Exercises on Functions

The following exercises involve writing functions designed to tell us things about flights, using the `nycflights13` package.

Exercise 1 Write a function that, for a given carrier identifier (e.g. `DL`), will retrieve the five most common airport destinations from NYC in 2013, and how often the carrier flew there.

Exercise 2 Use your function to find the top five destinations for Delta Airlines (`DL`)

Exercise 3 Use your function to find the top five destinations for American Airlines (AA). How many of these destinations are shared with Delta?

Exercise 4 Write a function that, for a given airport code (e.g. BDL), will retrieve the five most common carriers that service that airport from NYC in 2013, and what their average arrival delay time was.

Iteration

Computers are excellent at repetition, as long as you tell them *precisely* what to repeat.

Remember the example above where I called my function on a bunch of names of people in my family? I can make that even more efficient by creating the list of names I'm interested in up front, and then telling the computer "Call this function on each one of these names, and return the results".

In R, the `lapply()` (short for "list apply") is useful for this sort of thing, provided the list of argument values goes with the first argument of my function.

Code:

Code

```
## [[1]]
## # A tibble: 1 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Colin  2004              0.122
## 
## [[2]]
## # A tibble: 1 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Megan  1985              0.544
## 
## [[3]]
## # A tibble: 1 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Bruce  1951              0.369
## 
## [[4]]
## # A tibble: 1 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Mary   1880               3.63
## 
## [[5]]
## # A tibble: 1 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Arlo   2017             0.0287
## 
## [[6]]
## # A tibble: 1 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Esai   2002            0.00261
```

Now, this result is a bit inelegant; the function always returns a data frame with a single entry. Wouldn’t it be nice if we could “stack” these into a single data frame?

We can! The `bind_rows()` function will do this for us.

Code

```
## # A tibble: 6 x 3
##   name   year overall_percentage
##   <chr> <dbl>              <dbl>
## 1 Colin  2004            0.122  
## 2 Megan  1985            0.544  
## 3 Bruce  1951            0.369  
## 4 Mary   1880            3.63   
## 5 Arlo   2017            0.0287 
## 6 Esai   2002            0.00261
```

`for` loops

If you have programmed in another language before, you likely would have handled something like this using a "loop" such as a `for` loop. You can write for loops in R, but it is more "idiomatic" to use the above sort of "apply" construct; and in certain cases it's more efficient too (which is important when there are a lot of iterations involved).

If you find yourself wanting a `for` loop, ask yourself whether you could handle what you wanted to do with a function whose first argument is the thing you want to iterate over.

Exercise 5 Use `lapply()` and the function that you wrote in Exercise 1 to find the five most common airport destinations for Delta, American, and United.

Exercise 6 Use `lapply()` and the function that you wrote in Exercise 4 to find the five most common carriers to Bradley International, Los Angeles International, and San Francisco International airports.

Applying a function to a grouped data frame

The following function computes the top 10 most popular names in the dataset passed to it via the `data` argument:

Code:

Code

```
## # A tibble: 10 x 2
##    name     births
##    <chr>     <int>
##  1 James   5173828
##  2 John    5137142
##  3 Robert  4834915
##  4 Michael 4372536
##  5 Mary    4138360
##  6 William 4118553
##  7 David   3624225
##  8 Joseph  2614083
##  9 Richard 2572613
## 10 Charles 2398453
```

If we want to apply this function to find the most popular name in a particular decade, we could simply `filter` our data to keep only years in the range of interest, and call the function on the filtered data.

But suppose we want to do this for *every* decade in the 20th century. We could theoretically create 10 datasets, put them in a list, and use `lapply` on the list of datasets. But it's simpler to take advantage of the `do()` function for this. This is seen most easily by example:

Code:

Code

```
## # A tibble: 140 x 3
## # Groups:   decade [14]
##    decade name    births
##     <dbl> <chr>    <int>
##  1   1880 Mary     92030
##  2   1880 John     90395
##  3   1880 William  85246
##  4   1880 James    54323
##  5   1880 George   47980
##  6   1880 Charles  46879
##  7   1880 Anna     38320
##  8   1880 Frank    31135
##  9   1880 Joseph   26404
## 10   1880 Emma     25512
## # … with 130 more rows
```

Note that since `top10()` always returns a data frame with 10 rows, the result of this operation is a big "stacked" data frame with ten names per decade.

Exercise 7 Find the total number of rows in `top_by_decade`, and make sure you understand what each one represents.

Exercise 8 The 27th row of `top_by_decade` is

Code

```
## # A tibble: 1 x 3
## # Groups:   decade [1]
##   decade name  births
##    <dbl> <chr>  <int>
## 1   1900 Anna   55099
```

What does this tell us?

Note: If you have worked with the `mosaic` package, you likely used another function called `do()`. It's related to the `dplyr` one, but not identical, so if you are working in an R session with both packages loaded, it's a good idea to be explicit about which one you want to be using. You can do this by writing either `dplyr::do()` or `mosaic::do()`.

Exercise 9 Find the most popular year for the name of your choice in each 20 year span.

Getting credit

Revisit previous labs and/or projects, and identify a task you did where you could have used some combination of the tools you learned in this lab. Describe it on Slack in the #lab9 channel.