r-prog Documentation

Release 0.1

Ista Zahn

March 09, 2016

Contents:

CHAPTER 1

Workshop overview and materials

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

2.1 Workshop description

This is an intermediate/advanced R course appropriate for those with basic knowledge of R. It is intended for those already comfortable with using R for data analysis who wish to move on to writing their own functions. To the extent possible this workshop uses real-world examples. That is to say that concepts are introduced as they are needed for a realistic analysis task. In the course of working through a realistic project we will lean about interacting with web services, regular expressions, iteration, functions, control flow and more.

Prerequisite: basic familiarity with R, such as acquired from an introductory R workshop.

2.2 Materials and setup

Everyone should have R installed – if not:

- Open a web browser and go to http://cran.r-project.org and download and install it
- Also helpful to install RStudio (download from http://rstudio.com)

Materials for this workshop include slides, example data sets, and example code.

- Download materials from http://tutorials.iq.harvard.edu/R/RProgramming.zip
- Extract the zip file containing the materials to your desktop

2.3 Launch RStudio on Athena

- To start R **type these commands in the terminal**:
- Open up today's R script
  - In RStudio, Go to **File => Open Script**
  - Locate and open the `RProgramming.R` script in the Rintro folder in your home directory

- Go to **Tools => Set working directory => To source file location** (more on the working directory later)
- I encourage you to add your own notes to this file!

2.4 Example project overview

Throughout this workshop we will return to a running example that involves acquiring, processing, and analyzing data from the Displaced Worker Survery (DWS). In this context we will learn about finding and using R packages, importing and manipulating data, writing functions, and more. The web page has been mirrored at http://tutorials.iq.harvard.edu/cps-uniform-data-extracts/cps-displaced-worker-survey/cps-dws-data for convenience.

Extracting elements from html

It is common for data to be made available on a website somewhere, either by a government agency, research group, or other organizations and entities. Often the data you want is spread over many files, and retrieving it all one file at a time is tedious and time consuming. Such is the case with the CPS data we will be using today.

The Center for Economic and Policy Research has helpfully compiled DWS data going back to 1994 [1]. Although we could open a web browser and download these files one at a time, it will be faster and easier to instruct R to do that for us. Doing it this way will also give us an excuse to talk about html parsing, regular expressions, package management, and other useful techniques.

Our goal is to download all the Stata data sets from http://tutorials.iq.harvard.edu/cps-uniform-data-extracts/cps-displaced-worker-survey/cps-dws-data/. In order to do that we need a list of the Uniform Resource Locators (URLs) of those files. The URLs we need are right there as links in the ceprdata.org webpage. All we have to do is read that data in a way R can understand.

3.1 Packages for parsing html

In order extract the data URls from the ceprdata.org website we need a package for parsing XML and HTML. How do we find such a package?

Task views https://cran.r-project.org/web/views/WebTechnologies.html

R package search http://www.r-pkg.org/search.html?q=html+xml

Web search https://www.google.com/search?q=R+parse+html+xml&ie=utf-8&oe=utf-8

For parsing html in R I recommend either the `xml2` package or the `rvest` package, with the former being more flexible and the later being more user friendly. Let's use the friendlier one.

3.2 Extracting information from web pages with the `rvest` package

Our first task is to read the web page into R. We can do that using the `read_html` function. Next we want to find all the links (the `<a>` tags) and extract their `href` attributes. To give a better sense of this here is what the html for the 2010 data file link looks like:

We want the `href` part, i.e., "/wp-content/cps/data/cepr:sub:*dws2010dta*.zip".

We can get all the `<a>` elements using the `html_nodes` function, and then extract the `href` attributes usig the `html_attr` function, like this:

[1] Center for Economic and Policy Research. 2012. CPS Displaced Worker Uniform Extracts, Version 1.02. Washington, DC.

```
## install.packages("rvest")
library(rvest)

## read the web page into R
dataPage <- read_html("http://tutorials.iq.harvard.edu/cps-uniform-data-extracts/cps-displaced-worke

## find the link ("a") elements.
allAnchors <- html_nodes(dataPage, "a")
head(allAnchors, 15)

## extract the link ("href") attributes
allLinks <- html_attr(allAnchors, "href")
head(allLinks, 15)
```

3.3 Just the data please – regular expressions to the rescue

Looking at the output from the previous example you might notice a problem; we've matched **all** the URLs on the web page. Some of those (the ones that end in .zip) are the ones we want, others are menu links that we don't want. How can we separate the data links from the other links on the page?

One answer is to use regular expressions to idenfify the links we want. Regular expressions are useful in general (not just in R!) and it is a good idea to be familiar with at least the basics. For our present purpose it will be more than enough to use regular expression that matches strings starting with `/wp` and ending with `.zip`.

In regulars expression `^`, `.`, `*`, and `$` are special characters with the following meanings:

^ matches the beginning of the string

. matches any character

***** repeates the last caracter zero or more times

$ matches the end of the string

If you have not been introduced to regular expressions yet a nice interactive regex tester is available at http://www.regexr.com/ and an interactive tutorial is available at http://www.regexone.com/.

R comes with a `grep` function that can be used to search for patterns in strings, but for more sophisticated string manipulation I recommend the `stringi` package. The function names are more verbose, but it provides much more complete and robust string handling than is available in base R. For our relatively simple needs `grep` will suffice, but if you need to do extensive string manipulation in R the `stringi` package is the way to go.

```
dataLinks <- grep("^/wp.*\\.zip$", allLinks, value = TRUE)
head(dataLinks)
```

(Note that the backslashes in the above example are used to escape the `.` so that it is matched literally instead of matching any characters as it normallly would in a regular expression.)

Finally, the data links we've extracted are relative to the ceprdata website. To make them valid we need to prepend `http://tutorials.iq.harvard.edu/` to each one. We can do that using the `paste` function.

```
dataLinks <- paste("http://tutorials.iq.harvard.edu", dataLinks, sep = "")
head(dataLinks)
```

3.4 Getting the list of data links the easy way

If you look at the result from the previous two methods you might notice that the URLs are all the same save for the year number. This suggests an even easier way to construct the list of URLs:

```
(dataLinks <- paste("http://tutorials.iq.harvard.edu/wp-content/cps/data/cepr_dws_",
                    seq(1994, 2010, by = 2),
                    "_dta.zip",
                    sep = ""))
```

Wow, that was a **lot** easier. Why oh why didn't we just do that in the first place? Well, it works for this specific case, but it is much less general than the html parsing methods we discussed previously. Those methods will work in the general case, while pasting the year number into the URLs only works because the URLs we want have a very regular and consistent form.

CHAPTER 4

Exercise 0: html parsing

The http://ceprdata.org/ website provides code books for the DWS data in `.pdf` format. Links to these code books are available on the documentation page at http://ceprdata.org/cps-uniform-data-extracts/cps-displaced-worker-survey/cps-dws-documentation/. Parse this page and extract the links to the code books.

Exercise 0 prototype

```
## read in the html page
ceprDoc <- read_html(
  "http://ceprdata.org/cps-uniform-data-extracts/cps-displaced-worker-survey/cps-dws-documentation/"
)
## get the codebook links
ceprCodeBookLinks <- html_attr(#extract attributes
  html_nodes(#from nodes
    ceprDoc,#in ceprDoc
    ## This uses an xpath expression to select just the codebook links.
    ## You could alternatively download all the links and filter them
    ## with a regular expression. Use whatever works and is comfortable!
    ## There is more than one right way.
    xpath = '//*[@id="content"]/article/div/ul[1]//a'),#matching this xpath
  'href' #extract href attributes
)
```

CHAPTER 6

Downloading files in R

Now that we have a vector of URLs pointing to the data files we want to download, we want to iterate over the elements and download each file. We can use the `download.file` function to download the data files.

The `download.file` function requires a URL as the first argument, and a file name as the second argument. We can use the `basename` function to strip of the location part of the URL, leaving only the file name. We could do this verbosely by writing one line for each file:

```
## download.file(dataLinks[1], basename(dataLinks[1]))
## download.file(dataLinks[2], basename(dataLinks[2]))
## ...
## download.file(dataLinks[n], basename(dataLinks[3]))
```

but that is too much typing. First, it would be more convenient if the `download.file` function defaulted to `destfile `` basename(url)=. Fortunately it is very easy to write your own functions in R. We can write a wrapper around the ``download.file` function like this:

```
## a simple function to make downloading files easier
downloadFile <- function(url, # url to download
                         destfile = basename(url), # default name to save to
                         outdir = "./dataSets/", # default directory to save in
                         ... # other named arguments passed to download.file
                         ){
  ## create output directory if it doesn't exist
  if(!dir.exists(outdir)) {
    dir.create(outdir)
  }
  ## download the file using the specified url, output directory, and file name
  download.file(url = url, destfile = paste(outdir, destfile, sep = ""), ...)
}
```

Using this function we can download the data files more conveniently, but we haven’t yet addressed how to avoid typing out separate function calls for each file we need to download. For that we need iteration.

CHAPTER 7

Downloading all the files

To download all the files conveniently we want to iterate over the vector of URLs and download each one. We can carry out this iteration in several ways, including using a `for` loop, or using one of the `apply` family of functions.

`for` and `while` loops in R work much the same as they do in other programming languages. The `apply` family of functions apply a function to each element of an object.

7.1 Iterating using for-loop

One way to download the data files is to use a for-loop to iterate over the contents of our vector of URLs. Some people will tell you to avoid for-loops in R but this is nonsense. Loops are convenient and useful, and while they are not the best tool for all situations calling for iteration they are perfectly appropriate for downloading a series of files. If you've used a for loop in any other language you will probably find the R implementation to be very similar.

For loops in R have the following general structure: `for(<placeholder> in <thing to iterate over>) {do stuff with placeholder}`. In our case we want to iterate over `dataLinks` and download each one, so this becomes

```
str(dataLinks)

## make a directory to store the data
dir.create("dataSets")

for(link in dataLinks) {
    downloadFile(link, outdir = "dataSets/")
}
```

7.2 Iterating over vectors and lists with the `sapply` function

The `sapply` function iterates over a vector or list and applies a function to each element. To start, let's use `sapply` do download all the displaced worker survey data files:

```
## download all the dws data
sapply(dataLinks,
       downloadFile,
       outdir = "dataSets/")
```

7.3 Iterating in parallel with the `mclapply` function

The `mclapply` function iterates over a vector or list and applies a function to each element using multiple CPU cores (where available). Let's use `mclapply` do download all the displaced worker survey data files:

```
## download all the dws data
library(parallel)
mclapply(dataLinks,
       downloadFile,
       outdir = "dataSets/",
       mc.cores = detectCores())
```

We can now use what we've learned about iteration to unzip all the files in the `dataSets` directory, a task I leave to you.

CHAPTER 8

Exercise 1: Iterate and extract

Use a `for` loop or `*apply` function to unzip each of the `.zip` files in the `dataSets` directory.

BONUS (optional): calculate the size of each extracted file and calculate the difference in size between each `.dta` file and the `.zip` file it was extracted from.

Exercise 1 prototype

```
zipFiles <- list.files("dataSets", pattern = "\\.zip$", full.names=TRUE)

## using sapply
sapply(zipFiles, unzip, exdir = "dataSets")
## using a for loop
for(f in zipFiles) unzip(f, exdir = "dataSets")

## Calculating compression ratios
dataFiles <- list.files("dataSets", pattern = "\\.dta$", full.names = TRUE)

uncompSize <- round(file.size(dataFiles) / 1024^2)
compSize <- round(file.size(zipFiles) / 1024^2)

cbind(zipFile = paste0(basename(zipFiles), ": ", compSize, "Mb"),
      dtaFile = paste0(basename(dataFiles), ": ", uncompSize, "Mb"),
      diff = paste0(round(uncompSize - compSize), "Mb"),
      compression_ratio = round(uncompSize / compSize, digits = 3))
```

CHAPTER 10

Importing and inspecting data and meta-data

Our next goal is to read in the data that we downloaded and extracted earlier. The data are stored as Stata data sets, which can be imported using the `read.dta` function in the `foreign` package. Let's start by reading just the first data set.

```
## attach the foreign packge so we can read stata files
library(foreign)

## get a list of all the stata files in the dataSets directory
dataFiles <- list.files("dataSets", pattern = "\\.dta$", full.names=TRUE)

## read in the first one
ceprData1 <- read.dta(dataFiles[1])
```

Now that we've read in some of the data we want to get some more information about it.

10.1 Mode and length

Information about objects in R are stored as *attributes* of the object. All R objects have a storage *mode* and a *length*. Since all objects in R have these attributes we refer to them as *intrinsic attributes*. We can get the value of these intrinsic attributes using the `mode` and `length` functions respecively. For example, what is the mode and length of our `ceprData1` object?

```
mode(ceprData1)
length(ceprData1)
```

10.2 Other properties of data.frames

So far we've seen that `ceprData1` is a list of length 178. Actually `ceprData1` is a special kind of list called a `data.frame`. We can see that by asking R what the `class` of the object is.

```
class(ceprData1)
```

A *data.frame* in R is a list with elements of equal length. It is a rectangular structure with rows and columns. In addition to the *mode* and *length* that all object in R have, *data.frames* also have dimension, (col)names, and =rownames.

```
dim(ceprData1)
names(ceprData1)
c(head(rownames(ceprData1)), "...", tail(rownames(ceprData1)))
```

10.3 Additional attributes

OK, so far we know the ceprData1 is a *data.frame* with 156246 rows and 178 columns, and that the variables have terrible cryptic names like `cjpporg` and `ljagric`. What do we actually have here? One way to answer the question is

```
browseURL(ceprCodeBookLinks[1])
```

and that is a good answer actually. In our case the meta-data has also been embedded in the `.dta` files by our friends at ceprdata.org. This meta-data has been attached to the `ceprData1` data.frame in the form of additional attributes.

The system used by R for storing attributes of this kind is simple. Arbitrary attributes can be set using the `attr` function, and retrieved using either `attr` or the `attributes` function. Let's take a quick look a this system before using it to access the ceprData1 meta-data.

```
x <- 1:10
attributes(x)
attr(x, "description") <- "This is vector of integers from 1 to 10"
attributes(x)
attr(x, "how_many") <- "There are ten things in this vector"
attributes(x)
attr(x, "description")
```

As we've seen, additional attributes and be accessed vie the `attributes` function. Let's see what other attributes our `ceprData` object has.

```
ceprDataInfo <- attributes(ceprData1)
mode(ceprDataInfo)
class(ceprDataInfo)
length(ceprDataInfo)
names(ceprDataInfo)
```

Let's iterate over the attributes of =ceprData1- and get some more information about the available meta-data

```
t(sapply(attributes(ceprData1),
         function(x) {
           c(mode = mode(x), class = class(x), length = length(x))
         })
  )
```

10.4 Extracting useful meta-data

We can extract elements from lists in a few different ways:

```
## extract by name
ceprDataInfo$datalabel #using $
ceprDataInfo["datalabel"] #using [
ceprDataInfo[["datalabel"]] #using [[, note the difference

## by position
ceprDataInfo[1]; ceprDataInfo[[1]] # note the difference

## by logical index
ceprDataInfo[c(TRUE, TRUE, rep(FALSE, 10))]
ceprDataInfo[sapply(ceprDataInfo, length) == 1]
```

Note that `[` indexing on a list returns a list, and `[[` indexing returns whatever contained in a single element of the list.

This visual explanation may help: file:images/HadleyWickham_index_list.png [2]

You may have noticed during our earlier investigation of `ceprDataInfo` that many of the elements have length 178. That number might be familiar:

```
dim(ceprData1)
```

10.4.1 Exercise 2

Extract elements from `ceprDataInfo` that will help you understand what each column in `deprData` contains. Include at least the variable `names` and `var.labels` as well as any other information that you think will be useful.

Bonus (optional): supplement the `ceprDataInfo` you extracted in step one with the mode, class, etc. of each column in `ceprData1`

10.4.2 Exercise 2 prototype

```
ceprCodebook <- data.frame(
  ceprDataInfo[
    sapply(ceprDataInfo, length) == ncol(ceprData1)
  ])

ceprCodebook$mode <- sapply(ceprData1, mode)
ceprCodebook$class <- sapply(ceprData1, class)
ceprCodebook$n_distinct = sapply(ceprData1, function(x) length(unique(x)))

rbind(head(ceprCodebook), tail(ceprCodebook))
```

[2] Photo by Hadley Wickham via https://twitter.com/hadleywickham/status/643381054758363136/photo/1. Used by permission.

Aggregation

Now that we have the data read in, and we know what is in each column, I want to calculate the proportion displaced by year/rural/gender. I can do that using the `aggregate` function (the `data.table` and `dplyr` packages provide advanced aggregation capabilities, but `aggregate` is available in base R and works well for many things).

```
ceprData1 <- aggregate(ceprData1["dw"],
                       by = ceprData1[c("year", "rural", "female")],
                       FUN = mean, na.rm = TRUE)
ceprData1
```

CHAPTER 12

Exercise 3

Now that we have a process for importing and aggregating the data we can apply it to all the data files we downloaded earlier. We can do that by wrapping the `read.dta` and `aggregate` code in a function and applying that function to each element of `dataFiles` using the `sapply` function, or using a `for` loop. Go ahead and give it a try! Note that this exercise is intentionally challenging; read the documentation, search stackoverflow.com, and use any other resources at your disposal as you attempt it.

Exercise 3 prototype

```
library(foreign)

ceprData <- mclapply(dataFiles, function(x) {
  tmp <- read.dta(x)
  return(aggregate(tmp["dw"],
                   by = tmp[c("year", "rural", "female")],
                   FUN = mean, na.rm = TRUE))})
```

Finishing touches

We now have a list of aggregated data.frames. The next step is to stack each element of the list so that we end up with one big data.frame instead of a list of small ones. We can stack two data.frames using the `rbind` function:

```
ceprData[[1]]; ceprData[[2]]
rbind(ceprData[[1]], ceprData[[2]])
```

and we can apply this operation to every element in the list using the `do.call` function.

```
str(ceprData <- do.call("rbind", ceprData))
```

14.1 Data formatting

Our final step before plotting our data is to format the values for `rural` and `female`. Currently these values are stored as 0/1 dummy codes, but I would like for the values to be spelled out.

Earlier we saw how to extract elements of R objects using bracket notation. To replace elements we using the replacement form, which looks like this:

```
 ceprData[["rural"]] <- factor(ceprData[["rural"]],
                               levels = c(0, 1),
                               labels = c("Non-rural", "Rural"))

 ceprData[["gender"]] <- factor(ceprData[["female"]],
                               levels = c(1, 0),
                               labels = c("Female", "Male"))

ceprData$displaced_percent <- ceprData$dw * 100

str(ceprData)
```

14.2 Plotting

Now we can take a look at the trends in worker displacement over the last few years.

```
library(ggplot2)
library(directlabels)
ceprPlot <- ggplot(ceprData, aes(x = year, y = displaced_percent, color = gender)) +
  geom_line() +
  geom_point() +
  facet_wrap(~rural)
direct.label(ceprPlot)
```

What else?

If there is anything else you want to learn how to do, now is the time to ask!

CHAPTER 16

Go forth and code!

You now know everything you could possibly want to know about R. OK maybe not! But you do know how to manipulate character strings with regular expressions, write your own functions, execute code conditionally, iterate using `for` or `sapply`, inspect and modify attributes, and extract and replace object elements. There's a lot more to learn, but that's a pretty good start. As you go forth and write your own R code here are some resources that may be helpful.

16.1 Additional reading and resources

- Learn from the best: http://adv-r.had.co.nz/
- S3 system overview: https://github.com/hadley/devtools/wiki/S3
- S4 system overview: https://github.com/hadley/devtools/wiki/S4
- R documentation: http://cran.r-project.org/manuals.html
- Collection of R tutorials: http://cran.r-project.org/other-docs.html
- R for Programmers (by Norman Matloff, UC–Davis)

http://heather.cs.ucdavis.edu/~matloff/R/RProg.pdf

- Calling C and Fortran from R (by Charles Geyer, UMinn)

http://www.stat.umn.edu/~charlie/rc/

- State of the Art in Parallel Computing with R (Schmidberger et al.)

http://www.jstatsol.org/v31/i01/paper

- Institute for Quantitative Social Science: http://iq.harvard.edu
- Research technology consulting: http://projects.iq.harvard.edu/rtc

16.2 Things that may surprise you

There are an unfortunately large number of surprises in R programming. Some of these "gotcha's" are common problems in other languages, many are unique to R. We will only cover a few – for a more comprehensive discussion please see http://www.burns-stat.com/pages/Tutor/R_inferno.pdf

16.2.1 Floating point comparison

Floating point arithmetic is not exact:

```
.1 == .3/3
```

Solution: `use all.equal()`:

```
all.equal(.1, .3/3)
```

16.2.2 Missing values

R does not exclude missing values by default – a single missing value in a vector means that many thing are unknown:

```
x <- c(1:10, NA, 12:20)
c(mean(x), sd(x), median(x), min(x), sd(x))
```

NA is not equal to anything, not even NA

```
NA == NA
```

Solutions: use `na.rm = TRUE` option when calculating, and is.na to test for missing

16.2.3 Automatic type conversion

Automatic type conversion happens a lot which is often useful, but makes it easy to miss mistakes

```
# combining values coereces them to the most general type
(x <- c(TRUE, FALSE, 1, 2, "a", "b"))
str(x)

# comparisons convert arguments to most general type
1 > "a"
```

Maybe this is what you expect... I would like to at least get a warning!

16.2.4 Optional argument inconsistencies

Functions you might expect to work similarly don't always:

```
mean(1, 2, 3, 4, 5)*5
sum(1, 2, 3, 4, 5)
```

Why are these different?!?

```
args(mean)
args(sum)
```

Ouch. That is not nice at all!

16.2.5 Trouble with Factors

Factors sometimes behave as numbers, and sometimes as characters, which can be confusing!

```
(x <- factor(c(5, 5, 6, 6), levels = c(6, 5)))

str(x)

as.character(x)
# here is where people sometimes get lost...
as.numeric(x)
# you probably want
as.numeric(as.character(x))
```

16.3 Feedback

- Help Us Make This Workshop Better!
- Please take a moment to fill out a very short feedback form
- These workshops exist for you – tell us what you need!
- http://tinyurl.com/RprogrammingFeedback

CHAPTER 17

Footnotes

CHAPTER 18

Indices and tables

- genindex
- modindex
- search