Day Four: Functional Programming

Dillon Niederhut
Shinhye Choi

July 27, 2015

Introduction

Remember how we said that R started off as a functional language? This means that, underneath the hood, R is built on many small functions that can be grouped together in smart ways to do powerful things. It also means that, if you want to do more complicated things in R than run summary statistics and linear models, you'll need to learn how to create and use functions.

In R, functions are first-class citizens. This means that you can do anything to a function that you can do to an object, including using functions to create other functions.

Structuring a computer language around functions and their methods makes it easily parellelizable in ways that object oriented languages are usually not (for many complicated reasons that we don't have time to talk about). Key components of function-oriented languages are functions that write functions and the ability to map functions to data structures.

Finally, when we talk about the amazing extensibility of R, what we mean is that other people have written useful functions that you can find and download. If R is required in your field, it is likely because there are many functions specific to your field that have been developed in R. We'll close the intensive with a brief introduction to packaging and sharing R functions.

Looping

Before we get to understand how functions work and learn how to create functions, let us go over how for-loop works in R. Many functions often have for-loops embedded in them, hence it will be useful to understand looping first. the basic syntax looks like the following:

> side note - while you can use loops in R, the community strongly discourages explicit looping in favor of implicit loop functionals like `Map` and `lapply`

syntax: for (variable in sequence) {statement}

```
## I want to create a matrix of 2 to the power of n where n is 1 to 10.

mat <- c(rep(NA, 10))   # first create a null vector

# There are many ways to do the same task
mat <- c(rep(NA, 6))
for(i in 1:6){   # I want to create a matrix of 2 to the power of n where n is 5 to 10.
  mat[i] <- 2^(i+4)
}                # or

mat <- c(rep(NA, 6))
for(i in 5:10){
  mat[i-4] <- 2^i
}                # by setting sequence and statement accordingly
```

You can also loop over a non-numeric vector

```
a <- c("Berkeley", "SF", "Oakland")
b <- c(20, 18, 22)
city.temp <- data.frame(cbind(a, b))

for(city in c("Berkeley", "Walnut Creek", "Richmond")){
  if(sum(city==city.temp$a)>0){
    print(city.temp[which(city==city.temp$a),])
    # if we have the city in our data, then print it's temperature and the name of the city
  }
  if(sum(city==city.temp$a)==0){
    print(paste(city, "is NOT in the data. :(", sep=" "))
    # if not, then just print the name of the city next to "is Not in the data. :("
  }
}   # Loops can be as complicated and long as they could be. Often not so efficient.
```

```
##          a  b
## 1 Berkeley 20
## [1] "Walnut Creek is NOT in the data. :("
## [1] "Richmond is NOT in the data. :("
```

How can we make the running time shorter? The "break" command can be handy. We use the command when we want to "end the looping" once the variable reaches where we want it to stop.

```
system.time(
  for(i in 1:1000){
  print(i)
  })

system.time(
  for(i in 1:1000){
    print(i)
  if(i == 50) break
  })
```

Next we move on to control structures, such as if statements. "If" statements are very useful when you want to assign different tasks to different subsets of data using a single for-loop. The basic syntax looks like the following: if(condition){statement} else{other statement}

> side note - there is no **elseif** or **elif** keyword in R, which will be confusing to folks coming from Matlab and Python

```
x <- 7
if(x > 10){
  print(x)

  }else{                    # "else" should not start its own line.
                            # Always let it be preceded by a closing brace on the same line.
  print("NOT BIG ENOUGH!!")
}
```

```
## [1] "NOT BIG ENOUGH!!"
```

The “ifelse" function can be handy as long as you have two conditions that are mutually exclusive

```
# ifelse(test, yes, no)
gender <- sample(c("male", "female"), 100, replace=TRUE)
gender
```

```
##   [1] "female" "female" "male"   "female" "male"   "female" "male"
##   [8] "male"   "male"   "female" "female" "female" "male"   "female"
##  [15] "female" "male"   "female" "female" "female" "female" "male"
##  [22] "female" "female" "male"   "female" "female" "female" "male"
##  [29] "female" "male"   "male"   "male"   "female" "male"   "female"
##  [36] "female" "female" "male"   "female" "male"   "male"   "female"
##  [43] "male"   "male"   "male"   "female" "male"   "female" "male"
##  [50] "male"   "male"   "male"   "female" "female" "male"   "male"
##  [57] "female" "female" "female" "male"   "female" "male"   "male"
##  [64] "male"   "male"   "male"   "male"   "female" "female" "female"
##  [71] "male"   "female" "female" "male"   "male"   "male"   "female"
##  [78] "male"   "male"   "male"   "male"   "female" "female" "female"
##  [85] "female" "male"   "male"   "female" "male"   "female" "female"
##  [92] "female" "male"   "male"   "female" "male"   "female" "male"
##  [99] "female" "female"
```

```
gender <- ifelse(gender=="male", 1, 0)
gender
```

```
##   [1] 0 0 1 0 1 0 1 1 1 0 0 0 1 0 0 1 0 0 0 0 1 0 0 1 0 0 0 1 0 1 1 1 0 1 0
##  [36] 0 0 1 0 1 1 0 1 1 1 0 1 0 1 1 1 1 0 0 1 1 0 0 0 1 0 1 1 1 1 1 1 0 0 0
##  [71] 1 0 0 1 1 1 0 1 1 1 1 0 0 0 0 1 1 0 1 0 0 0 1 1 0 1 0 1 0 0
```

“While" loops are used less frequently. The basic syntax: while(condition) {statement}

```
# if there are multiple statements, then use ; to separate each statement
x <- 0
while(x < 5) {print(x <- x+1)}
x <- 1
while(x < 5) {x <- x+1; if (x == 3) break; print(x)}  # break the loop when x=3
```

Functions

it’s really easy to create functions in R

```
f <- function(x) x + 1
class(f)
```

```
## [1] "function"
```

every function has three parts

every function needs inputs - these are called `arguments` (or, here, formal arguments)

every function has stuff it does, and this stuff is contained in the body

every function has an `environment` that it executes in

```
formals(f)
```

```
## $x
```

```
body(f)
```

```
## x + 1
```

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

environments are where the function was defined

see how our function has `R_GlobalEnv` as it's environment? that's because we defined it in the global environment

this means that if you tell a function to look for an `object`, it will look in the global namespace

```
f <- function(x) x + y
y <- 1
f(x = 1)
```

```
## [1] 2
```

it's very common for functions to be declared within another function

```
y <- 9001
f <- function(x) {
  y <- 1
  g <- function (x) {
    x + y
  }
  g(x)
}
f(1)
```

```
## [1] 2
```

this is important because it means that functions can be separated from the state of your computer (which is what makes them easy to parallelize)

functions don't modify your computer state (usually)

an obvious exception is writing/reading from disk, but what we really mean here is that anything created inside the function environment doesn't show up in the global environment

```
h <- function(){
  if (!exists('a')) {
    a <- 1
  }
  else {
    a <- 9000
  }
  print(a)
}
h()
```

```
## [1] 9000
```

```
h()
```

```
## [1] 9000
```

there are ways to make functions modify global variables, but this is generally not a good idea - anything that needs to go into a function should be in the arguments, and anything that needs to come out of the function should be returned

> side note - R automatically returns the value of the last expression, so there is no need for an explicit `return` statement unless you want to break the function early

a couple of days ago, we were dealing with data that came in several different units of length - let's try writing a function that converts inches to centimeters

```
in_to_cm <- function(x) x * 2.5
in_to_cm(69)
```

```
## [1] 172.5
```

that's not juvenile humor - it's actually Dillon's height in inches

what if we want to know how tall we are in meters?

you could do `function(x) x * 2.5 / 100` but this would be repeating yourself

then, when you figure out that the conversion factor is really *2.54*, not 2.5, you might update one and forget to update the other

```
in_to_m <- function(x){
  in_to_cm(x) / 100
}
in_to_m(69)
```

```
## [1] 1.725
```

now, if we go back and update `in_to_cm`, those changes automatically get propogated to `in_to_m`

```
in_to_cm <- function(x) x * 2.54
in_to_m(69)
```

```
## [1] 1.7526
```

if you were here for the intro to Unix, this idea of small functions combined together should sound awfully familiar

R is a bit quirky in that there is no such thing as an uncontained value, e.g. `4` is really a vector with length of one, and a value of 4 in position 1

```
69 == c(69)
```

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

this means that R automatically broadcasts functions across vectors of any length

```
heights <- c(69,54,73,82)
in_to_m(heights)
```

```
## [1] 1.7526 1.3716 1.8542 2.0828
```

this doesn't work with lists

```
heights <- list(69,54,73,82)
in_to_m(heights)
```

Functionals

a functional is a function that takes functions as arguments

the wrong way to be functional

imagine you want to apply a function to the columns of a dataframe (which is a list!)

you could do something like this:

```
in_to_m(heights[[1]])
```

```
## [1] 1.7526
```

```
in_to_m(heights[[2]])
```

```
## [1] 1.3716
```

```
in_to_m(heights[[3]])
```

```
## [1] 1.8542
```

but this is clunky, prone to errors, and can't acommodate changes in your list - if you added another item in the list, you would need to find every place you tried to do this and `cntrl-c cntrl-v` a whole bunch of crap

the right way to be functional

```
lapply(heights, in_to_m)
```

```
## [[1]]
## [1] 1.7526
##
## [[2]]
## [1] 1.3716
##
## [[3]]
## [1] 1.8542
##
## [[4]]
## [1] 2.0828
```

it's not always smart to name functions

these are called anonymous functions - they aren't actually any different, but you should know they exist

```
lapply(heights, FUN = function(x) x %/% 12)
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 6
##
## [[4]]
## [1] 6
```

lapply has limits

```
dat <- read.csv('data/large.csv')
str(dat)
```

```
## 'data.frame':    1000 obs. of  3 variables:
##  $ a: num  1.358 11.358 -8.956 1.881 0.544 ...
##  $ b: num  501 510 489 499 497 ...
##  $ c: num  -249999 -248990 -248013 -247007 -246015 ...
```

```
lapply(dat, mean)
```

```
## $a
## [1] NA
##
## $b
## [1] NA
##
## $c
## [1] NA
```

we *know* there are numbers there - why are the means all missing?

a. we didn't use amelia
b. `mean` has an argument named `na.rm` that ignores missingness

and for Hadley knows what reason, the default is FALSE

but we can't do this

```
lapply(dat, mean(na.rm = TRUE))
```

so we use Map

Map is a function similar to those found in other functional languages

```
Map(mean, dat, na.rm=TRUE)
```

this can be parallelized

> side note - previous versions of these materials imported the `parallel` library, which is no longer supported as of R versions >= 3.2

```
install.packages('parallelMap')
```

```
##
## The downloaded binary packages are in
##  /var/folders/rj/8gpcssqd52z9yrqw7f8xxfym0000gn/T//RtmpzGB5Ys/downloaded_packages
```

```
library(parallelMap)
system.time(Map(median, dat, na.rm=TRUE))
```

```
##    user  system elapsed
##       0       0       0
```

```
system.time(parallelMap(median, dat, na.rm=TRUE))
```

```
##    user  system elapsed
##   0.001   0.000   0.001
```

parallel processing incurs time costs from memory management and message passing that can make small jobs take longer in parallel than in serial

Packages

** Real artists ship - Steve Jobs **

why to package

1. embrace your inner sloth
2. Linus's law
3. R packages get cited

how to package

1. document first
2. avoid feature creep
3. release early, release often

we're going to be using devtools, an R package which makes it easier to build, install, and share packages

```
install.packages('devtools')
```

```
library(devtools)
# has_devel() # this is currently returning a clang compiler error
```

basic components of a package

1. the only thing a package needs to be a package is `Description` - this should tell you something about the importance of documentation (namely, that code without documentation is worthless)
2. your package will also need a name - keep it simple but unique (i.e. easily Googleable)
3. *the code*
4. namespace

getting started

we'll use devtools to create the boilerplate for us

```
devtools::create("convertR")
```

editing the DESCRIPTION

the `DESCRIPTION` is a plaintext file in DCF format (similar to YAML). if you open it up, you should see something like this:

```
Package: convertR
Title: What the Package Does (one line, title case)
Version: 0.0.0.9000
Authors@R: person("First", "Last", email = "first.last@example.com", role = c("aut", "cre"))
Description: What the package does (one paragraph)
Depends: R (>= 3.2.1)
License: What license is it under?
LazyData: true
```

it's your job to edit this to contain the correct information

adding dependencies

devtools automatically adds in that your code depends on the ability to run R, and at a version number equal to or greater than the one you are currently using

what if there are other packages that your package uses? like ggplot2? do

```
Imports: ggplot
```

and if you want list optional packages, you can do so like this:

```
Suggests:
  reshape2 (>=1.4.1)
  plyr (>=1.8.3)
```

> side note - moving the packages a line below is a stylistic choice so that they line up if there is more than one - be sure to indent!

adding your code

generally speaking, your package should only contain definitions of objects, most of which are functions

these are placed in `.R` files in the `/R` directory

we can take the functions that we defined above:

```
in_to_cm <- function(x) x * 2.54

in_to_m <- function(x){
  in_to_cm(x) / 100
}
```

put them in a file in `/R`, and save it with an informative name like `lengths.R`

creating man pages

these are descriptions of each object in your .R script

you could write these in R's semi-LaTeX format yourself, but that's time consuming

as Raymond Hettinger would say "there must be a better way"

```
install.packages('roxygen2')
```

```
library(roxygen2)
```

now we're going to add specialized comments to our length.R file

```
#' Converts inches to centimeters
#'
#' @param x A numeric
#' @return Converted numeric
#' @examples
#' in_to_cm(1)
#' in_to_cm(c(1,2,3))
in_to_cm <- function(x) x * 2.54

#' Converts inches to meters
#'
#' @param x A numeric
#' @return Converted numeric
#' @examples
#' in_to_m(1)
#' in_to_m(c(1,2,3))
in_to_m <- function(x){
  in_to_cm(x) / 100
}
```

and now create the documentation with

```
devtools::document('convertR')
```

NAMESPACE

this is a file that tells R what `names` from the `environment` your package calls, and what `names` your package is going to put into the `global environment` for the user

again, you can write this yourself in `NAMESPACE`:

```
export(in_to_cm)
export(in_to_m)
```

or you can have roxygen2 handle it for you by adding `#' @export` in the function blocks you want to have exported

```
#' Converts inches to centimeters
#'
#' @param x A numeric
#' @return Converted numeric
#' @examples
#' in_to_cm(1)
#' in_to_cm(c(1,2,3))
#' @export
in_to_cm <- function(x) x * 2.54

#' Converts inches to meters
#'
#' @param x A numeric
#' @return Converted numeric
#' @examples
#' in_to_m(1)
#' in_to_m(c(1,2,3))
#' @export
in_to_m <- function(x){
  in_to_cm(x) / 100
}
```

then running:

```
devtools::document('convertR')
```

roxygen is careful in that it will only write files if they:

a. do not exist yet
b. were created by roxygen

you can see this in the header:

```
# Generated by roxygen2 (4.1.1): do not edit by hand
```

data

lastly, if you are shipping data with your code, it goes in the `/data` directory

CRAN expects this to contain a single `.Rdata` file created by `save()`

the other option is to use `devtools::use_data()`

source packages are not bundled packages

now you have a package!

but no one else can use it

this might be want you want, but you may want to share it

first, we'll have to tell R what things are not a part of our shipped package

adding .Rbuildignore

CRAN is very fussy about what they allow to be uploaded, and in what format

remember .gitignore? there's a similar function for R packages

you can add regex to .Rbuildignore via devtools

```
devtools::use_build_ignore("Rproj", pkg = "convertR")
```

devtools automatically initiates this as a project file for RStudio, which no one else wants to see

if you have a README, NEWS, or UPDATES file, you should add them to .Rbuildignore

checking

before you ship your code anywhere, you should check it to make sure it works the way it's supposed to

ideally, you will have been doing this with unit testing all along, but that's beyond the scope of this class

```
devtools::check("convertR")
```

this checks over 50 compatibility issues, and takes a bit of time, even with our two tiny functions

shipping to Github

if you're planning on shipping to github, you're pretty much set

initialize git in your package, add the contents, and push to a repo

```
git init
git add *
git commit -m "initial commit"
git remote add origin git@github.com:deniederhut/convertR
git push
```

shipping to CRAN

this is a bit more involved

first, you'll need to build your package

```
devtools::build('convertR')
```

this creates a compressed tarball, which should be called something like `convertR_0.0.0.9000.tar.gz"`

which you'll submit via https://cran.r-project.org/submit.html

in the comments, you should include:

1. the environments you checked on
2. the results of `devtools::check()`, with an explanation for any errors

Practice

Assignment

Remember how we read lines from an html document of Romeo and Juliet on day two?

```
RJ <- readLines("http://shakespeare.mit.edu/romeo_juliet/full.html")
```

Write functions to parse this document into acts, then count the number of times the words "Romeo" and "Juliet" appear in each act. Then, package these functions.

Example code

Take a look at the data and look for some pattern.

```
RJ[grep("<title>", RJ, perl=TRUE)]
```

```
## [1] " <title>Romeo and Juliet: Entire Play"
```

```
RJ[grep("<H3>", RJ, perl=TRUE)]
```

```
## [1] "<H3>ACT I</h3>"   "<H3>ACT II</h3>"  "<H3>ACT III</h3>"
## [4] "<H3>ACT IV</h3>"  "<H3>ACT V</h3>"
```

```
RJ[grep("<h3>", RJ, perl=TRUE)]
```

```
##  [1] "<h3>PROLOGUE</h3>"
##  [2] "<h3>SCENE I. Verona. A public place.</h3>"
##  [3] "<h3>SCENE II. A street.</h3>"
##  [4] "<h3>SCENE III. A room in Capulet's house.</h3>"
##  [5] "<h3>SCENE IV. A street.</h3>"
##  [6] "<h3>SCENE V. A hall in Capulet's house.</h3>"
##  [7] "<h3>PROLOGUE</h3>"
##  [8] "<h3>SCENE I. A lane by the wall of Capulet's orchard.</h3>"
##  [9] "<h3>SCENE II. Capulet's orchard.</h3>"
## [10] "<h3>SCENE III. Friar Laurence's cell.</h3>"
## [11] "<h3>SCENE IV. A street.</h3>"
## [12] "<h3>SCENE V. Capulet's orchard.</h3>"
## [13] "<h3>SCENE VI. Friar Laurence's cell.</h3>"
## [14] "<h3>SCENE I. A public place.</h3>"
## [15] "<h3>SCENE II. Capulet's orchard.</h3>"
## [16] "<h3>SCENE III. Friar Laurence's cell.</h3>"
## [17] "<h3>SCENE IV. A room in Capulet's house.</h3>"
## [18] "<h3>SCENE V. Capulet's orchard.</h3>"
## [19] "<h3>SCENE I. Friar Laurence's cell.</h3>"
## [20] "<h3>SCENE II. Hall in Capulet's house.</h3>"
## [21] "<h3>SCENE III. Juliet's chamber.</h3>"
## [22] "<h3>SCENE IV. Hall in Capulet's house.</h3>"
## [23] "<h3>SCENE V. Juliet's chamber.</h3>"
## [24] "<h3>SCENE I. Mantua. A street.</h3>"
## [25] "<h3>SCENE II. Friar Laurence's cell.</h3>"
## [26] "<h3>SCENE III. A churchyard; in it a tomb belonging to the Capulets.</h3>"
```

Now that we know that the first line of each act begins with the string “

“, we will create a null list object called x, and then assign all the lines in each act to each component in x.

```
x <- list(NA)
y <- grep("<H3>", RJ, perl=TRUE)
for(i in 1:length(y)){
  if(i < length(y)){
  x[[i]]  <- RJ[c(y[i]:(y[i+1]-1))]
  }
  if(i == length(y)){
    x[[i]]  <- RJ[c(y[i]:length(RJ))]
  }
}
```

How should we count the number of the words appear in each act? Create a wrapper function that counts the number of the words and returns the number.

```
countR <- function(z){
  return(c(length(grep("Romeo", z, perl=T)), length(grep("Juliet", z, perl=T))))
}
lapply(x, countR)
```

```
## [[1]]
## [1] 8 4
##
## [[2]]
## [1] 30  3
##
## [[3]]
## [1] 54 13
##
## [[4]]
## [1] 9 8
##
## [[5]]
## [1] 20 19
```

Now count the lines in each scene

```
# now count the lines in each scene
countL <- function(z){
  return(length(grep("</A><br>$", z, perl=T)))
}
lapply(x, countL)
```

```
## [[1]]
## [1] 739
##
## [[2]]
## [1] 685
##
## [[3]]
```

```
## [1] 821
##
## [[4]]
## [1] 407
##
## [[5]]
## [1] 441
```

Acknowledgements

Materials taken from:

Software Carpentry

Hadley Wickham

more Hadley Wickham