Introduction to Functional Programming

Jeff Oliver

20 May, 2020

An introduction to writing functions to improve your coding efficiency and optimize performance.

Learning objectives

1. Write and use functions in R
2. Document functions for easy re-use
3. Replace loops with functions optimized for vector calculations

Don't Repeat Yourself

The DRY principle aims to reduce repetition in software engineering. By writing and using functions to accomplish a set of instructions multiple times, you reduce the opportunities for mistakes and often improve performance of the code you write. Functional programming makes it easy to apply the same analyses to different sets of data, without excessive copy-paste-update cycles that can introduce hard-to-detect errors.

Writing functions

Why do we write functions? Usually, we create functions after writing some code with certain variables, then copy/pasting that code and changing the variable names. Then more copy/paste. Then we forget to change one of the variables and spend a day figuring out why our results don't make sense.

For example, consider the `iris` data set:

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

And if we look at the values for each of the four measurements, some are quite different in their ranges.

```
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
```

```
##  versicolor:50
##  virginica :50
##
##
##
```

Given this variation in ranges and magnitudes, we will have to standardize the variables if we want to run analyses like principle components analysis (PCA) (PCA is beyond the scope of this lesson, but it is covered in a separate lesson). Standardizing a variable means we transform the data so the mean value is zero and the standard deviation is one.

We'll start by creating an R script to hold our work. R scripts are simple text files with a series of R commands and are a great way to make our work reproducible. For all scripts, it is a good idea to start off with a little information about what the script does (and who is responsible for it). So create a script (from the File menu, select New File > R Script or use the keyboard shortcut of Ctrl+Shift+N or Cmd+Shift+N).

```
# Standardize iris values
# Jeff Oliver
# jcoliver@email.arizona.edu
# 2018-08-16
```

R will ignore anything that appears to the right of the pound sign, `#`. This information allows anyone who is reading the script to quickly know (1) what the script does and (2) who to contact with questions. While we are at it, go ahead and save this file as "standardize-iris.R".

To standardize a variable, we subtract observed values by the mean, and divide that difference by the standard deviation ($(x_i - \mu)/\sigma$).

We start by performing the calculation on the petal length data, first calculating the two summary statistics (mean and standard deviation), then using them in the formula for standardization.

```
# Standardize petal length
mean_petal_length <- mean(iris$Petal.Length)
sd_petal_length <- sd(iris$Petal.Length)
standard_petal_length <- (iris$Petal.Length - mean_petal_length)/sd_petal_length
```

We can use the `summary` command to ensure the mean is zero,

```
summary(standard_petal_length)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -1.5623 -1.2225  0.3354  0.0000  0.7602  1.7799
```

and the `sd` function to make sure the standard deviation is one.

```
sd(standard_petal_length)
```

```
## [1] 1
```

Now that we have the calculation for petal length, we can copy the code and update it to perform calculation for petal width.

```
# Standardize petal width
mean_petal_width <- mean(iris$Petal.Width)
sd_petal_width <- sd(iris$Petal.Width)
standard_petal_width <- (iris$Petal.Width - mean_petal_width)/sd_petal_width
summary(standard_petal_width)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -1.4422 -1.1799  0.1321  0.0000  0.7880  1.7064
```

But we had to change three instances of "Length" to "Width" and seven (seven!?) instances of "length" to "width". Did copy/paste really save time? And how many opportunities for mistakes?

Here we have a perfect opportunity to write a function that will do those three steps for any data we want. *And* we can avoid the inevitable errors that arise with the copy-paste-update process.

Write a function!

When we write a function, it is very much like assigning a value to a variable. However, in this instance, we assign the value produced by a special function in R called `function` (shocking, I know). When writing functions, I generally start by naming the function and creating a function with nothing in it. We will call our function `standardize`:

```
standardize <- function(data) {

}
```

Note that there is a variable `data` in the call to `function`. This is referring to the data that we will perform our standarization calculation on. The next step is to outline what actions the function will peform. I start with comments to help with this process.

```
standardize <- function(data) {
  # Calculate mean of data
  # Calculate standard deviation of data
  # Calculate standardized scores
}
```

These comments then provide a roadmap of the code that needs to be written inside the function. We start by adding the mean calculation to the function:

```
standardize <- function(data) {
  # Calculate mean of data
  mean_data <- mean(data)
  # Calculate standard deviation of data
  # Calculate standardized scores
}
```

Next we fill in the standard deviation and standardization code:

```
standardize <- function(data) {
  # Calculate mean of data
  mean_data <- mean(data)
  # Calculate standard deviation of data
  sd_data <- sd(data)
  # Calculate standardized scores
  stand_data <- (data - mean_data)/sd_data
}
```

At this point, remember that we want to use this function with the iris data. However, at no point in the function are we referring to the iris data frame. This is because we want our function to be *generalizable* to other data as well. The last thing we need to do is to send the standardized values back to the user. This is known as "returning" a value, and we use the function `return` to do so:

```
standardize <- function(data) {
  # Calculate mean of data
  mean_data <- mean(data)
  # Calculate standard deviation of data
  sd_data <- sd(data)
```

```
  # Calculate standardized scores
  stand_data <- (data - mean_data)/sd_data
  return(stand_data)
}
```

At this point, our function is almost ready for use. The last thing we need to do is to read it into memory. We do this like we are running any other line of code in R: place your cursor on one of the curly braces and run the code with Ctrl-Enter or Cmd-Enter.

Now we can use the function. We can replace the three lines that performed the calculation for petal length with a single line that calls our `standardize` function:

```
standard_petal_length <- standardize(data = iris$Petal.Length)
```

Like all code, we want to provide enough information so it is easy for someone else to use. So we want to add documentation that consists at a minimum of three things:

1. A brief explanation of what the function does
2. A description of the information needs to be provided to the function
3. An explanation of what the function returns

```
# Standardize a vector of numerical data
# data: vector of numerical data
# returns: vector of original data standardized to have mean = 0, sd = 1
standardize <- function(data) {
  # Calculate mean of data
  mean_data <- mean(data)
  # Calculate standard deviation of data
  sd_data <- sd(data)
  # Calculate standardized scores
  stand_data <- (data - mean_data)/sd_data
  return(stand_data)
}
```

Now we have a function we can use on each of the columns in our data frame. Yay, right? Well, almost. We still have to run the `standardize` function four times on the iris data frame. What if we had 100 columns of data? Or 1000 columns? That is simply too much copy and paste. We can use the `apply` function to run the `standardize` function on each column of data.

We need to tell `apply` three things:

1. The data frame to use, in this case we are using the `iris` data
2. Whether to do calculations for each row or each column; we use the `MARGIN` parameter for this: `MARGIN = 1` means do calculation for each *row* while `MARGIN = 2` means do calculation for each *column*. We want to do calculations for each column
3. The function to use; here we use the function we just wrote, `standardize`

Note the `apply` function arguments `X`, `MARGIN`, and `FUN` are all capitalized. I have no idea why.

```
iris_stand <- apply(X = iris, MARGIN = 2, FUN = standardize)
```

```
## Warning in mean.default(data): argument is not numeric or logical:
## returning NA
```

```
## Error in data - mean_data: non-numeric argument to binary operator
```

Well that didn't work. Why not? Let us look at the `iris` data again:

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
```

```
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

The error occurs because the `standardize` function is run on *all* columns, including the "Species" column, which contains non-numerical data. We need to skip that column when we send data to the `apply` function. One way to do this is to use the negation operator (the minus sign, "-") with the index of the column we want to skip. In our case, the "Species" column is the fifth column, so we drop that column in our call to `apply` via `iris[, -5]`.

```
# Run standardize on all numeric columns
iris_stand <- apply(X = iris[, -5], MARGIN = 2, FUN = standardize)
```

We also need to convert the output back to a data frame with `as.data.frame`. We can then look at the first few rows with `head`.

```
# Convert output to data frame
iris_stand <- as.data.frame(iris_stand)
head(iris_stand)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1   -0.8976739  1.01560199    -1.335752   -1.311052
## 2   -1.1392005 -0.13153881    -1.335752   -1.311052
## 3   -1.3807271  0.32731751    -1.392399   -1.311052
## 4   -1.5014904  0.09788935    -1.279104   -1.311052
## 5   -1.0184372  1.24503015    -1.335752   -1.311052
## 6   -0.5353840  1.93331463    -1.165809   -1.048667
```

Great. But what's missing? The "Species" column is not included in the results because we did not send it to the `apply` function (remember, we had to skip it because it contained non-numerical data). We can add it back, though, copying from the "Species" column in the original data.

```
# Add back species column
iris_stand$Species <- iris$Species
head(iris_stand)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1   -0.8976739  1.01560199    -1.335752   -1.311052  setosa
## 2   -1.1392005 -0.13153881    -1.335752   -1.311052  setosa
## 3   -1.3807271  0.32731751    -1.392399   -1.311052  setosa
## 4   -1.5014904  0.09788935    -1.279104   -1.311052  setosa
## 5   -1.0184372  1.24503015    -1.335752   -1.311052  setosa
## 6   -0.5353840  1.93331463    -1.165809   -1.048667  setosa
```

So to summarize at this point, we wrote a function, `standardize`, and applied that to all the numeric data columns in the iris data.

```
# Run standardize on all numeric columns
iris_stand <- apply(X = iris[, -5], MARGIN = 2, FUN = standardize)
# Convert output to data frame
iris_stand <- as.data.frame(iris_stand)
# Add back species column
iris_stand$Species <- iris$Species
```

At this point, though, we have to ask ourselves two questions:

1. How can I use this function with other projects?
2. Can we have a function do all that business with `apply`, converting back to a data frame, and add on those missing columns?

The answer to both is, of course, yes.

Use the `source`

Right now, the function we wrote, `standardize`, is locked up inside the standardize-iris.R file. This makes it hard to use outside of this file. In order to make the `standardize` function useable elsewhere, we need to move the function to a separate file and use the `source` command in any script that we would like to use our function. For the first step, create a new R script through the File menu or with the keyboard shortcut (Ctrl+Shift+N or Cmd+Shift+N). Let us call this file "standardization-functions.R".

As we did with our script, we want to start the file with a header that has a little information about what is included in the script.

```
# Functions for standardizing data
# Jeff Oliver
# jcoliver@arizona.edu
# 2020-05-20
```

Now cut (Ctrl-X or Cmd-X) the function from our original script (the file called "standardize-iris.R") and paste (Ctrl-V or Cmd-V) it into this new script (the file called "standardization-functions.R").

So we have done the first part (moving our function to a separate file), but how do we actually use it?

The `source` command.

But we should pause for a moment to clean up our workspace. We can remove all variables from memory by running:

```
rm(list = ls())
```

We are doing this now just to ensure we are using the `standardize` function defined in the file we just created.

When we use `source`, we tell R to run *all* the code in a specific file. Go to your first script ("standardize-iris.R") and add the `source` command, setting `file` to the name of our script with the `standarize` function ("standardization-functions.R").

```
source(file = "standardization-functions.R")
```

An important thing to remember about the `source` function is that it will run *all* the code in that file. We can see this behavior by adding a line before our `standardize` function and re-running the `source` function.

First, add this line to the standardization-functions.R file:

```
message("Loading functions from standarization-functions.R")
```

Save the standardization-functions.R file, then re-run the line with `source` in "standardize-iris.R":

```
source(file = "standardization-functions.R")
```

```
## Loading functions from standarization-functions.R
```

If there are additional commands in the file you call `source` on, such as variable declarations, it will run the code and those variables will be added to memory. Sometimes that is OK, but I will caution against using `source` for anything other than loading functions into memory.

Functions within functions

Now that we have a separate file for functions, let us return to the other point, about writing a function that standardizes *all* numeric columns of a data frame.

So let us go back to our file with the `standardize` function ("standardization-functions.R") and at the end of the file, start defining a new function we will call `standardize_df`. We will pass it one argument, `df`.

```
standardize_df <- function(df) {

}
```

Just as we did with `standardize` let us start by enumerating the steps to take to accomplish this. Since we already wrote the code to do each of these steps, we can paste the code in the function and comment it out. We can use this to guide the code we write. (If you don't have code that is easy to copy/paste, you can skip to writing comments about the steps that need to occur in the function.)

```
standardize_df <- function(df) {
  # Run standardize on all numeric columns
  # iris_stand <- apply(X = iris[, -5], MARGIN = 2, FUN = standardize)
  # Convert output to data frame
  # iris_stand <- as.data.frame(iris_stand)
  # Add back species column
  # iris_stand$Species <- iris$Species
}
```

Looking at what we did before, there presents a challenge: How do we know which columns to skip? That is, how can we tell which columns are not numeric? We "hard-coded" the script to skip the fifth column, which holds the species' names. But how do we make this flexible enough so we do not need users to enter which columns to skip? We can use the function `is.numeric` on each column to figure out which columns have numeric data and thus are appropriate for the standardization transformation.

So we need to add another step to our function, where we look at the incoming data (a data frame called `df`). The first thing the function needs to do is figure out which columns in `df` are numeric and which columns are not numeric.

```
standardize_df <- function(df) {
  # Identify numeric columns
  # Run standardize on all numeric columns
  # iris_stand <- apply(X = iris[, -5], MARGIN = 2, FUN = standardize)
  # Convert output to data frame
  # iris_stand <- as.data.frame(iris_stand)
  # Add back species column
  # iris_stand$Species <- iris$Species
}
```

To extract this information, we can use the `sapply` function, which is related to `apply`; if you want to read more about `sapply` (and `apply` and `lapply`), there is a nice discussion at https://nsaunders.wordpress.com/2010/08/20/a-brief-introduction-to-apply-in-r/.

The function `is.numeric` will tell us if an individual column has numeric data, so we can test out `sapply` and `is.numeric` on the iris data:

```
sapply(X = iris, FUN = is.numeric)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##         TRUE         TRUE         TRUE         TRUE        FALSE
```

This is great! The value returned is a logical vector (a vector of TRUE/FALSE values) that tells us which

columns are numeric (Sepal.Length, Sepal.Width, Petal.Length, Petal.Width) and which columns are not numeric (Species).

We can update our function to extract the information about whether or not a column is numeric.

```
standardize_df <- function(df) {
  # Identify numeric columns
  col_numeric <- sapply(X = df, FUN = is.numeric)
  # Run standardize on all numeric columns
  # iris_stand <- apply(X = iris[, -5], MARGIN = 2, FUN = standardize)
  # Convert output to data frame
  # iris_stand <- as.data.frame(iris_stand)
  # Add back species column
  # iris_stand$Species <- iris$Species
}
```

But now, the information stored within the function in the variable `col_numeric` allows us some efficiencies. Because `col_numeric` has as many elements as there are columns in `df`, we can get crafty with R to use our `standardize` function to *update* columns of interest. If we can update columns rather than creating a new data frame, we can skip the step where we add back those non-numeric column. What does this look like, then? We change the `apply` command to use the `col_numeric` vector and update the data frame `df` rather than creating an entirely new data frame.

```
standardize_df <- function(df) {
  # Identify numeric columns
  col_numeric <- sapply(X = df, FUN = is.numeric)
  # Run standardize on all numeric columns
  # iris_stand <- apply(X = iris[, -5], MARGIN = 2, FUN = standardize)
  df[, col_numeric] <- apply(X = df[, col_numeric],
                             MARGIN = 2,
                             FUN = standardize)
  # Convert output to data frame
  # iris_stand <- as.data.frame(iris_stand)
  # Add back species column
  # iris_stand$Species <- iris$Species
}
```

Note that I did not remove the comment with the old code. Take a moment to notice an important difference: we did not create a new data frame (in our hard-coded version, this was `iris_stand`). Instead, we assigned values to the numeric columns of `df` (via `df[, col_numeric]`). Similar to our previous code, though, inside the `apply` command we *did* skip all non-numeric columns in the data frame (also via `df[, col_numeric]`).

At this point, our function is almost done; we need to do two more things:

1. Use the `return` function
2. Document the function

While we add the `return` function, we can also clean up the commented-out code from inside the function.

```
standardize_df <- function(df) {
  # Identify numeric columns
  col_numeric <- sapply(X = df, FUN = is.numeric)
  # Run standardize on all numeric columns
  df[, col_numeric] <- apply(X = df[, col_numeric],
                             MARGIN = 2,
                             FUN = standardize)
  return(df)
}
```

And before we are done, we add the minimal documentation.

```
# Standardize all numeric columns of a data frame
# df: data frame
# returns: data frame with numeric columns standardized to mean = 0, sd = 1
standardize_df <- function(df) {
  # Identify numeric columns
  col_numeric <- sapply(X = df, FUN = is.numeric)
  # Run standardize on all numeric columns
  df[, col_numeric] <- apply(X = df[, col_numeric],
                             MARGIN = 2,
                             FUN = standardize)
  return(df)
}
```

At this point, we are ready to try it out, so save this file ("standardization-functions.R") and go back to our first script where we are doing the standardization of the iris data ("standardize-iris.R").

Now we need one line:

```
iris_stand <- standardize_df(df = iris)
```

This probably did not work. Why not?

Remember, when we use functions that are located in another file, we will need to run the `source` command any time we have made changes. So after saving standardization-functions.R, we need to re-run the `source` command to load any changes we made to standardization-functions.R.

```
source(file = "standardization-functions.R")
```

And re-run the line calling `standardize_df`.

```
iris_stand <- standardize_df(df = iris)
```

We can use `head` once more to see the first few rows of our new data frame.

```
head(iris_stand)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1   -0.8976739  1.01560199    -1.335752   -1.311052  setosa
## 2   -1.1392005 -0.13153881    -1.335752   -1.311052  setosa
## 3   -1.3807271  0.32731751    -1.392399   -1.311052  setosa
## 4   -1.5014904  0.09788935    -1.279104   -1.311052  setosa
## 5   -1.0184372  1.24503015    -1.335752   -1.311052  setosa
## 6   -0.5353840  1.93331463    -1.165809   -1.048667  setosa
```

Woo-hoo!

In summary, we created two files. One file, "standardization-functions.R", includes the two functions we wrote:

```
# Functions for standardizing data
# Jeff Oliver
# jcoliver@arizona.edu
# 2020-05-20

message("Loading functions from standarization-functions.R")

# Standardize a vector of numerical data
# data: vector of numerical data
```

```r
# returns: vector of original data standardized to have mean = 0, sd = 1
standardize <- function(data) {
  # Calculate mean of data
  mean_data <- mean(data)
  # Calculate standard deviation of data
  sd_data <- sd(data)
  # Calculate standardized scores
  stand_data <- (data - mean_data)/sd_data
  return(stand_data)
}

# Standardize all numeric columns of a data frame
# df: data frame
# returns: data frame with numeric columns standardized to mean = 0, sd = 1
standardize_df <- function(df) {
  # Identify numeric columns
  col_numeric <- sapply(X = df, FUN = is.numeric)
  # Run standardize on all numeric columns
  df[, col_numeric] <- apply(X = df[, col_numeric],
                             MARGIN = 2,
                             FUN = standardize)
  return(df)
}
```

Our analysis file is now quite small, as we have shifted much of the code to the standardization-functions.R file.

```r
# Standardize iris values
# Jeff Oliver
# jcoliver@email.arizona.edu
# 2018-08-16

source(file = "standardization-functions.R")

iris_stand <- standardize_df(df = iris)
```

Actually developing a function

So I need to come clean about writing these functions. In the lesson above, they are consise and fairly clean. They were certainly not born that way. In writing the `standardize_df` function, which is effectively three lines of code (find the numeric columns, run the `standardize` function on each column, return the values), I probably wrote 30 lines of code testing different ways of performing those steps. When writing a function, here are a few things to consider:

1. Clean out your workspace with `rm(list = ls())`. The code inside your function should not be dependent on *any* "outside" variables or data. That is, you only want to use varibles that are (1) passed to the function (those variables we name when using `function`, e.g. the `data` variable in the `standardize` function, `standardize <- function(data)`) or (2) defined *within* the function (like `mean_data <- mean(data)`).
2. Step through the function one line at a time. The process for this does require you to assign values to all those arguments named in the function definition. So to test the `standardize` function, I first assigned a value to the `data` variable, in this case, the petal length column, by running `data <- iris$Petal.Length`. Then I ran the first line of code in the `standardize` function (`mean_data <- mean(data)`), then the second (`sd_data <- sd(data)`), and so on (in this development process, you

do not run the line defining the function, `standardize <- function(data)`). Along the way, do some reality checks to make sure things work the way they should. For example, you can compare the output of `mean(data)` for petal length with the output of `mean(iris$Petal.Length)` with `all(mean(data) == mean(iris$Petal.Length))`.
3. Think from the point of a naive user. This is especially important when writing your documentation. Also, if you wait six months and come back to your functions, you will be that naive user.
4. Clean out your workspace.

Debugging tips

It is not working.

Why is it not working?

This will happen. Guaranteed. When debugging a function, one of the best functions to know about is `cat`. This allows you to print little messages to the console, building in those reality checks. Find points in your code where you know a variable should have a certain value and use the `cat` function (or multiple calls to `cat`) to print out what the value of the variable *really* is. For example, if we are debugging the `standardize` function, we can `cat` the values of the mean for a column in our data frame to make sure the values are what we expect. We would first update the function with a call to `cat`:

```
# Standardize a vector of numerical data
# data: vector of numerical data
# returns: vector of original data standardized to have mean = 0, sd = 1
standardize <- function(data) {
  # Calculate mean of data
  mean_data <- mean(data)
  cat("Value of mean: ", mean_data, "\n")
  # Calculate standard deviation of data
  sd_data <- sd(data)
  # Calculate standardized scores
  stand_data <- (data - mean_data)/sd_data
  return(stand_data)
}
```

Note that in this case we provide three things to the `cat` function (which I am pretty sure stands for "concatenate"):

1. `"Value of mean: "` is a bit of text that reminds me what is being printed
2. `mean_data` is the name of the variable that stores the mean of the column of data that was passed to the function as `data`; `cat` will not print "mean_data", but rather the value stored in `mean_data`
3. `\n` is the end of line character so whatever R outputs *after* this call to `cat` appears on a new line

Now we can run this function on the petal length column of the iris data (remember to reload `standardize` into memory first!) and compare the values to calcuations directly on the iris data:

```
# Test standardize function on petal length
petal_length_stand <- standardize(data = iris$Petal.Length)
# Calculate and print actual mean value of petal length
mean(iris$Petal.Length)
```

In addition to printing values, you can also use functions like `is.numeric` or `typeof` to ensure the data you *think* you are working with really is in the form you want it to be.

You will eventually want to remove those calls to `cat`, or at the very least comment them out so they don't print every time you run your function.

When debugging, also remember to make incremental changes. Change one line. Check to make sure it works as expected. Check it again with different data. Change the next line. Avoid taking a 50-line function, and changing 30 lines all at once. Where there was once one bug, now there are three.

Finally, remember to keep functions small (when you can). We could have written a single function to do all the work. That is, we could have just written `standardize_df`, but it would have made it tougher to debug. Some zealots even assert that a function should do one thing and only one thing. While that is a bit extreme, it is a useful guiding principle that helps streamline the development process. Check out the resources below for more information about writing functions and best practices.

Additional resources

- A deeper dive to functional programming
- An *even deeper* dive into functional programming
- Some opinions and suggestions for naming things like functions (see the 'Object names' section)
- A PDF version of this lesson

Back to learn-r main page

Questions? e-mail me at jcoliver@email.arizona.edu.