Don't Repeat Yourself - Eliminate repetition in your code

Learning Objectives

After completing this tutorial, you will be able to:

- Be able to define the DRY principle.
- Describe how functions can make your code easier to read
- Identify repeated lines of code that could be replaced by functions.

What you need

You will need a computer with internet access to complete this lesson.

Efficient coding 101

If you remember from the first week of class, we discussed the concept of reproducibility. One component of reproducibility that we emphasized was making your code easier for your colleagues and your future self to work with. This is important, but there is an even more selfish reasons to consider writing efficient code.

Efficient coding will make your life easier too.

> Reproducibility is actually all about being as lazy as possible! – Hadley Wickham (via Twitter, 2015-05-03)

Don't Repeat Yourself - DRY

DRY (Don't Repeat Yourself) is a principle of software development. The focus of DRY is to avoid repetition of information.

Why?

When you write code that performs the same tasks over and over again, any modification of one task requires the same change to be made to every single instance of that task! Editing every instance of a task is a lot of work.

We can use functions in our code to replace tasks that we are performing over and over. Source: Francois Michonneau

Instead, we can create **functions** that perform those tasks, using sets of **arguments** or inputs to specify how the task is performed.

The benefits of functions

- **Modularity:** If you write function for specific individual tasks, you can use them over and over. A function that you write for one script can even be reused in other scripts!
- **Fewer global variables:** When you run a function, the intermediate variables that it creates are not stored in your global environment. This saves memory and keeps your global environment cleaner.
- **Better documentation:** Well documented functions help the user understand the steps of your processing.

- **Easier to maintain / edit:** When you create a function for a repeated task, it is easy to edit that one function. Then every location in your code where that same task is performed is automatically updated.
- **Testing:** We don't discuss this in our class this week, but writing functions allows you to more easily test your code to identify bugs.

Modularity

Good functions only do one thing, but they do it well and often in a variety of contexts. Often the operations contained in a good function are generally useful for many tasks. Take for instance the R function `mean()`, which computes sample means. This function only does one thing (computes a mean), but it can do it very well in many different circumstances. This function is highly **modular** in the sense that it can be easily combined with other functions to accomplish a variety of tasks.

When you write **modular** functions, you can even re-use them for other projects. Some people even write their own R packages for personal use that contain custom functions for their work!

Fewer global variables

Global variables are objects in R that exist within the **global environment**, which you can think of as a storage container for all of the objects in your R session. When you code line by line, you end up creating numerous intermediate variables that you don't need to use again. This is suboptimal for a few reasons:

1. Global variables accumulate in your environment, but you may never need to access intermediate results.
2. Global variables consume memory, which can lead to reduced performance. In the extreme, you may run out of RAM if you have too many objects in your global environment.
3. Having many global variables increases the odds that you will define another global variable with the same name by chance, inadvertently overwriting existing objects and leading to a variety of problems, some of which can be difficult to diagnose.

Functions are useful in part because they have their own environments which are created when the function is called, and destroyed (by default) once the function returns it result. Objects defined inside of functions are thus created inside of that function's environment. Once the function is done running, those objects do not persist. In addition to minimizing the number of objects in your global environment, defining objects within functions is good because those objects will not overwrite previously existing objects in the global environment.

For instance, if we define an object in our global environment:

```
# first define x -x is a global variable
x <- 10
```

Then define and call a function which defines an object called `x`:

```
modify_x <- function(x) {
  x <- 2 * x
  x
}

modify_x(x)
## [1] 20

x
## [1] 10
```

Notice that the `x` in our global environment is unchanged, even though it was changed in our function. The `x` in the function environment was removed once the function returned its result. Functions allow you to focus on the inputs and the outputs of your workflow rather than the intermediate steps.

Better documentation

Ideally, your code should be written so that it's easy to understand. However, what might seem clear to you now might be clear as mud 6 months from now or even 3 weeks from now (remember we discussed your future self in week 1 of this class).

Well written functions are documented with inputs and outputs clearly defined. Well written functions also use names that help you better understand what task the function performs. It is good to name functions using verbs to make your code more expressive and indicate what the function *does*. This makes your code easier to read for both you, your future self and your colleagues.

Easier to maintain / edit

If all your code is written line by line, with repeated code in multiple parts of your document, it can be challenging to maintain.

Imagine having to fix one element of a line of code that is repeated many times. You will have to find and replace that code to implement the fix in EVERY INSTANCE it occurs in your code. This makes your code difficult to maintain.

Do you also duplicate your comments where you duplicate parts of your scripts? How do you keep the duplicated comments in sync? **A comment that is misleading because the code changed is worse than no comment at all.**

Re-organizing your code using functions (or organizing your code using functions from the beginning) allows you to explicitly document the tasks that your code performs.

Testing

While we won't cover this in our class this week, functions are also useful for testing. As your code gets longer and more complex, it is more prone to mistakes. For example, if your analysis relies on data that gets updated often, you may want to make sure that all the columns in your spreadsheet are present before performing an analysis. Or that the new data are not formatted in a different way.

Changes in data structure and format could cause your code to not run. Or in the worse case scenario, your code may run but return the wrong values!

If all your code is made up of functions, that have built in tests to ensure that they run as expected, then you can control the input and test for the output. It is something that would be difficult to do if all of your code is written, line by line with repeated steps.

Summary

In short, it is a good idea to learn how to

1. Think about how to **modularize** your code and identify generalizable tasks
2. Write functions for parts of your code which include repeated steps
3. Document your functions clearly, specifying the structure of the inputs and outputs.

   input –> function does something –> output