R basics 2 - Automating analyses

author: Kevin Shook
date:
autosize: true

Automating your work

- Using scripts, you can do more work in less time
- Code can be re-used, making you more productive

Organizing your R code

Principles of RR

5. Organization!
   - Follow consistent approach/template for all projects
   - Easier to pick up where you last left off
   - Portability
   - Basic project directory layout

```
proj/
    code/       -function definition files
    data/       -all input data for project
    doc/        -the paper (thesis)
    figs/       -all figures generated by code
    output/     -all analyzed data generated by code
    analysis.R  -the single R script which runs everything
```

   - Only define working directory once

Projects

- It's a good idea to create a new **R** project for each new work project
- Can go in the code folder
- R studio command is `File|New project`
- Creates a file with the extension `` `.RProj ``
  - Contains all of the project settings
- Creates a `.Rhistory` file

    - Contains all of the commands you execute
- Creates a `.RData` file
    - Contains all of the variables in the session

Projects cont'd

- Projects allow the use of **git** to manage versions of files
- Also allow you to set options for just this project
- Once you have created a project file, just double-click to load it and run RStudio
- Will remember all of your project settings

R script files

- Text files with the extension .R
- Can be run in several ways
- All at once
- One line at a time

Loading R command files

- Typing `source(filename)` loads in a .R file and runs all commands
- Using the menu `File|OpenFile...` loads in .R file **without** automatically running the commands

Example #1

- Using `File|Open File`, load in file `Example1.R`
- Press **[Ctrl][Enter]** to step through the code one line at a time
- Now check `Source on Save`

- click on the save icon
- What happened?

Running all lines

- When you execute *all* of the lines, they are not echoed to the screen
- If you want to output a value, you have to tell **R** to do it
- Use the command `cat()` to print things out
- All of the values will be printed on one line
- Have to add line breaks using the symbol `"\n"`

Editing the file

class: small-code
- Make these changes to lines **8** and **9**:
Change the lines

```
actual_mean
```

```
actual_sd
```

to

```
cat("actual mean:", actual_mean, "\n")
cat("actual sd:", actual_sd, "\n")
```

- Save, and re-run
- Congratulations! You have now built an **R** program

Editing R script files

- R Studio has a very good built-in editor
- You can change the appearance (colours, fonts, other settings) using the menu `Tools|Global Options`

Programming in R

- **R** scripts are really computer programs
- Can include all of the elements of other programs
- data input and output
- branches
- loops
- functions

Functions

- Writing your scripts as functions makes them more repeatable
- Can build your own library of functions
- functions are not executed automatically when the file is sourced

```
funcName <- function(parameters) {

}
```

Parameters

- All function variables are separate from the rest of your **R** code
- Makes functions secure and repeatable
- You pass values into the function using parameters
- Parameters can have default values

Returning values

- A function can only return a single value
- Can be a scalar, vector, data frame etc.
- To return more than 1 value, put them in a list
- By default the last variable is returned
- sloppy and potentially dangerous
- use `` `return() `` instead

Function example

```
cv <- function(values) {
  coeff_of_var <- sd(values) / mean(values)
  return(coeff_of_var)
}
```

```
x <- runif(5)   # random numbers
x
```

```
[1] 0.9767598 0.2182011 0.5251503 0.8082567 0.1810018
```

```
cv(x)
```

```
[1] 0.6494285
```

Loading a function file

class: small-code

```
source('Example2.R')
saturatedVP
```

```
function (airTemp)
{
    if (airTemp <= 0) {
        estar <- 0.611 * exp((21.88 * airTemp)/(airTemp + 265.5))
    }
    else {
        estar <- 0.611 * exp((17.27 * airTemp)/(airTemp + 237.3))
    }
    return(estar)
}
```

```
saturatedVP(20)
```

```
[1] 2.339047
```

When to write a function?

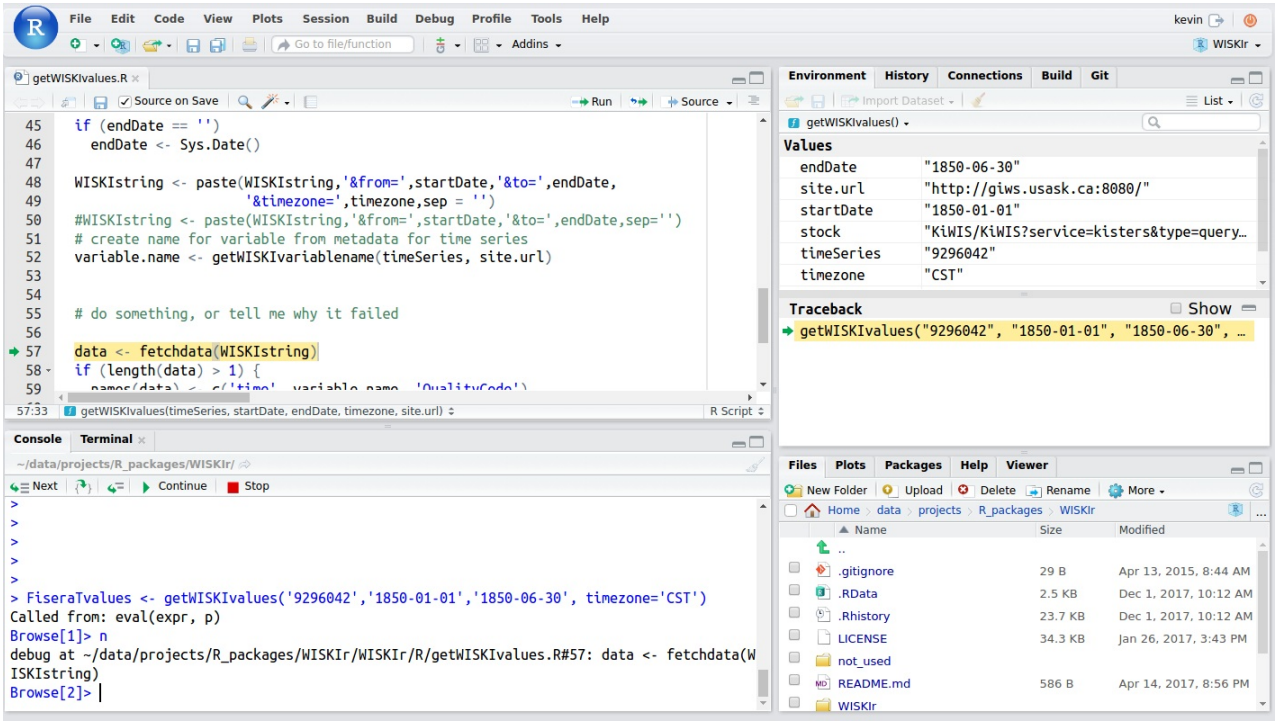
- If the code will be used several times
- especially if it will be used with different inputs
- or it will be used by someone else
- If the logic is very complex

- If it calls many other functions
- If the order of operation is complex
- This type of code needs a debugger

Debugger

- R has a built-in debugger
- Allows you to step through a function, examining the values of variables
- Set break-points by clicking on a line
- Save the file
- Run the function, passing the parameters
  - Will run to that line
  - Variable values are listed in the Environment pane
  - Step through the function with **[F10]**

Debugging



Complex statements

- These statements are usually used in **R** *programs*
- Control the order of execution of code
- Especially useful in functions

If statements

- need a condition which evaluates to be `TRUE` or `FALSE`
- any number of lines can be between the braces
- good style to indent

```
a <- 2
if (a %%2 == 0) {
  cat("even")
} else {
  cat("odd")
}
```

```
even
```

Loops

- **R** is interpreted, meaning that each line is converted to machine-language and then executed
- Much slower than compiled languages (C or Fortran)
- Loops are generally a bad idea, as they are very slow in **R**
- Can often be avoided, but sometimes you have to use them

For loops

- Used when loop will execute a fixed number of times

```
for (i in 1:5) {
  cat(i, " ")
}
```

```
1  2  3  4  5
```

Avoiding loops

- “If you’re using a loop, you’ve failed”

- **R** is written in C and Fortran
- Any function which loops *automatically* is much faster than doing it yourself
- Most functions can be applied automatically to all of the rows or columns in a data frame or matrix
- Some functions also do the types of things loops are often used for

ifelse

```
ifelse(test, true_val, false_val)
```

- applies test to each element in a variable and returns either the true or false value

cumsum() and diff()

class: small-code

```
x
```

```
[1] 0.9767598 0.2182011 0.5251503 0.8082567 0.1810018
```

```
cumulative <- cumsum(x)
cumulative
```

```
[1] 0.9767598 1.1949609 1.7201112 2.5283679 2.7093697
```

```
diff(cumulative)
```

```
[1] 0.2182011 0.5251503 0.8082567 0.1810018
```

Apply functions

- Some functions cannot use vectors or data frames
- For these, use the `apply` series of functions to loop over your values
- Much faster as looping is compiled
- `sapply` (simple apply) is the easiest to use

Example #2 - sapply() vs loop

```
load("R_basics_2.RData")
head(Saskatoon, 3)
```

```
             datetime  u.1   t.1 rh.1 ppt.1
1 1960-01-01 01:00:00 6.67 -13.3   84   1.3
2 1960-01-01 02:00:00 8.89 -12.8   85   0.0
3 1960-01-01 03:00:00 7.22 -12.8   83   0.0
```

```
rows <- nrow(Saskatoon)
rows
```

```
[1] 406607
```

Using a loop

```
system.time({for (i in 1:rows)
  Saskatoon$satVP[i] <- saturatedVP(Saskatoon$t.1[i])})
```

```
   user  system elapsed
 287.476  56.872 344.366
 ```

Using sapply()
==========

```r
system.time({Saskatoon$satVP <- sapply(Saskatoon$t.1, saturatedVP)})
```

```
   user  system elapsed
  0.420   0.000   0.418
```

Mixed language programming

- You can call code written in C, or Fortran from R
- Use the compiled code for speed
- Need to have a compiler

```
meanepsilon2d <- function(x, r, q){
  xsize <- as.integer(nrow(x))
  meanepsilon <- 0
  r <- as.integer(r)
  retdata <- .Fortran("meanepsilon2d", x, xsize, r, q, meanepsilon, PACKAGE = 'multifRactal')
  return(retdata[[5]])
}
```

A useful loop

class: small-code
- This will loop though all of the files in the directory whose names fit the specifications
- I have added it to my copy of **RStudio** as a snippet
- Overhead of looping is small compared to reading/processing data

```
filespec <- '*.csv'                # using wildcards
FilePattern <- glob2rx(filespec)   # wild cards to regular expression
FileList <- list.files(pattern = FilePattern)

NumFiles <- length(FileList)      # get number of file names
for (i in 1:NumFiles){            # loop through all file names
  filename <- FileList[i]
  ...
}
```

Summary

- The real power of **R** comes from writing your own code
- Eventually you will build up a library of frequenty-used functions
- We will be seeing how to combine your code with outputs, graphs and images
- produces very reproducible research

Questions/answers

- Bulk downloading data from EC?
- check out `downloadMSCobs` in package `MSCr`
- Several box-plots in one plot?
- check out `facet_grid` or `facet_wrap` in package `ggplot2`

- Navigating a directory tree?
- check out `list.files(recursive=TRUE)`

Questions/answers cont'd

- Huge data structures?
- check out https://www.r-bloggers.com/five-ways-to-handle-big-data-in-r/
- GUI for data QA/QC
- can build web apps using package `shiny`
- **R** is not the right program for complex GUI applications
- GUI applications in other languages can call **R** code