Introduction to Functional Programming

Jeff Oliver

09 September, 2019

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:

```r
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.

```r
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 variance 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) and add a few lines at the beginning of the script:

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

R will ignore anything that appears to the right of the pound sign, `#`.

We want to transform all the values so they have a mean of 0 and a variance of 1. The formula for this transformation is

$$z_i = (x_i - \mu)/\sigma$$

where

$$\begin{aligned}
&z_i = \text{transformed value of the } i^{th} \text{ observation}\\
&x_i = \text{the } i^{th}\text{observation of } x\\
&\mu = \text{mean value of all } x\\
&\sigma = \text{standard deviation value of all } x
\end{aligned}$$

We can compute all these values, starting with the data in the `Petal.Length` column:

```
# 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 check our calculations using the summary command to show the mean of the standardized variable is now equal to 0:

```
summary(standard.petal.length)
```

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

We can do the same thing again for the `Sepal.Length` column, starting by copying the code we used for petal lengths and pasting below those calculations.

```
# 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

# Standarize petal length
```

```r
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
```

Then we can update the values in the second block of code to use the `Sepal.Length` column:

```r
# 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

# Standarize sepal length
mean.sepal.length <- mean(iris$Sepal.Length)
sd.sepal.length <- sd(iris$Sepal.Length)
standard.sepal.length <- (iris$Petal.Length - mean.sepal.length)/sd.sepal.length
```

Again, we check our work with the `summary` function:

```r
summary(standard.sepal.length)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -5.8490 -5.1244 -1.8034 -2.5183 -0.8977  1.2761
```

Uh oh. The mean is *not* equal to zero… let's look at that code we updated again:

```r
# Standarize sepal length
mean.sepal.length <- mean(iris$Sepal.Length)
sd.sepal.length <- sd(iris$Sepal.Length)
standard.sepal.length <- (iris$Petal.Length - mean.sepal.length)/sd.sepal.length
```

The problem lies in the last line - we forgot to replace `Petal.Length` with `Sepal.Length`. When we update the code and run `summary`, we can see that it fixes the problem and that, as expected, our standardized variable now has a mean of zero.

```r
# Standarize sepal length
mean.sepal.length <- mean(iris$Sepal.Length)
sd.sepal.length <- sd(iris$Sepal.Length)
standard.sepal.length <- (iris$Sepal.Length - mean.sepal.length)/sd.sepal.length
summary(standard.sepal.length)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -1.86378 -0.89767 -0.05233  0.00000  0.67225  2.48370
```

Did copy/paste really save time? While copy/paste can be useful, it can also be a great opportunity to introduce mistakes into our code. If we want to avoid those mistakes, we can take advantage of writing our own functions in R.

Behold, a function

We start writing a function in a way very similar to assigning values to variables. We are going to write a function that will perform the standardization transformation, but first we add some comments about (1) what the function will do, (2) what input the function will take, and (3) what the function will output. Looking back at what we did previously, it's probably easiest to start with the last one (output).

```r
# goal:
# input:
# output: standarized variables
```

What information will we need to provide the function? The "input" we used above was a single column of numeric data.

```r
# goal:
# input: one column of numeric data
# output: standarized variables
```

Finally, what is the function going to do? Looking back at our code above, we see three steps:

1. Calculate the mean value for the data in the column
2. Calculate the standard deviation for the data in the column
3. Transform data in the column to mean of 0 and variance of 1

We can update our comments with this information:

```r
# goal: calculate mean, sd, and transform to mean = 0, var = 1
# input: one column of numeric data
# output: standarized variables
```

Now we can can start to write the function. We begin like we are defining any other variable in R, naming the function something useful but not too long. We'll call our function `standardize`:

```r
# goal: calculate mean, sd, and transform to mean = 0, var = 1
# input: one column of numeric data
# output: standarized variables
standardize <- function() {

}
```

The command `function` will create a function and anything we place between the parentheses will be the input to the function. Looking back to the comments we wrote, our function will take one column of data, so we can add one argument to the call to `function`:

```r
# goal: calculate mean, sd, and transform to mean = 0, var = 1
# input: one column of numeric data
# output: standarized variables
standardize <- function(data) {

}
```

Our function does not do anything yet. Looking at the `goal` comment, we see we need to perform three calculations:

```r
# goal: calculate mean, sd, and transform to mean = 0, var = 1
# input: one column of numeric data
# output: standarized variables
standardize <- function(data) {
  mean.value <- mean(data)
  sd.value <- sd(data)
  z.value <- (data - mean.value)/sd.value
}
```

And finally, we need to provide output. This is the very last part of our function definition and we use the `return` function to indicate what the output of our function is. Note that any code in your function that appears *after* the call to `return` will not be executed.

```r
# goal: calculate mean, sd, and transform to mean = 0, var = 1
# input: one column of numeric data
# output: standarized variables
standardize <- function(data) {
  mean.value <- mean(data)
  sd.value <- sd(data)
  z.value <- (data - mean.value)/sd.value
  return(z.value)
}
```

Once we have our function written, we need to make it available for use by executing the code (via Ctrl-Enter or Cmd-Enter). If you're using RStudio, you can see your new function in the Environment tab. Now we can perform the calculations using the function we just defined.

Now we are ready to use the function to perform the calculations. We can replace the three lines of code for the petal length calculation with just one, passing in that one column of data for the `data` argument.

```
standard.petal.length <- standardize(data = iris$Petal.Length)
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 we can add the transformation for sepal length, too:

```
standard.sepal.length <- standardize(data = iris$Sepal.Length)
summary(standard.sepal.length)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -1.86378 -0.89767 -0.05233  0.00000  0.67225  2.48370
```

Documentation is key

The last thing we need to do is to document our function. That is, we need to write comments so that anyone who wants to use the function can do so without reading all the code to figure out what the input should be, what the output is going to be, and what actually happens in the function. Because we already wrote out some comments, we can use those as a starting point.

```
# goal: calculate mean, sd, and transform to mean = 0, var = 1
# input: one column of numeric data
# output: standarized variables
```

First, we replace "input" with the name of each argument in the function and what it should be. We only have one argument, `data`, and it is a column of numeric data, so our comments just need to replace the word "input" with "data"

```
# goal: calculate mean, sd, and transform to mean = 0, var = 1
# data: one column of numeric data
# output: standarized variables
```

Next, we focus on the output. In the R language, we generally refer to this as the data that the function *returns* (remember that we used the `return` command at the end of our function). In this case, we want to replace "output" with "returns" and add a little more detail about the data that are produced by this function.

```
# goal: calculate mean, sd, and transform to mean = 0, var = 1
# data: one column of numeric data
# returns: vector of numeric values, transformed to have mean of zero and
#          variance of 1
```

And lastly we replace the "goal" with a brief, declarative statement of the function.

```r
# Standardize a vector of numerical data
# data: one column of numeric data
# returns: vector of numeric values, transformed to have mean of zero and
#          variance of 1
standardize <- function(data) {
  mean.value <- mean(data)
  sd.value <- sd(data)
  z.value <- (data - mean.value)/sd.value
  return(z.value)
}
```

TO HERE

Use apply to do it to multiple columns

```r
iris.stand <- apply(X = iris, MARGIN = 2, FUN = standardize)
```

```r
iris.stand <- apply(X = iris[, 1:4], MARGIN = 2, FUN = standardize)
iris.stand <- as.data.frame(iris.stand)
```

Could also instead use `is.numeric` to *only* tranform numeric columns.

```r
iris.stand <- iris
numeric.cols <- which(sapply(iris, is.numeric))
iris.stand[, numeric.cols] <- apply(X = iris[, numeric.cols],
                                    MARGIN = 2,
                                    FUN = standardize)
```

Add the species column

```r
iris.stand$Species <- iris$Species
```

Write a function to do it all (but do this in a separate file - move our other function there, too)

```r
# data: data frame to standardize
# ignore: (optional) vector of columns to skip in standardization process
# returns: data frame with all non-ignored columns standardized to have mean = 0, sd = 1
standardize.df <- function(data, ignore = c()) {
  ignored.cols <- data[, ignore]
  standardized.df <- apply(X = data[, -ignore], MARGIN = 2, FUN = standardize)
  standardized.df <- cbind(standardized.df, ignored.cols)
  return(standardized.df)
}

iris.stand <- standardize.df(data = iris, ignore = 5)
```

```r
# data: data frame to standardize
# ignore: (optional) vector of columns to skip in standardization process
# returns: data frame with all non-ignored columns standardized to have mean = 0, sd = 1
standardize.df <- function(data, ignore = c()) {
  ignored.cols <- as.data.frame(data[, ignore])
  colnames(ignored.cols) <- colnames(data)[ignore]
  standardized.df <- apply(X = data[, -ignore], MARGIN = 2, FUN = standardize)
  standardized.df <- cbind(standardized.df, ignored.cols)
  return(standardized.df)
}

iris.stand <- standardize.df(data = iris, ignore = 5)
```

Old Lesson

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 `airquality` data set:

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

Let's start by writing a script to run linear regression to see if the solar radiation (the `Solar.R` column) predicts the ozone level (the `Ozone` column). Create a new file called "airquality-regression.R":

```r
# Analyze air quality data
# Jeffrey Oliver
# jcoliver@email.arizona.edu
# 2017-06-22

# Relationship between ozone and solar radiation
simple <- lm(airquality[, "Ozone"] ~ airquality[, "Solar.R"])
```

Now we are interested in just the correlation coefficient (r^2) and the p-value for this relationship, so we use `summary` and extract those values:

```r
# Extract the model parameters
simple.summary <- summary(simple)
simple.r2 <- simple.summary$r.squared
simple.p <- simple.summary$coefficients[2, 4]
```

And finally we can print out these two values with the `cat` command:

```r
cat("Solar r^2 =", simple.r2)
cat("Solar p =", simple.p)
```

```
## Solar r^2 = 0.1213419Solar p = 0.0001793109
```

If we want to do the same thing for the relationship between ozone and wind, we can copy/paste this code and just change the predictor column specification in the `lm` command from `"Solar.R"` to `"Wind"` and update the message in the `cat` commands:

```r
simple <- lm(airquality[, "Ozone"] ~ airquality[, "Wind"])
simple.summary <- summary(simple)
simple.r2 <- simple.summary$r.squared
simple.p <- simple.summary$coefficients[2, 4]
cat("Wind r^2 =", simple.r2)
cat("Wind p =", simple.p)
```

This isn't too great an effort, but as the number of predictors grows, the time and opportunty for mistakes also grows. Since we are doing the same exact process for each predictor variable (run `lm`, run `summary`, extract values of interest, then print values of interest), we can encapsulate this in a function.

Behold, a function!

To define a function, create a new file called "regression-functions.R". While functions do not necessarily have to exist outside of the R script in which they are called, it is generally considered good practice. Start this file with the usual header containing information about the contents:

```r
# Functions to automate linear regression
# Jeff Oliver
# jcoliver@email.arizona.edu
# 2017-06-15
```

And we define a function pretty much the same way we assign values to a variable, but here we use the `function` function:

```r
RegressSimple <- function() {
}
```

We have a general idea of what the function should do, so write that as a brief comment above the function:

```r
# Run linear regression for all predictors and a response variable in a data frame
RegressSimple <- function() {
}
```

Now what do we do? How does one actually go about writing functions? Maybe you know exactly what your function should do and all the inputs and outputs. I rarely find myself in that case. Rather, I start by doing the same thing as I did before: I copy paste the portion of code I want to run. So in this case, from the airquality-regression.R file, copy the section for the solar radiation analysis and paste it into the body of the `RegressSimple` function. That is, paste it between the pair of curly braces `{ }`:

```r
# Run linear regression for all predictors and a response variable in a data frame
RegressSimple <- function() {
  simple <- lm(airquality[, "Ozone"] ~ airquality[, "Solar.R"])
  simple.summary <- summary(simple)
  simple.r2 <- simple.summary$r.squared
  simple.p <- simple.summary$coefficients[2, 4]
  cat("Solar r^2 =", simple.r2)
  cat("Solar p =", simple.p)
}
```

We don't want the function to print out values from the models, so delete the code that extracts the values and prints them with the two `cat` statements:

```r
# Run linear regression for all predictors and a response variable in a data frame
RegressSimple <- function() {
  simple <- lm(airquality[, "Ozone"] ~ airquality[, "Solar.R"])
  simple.summary <- summary(simple)
}
```

At this point, we can consider input and output of the function. We'll need to give the function two pieces of input: the data to work with and which variable to use as the response. We specify input by declaring the names of the values (`data` and `response`) in the `function` call:

```r
# Run linear regression for all predictors and a response variable in a data frame
RegressSimple <- function(data, response) {
  simple <- lm(airquality[, "Ozone"] ~ airquality[, "Solar.R"])
  simple.summary <- summary(simple)
}
```

And while we're at it, we need to document these inputs:

```r
# Run linear regression for all predictors and a response variable in a data frame
# data: the data frame to analyze
# response: the name of the response variable
RegressSimple <- function(data, response) {
  simple <- lm(airquality[, "Ozone"] ~ airquality[, "Solar.R"])
  simple.summary <- summary(simple)
}
```

Now that we have inputs, we can update the variables in the code we copied from the regression script. What do we need to update?

- The name of the data frame we used, `airquality` is replaced by `data`
- The model specification now uses the abstracted response variable, stored in `response` instead of the hard coded `"Ozone"`; note that we *do not* put `response` in double quotes

```r
# Run linear regression for all predictors and a response variable in a data frame
# data: the data frame to analyze
# response: the name of the response variable
RegressSimple <- function(data, response) {
  simple <- lm(data[, response] ~ data[, "Solar.R"])
  simple.summary <- summary(simple)
}
```

The goal of the function is to run linear regression for *all* the predictors in the data frame, so how can we do this? The simplest way is to use a `for` loop for all the columns in the data frame, updating the `lm` call with the column specification of the predictor variable:

```r
# Run linear regression for all predictors and a response variable in a data frame
# data: the data frame to analyze
# response: the name of the response variable
RegressSimple <- function(data, response) {
  for (predictor in 1:ncol(data)) {
    simple <- lm(data[, response] ~ data[, predictor])
    simple.summary <- summary(simple)
  }
}
```

For output, we probably want the results of the linear model for each of our predictors. In this case, we'll use a `list` object and assigning the output of `summary` to an element in that list. Note because we are using a list, we use two-bracket notation `[[ ]]` to indicate an element in the list. And finally, we sent back these results with the `return` function:

```r
# Run linear regression for all predictors and a response variable in a data frame
# data: the data frame to analyze
# response: the name of the response variable
RegressSimple <- function(data, response) {
  model.summaries <- list()
  for (predictor in 1:ncol(data)) {
    simple <- lm(data[, response] ~ data[, predictor])
    element.name <- colnames(data)[predictor]
    model.summaries[[element.name]] <- summary(simple)
  }
  return(model.summaries)
}
```

Before we try this out, update the documentation with a description of the output

```r
# Run linear regression for all predictors and a response variable in a data frame
# data: the data frame to analyze
# response: the name of the response variable
# returns: a list where each element is the output from summary(lm)
RegressSimple <- function(data, response) {
  model.summaries <- list()
  for (predictor in 1:ncol(data)) {
    simple <- lm(data[, response] ~ data[, predictor])
    element.name <- colnames(data)[predictor]
    model.summaries[[element.name]] <- summary(simple)
  }
  return(model.summaries)
}
```

Using our function

So how do we use this? We need to load this function into memory so we can use it. Go back to the script with our original regression analyses, "airquality-regression.R". Comment out our previous linear regression code and add a call to `source` to load our function file. Hint: in RStudio, you can select multiple lines and comment them out with the shortcut Shift-Ctrl-C.

```r
# Analyze air quality data
# Jeffrey Oliver
# jcoliver@email.arizona.edu
# 2017-06-22

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

```
# Relationship between ozone and solar radiation
# simple <- lm(airquality[, "Ozone"] ~ airquality[, "Solar.R"])

# Extract the model parameters
# simple.summary <- summary(simple)
# simple.r2 <- simple.summary$r.squared
# simple.p <- simple.summary$coefficients[2, 4]
# cat("Solar r^2 =", simple.r2)
# cat("Solar p =", simple.p)
```

And we can now use this function, passing `airquality` as the `data` and `"Ozone"` as the `response`:

```
airquality.models <- RegressSimple(data = airquality, response = "Ozone")
```

```
## Warning in summary.lm(simple): essentially perfect fit: summary may be
## unreliable
```

Hmmmm…that's an odd warning. Let's add some reporting code to see if we can figure out what it's doing. Update the `RegressSimple` function to print the name of the predictor using the `cat` function:

```
# Run linear regression for all predictors and a response variable in a data frame
# data: the data frame to analyze
# response: the name of the response variable
# returns: a list where each element is the output from summary(lm)
RegressSimple <- function(data, response) {
  model.summaries <- list()
  for (predictor in 1:ncol(data)) {
    simple <- lm(data[, response] ~ data[, predictor])
    element.name <- colnames(data)[predictor]
    model.summaries[[element.name]] <- summary(simple)
    cat("Predictor:", element.name, "\n")
  }
  return(model.summaries)
}
```

Now go back our airquality-regression.R script and run the `RegressSimple` command again.

```
airquality.models <- RegressSimple(data = airquality, response = "Ozone")
```

```
## Warning in summary.lm(simple): essentially perfect fit: summary may be
## unreliable
```

Drat. Same warning, but no message printed? Why not? Because we made changes to the regression-functions.R file, but did not load them into memory with the `source` command, R is using the old version of the `RegressSimple` function. So we need to run the `source` command first:

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

Then the `RegressSimple` command:

```
airquality.models <- RegressSimple(data = airquality, response = "Ozone")
```

```
## Warning in summary.lm(simple): essentially perfect fit: summary may be
## unreliable
```

```
## Predictor: Ozone
## Predictor: Solar.R
## Predictor: Wind
## Predictor: Temp
## Predictor: Month
## Predictor: Day
```

OK, so there are some problems. First, we aren't really interested in the effect of `Day` or `Month` on ozone levels, so when we call `RegressSimple`, we should only pass it data we want to analyze. Here we drop the fifth and sixth columns, which are the Month and Day columns, respectively:

```r
airquality.models <- RegressSimple(data = airquality[, -c(5:6)], response = "Ozone")
```

But look at the output from function call again. Our function actually ran regression on a model using ozone to predict ozone - that's probably what caused the warning message "summary may be unreliable". But we don't want to exclude ozone from the data we pass to the function, because that is the response variable. We therefore need to update our function definition so we don't run an `ozone ~ ozone` model. More generally, we need to make sure our response variable is not treated as a predictor. To do this, we:

1. Find out which column is the `response` variable
2. Create a vector of predictor variable names using `colnames`
3. Update our `for` loop to only use those predictor variables
4. Use the predictor variable name for the `model.summaries` list element name

Open the regression-functions.R file and update `RegressSimple`:

```r
# Run linear regression for all predictors and a response variable in a data frame
# data: the data frame to analyze
# response: the name of the response variable
# returns: a list where each element is the output from summary(lm)
RegressSimple <- function(data, response) {
  response.index <- which(colnames(data) == response)
  predictors <- colnames(data)[-response.index]
  model.summaries <- list()
  for (predictor in predictors) {
    simple <- lm(data[, response] ~ data[, predictor])
    model.summaries[[predictor]] <- summary(simple)
    cat("Predictor:", predictor, "\n")
  }
  return(model.summaries)
}
```

Now when we run `RegressSimple`, there are only three predictors used in the models, as we expect:

```r
source(file = "regression-functions.R")
airquality.models <- RegressSimple(data = airquality[, -c(5:6)], response = "Ozone")
```

```
## Predictor: Solar.R
## Predictor: Wind
## Predictor: Temp
```

Since it `RegressSimple` is now only running the models we want, remove the `cat` command from the function:

```r
# Run linear regression for all predictors and a response variable in a data frame
# data: the data frame to analyze
# response: the name of the response variable
# returns: a list where each element is the output from summary(lm)
RegressSimple <- function(data, response) {
  response.index <- which(colnames(data) == response)
  predictors <- colnames(data)[-response.index]
  model.summaries <- list()
  for (predictor in predictors) {
    simple <- lm(data[, response] ~ data[, predictor])
    model.summaries[[predictor]] <- summary(simple)
  }
  return(model.summaries)
}
```

Because `airquality.models` is a list, we can access the objects using double-bracket notation:

```
solar.model <- airquality.models[["Solar.R"]]
solar.corr <- solar.model$r.squared
solar.p <- solar.model$coefficients[2, 4]
cat("Solar r^2 = ", solar.corr, "\n")
cat("Solar p = ", solar.p, "\n")
```

```
## Solar r^2 =  0.1213419
## Solar p =  0.0001793109
```

Make it Class-y

But now we're back to the copy-paste-update cycle if we want to get all the correlation coefficients and p-values. If we know that's all we want, we can create another function, one that does the work of heavy lifting of extracting coefficients and printing them out to the screen.

In the file with our `RegressSimple` function, create *another* function, and call it `print.RegressSimple`:

```
print.RegressSimple <- function(x, ...) {
}
```

Briefly, what we are doing is creating a function that specifies the output of anything that is `class RegressSimple` (we'll get to how we make that happen in a moment). The thing to note now is that in a `print.____` function, the first argument, `x` is the object we wish to print; in this case, a product of the function `RegressSimple`. Stick with me here, it will become clearer. For the purposes of this lesson, just remember that in the `print.RegressSimple` function, the variable `x` is the list that is produced from a call to `RegressSimple`. We want this function to extract the r^2 and p-values for each model and display a table of those values for each predictor variable. Something like:

```
##      variable    r2     p
##      Solar.R     0.12   1.7e-04
##      Wind        0.36   9.2e-13
##      Temp        0.48   2.9e-18
```

Add a brief explanation of what this function does, then add code to get the names of the variables and set up a data frame to hold the values we want to print:

```
# Print values of interest from each predictor in a RegressSimple object
print.RegressSimple <- function(x, ...) {
  # Get a vector of the elements' names
  predictors <- names(x)
  # Set up a dataframe for results of interest
  model.results <- data.frame(variable = predictors,
                              r2 = 0,
                              p = 0)
}
```

Now extract the values we want to print for each element in the `RegressSimple` object (which is the variable `x` in `print.RegressSimple`):

```r
# Print values of interest from each predictor in a RegressSimple object
print.RegressSimple <- function(x, ...) {
  # Get a vector of the elements' names
  predictors <- names(x)
  # Set up a dataframe for results of interest
  model.results <- data.frame(variable = predictors,
                              r2 = 0,
                              p = 0)
  # Extract r-squared and p-values
  model.results$r2 <- sapply(x, "[[", "r.squared")
  model.coeffs <- sapply(x, "[[", "coefficients")
  model.results$p <- model.coeffs[8, ]
}
```

Finally, we add `cat` and `print` statements to output the values.

```r
# Print values of interest from each predictor in a RegressSimple object
print.RegressSimple <- function(x, ...) {
  # Get a vector of the elements' names
  predictors <- names(x)
  # Set up a dataframe for results of interest
  model.results <- data.frame(variable = predictors,
                              r2 = 0,
                              p = 0)
  # Extract r-squared and p-values
  model.results$r2 <- sapply(x, "[[", "r.squared")
  model.coeffs <- sapply(x, "[[", "coefficients")
  model.results$p <- model.coeffs[8, ]

  # Print values
  cat("Regression results: ", "\n")
  print(as.matrix(model.results), quote = FALSE)
}
```

Now when we run our code, we can just enter the name of the variable to have it print out our nice table of just the r^2 and p-values.

```r
source(file = "regression-functions.R")
airquality.models <- RegressSimple(data = airquality[, -c(5:6)], response = "Ozone")
airquality.models
```

```
## $Solar.R
##
## Call:
## lm(formula = data[, response] ~ data[, predictor])
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -48.292 -21.361  -8.864  16.373 119.136
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)       18.59873    6.74790   2.756 0.006856 **
## data[, predictor]  0.12717    0.03278   3.880 0.000179 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 31.33 on 109 degrees of freedom
##   (42 observations deleted due to missingness)
## Multiple R-squared:  0.1213, Adjusted R-squared:  0.1133
## F-statistic: 15.05 on 1 and 109 DF,  p-value: 0.0001793
##
##
## $Wind
##
## Call:
## lm(formula = data[, response] ~ data[, predictor])
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -51.572 -18.854  -4.868  15.234  90.000
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)        96.8729     7.2387   13.38  < 2e-16 ***
## data[, predictor]  -5.5509     0.6904   -8.04 9.27e-13 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 26.47 on 114 degrees of freedom
##   (37 observations deleted due to missingness)
## Multiple R-squared:  0.3619, Adjusted R-squared:  0.3563
## F-statistic: 64.64 on 1 and 114 DF,  p-value: 9.272e-13
##
##
## $Temp
##
## Call:
## lm(formula = data[, response] ~ data[, predictor])
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -40.729 -17.409  -0.587  11.306 118.271
##
## Coefficients:
##                    Estimate Std. Error t value Pr(>|t|)
## (Intercept)       -146.9955    18.2872  -8.038 9.37e-13 ***
## data[, predictor]    2.4287     0.2331  10.418  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 23.71 on 114 degrees of freedom
##   (37 observations deleted due to missingness)
## Multiple R-squared:  0.4877, Adjusted R-squared:  0.4832
## F-statistic: 108.5 on 1 and 114 DF,  p-value: < 2.2e-16
```

Well that didn't work. It just printed out the each element of the list. That's because we need to take one final step. The `print.RegressSimple` only works on objects that are of `class RegressSimple`. So what is the class of our `airquality.models`?

```
class(airquality.models)
```

```
## [1] "list"
```

It's a list, *not* a `RegressSimple` object. So how do we instruct R to make the output of `RegressSimple` to be an object of `class RegressSimple`? Big surprise here, we use the `class` function! In our `RegressSimple` function, right before we return the `model.summaries` list, we set the class to be "RegressSimple":

```
# Run linear regression for all predictors and a response variable in a data frame
# data: the data frame to analyze
# response: the name of the response variable
# returns: a list where each element is the output from summary(lm)
RegressSimple <- function(data, response) {
  response.index <- which(colnames(data) == response)
  predictors <- colnames(data)[-response.index]
  model.summaries <- list()
  for (predictor in predictors) {
    simple <- lm(data[, response] ~ data[, predictor])
    model.summaries[[predictor]] <- summary(simple)
  }

  # Set class and return results
  class(model.summaries) <- "RegressSimple"
  return(model.summaries)
}
```

Now re-run the lines in our regression script:

```
source(file = "regression-functions.R")
airquality.models <- RegressSimple(data = airquality[, -c(5:6)], response = "Ozone")
airquality.models
```

```
## Regression results:
##      variable r2        p
## [1,] Solar.R  0.1213419 1.793109e-04
## [2,] Wind     0.3618582 9.271974e-13
## [3,] Temp     0.4877072 2.931897e-18
```

There it is, our table of values! Our final two files will then be:

airquality-regression.R:

```
# Analyze air quality data
# Jeffrey Oliver
# jcoliver@email.arizona.edu
# 2017-06-22

source(file = "regression-functions.R")
airquality.models <- RegressSimple(data = airquality[, -c(5:6)], response = "Ozone")
airquality.models
```

regression-functions.R:

```
# Functions to automate linear regression
# Jeff Oliver
# jcoliver@email.arizona.edu
# 2017-06-15
```

```r
################################################################################
# Run linear regression for all predictors and a response variable in a data frame
# data: the data frame to analyze
# response: the name of the response variable
# returns: a list where each element is the output from summary(lm)
RegressSimple <- function(data, response) {
  response.index <- which(colnames(data) == response)
  predictors <- colnames(data)[-response.index]
  model.summaries <- list()
  for (predictor in predictors) {
    simple <- lm(data[, response] ~ data[, predictor])
    model.summaries[[predictor]] <- summary(simple)
  }

  # Set class and return results
  class(model.summaries) <- "RegressSimple"
  return(model.summaries)
}

################################################################################
# Print values of interest from each predictor in a RegressSimple object
print.RegressSimple <- function(x, ...) {
  # Get a vector of the elements' names
  predictors <- names(x)
  # Set up a dataframe for results of interest
  model.results <- data.frame(variable = predictors,
                              r2 = 0,
                              p = 0)
  # Extract r-squared and p-values
  model.results$r2 <- sapply(x, "[[", "r.squared")
  model.coeffs <- sapply(x, "[[", "coefficients")
  model.results$p <- model.coeffs[8, ]

  # Print values
  cat("Regression results: ", "\n")
  print(as.matrix(model.results), quote = FALSE)
}
```

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.