WEEK 3: SOURCE FILES AND PACKAGES

THOMAS ELLIOTT

1. Source Files

Source files function exactly like Stata Do files. They are plain text files containing lists of commands to execute. Source files typically end with the file extension `.R`. To run a source file within R, you use the `source()` command. For example, say we had the following commands in a source file we've saved as "my-source-file.R":

```
x<-c(1,2,3,4,5)
y<-c(6,7,8,9,10)

z<-x*y

print(z)
```

We can run all the commands in the source file with the following command:

```
> source("my-source-file.R")
[1]  6 14 24 36 50
```

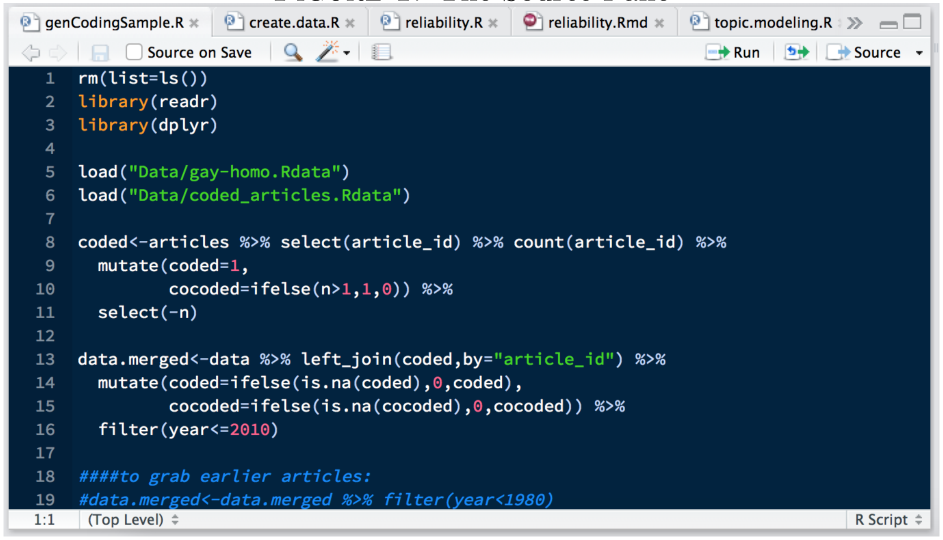
By default, sourcing a source file will not output the commands it is running. It will, however, output any messages, warnings, errors, and print statements. So in the source file above, the only thing that gets output to the console is the contents of z from the `print(z)` command. If you want to see the commands it is running, you can supply the `echo=TRUE` argument to the source command:

```
> source('my-source-file.R', echo=TRUE)

> x<-c(1,2,3,4,5)

> y<-c(6,7,8,9,10)

> z<-x*y

> print(z)
[1]  6 14 24 36 50
```

This is another area where RStudio is particularly helpful. As you've noticed by now, RStudio has four panes to show various information, outputs, etc. One of these panes is for working

Date: January 27, 2016.

FIGURE 1. The Source Pane



```r
rm(list=ls())
library(readr)
library(dplyr)

load("Data/gay-homo.Rdata")
load("Data/coded_articles.Rdata")

coded<-articles %>% select(article_id) %>% count(article_id) %>%
  mutate(coded=1,
         cocoded=ifelse(n>1,1,0)) %>%
  select(-n)

data.merged<-data %>% left_join(coded,by="article_id") %>%
  mutate(coded=ifelse(is.na(coded),0,coded),
         cocoded=ifelse(is.na(cocoded),0,cocoded)) %>%
  filter(year<=2010)

####to grab earlier articles:
#data.merged<-data.merged %>% filter(year<1980)
```

on source files (see Figure 1. This pane will do syntax highlighting, which means it can tell the difference between different parts of the code and display the text in different colors to help with reading your code. RStudio's source editor is also smart enough to figure out how things should be indented, and so will automatically indent lines of code the appropriate amount to facilitate reading your code. The source pane can have multiple source files open at once, and will display tabs to change between them. While you are working in the source pane, there are keyboard shortcuts to run the line your cursor is on in the console, making checking that your code works really easy.

1.1. **Logging Output.**

One unfortunate aspect of R is that it's logging capabilities are not as intuitive as Stata's. You can redirect console output to a text file using the `sink()` command, but it is easy mess things up with sink — again, R requires more user vigilance than Stata does. Sink works by creating a new output stream - think of what gets printed to the console as originating from a lake or dam. The console is one stream from the lake, but you can, on the fly, create new ones to redirect the output. To use `sink()` you insert the command where you want to start redirecting the output, supplying a filename as an argument for the text file where the output will be saved. By default, redirecting output means the output will NOT show up on the console, only the text file. If you want the output to show up in both, you can supply the `split=TRUE` argument. To stop redirecting the output, you can type `sink()` with no arguments, or `sink(NULL)`, which is equivalent. This closes the current redirection of the output. As an example, if we wanted to source our source file, echoing the commands, and outputting the results to both the console and a text file, we would type:

```
> sink("test.txt",split=TRUE)
> source('my-source-file.R', echo=TRUE)

> x<-c(1,2,3,4,5)

> y<-c(6,7,8,9,10)
> z<-x*y

> print(z)
[1]  6 14 24 36 50
> sink()
```

Now, the tricky thing with sink is that you can create new redirections while you are already redirecting output. For example, say you sink to a text file called "cleaning-log.txt" for a source file that cleans your data. You could, while that sink is still active, create a new sink that redirects output to "cleaning-errors.txt" where you output any errors the cleaning program found. At this point, typing `sink(NULL)` closes the redirection to "cleaning-errors.txt" and returns to the redirection to "cleaning-log.txt". As a result, you should pay careful attention to what redirections of output you already have in place as you write code in R. You can query the number of diversions of output with the command `sink.number()`, which returns the number of diversions that are currently active.

2. Packages

In Stata you can install third-party functions, often stored in ado files, to extend the capabilities of Stata. You can do the same thing with R using packages. In fact, what makes R so powerful is the richness of its packages archive. When you installed R, it came with some packages that were also installed and these are automatically loaded whenever you start R. Examples of these packages include the `stats` package, which contains the bulk of the basic statistical analysis functions, including `lm()`; and the `utils` package, which contains a bunch of functions for working in R, including the `help()` functions discussed last week. However, there are some packages you may want to use that are not automatically installed and loaded when you start R. Two such packages I strongly recommend installing and using are the `dplyr` and `tidyr` packages, which makes data wrangling much easier than base R. Once you know the name of the package you want to install, you can install it using the `install.packages()` command in R:

```
install.packages(c("dplyr","tidyr"))
```

If you are only installing one package, you can just enter the name of the package with quotes as an argument to `install.packages()`. However, if you want to install more than one package at once, you'll need to supply a character vector. You do this with the `c()` function, which combines values into a vector.

If this is the first time you are installing packages and you are not using RStudio, you will be asked which CRAN mirror to use. CRAN is the archive which stores the official versions of R packages, and it is from this archive that packages are downloaded and updated. You will want to choose a mirror that is close to you - you are more likely to get a faster connection to servers that are geographically closer to you than farther away. RStudio hosts its own CRAN mirror and so will automatically use this when installing packages, which is fine.

If you do not have administer privileges on the user you are currently logged into, you will likely get a message saying R cannot write to a folder where it defaults to installing packages. It should ask if you would like R to, instead, install to a folder in your user folder - this is fine to do, but means that the installed packages will not be accessible to other users on your computer. They will have to install the packages themselves, which is fine.

You only need to install a package once, but in order to use the functions in the package during a R session, you need to tell R to load the package into the workspace. You do this with the `library()` function. Once you've installed `dplyr` and `tidyr`, you can load them into your session with:

```
library(tidyr)
library(dplyr)
Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

Notice that some packages will output messages when they are loaded. In the above example, dplyr contains functions that are named the same as functions in other packages that are already loaded. Loaded packages stack, so that the most recently loaded package takes precedence over older loaded packages. When you use a function and R goes to find where that function is located, it traverses down this stack and will return the first function definition it finds. Thus, using the `filter()` function after loading `dplyr` will always return `dplyr`'s version of `filter()`, not `stats`'. This is *usually* not a problem, as package developers are usually good about not using the same names for functions as other common functions, but it is still something to pay attention to - the load order is important for packages. It is possible to use functions that have been masked by another package, by telling R which package it should be looking for the function in. For example, to use `stats`' filter function, you would type `stats::filter()` — the `stats::` is telling R to look in the stats package for the filter function.

2.1. Updating Packages.

You can check for updates to installed packages with the `update.packages()` function. This will check the version number of your installed packages with the version number of the package on your CRAN mirror and will offer to download any newer packages it finds.

2.2. Creating Your Own Packages.

Creating your own packages is a fairly easy process, made even easier by RStudio. I won't go into details about how to create packages here, but I highly recommend Hadley Wickham's book on creating R packages with RStudio, which you can read for free online here: `http://r-pkgs.had.co.nz`