14 Functions

Writing functions is a core activity of an R programmer. It represents the key step of the transition from a mere "user" to a developer who creates new functionality for R. Functions are often used to encapsulate a sequence of expressions that need to be executed numerous times, perhaps under slightly different conditions. Functions are also often written when code must be shared with others or the public.

The writing of a function allows a developer to create an interface to the code, that is explicitly specified with a set of parameters. This interface provides an abstraction of the code to potential users. This abstraction simplifies the users' lives because it relieves them from having to know every detail of how the code operates. In addition, the creation of an interface allows the developer to communicate to the user the aspects of the code that are important or are most relevant.

14.1 Functions in R

Functions in R are "first class objects", which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions. This is very handy for the various apply functions, like `lapply()` and `sapply()`.
- Functions can be nested, so that you can define a function inside of another function

If you're familiar with common language like C, these features might appear a bit strange. However, they are really important in R and can be useful for data analysis.

14.2 Your First Function

Functions are defined using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class "function".

Here's a simple function that takes no arguments and does nothing.

```
> f <- function() {
+         ## This is an empty function
+ }
> ## Functions have their own class
> class(f)
[1] "function"
> ## Execute this function
> f()
NULL
```

Not very interesting, but it's a start. The next thing we can do is create a function that actually has a non-trivial *function body*.

```
> f <- function() {
+         cat("Hello, world!\n")
+ }
> f()
Hello, world!
```

The last aspect of a basic function is the *function arguments*. These are the options that you can specify to the user that the user may explicity set. For this basic function, we can add an argument that determines how many times "Hello, world!" is printed to the console.

```
> f <- function(num) {
+         for(i in seq_len(num)) {
+                 cat("Hello, world!\n")
+         }
+ }
> f(3)
Hello, world!
Hello, world!
Hello, world!
```

Obviously, we could have just cut-and-pasted the `cat("Hello, world!\n")` code three times to achieve the same effect, but then we wouldn't be programming, would we? Also, it would be un-neighborly of you to give your code to someone else and force them to cut-and-paste the code however many times the need to see "Hello, world!".

> In general, if you find yourself doing a lot of cutting and pasting, that's usually a good sign that you might need to write a function.

Finally, the function above doesn't *return* anything. It just prints "Hello, world!" to the console `num` number of times and then exits. But often it is useful if a function returns something that perhaps can be fed into another section of code.

This next function returns the total number of characters printed to the console.

```
> f <- function(num) {
+         hello <- "Hello, world!\n"
+         for(i in seq_len(num)) {
+                 cat(hello)
+         }
+         chars <- nchar(hello) * num
+         chars
+ }
> meaningoflife <- f(3)
Hello, world!
Hello, world!
Hello, world!
> print(meaningoflife)
[1] 42
```

14 characters technically. \n means return.

In the above function, we didn't have to indicate anything special in order for the function to return the number of characters. In R, the return value of a function is always the very last expression that is evaluated. Because the `chars` variable is the last expression that is evaluated in this function, that becomes the return value of the function.

Note that there is a `return()` function that can be used to return an explicity value from a function, but it is rarely used in R (we will discuss it a bit later in this chapter).

Finally, in the above function, the user must specify the value of the argument `num`. If it is not specified by the user, R will throw an error.

```
> f()
Error in f(): argument "num" is missing, with no default
```

We can modify this behavior by setting a *default value* for the argument `num`. Any function argument can have a default value, if you wish to specify it. Sometimes, argument values are rarely modified (except in special cases) and it makes sense to set a default value for that argument. This relieves the user from having to specify the value of that argument every single time the function is called.

Here, for example, we could set the default value for `num` to be 1, so that if the function is called without the `num` argument being explicitly specified, then it will print "Hello, world!" to the console once.

```
> f <- function(num = 1) {
+         hello <- "Hello, world!\n"
+         for(i in seq_len(num)) {
+                 cat(hello)
+         }
+         chars <- nchar(hello) * num
+         chars
+ }
> f()    ## Use default value for 'num'
Hello, world!
[1] 14
> f(2)   ## Use user-specified value
Hello, world!
Hello, world!
[1] 28
```

Remember that the function still returns the number of characters printed to the console.

At this point, we have written a function that

- has one *formal argument* named `num` with a *default value* of 1. The *formal arguments* are the arguments included in the function definition. The `formals()` function returns a list of all the formal arguments of a function
- prints the message "Hello, world!" to the console a number of times indicated by the argument `num`
- *returns* the number of characters printed to the console

Functions have *named arguments* which can optionally have default values. Because all function arguments have names, they can be specified using their name.

```
> f(num = 2)
Hello, world!
Hello, world!
[1] 28
```

Specifying an argument by its name is sometimes useful if a function has many arguments and it may not always be clear which argument is being specified. Here, our function only has one argument so there's no confusion.

14.3 Argument Matching

Calling an R function with arguments can be done in a variety of ways. This may be confusing at first, but it's really handing when doing interactive work at the command line. R functions arguments can be matched *positionally* or by name. Positional matching just means that R assigns the first value to the first argument, the second value to second argument, etc. So in the following call to `rnorm()`

```
> str(rnorm)
function (n, mean = 0, sd = 1)
> mydata <- rnorm(100, 2, 1)              ## Generate some data
```

100 is assigned to the `n` argument, 2 is assigned to the `mean` argument, and 1 is assigned to the `sd` argument, all by positional matching.

The following calls to the `sd()` function (which computes the empirical standard deviation of a vector of numbers) are all equivalent. Note that `sd()` has two arguments: `x` indicates the vector of numbers and `na.rm` is a logical indicating whether missing values should be removed or not.

```
> ## Positional match first argument, default for 'na.rm'
> sd(mydata)
[1] 1.014325
> ## Specify 'x' argument by name, default for 'na.rm'
> sd(x = mydata)
[1] 1.014325
> ## Specify both arguments by name
> sd(x = mydata, na.rm = FALSE)
[1] 1.014325
```

When specifying the function arguments by name, it doesn't matter in what order you specify them. In the example below, we specify the `na.rm` argument first, followed by `x`, even though `x` is the first argument defined in the function definition.

```
> ## Specify both arguments by name
> sd(na.rm = FALSE, x = mydata)
[1] 1.014325
```

You can mix positional matching with matching by name. When an argument is matched by name, it is "taken out" of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

```
> sd(na.rm = FALSE, mydata)
[1] 1.014325
```

Here, the `mydata` object is assigned to the `x` argument, because it's the only argument not yet specified.

Below is the argument list for the `lm()` function, which fits linear models to a dataset.

```
> args(lm)
function (formula, data, subset, weights, na.action, method = "qr",
    model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
    contrasts = NULL, offset, ...)
NULL
```

The following two calls are equivalent.

```
lm(data = mydata, y ~ x, model = FALSE, 1:100)
lm(y ~ x, mydata, 1:100, model = FALSE)
```

Even though it's legal, I don't recommend messing around with the order of the arguments too much, since it can lead to some confusion.

Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list. Named arguments also help if you can remember the name of the argument and not its position on the argument list. For example, plotting functions often have a lot of options to allow for customization, but this makes it difficult to remember exactly the position of every argument on the argument list.

Function arguments can also be *partially* matched, which is useful for interactive work. The order of operations when given an argument is

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

Partial matching should be avoided when writing longer code or programs, because it may lead to confusion if someone is reading the code. However, partial matching is very useful when calling functions interactively that have very long argument names.

In addition to not specifying a default value, you can also set an argument value to `NULL`.

```
f <- function(a, b = 1, c = 2, d = NULL) {

}
```

You can check to see whether an R object is `NULL` with the `is.null()` function. It is sometimes useful to allow an argument to take the `NULL` value, which might indicate that the function should take some specific action.

14.4 Lazy Evaluation

Arguments to functions are evaluated *lazily*, so they are evaluated only as needed in the body of the function.

In this example, the function `f()` has two arguments: `a` and `b`.

```
> f <- function(a, b) {
+         a^2
+ }
> f(2)
[1] 4
```

This function never actually uses the argument `b`, so calling `f(2)` will not produce an error because the 2 gets positionally matched to `a`. This behavior can be good or bad. It's common to write a function that doesn't use an argument and not notice it simply because R never throws an error.

This example also shows lazy evaluation at work, but does eventually result in an error.

```
> f <- function(a, b) {
+         print(a)
+         print(b)
+ }
> f(45)
[1] 45
Error in print(b): argument "b" is missing, with no default
```

Notice that "45" got printed first before the error was triggered. This is because `b` did not have to be evaluated until after `print(a)`. Once the function tried to evaluate `print(b)` the function had to throw an error.

14.5 The ... Argument

There is a special argument in R known as the `...` argument, which indicate a variable number of arguments that are usually passed on to other functions. The `...` argument is often used when extending another function and you don't want to copy the entire argument list of the original function

For example, a custom plotting function may want to make use of the default `plot()` function along with its entire argument list. The function below changes the default for the `type` argument to the value `type = "l"` (the original default was `type = "p"`).

```
myplot <- function(x, y, type = "l", ...) {
        plot(x, y, type = type, ...)         ## Pass '...' to 'plot' function
}
```

Generic functions use `...` so that extra arguments can be passed to methods.

```
> mean
function (x, ...)
UseMethod("mean")
<bytecode: 0x7fe2e9aead40>
<environment: namespace:base>
```

The `...` argument is necessary when the number of arguments passed to the function cannot be known in advance. This is clear in functions like `paste()` and `cat()` .

```
> args(paste)
function (..., sep = " ", collapse = NULL, recycle0 = FALSE)
NULL
> args(cat)
function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
    append = FALSE)
NULL
```

Because both `paste()` and `cat()` print out text to the console by combining multiple character vectors together, it is impossible for those functions to know in advance how many character vectors will be passed to the function by the user. So the first argument to either function is `...` .

14.6 Arguments Coming After the `...` Argument

One catch with `...` is that any arguments that appear *after* `...` on the argument list must be named explicitly and cannot be partially matched or matched positionally.

Take a look at the arguments to the `paste()` function.

```
> args(paste)
function (..., sep = " ", collapse = NULL, recycle0 = FALSE)
NULL
```

With the `paste()` function, the arguments `sep` and `collapse` must be named explicitly and in full if the default values are not going to be used.

Here I specify that I want “a” and “b” to be pasted together and separated by a colon.

```
> paste("a", "b", sep = ":")
[1] "a:b"
```

If I don’t specify the `sep` argument in full and attempt to rely on partial matching, I don’t get the expected result.

```
> paste("a", "b", se = ":")
[1] "a b :"
```

14.7 Summary

- Functions can be defined using the `function()` directive and are assigned to R objects just like any other R object

- Functions have can be defined with named arguments; these function arguments can have default values
- Functions arguments can be specified by name or by position in the argument list
- Functions always return the last expression evaluated in the function body
- A variable number of arguments can be specified using the special `...` argument in a function definition.