Demography 213
Branching and looping: Programming in R

Carl Mason
carlm@demog.berkeley.edu

September 11, 2016

Abstract

This week we will learn how to program. Computer programming is a moderately complicated skill which one can spend a lifetime, and a relatively lucrative career developing. An hour of lecture, and one assignment *might* therefore, not be quite sufficient for complete mastery, but we'll give it a try.

Though one might think of R as primarily a statistical and graphics package, it is in fact, a more or less complete programming environment. By "complete", I mean that it contains most of the basic elements of a serious low level programming language such as C or FORTRAN. For most things that Demographers do, R is a better than average replacement for Java or C or FORTRAN – which by the way cannot be mastered in one lesson either.

Contents

1 Read Sections 9 and 10 of *Introduction to R* and Sections 5.4 and 5.5 of *R In Action*

Section 9 (4.4) introduces *grouped expressions*, *loops*, branching structures; Section 10 (5.5) discuses functions. These are the basic things, along with data structures (next week's topic) that pretty good programming languages need.

In addition to the syntactic information are three important points about programming in R:

Always document your stuff. This is not a new idea in this class, but when we begin to use and hopefully re-use functions it becomes all the more important that documentation is clear enough to be understood by a moron in a hurry (that is to say, yourself).

Loops in R are inefficient. Although we will concentrate this week on writing `for()` loops and `while()` loops, it should be kept in mind that R does not do loops particularly fast. This *only* becomes a problem with large data sets and as hardware improves, *large* becomes ever larger. But in case large never becomes "large enough" later in this course we will learn speed up our code replacing loops with functions such as `apply()`, `tapply()`, `sapply()`, `lapply()` and also the family of `plyr` function.

Functions should be self contained. When writing functions, it is possible to read (and to a lesser extent assign to) variables which are part of the ***Global Environment***. This is a very bad idea. Functions should get what they need as ***arguments*** and return everything they produce via the `return()` function. Functions that rely on variables generated outside of themselves and **not** passed in as function arguments, are error prone and not very portable.

2 Rstudio/Unix tricks of the week

2.1 TAB completion

Although Unix aficionados prefer typing to pointing and clicking, that does not mean that we actually *like* to type. One of the neat things about clever command line oriented tools (like the Unix Shell and Rstudio) is the way that they save us from actually typing.

TAB competition means hitting `TAB` to ask Unix or Rstudio to guess what you were thinking of typing next. From the context, Unix/Rstudio can generally at least partially complete your thought.

For example, at the Unix shell, after typing `cp ~carlm/213/Week4/dem` , Unix knows that you want to copy a file from my 213/Week4 directory and that that file begins with "dem". Typing `TAB` at this point asks the shell to guess what's your brain: what's the rest of the filename. You've done this a few times now – Unix is pretty good at guessing. It is especially good in cases where I have spelled the file wrong either in the handout or in the directory.

On the Unix command line, `TAB` works even before you give the shell much to work with. For example if you type:

@:> `fi` `TAB`

The shell will suggest all the possible commands that start with "fi":

```
fi           file2brl     findfs       findtrash    fitsdiff     fixmsxpart
fiascotopnm  filefrag     findhyph     firefox      fitsheader   fixparts
fig4latex    file-rename  findmnt      fits2bitmap  fitsinfo     fix-qdf
file         find         findtr       fitscheck    fitstopnm
```

The shell can tell from the context that you are looking to type a command.

2.2 TAB in R/Rsudio

In Rstudio you have already seen the power of the `TAB` completion. It works in both the source pane and the R console pane. Just like at the Unix prompt, Rstudio can guess from the context what sort of thing your are looking for. It can complete the name of an object, or it can prompt you for the arguments to the function you are trying to use.

Experiment with `TAB` and see how much time and effort it can save you.

3 Assignment

This week's assignment consists of a demonstration.r file (as usual) but also some short problems with answers thoughtfully provided below in this handout. With luck we'll get through some of the demonstration.r file in class. You should pick up where we left off.

As usual, you should start by

- copying the demonstration file from `~carlm/213/ProgrammingI/demonstration.r` into your `213/WeekN` directory.
- browsing over to `http://rstudio.demog.berkeley.edu` and creating a ***new project in an existing directory***

When you have extracted all the knowledge and wisdom that exists in the demonstration.r file, move on to the short problems ... with answers. Please start a new R script file for these problems, rather than just sticking them at the end of your demonstration.r file.

4 Short problems with answers

Below are illustrative examples of R's programming structures. They are posed as problems with answers in bacterial sized font– in the hopes that you will think about each problem before looking at the answer.

You have already created your 213/WeekN directory for this week so, once Rstudio is running with the proper working directory set, simply open a new "R script window" and go to it. Don't forget to save the file occasionally and maybe name it something like exercise4.r

Also remember to send email to cmaason@berkeley.edu pointing out one thing in this exercise that you believe to be worthy of deeper consideration.

1. Write a `for` loop the counts the integers between 123 and 3456 that are divisible by 3. HINT:

```
## The %% operator (NOT %*%) gives the remainder
5 %% 2

## [1] 1

111 %% 3

## [1] 0
```

```
## A for() loop that counts the integers divisible by 3 between 123 and 245
answer<-0
for(i in 123:3456){
  answer<- answer + (i %% 3  == 0)
}
answer

## [1] 1112
```

2. Write a `while` loop that counts the integers between 123 and 3456.

```
## Counting the integers between 123 and 2345 with while()

answer<-0
count<-123
while( count <= 3456){
  answer<-answer+1
  count<-count+1
}
```

3. Write a ***function***, that counts the number of integers between **any** two values which are divisible by a third value. In other words create a function that executes the code that you just wrote but in a more general way. So that any 3 numbers can be substituted for 123, 3456 and 3 in the above problem.

   HINT: the function you write should take 3 arguments and return an integer.

```
## function() is itself a function which returns and object of mode
## ``function''  -- is that confusing enough?
## countdivs is the object produced by the function ``function()''
## ``from'', ``to'' and ``by''  will be parameters of the countdivs().
## when countdivs is executed, the user must pass a value for each of
## these parameters as arguments

countdivs<- function(from,to,by){
  answer<-0
  for (i in from:to){
    answer<- answer + (i %% by == 0)*1
  }
  return(answer)
}
countdivs(from=123,to=3456,by=3)

## [1] 1112
```

4. Use an if() to modify the loop you wrote in example 1 so that it counts the integers divisible by either 3 or 4.

```
answer<-0
for(i in 123:345){
  if((i %% 3 == 0) | (i %% 4 == 0)){
     answer<-answer+1
  }
}
answer

## [1] 113

## check answer
sum( ((x<-123:345) %% 3 == 0) | (x %% 4 == 0))

## [1] 113
```

5. Use a `while()` loop to count the number "tosses" of a simulated coin until it comes up tails for the first time.

   HINT: `runif(1) > .5` makes a pretty fair simulated coin. `runif()` generates ***random variables, uniformly distributed over*** (0..1). `? runif` will tell you more.

```
heads<-0
while(runif(1) < .5){
## the first time toss is less than .5 we are done
  heads<-heads+1
}
heads

## [1] 0
```

6. Write a function that returns the number of consecutive heads that a coin with probability p produces. In other words, put the algorithm of the previous problem into a function. But let that function take an argument to determine the probability of heads.

```
## stopifnot() provides a simple way of testing the validity of
## the parameters.
ctoss<-function(p=.5){
  stopifnot( (p >= 0) & (p <= 1))
  heads<-0
  while(runif(1) < p ){
    heads<-heads+1
  }
  return(heads)
}
```

7. Consider a game wherein a player tosses a fair coin and the "house" pays the player 2^n dollars where n is the number of consecutive heads the player tosses. In other words, the player keeps tossing a coin ***until*** she throws a tail. The house pays nothing if the first toss is a tail; $2 for one head followed by a tail; $8 for three heads followed by a tail... and so on.

   For the privilege of playing this game one time, a smart economist, should be willing to pay anything less than the *expected value* of the payoff. The expected value of the payoff is just the sum of all possible payoffs weighted by their probability of occurring. For example, the payoff of tossing 6 heads in a row is 2^6 and the probability of tossing 6 heads in a row is $(\frac{1}{2})^6$. Since it is technically possible to throw an infinite number of heads in a row, the expected value of one game is conveniently calculated in Equation 1

$$E(\text{payoff}) = \sum_{n=1..\infty} 2^n (\frac{1}{2})^n = \sum_{i=1..\infty} 1 = \infty \tag{1}$$

That's right, the expected value of this game is very very high.

If you were to offer a bright, young, economist the opportunity to play this game for only say $20 –considerably less than infinity – a well trained economist simply could not say "no". Suppose you offer to let her play as many games as she would like for $20 per game... How much money will change hands if the economist agrees to play one time per minute for an entire day? Obviously, it would be prohibitively boring to solve this analytically. So how about writing some R code to simulate it.

```
payoff<-0

for(i in 1:(24*60)){
  heads<-ctoss(.5)
  ## (heads >0) is a logical but gets coerced to 1/0
  ## otherwise the house would pay £1 for zero heads in a row
  payoff<- payoff +  ((heads >0)*(2^heads)) -20
}

payoff

## [1] -15858
```

8. Modify the code you wrote in the preceding problem to collect a vector containing the payoff from each trial so you can further humiliate your "colleague" in next week's seminar.

```
seminar<-NULL
  position<-0
  for(i in 1:(24*60)){
    heads<-ctoss(.5)
    payoff<- ((heads>0)*(2^heads)) -20
    position<-position + payoff
    seminar<-rbind(seminar,c(position, payoff)  )
  }
  colnames(seminar)<-c("position","payoff")

summary(seminar)

##     position          payoff
##  Min.   :-19110   Min.   : -20.00
##  1st Qu.:-14558   1st Qu.: -20.00
##  Median : -9081   Median : -18.00
##  Mean   : -9343   Mean   : -13.27
##  3rd Qu.: -4045   3rd Qu.: -16.00
##  Max.   :    -4   Max.   :1004.00

library(Hmisc)

## Loading required package:  lattice
## Loading required package:  survival
## Loading required package:  Formula
```

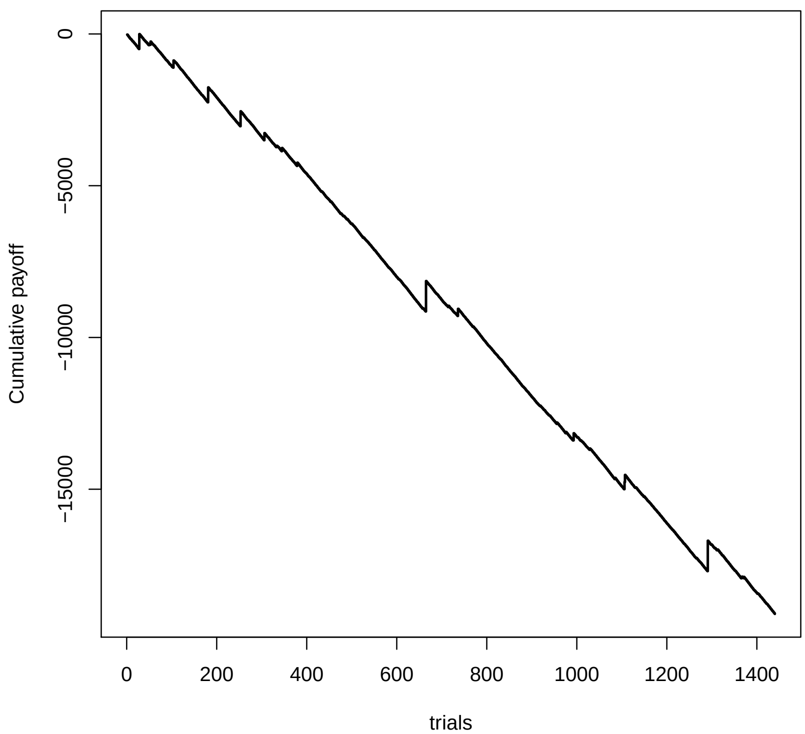
```
## Loading required package:  ggplot2
##
## Attaching package:  'Hmisc'
## The following objects are masked from 'package:base':
##
##     format.pval, round.POSIXt, trunc.POSIXt, units
```

```
describe(seminar)
```

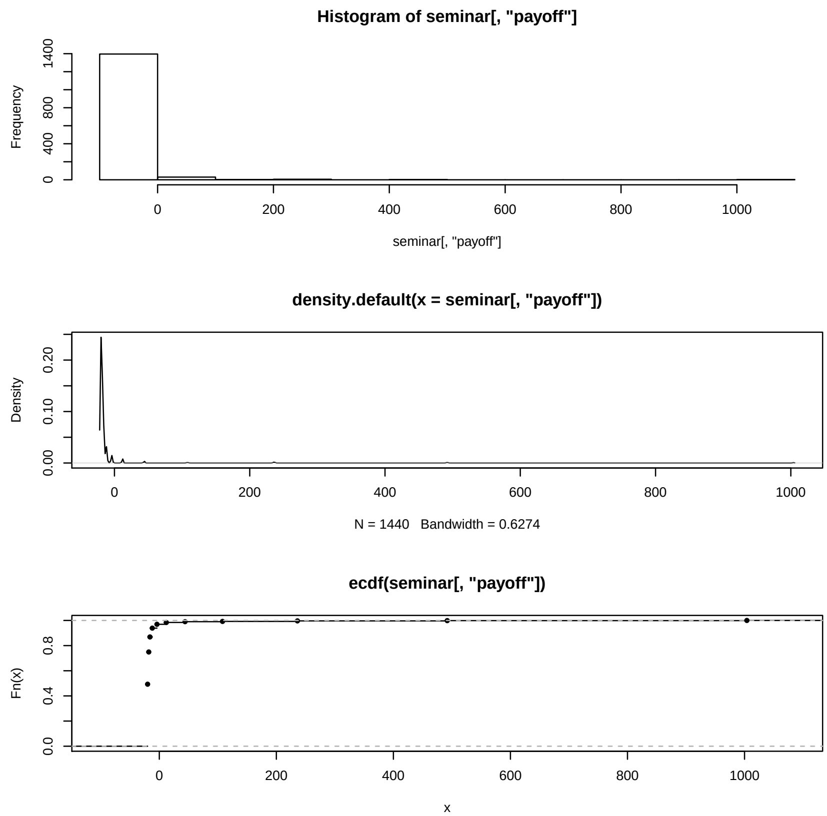
```
## seminar
##
##  2  Variables      1440  Observations
## ---------------------------------------------------------------------------
## position
##         n  missing   unique     Info     Mean      .05      .10      .25
##      1440        0     1402        1    -9343 -17908.4 -17211.0 -14557.5
##       .50      .75      .90      .95
##   -9081.0  -4045.0  -1608.4   -593.4
##
## lowest : -19110 -19090 -19072 -19056 -19036
## highest:    -42    -40    -22    -20     -4
## ---------------------------------------------------------------------------
## payoff
##        n missing  unique    Info    Mean     .05     .10     .25     .50
##     1440       0      11    0.86  -13.27     -20     -20     -20     -18
##      .75     .90     .95
##      -16     -12      -4
##
##           -20 -18 -16 -12 -4 12 44 108 236 492 1004
## Frequency 710 369 172 101 44 21  9   3   6   3    2
## %          49  26  12   7  3  1  1   0   0   0    0
## ---------------------------------------------------------------------------
```

And of course you'll want to draw some graphs to help your economist friend benefit as much as possible from this experience...

```
  plot(seminar[,"position"],
      type='s',lwd=2,ylab="Cumulative payoff",xlab="trials")
```




```
par(mfrow=c(3,1))
hist(seminar[,"payoff"])
plot(density(seminar[,"payoff"]))
plot(ecdf(seminar[,"payoff"]))
```



Your all done. Except for a short email to cmason@berkeley.edu describing the most exhilarating part of this exercise.