

# day\_\_four

*Dillon Niederhut*

*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.

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`

Structuring a computer language around functions and their methods makes it easily parallelizable 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.

## 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 <- a + 9000
  }
  print(a)
}
h()
```

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

```
h()
```

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

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 propagated 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 accommodate 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 `ctrl-c ctrl-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

```
install.packages('parallel')
```

```
## Warning: package 'parallel' is not available (for R version 3.2.1)
```

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

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

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

```
##      user  system elapsed
##    0.005    0.013    0.020
```

so that's not any better, but this is a small dataset

## Map can be combined with other Filter and Reduce

this is a quick, powerful, and parallelizable way to handle data

# 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() # if this returns false, you will be unhappy
```

```
## '/Library/Frameworks/R.framework/Resources/bin/R' --no-site-file \  
## --no-environ --no-save --no-restore CMD SHLIB foo.c
```

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

## 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 what 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")
```

```
## Updating convertR documentation
## Loading convertR
## '/Library/Frameworks/R.framework/Resources/bin/R' --no-site-file \
## --no-environ --no-save --no-restore CMD build \
## '/Users/dilloniederhut/Dropbox/dlab/workshops/Rintensive/convertR' \
## --no-resave-data --no-manual
##
## '/Library/Frameworks/R.framework/Resources/bin/R' --no-site-file \
## --no-environ --no-save --no-restore CMD check \
## '/var/folders/ml/n7zzqlx55l53vt2xqbm2k980000gn/T/RtmpbNSXrH/convertR_0.0.0.9000.tar.gz' \
## --timings
```

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

## Acknowledgements

### Materials taken from:

[Software Carpentry](#) [Hadley Wickham](#) [more Hadley Wickham](#)