Functions in R

HTML Version: https://stirlingcodingclub.github.io/Functions_in_R

Brad Duthie

27 November 2024

Contents

After reading through this, you should have a working understanding of what functions are in R, how to use them, and how to write your own. Parts of these notes were inspired by what Thiago Silva taught in 2021.

Installing packages in R

One of the most useful attributes of R is the ability to make and use packages. Packages allow custom code, data, and documentation to be published and used by anyone. Up until now, we have been looking at functions (e.g., `hist` or `plot`) that are available in base R (i.e., they are part of the R language, and anyone who installs R can use these functions). But by downloading and installing a package, it is possible to use functions written by anyone in the R community. As an example, we can download the swirl package (https://swirlstats.com/). The swirl package is a useful package for learning R step-by-step. You can get started by running the code below.

```r
install.packages("swirl");
library("swirl");
swirl();
```

```
##
## | Hi! Type swirl() when you are ready to begin.
```

The `install.packages` command is all that you need to install a package directly from the Comprehensive R Archive Network (CRAN; https://cran.r-project.org/). After R installs the package, the `library` function is used to deploy the package functions into the console. There are 18000+ packages in R, all free to download and use. If you use one of these packages, you can cite it with the `citation()` function in base R.

```r
citation("swirl");
```

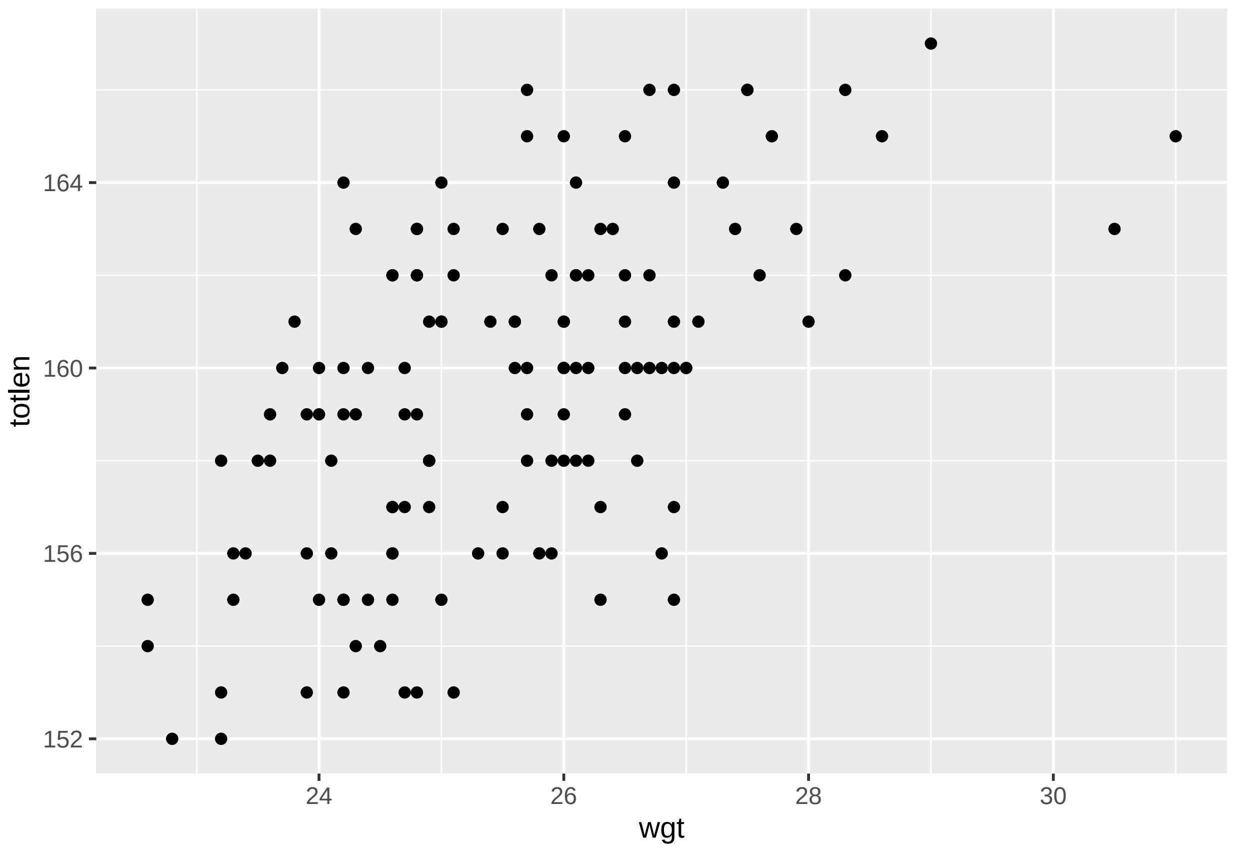
```
## To cite package 'swirl' in publications use:
##
##   Kross S, Carchedi N, Bauer B, Grdina G (2020). _swirl: Learn R, in
##   R_. R package version 2.4.5,
##   <https://CRAN.R-project.org/package=swirl>.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {swirl: Learn R, in R},
##     author = {Sean Kross and Nick Carchedi and Bill Bauer and Gina Grdina},
##     year = {2020},
##     note = {R package version 2.4.5},
##     url = {https://CRAN.R-project.org/package=swirl},
##   }
```

As one more example, a popular package is `ggplot2`, which some people prefer to use in plotting instead of the available base R functions. We can return to the Bumpus dataset from the sessions on Getting Started and Coding Types.

```r
dat <- read.csv("Bumpus_data.csv"); # Read in the data
```

In the session on Getting Started, we mad a scatterplot to show the relationship between sparrow body mass (`wgt`) and total length (`totlen`) using the `plot` function. If we wanted to plot sparrow weight versus total length using a slightly different plotting style from `plot`, then we could download `ggplot2` and run the `ggplot` function.

```r
install.packages("ggplot2");
library("ggplot2");
ggplot(data = dat, mapping = aes(x = wgt, y = totlen)) +
    geom_point();
```



There are ways to make the plot above look much nicer, but I am not an expert in `ggplot`. For your own data, I recommend exploring and trying new tools and new code. Most of what you try will not work out they way you want it to, but you will almost always learn something new and get practice.

Custom functions in R

Functions can be written in R, then used in ways that make data wrangling, analysis, and all kinds of other tasks much more easy and efficient. Being able to write your own functions is a highly useful skill. For example, if you collect data in a particular way more than once, and need to reorganise those data to a different format than you collected for statistical analysis, then it might be useful to write a function that automates this so that you can simply run the function on your collected dataset. To write a function, you use the `function` function, which specifies the function name, the arguments that your custom function will take (along with any defaults), what the function does, and what the function returns. Below is a very simple example that converts a Fahrenheit value to Celsius.

```r
F_to_C <- function(F_temp = 70){
    C_temp <- (F_temp - 32) * 5/9;
    return(C_temp);
}
```

The function is assigned to the name `F_to_C`. The function takes one argument `F_temp`, which has a default value of 70. It then calculates `(F_temp - 32) * 5/9` and assigns it to the variable `C_temp`. It then returns `C_temp`. To input this function into R, we just need to highlight the whole function and run it. After it is run, then we can then use `F_to_C` just as we would any other function. This is demonstrate below with a Fahrenheit temperature of 100.

```r
F_to_C(F_temp = 100);
```

```
## [1] 37.77778
```

Note that we do not need to actually write out `F_temp` as an argument. R recognises the order of arguments, or in this case, that there is only one argument in the `F_To_C` function, so it must be `F_temp`. We can therefore do the same thing as above without specifying the argument.

```
F_to_C(100);
```

```
## [1] 37.77778
```

Also note that because we gave the argument `F_temp` a default value of 70, if we do not specify any argument, then it runs the function with this default `F_temp = 70`.

```
F_to_C();
```

```
## [1] 21.11111
```

If we realy want to, we can also set the function without `return`. In this case, R simply returns the last variable assigned.

```
F_to_C <- function(F_temp = 70){
    C_temp <- (F_temp - 32) * 5/9;
}
```

The above leaves out the return argument, but it still returns `C_temp` because this is the last variable assigned.

```
converted_temp <- F_to_C(100);
print(converted_temp);
```

```
## [1] 37.77778
```

It is general best to use `return` to specify what a function should return.

Functions in functions

We can also create custom functions that use custom functions. For example, assume that we wanted to convert from temperature Fahrenheit to temperature Kelvin. Temperature Kelvin is simply Celsius plus 273.15, $K = C + 273.15$. We could convert from Fahrenheit to Kelvin by calculating `((F_temp - 32) * 5/9) + 273.15`, or we could make use of the `F_to_C` function within a new function `F_to_K`.

```
F_to_K <- function(F_temp){
  K_temp <- F_to_C(F_temp = F_temp) + 273.15;
  return(K_temp);
}
```

What is going on above? We are defining the new function `F_to_K`, which takes an argument `F_temp` and converts it to Kelvin. It does this by passing this `F_temp` into the previously written function `F_to_C`. It calculates the output of `F_to_C(F_temp)` and then adds 273.15 and assigns the value to `K_temp`. Then it returns the calculated `K_temp`.

```
F_to_K(F_temp = 100);
```

```
## [1] 310.9278
```

Note that I have not given `F_temp` a default value in the `F_to_K` function above, so we cannot just call it without defining a value. If we do, then we get an error message.

```
F_to_K();
```

```
## Error in F_to_K(): argument "F_temp" is missing, with no default
```

We can try something else. Maybe we want to write a function that converts degrees Fahrenheit to *either* Celsius or Kelvin. We can do this with a new function that includes two arguments, one specifying the degrees Fahrenheit, and the second specifying whether the function should convert to Celsius or Fahrenheit.

```
F_convert <- function(F_temp = 70, conversion = "Celsius"){
  if(conversion == "Celsius"){
    converted <- F_to_C(F_temp = F_temp);
  }
  if(conversion == "Kelvin"){
    converted <- F_to_K(F_temp = F_temp);
  }
  return(converted);
}
```

The below will get the job done, provided we use it correctly.

```
F_convert(F_temp = 70, conversion = "Kelvin");
```

```
## [1] 294.2611
```

But the function could be written better. If, for example, we spell "Kelvin" incorrectly, or even forget to capitalise it, then we will get an uninformative error.

```
F_convert(F_temp = 70, conversion = "kelvin");
```

```
## Error in F_convert(F_temp = 70, conversion = "kelvin"): object 'converted' not found
```

To improve the function, we can redefine it to make sure it gives an informative error message. We can check to make sure that `conversion` is either "Celsius" or "Kelvin", and that "F_temp" is a numeric input. If this is not the case, then the function should `stop` with an error message.

```
F_convert <- function(F_temp = 70, conversion = "Celsius"){
  if(conversion != "Celsius" & conversion != "Kelvin"){
    stop("conversion argument must be 'Celsius' or 'Kelvin'.")
  }
  if(is.numeric(F_temp) == FALSE){
    stop("F_temp argument must be numeric");
  }
  if(conversion == "Celsius"){
    converted <- F_to_C(F_temp = F_temp);
  }else{
    converted <- F_to_K(F_temp = F_temp);
  }
  return(converted);
}
```

Note that the first `if` statement in the function above checks to see if the argument `conversion` does *not* equal "Celsius" (`conversion != "Celsius"`) **and** (`&`) does *not* equal "Kelvin" (`conversion != "Kelvin"`). If this is `TRUE`, then `conversion` must be incorrectly specified because it does not equal one of the two permissible options. We therefore need to return an error message. Note that since we have established that `conversion` must be either "Celsius" or "Kelvin" (returning an error message if not), then we can just use the `else` at the end of the `if(conversion == "Celsius")` statement. We can try the incorrect specification of `conversion = "kelvin"` again.

```
F_convert(F_temp = 70, conversion = "kelvin");
```

```
## Error in F_convert(F_temp = 70, conversion = "kelvin"): conversion argument must be 'Celsius' or 'Ke
```

Similarly, if we specify a value of `F_temp` that is not numeric, then `is.numeric(F_temp)` will be `FALSE`, causing the function to `stop` with a relevant error message.

```
F_convert(F_temp = "seventy", conversion = "Kelvin");
```

```
## Error in F_convert(F_temp = "seventy", conversion = "Kelvin"): F_temp argument must be numeric
```

We can run the function correctly now.

```
F_convert(F_temp = 70, conversion = "Kelvin");
```

```
## [1] 294.2611
```

Note that we could also assign the output of the above to a new variable.

```
new_variable <- F_convert(F_temp = 70, conversion = "Kelvin");
print(new_variable);
```

```
## [1] 294.2611
```

It is often useful to write error messages like this into functions even if you know that you will be the only person who uses them. Next, try to write a function that converts from Celsius to either Fahrenheit or Kelvin. The 'Details' pulldown below gives one potential way to do it.

```
C_convert <- function(C_temp, conversion = "Kelvin"){
  if(conversion != "Fahrenheit" & conversion != "Kelvin"){
    stop("conversion argument must be 'Fahrenheit' or 'Kelvin'.")
  }
  if(is.numeric(C_temp) == FALSE){
    stop("C_temp argument must be numeric");
  }
  if(conversion == "Celsius"){
    converted <- C_temp + 273.15;
  }else{
    converted <- (C_temp * 9/5) + 32;
  }
  return(converted);
}
```

Try creating some other functions that might be useful for your own purposes, or that do something fun.

Recursive functions

There is one trick that we can do with functions called *recursion*. Recursion occurs when a function calls *itself* until some kind of stopping condition is met. This is almost never necessary to use once you know how to use a loop (which we will learn in the next session). Since learning to code, there has been exactly one time I can recall in which I basically **needed** to use recursive programming, and this was programming in C, not R. Nevertheless, it is a bit of a cool trick to know.

Suppose we wanted to create a function that samples from a range of values `interval` iteratively until it finds the value that makes it stop (`stop_number`). Further suppose that we want to record all of the values tried until sampling the `stop_number`. There are many ways that we could do this in R, but the program below shows how to do it with recursive programming.

```
recursive_function <- function(stop_number, interval = 1:10, rand_tries = NULL){
  rand_number <- sample(x = interval, size = 1);
  rand_tries  <- c(rand_tries, rand_number);
  if(rand_number == stop_number){
    return(rand_tries);
  }else{
    new_try    <- recursive_function(stop_number = stop_number,
                                     interval    = interval,
                                     rand_tries  = rand_tries);
  }
}
```

Verbally, the program first randomly samples a number `rand_number` (by default the random number is from 1 to 10). The number is added to a growing list called `rand_tries`, which starts out as `NULL` with the first call to the function. If the `rand_number` equals the `stop_number`, then the function returns the list `rand_tries`. But if it does not equal `stop_number`, then the function runs again, so a new `rand_number` is selected and added to `rand_tries`. This continues until the `rand_number == stop_number` condition is met, at which point the function returns the list `rand_tries`. This is a bit confusing and counter-intuitive at first, and, as already mentioned, there almost never any need for it once you know how to use a loop. Nevertheless, we can run the function to see the output.

```
new_rec1 <- recursive_function(stop_number = 4, interval = 1:10);
return(new_rec1);
```

```
## [1] 1 9 5 4
```

We can try it again.

```
new_rec2 <- recursive_function(stop_number = 4, interval = 1:10);
return(new_rec2);
```

```
## [1] 5 4
```

One more time.

```
new_rec3 <- recursive_function(stop_number = 4, interval = 1:10);
return(new_rec3);
```

```
##  [1]  2  3  9  5  9  9  6 10  6  9  8  7 10  4
```

Note that we get a different answer each time, with a vector of varying lengths.

Important considerations

There are a few additional things to keep in mind when writing functions (note, these are useful points from Thiago that I would have probably otherwise forgotten to include).

Function environment

The code run within an argument executes within its own local environment. What this means is that any objects assigned in a function cannot be used unless they are returned by the function. Here is a quick example.

```
sum_three_vals <- function(starting_val){
  aa <- starting_val + 0;
  bb <- starting_val + 2;
  cc <- starting_val + 4;
  dd <- aa + bb + cc;
  return(dd);
}
```

The function `sum_three_vals` defined above takes one argument `starting_val` and uses it to assign a value to `aa`, `bb`, and `cc` in the function. It then sums `aa`, `bb`, and `cc` and returns the summed value.

```
sum_three_vals(starting_val = 1);
```

```
## [1] 9
```

We get the answer of 9 as expected, but note that the 'global' environment outside of the function does not retain `aa`, `bb`, or `cc`. If we try to print one of these values, R cannot find it.

```
print(aa);
```

```
## Error in eval(expr, envir, enclos): object 'aa' not found
```

This is not true in the other direction, going from the global to the local. For example, say that we first define `xx`, `yy`, and `zz`, then a function summing them up.

```
xx         <- 1;
yy         <- 2;
zz         <- 3;
sum_xxyyzz <- function(){
  dd <- xx + yy + zz;
  return(dd);
}
```

We can now see that the function does technically work.

```
sum_xxyyzz()
```

```
## [1] 6
```

Nevertheless, this is *really* not good practice. The `sum_xxyyzz` function is not portable. You could not actually use it unless the three `xx`, `yy`, and `zz` were already calculated, and if these values change, then the function could return unexpected values. It is important to always make functions self contained, so we can rewrite the above.

```
sum_xxyyzz <- function(xx, yy, zz){
  dd <- xx + yy + zz;
  return(dd);
}
```

Now we specify all of the arguments that the function takes, and the funciton is not reliant upon any other objects in the global environment. Note that the function no longer works by itself as before.

```
sum_xxyyzz();
```

```
## Error in sum_xxyyzz(): argument "xx" is missing, with no default
```

This is because the function is looking for `xx`, `yy`, and `zz` to be assigned as arguments, and with no defaults, it will (correctly and helpfully) produce an error message.

Order of arguments

Note that if arguments are not specified, then R assumes that any values input are defined in the order of the function arguments. We can consider a function that just adds two values, `val1` and two times `val2`.

```
add_two_eg <- function(val1, val2){
  out_val <- val1 + (2 * val2);
  return(out_val);
}
```

We can specify what we want the values to be in any order. Consider `val1 = 1` and `val2 = 2`.

```r
add_two_eg(val1 = 1, val2 = 2);
```

```
## [1] 5
```

We can write this with the order of arguments flipped and get the same answer.

```r
add_two_eg(val2 = 2, val1 = 1);
```

```
## [1] 5
```

But if we do not specify the arguments explicitly, then R just assumes that they are in the order of the original function `val1, val2`. We can see this below, where we get a different answer if we set `2` first.

```r
add_two_eg(2, 1);
```

```
## [1] 4
```

In the above, R just assumes that we meant `add_two_eg(val1 = 2, val2 = 1);`.

Function returns

Note that all of the examples that we have used return a single value. This was done for simplicity, but it is not necessary. For example, we might want to return a list of values instead.

```r
func_list <- function(name = "A name", value = 3, vector = 1:10){
  new_list <- list(name, value * 2, vector);
  return(new_list);
}
```

The above function is a very simple example. It simply returns the three arguments `name`, `value` (times 2), and `vector` as a list.

```r
func_list(name = "Grace Hopper", value = 85, vector = 1906:1992);
```

```
## [[1]]
## [1] "Grace Hopper"
##
## [[2]]
## [1] 170
##
## [[3]]
##  [1] 1906 1907 1908 1909 1910 1911 1912 1913 1914 1915 1916 1917 1918 1919 1920
## [16] 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935
## [31] 1936 1937 1938 1939 1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950
## [46] 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965
## [61] 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980
## [76] 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992
```

Do call

Lastly, we can use the function `do.call` to read arguments into a function as a list. Here is what that looks like, using the `func_list` example from above.

```r
my_args <- list("Grace Hopper", 85, vector = 1906:1992)
do.call(what = func_list, args = my_args);
```

```
## [[1]]
## [1] "Grace Hopper"
##
```

```
## [[2]]
## [1] 170
##
## [[3]]
##  [1] 1906 1907 1908 1909 1910 1911 1912 1913 1914 1915 1916 1917 1918 1919 1920
## [16] 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935
## [31] 1936 1937 1938 1939 1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950
## [46] 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965
## [61] 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980
## [76] 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992
```

The `do.call` function seems a bit unnecessary, bit it is actually quite useful in some context. I have used it when writing R packages that include functions with many arguments, and a need for flexibility given user input. But it is rarely needed, in my experience, for data analysis.