Introduction to Functions

Biostatistics 140.776

September 27, 2016

Functions

The development of a functions in R represents the next level of R programming, beyond writing code at the console or in a script.

1. Code
2. Functions
3. Packages
4. Software
5. Software System
6. ???

Functions

What's the point?

- Functions are used to **encapsulate** a sequence of expressions that are executed together to achieve a specific goal.
- A single function typically does "one thing well"—often taking some input and the generating output that can potentially be handed off to another function for further processing.
- Drawing the lines where functions **begin** and **end** is a key skill for writing functions.

Functions

Things to consider when writing functions:

- There is going to be a **user** who will desire the ability to modify certain aspects of your code to match their specific needs or application.
- Aspects of your code that can be modified often become *function arguments* that can be specified by the user.
- Specifying what the user can modify leads to the function's **interface**.

Code

Often we start out analyzing data by writing straight R code at the console. This code is designed to accomplish a single task—whatever it is that we are trying to do *right now*.

```
library(readr)
library(dplyr)

## Download data from RStudio (if we haven't already)
if(!file.exists("data/2016-07-20.csv.gz")) {
        download.file("http://cran-logs.rstudio.com/2016/20
                      "data/2016-07-20.csv.gz")
}
cran <- read_csv("data/2016-07-20.csv.gz",
                 col_types = "ccicccccci")
cran %>% filter(package == "filehash") %>% nrow
[1] 179
```

Code

There are a few aspects of this code that one might want to modify or expand on:

- the **date**: this code only reads data for July 20, 2016. But what about data from other days? Note that we would first need to obtain that data if we were interested in knowing download statistics from other days.
- the **package**: this code only returns the number of downloads for the `filehash` package. However, there are many other packages on CRAN and we may want to know how many times these other packages were downloaded.

Once we've identified which aspects of a block of code we might want to modify or vary, we can take those things and abstract them to be arguments of a function.

Function Interface

The following function has two arguments:

- `pkgname`, the name of the package as a character string
- `date`, a character string indicating the date for which you want download statistics, in year-month-day format
- R Functions return a value that is the **last expression** that is evaluated

Function Interface

```
num_download <- function(pkgname, date) {
        year <- substr(date, 1, 4)
        src <- sprintf("http://cran-logs.rstudio.com/%s/%s
                       year, date)

        ## Construct path for storing local file
        dest <- file.path("data", basename(src))
        if(!file.exists(dest))
                download.file(src, dest, quiet = TRUE)

        cran <- read_csv(dest, col_types = "cciccccccci",
                         progress = FALSE)
        ## Return this
        cran %>% filter(package == pkgname) %>% nrow
}
```

Function Interface

Now we can call our function using whatever date or package name we choose.

```
num_download("filehash", "2016-07-20")
[1] 179
```

We can look up the downloads for a different package on a different day.

```
num_download("Rcpp", "2016-07-19")
[1] 13572
```

Note that for this date, the CRAN log file had to be downloaded separately because it had not yet been downloaded.

Default Values

- The way that the `num.download()` function is currently specified, the user must enter the date and package name each time the function is called.
- It may be that there is a **default date** for which we always want to know the number of downloads, for any package.
- We can set a **default value** for the date argument, for example, to be July 20, 2016.
- If the `date` argument is not explicitly set by the user, the function can use the default value.

Default Values

The revised function with default value for `date`.

```
num_download <- function(pkgname, date = "2016-07-20") {
        year <- substr(date, 1, 4)
        src <- sprintf("http://cran-logs.rstudio.com/%s/%s
                       year, date)
        dest <- file.path("data", basename(src))
        if(!file.exists(dest))
                download.file(src, dest, quiet = TRUE)
        cran <- read_csv(dest, col_types = "cciccccccci",
                         progress = FALSE)
        cran %>% filter(package == pkgname) %>% nrow
}
```

Default Values

Now we can call the function in the following manner. Notice that we do not specify the `date` argument.

```
num_download("Rcpp")
[1] 14761
```

- Default values play a critical role in R functions because R functions are often called *interactively*.
- It can be a pain to have to specify the value of every argument in every instance of calling the function.
- Sometimes we want to call a function multiple times while varying a single argument (keeping the other arguments at a sensible default).

Default Values

```
library(ggplot2)
args(qplot)
function (x, y = NULL, ..., data, facets = NULL, margins =
    geom = "auto", xlim = c(NA, NA), ylim = c(NA, NA), log
    main = NULL, xlab = deparse(substitute(x)), ylab = depa
    asp = NA, stat = NULL, position = NULL)
NULL
```

Default Values

Function arguments have a tendency to proliferate.

- As functions are continuously developed, one way to add more functionality is to increase the number of arguments.
- If these new arguments do not have sensible default values, then users will generally have a harder time using the function.
- You have tremendous influence over the user's behavior by specifying defaults, so take care in choosing them.
- Judicious use of default values can greatly improve the user experience with respect to your function.

Re-factoring code

Now that we have a function written that handles the task at hand in a more general manner (i.e. it can handle any package and any date), it is worth taking a closer look at the function and asking whether it is written in the most useful possible manner.

In particular, it could be argued that this function does too many things:

1. Construct the path to the remote and local log file
2. Download the log file (if it doesn't already exist locally)
3. Read the log file into R
4. Find the package and return the number of downloads

Re-factoring

It might make sense to abstract the first two things on this list into a separate function.

```
check_for_logfile <- function(date) {
        year <- substr(date, 1, 4)
        src <- sprintf("http://cran-logs.rstudio.com/%s/%s
                       year, date)
        dest <- file.path("data", basename(src))
        if(!file.exists(dest)) {
                val <- download.file(src, dest,
                                     quiet = TRUE)
                if(!val)
                        stop("unable to download ",
                             src)
        }
        dest
}
```

Re-factoring

Now the `num_download()` function is much simpler.

```
num_download <- function(pkgname, date = "2016-07-20") {
        dest <- check_for_logfile(date)
        cran <- read_csv(dest, col_types = "ccicccccci", pr
        cran %>% filter(package == pkgname) %>% nrow
}
```

- Now the `num_download()` function does not need to know anything about downloading or URLs or files
- All it knows is that there is a function `check_for_logfile()` that just deals with getting the data to your computer.

Dependency Checking

The `num_downloads()` function depends on the `readr` and `dplyr` packages. Without them installed, the function won't run.

- We can write a separate function to check that the packages are installed

```
check_pkg_deps <- function() {
        if(!require(readr)) {
                message("installing the 'readr' package")
                install.packages("readr")
        }
        if(!require(dplyr))
                stop("the 'dplyr' package needs to be insta
        ## This function returns nothing useful
}
```

Dependencies

- The `library()` function stops with an error if the package is not installed
- The `require()` function returns `TRUE` or `FALSE` depending on whether the package is installed or not
- `library()` is usually good for interactive work because you usually can't go on without a package
- `require()` is good for programming because you may want to engage in different behaviors depending on which packages are not available (i.e. install vs. stop)

Dependencies

Now our revised function looks as follows.

```
num_download <- function(pkgname, date = "2016-07-20") {
        check_pkg_deps()
        dest <- check_for_logfile(date)
        cran <- read_csv(dest, col_types = "ccicccccci", pr
        cran %>% filter(package == pkgname) %>% nrow
}
```

Vectorization

One final aspect of this function that is worth noting is that as currently written it is not *vectorized*.

- Each argument must be a single value—a single package name and a single date.
- In R, it is a common paradigm for functions to take vector arguments and for those functions to return vector or list results.
- Users are often bitten by unexpected behavior because a function is assumed to be vectorized when it is not.

Vectorization

One way to vectorize our function is to allow the `pkgname` argument to be a character vector of package names. This way we can get download statistics for multiple packages with a single function call. Luckily, this is fairly straightforward to do.

The two things we need to do are

1. Adjust our call to `filter()` to grab rows of the data frame that fall within a vector of package names
2. Use a `group_by() %>% summarize()` combination to count the downloads *for each* package.

Vectorization

The revised function is now as follows.

```
## 'pkgname' can be a character vector of names
num_download <- function(pkgname, date = "2016-07-20") {
        check_pkg_deps()
        dest <- check_for_logfile(date)
        cran <- read_csv(dest, col_types = "ccicccccci",
                         progress = FALSE)
        cran %>% filter(package %in% pkgname) %>%
                group_by(package) %>%
                summarize(n = n())
}
```

Vectorization

Now we can call the following

```
num_download(c("filehash", "weathermetrics"))
# A tibble: 2 × 2
         package     n
           <chr> <int>
1       filehash   179
2 weathermetrics     7
```

- Note that now the **output** has changed from an integer to a data frame.

Vectorizing the `date` argument is similarly possible, but it has the added complication that for each date you need to download another log file.

Argument Checking

Checking that arguments are correctly specified can be useful to intelligently allow the user know when their inputs are improper.

- Both `pkgname` and `date` are required to be *character* inputs
- The types of objects in R can be checked with the various `is.*` functions.
- To check if an object is character we can use `is.character()`
- Because the `date` argument is not vectorized, we might also want to check that it is of length 1.

Argument Checking

The revised function with argument checking is:

```
num_download <- function(pkgname, date = "2016-07-20") {
        check_pkg_deps()
        if(!is.character(pkgname))
                stop("'pkgname' should be character")
        if(!is.character(date) || length(date) != 1)
                stop("'date' should be character, length 1"
        dest <- check_for_logfile(date)
        cran <- read_csv(dest, col_types = "ccicccccci",
                         progress = FALSE)
        cran %>% filter(package %in% pkgname) %>%
                group_by(package) %>%
                summarize(n = n())
}
```

Argument Checking

```
num_download("filehash", c("2016-07-20", "2016-0-21"))

Error in num_download("filehash", c("2016-07-20",
"2016-0-21")) : 'date' should be character, length 1
```

R Packages

- R packages are collections of functions that together allow one to conduct a series of related operations
- The natural evolution of writing many functions
- R packages similarly have an interface or API which specifies to the user what functions he/she can call in their own code

When Should I Write a Function?

Deciding when to write a function depends on the context in which you are programming in R.

- For a one-off type of activity, it's probably not worth considering the design of a function or set of function.
- There are relatively few one-off scenarios
- We often have to repeat certain tasks or we have to share code with others. Sometimes those "other people" are simply ourselves 3 months later.
- Will your code ever have any **users**, including yourself later on?
- If the code will have more than one user, they will benefit from the abstraction and simplification afforded by encapsulating the code in functions and providing a clean interface.

Your closest collaborator is you six months ago, but you don't reply to emails. —Karl Broman

When Should I Write a Function?

Some rules of thumb:

1. If you're going to do something **once**, just write some code and *document it very well*. You want to be able to reproduce it later on if you ever come back to it, or if someone else comes back to it.
2. If you're going to do something **twice**, write a function. This forces you to define an interface so you have well defined inputs and outputs.
3. If you're going to do something **three** times or more, you should think about writing a small package.

Summary

Developing functions is a key aspect of programming in R and typically involves a bottom-up process.

1. Code is written to accomplish a specific task or a specific instance of a task.
2. The code is examined to identify key aspects that may be modified by other users; these aspects are abstracted out of the code and made arguments of a function.
3. Functions are written to accomplish more general versions of a task; specific instances of the task are indicated by setting values of function arguments.
4. Function code can be re-factored to provide better modularity and to divide functions into specific sub-tasks.
5. Functions can be assembled and organized into R packages.