9. Writing Functions

Ken Rice
Thomas Lumley

Universities of Washington and Auckland

NYU Abu Dhabi, January 2017

In this session

One of the most powerful features of R is the user's ability to expand existing functions and write custom functions. We will give an introduction to writing functions in R.

- Structure of a function
- Creating your own function
- Examples and applications of functions

Introduction

Functions are an important part of R because they allow the user to customize and extend the language.

- Functions allow for reproducible code without copious/error prone retyping
- Organizing code into functions for performing specified tasks makes complex programs tractable
- Often necessary to develop your own algorithms or take existing functions and modify them to meet your needs

Structure of a function

Functions are created using the `function()` directive and are stored as R objects.

Functions are defined by;

1. A function name with assignment to the `function()` directive. (Function names can be almost anything. However, the usage of names of existing functions should be avoided.)

2. The declaration of arguments/variables 'passed' to the function

3. Finally, giving the operations (the function body) that perform computations on the provided arguments

Structure of a function

The basic structure of a function is:

```
my.func <- function(arg1, arg2, arg3, ...) {
 <commands>
 return(output.object)
}
```

- Function arguments (`arg1`, `arg2`, ...) are the objects 'passed' to the function and used by the function's code to perform calculations.
- The <`commands`> part describes what the function will do to `arg1`, `arg2`
- After doing these tasks, `return()` the output of interest. (If this is omitted, output from the last expression evaluated is returned)

Calling a function

Functions are called by their name followed by parentheses containing possible argument names.

A call to the function generally takes the form;

```
my.func(arg1=expr1, arg2=expr2, arg3=exp3, ...)
```

or

```
my.func(expr1, expr2, expr3, ...)
```

- Arguments can be 'matched' by name or by position (recall Session 2, and use of defaults when calling functions)
- A function *can* also take no arguments; entering `my.func()` will just execute its commands. This can be useful, if you do *exactly* the same thing repeatedly
- Typing just the function name *without* parentheses prints the definition of a function

Function body – more details

- The function body appears within {curly brackets}. For functions with just one expression the curly brackets {} are not required – but they may help you read your code

- Individual commands/operations are separated by new lines

- An object is returned by a function with the `return()` command, where the object to be returned appears inside the parentheses. Experts: you can `return()` from any place in the function, not just in the final line

- Variables that are created inside the function body exist *only* for the lifetime of the function. This means they are not accessible outside of the function, in an `R` session

Example: returning a single value

Here's a function for calculating the coefficient of variation (the ratio of the standard deviation to the mean) for a vector;

```
coef.of.var <- function(x){
   meanval <- mean(x,na.rm=TRUE) # recall this means "ignore NAs"
   sdval   <- sd(x,na.rm=TRUE)
   return(sdval/meanval)
   }
```

Translated, this function says "if you give me an object, that I will call `x`, I will store its `mean()` as `meanval`, then its `sd()` as `sdval`, and then return their ratio `sdval/meanval`."

Doing this to the `airquality`'s 1973 New York ozone data;

```
> data(airquality) # make the data available in this R session
> coef.of.var(airquality$Ozone)
[1] 0.7830151
```

Example: returning multiple values

A function can return multiple objects/values by using `list()` – which collects objects of (potentially) different types.

The function below calculates estimates of the mean and standard deviation of a population, based on a vector (`x`) of observations;

```
popn.mean.sd <- function(x){
      n        <- length(x)
      mean.est <- mean(x,na.rm=TRUE)
      var.est  <- var(x,na.rm=TRUE)*(n-1)/n
      est      <- list(mean=mean.est, sd=sqrt(var.est))
      return(est)
}
```

- The in-built `var()` applies a bias correction term of $n/(n-1)$, which we don't want here
- Easier to write a new function than correct this every time

Example: returning multiple values

Applying our `popn.mean.sd()` function to the daily ozone concentrations in New York data;

```
> results <- popn.mean.sd(airquality$Ozone)
> attributes(results)  #list the attributes of the object returned
$names
[1] "mean" "sd"
> results$mean
[1] 42.12931
> results$sd
[1] 32.8799
```

- Elements of lists can also be obtained using *double* square brackets, e.g. `results[[1]]` or `results[[2]]`.
- Can also use `str()` to see what's in a list

Declaring functions within functions

Usually, functions that take arguments, execute R commands, and return output will be enough. But functions can be declared and used inside a function;

```
square.plus.cube <- function(y) {
   square <- function(x) { return(x*x) }
   cube   <- function(x) { return(x^3) }
   return(square(y) + cube(y))
 }
```

Translated; "if you given me a number, that I will call `y`, I will define a function I call `square` that takes a number that *it* calls `x` and returns `x`-squared, then similarly one I call `cube` that cubes, then I will return the sum of applying `square` to `y` and `cube` to `y`".

```
> square.plus.cube(4)
[1] 80
```

Example: function returning a function

And functions can also return other functions, as output;

```
make.power <- function(n){
   pow <- function(x){x^n}
   pow
   }
```

Translated; "if you given me a number, that I will call `n`, I will define a function that takes a number that *it* calls `x` and raises `x` to the `n`th power, and I will return this function".

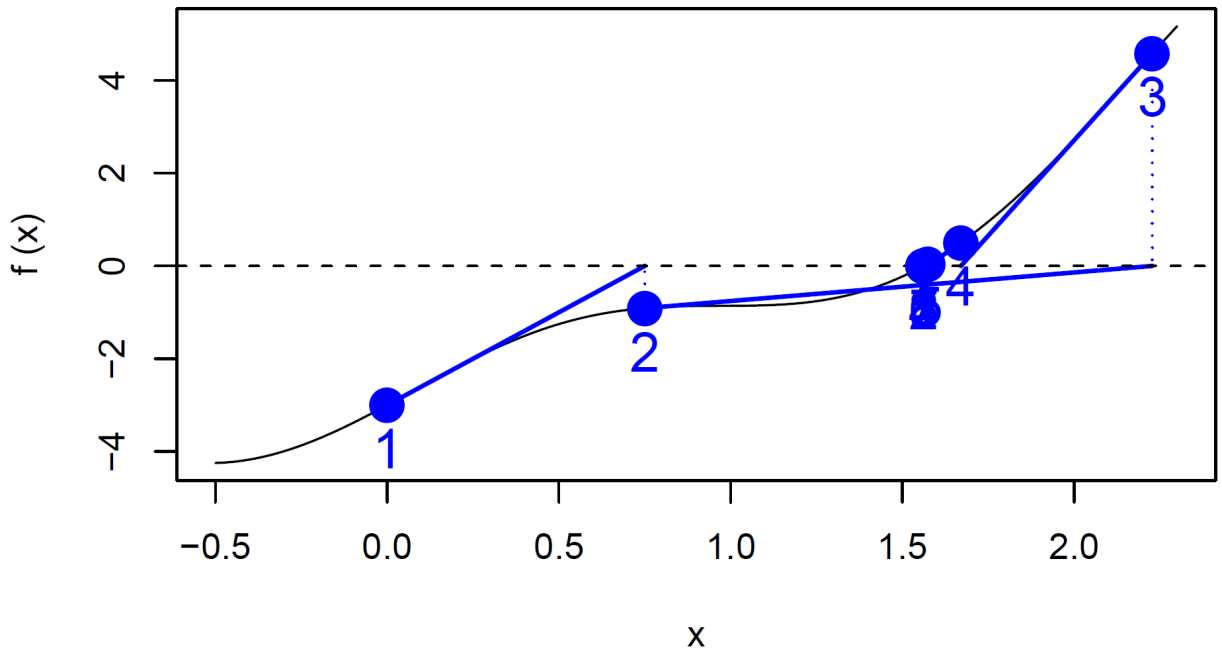
```
cube <- make.power(3)
square <- make.power(2)
> cube(3)
[1] 27
> square(3)
[1] 9
```

Example: functions as arguments

Functions can take other functions as arguments. This is helpful with finding *roots* of a function; values of x such that $f(x) = 0$.

The *Newton-Raphson* method finds roots of $f(x) = 0$ by the following iteration procedure:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$



Example: functions as arguments

A function to implement the Newton-Raphson method, given input of arguments, a place to start, and convergence tolerance:

```
newton.raphson <- function(f,fprime,x0,thresh){
   myabsdiff <- Inf
   xold      <- x0
   while(myabsdiff>thresh){ # have we converged yet? If no, move;
      xnew      <- xold-f(xold)/(fprime(xold))
      myabsdiff <- abs(xnew-xold)
      xold      <- xnew
   }
return(xnew)
}
```

- `Inf` is (positive) infinity – here, it ensures we go round the loop at least once
- Recall we saw `while()` loops in Session 6
- We could also use `repeat()` here

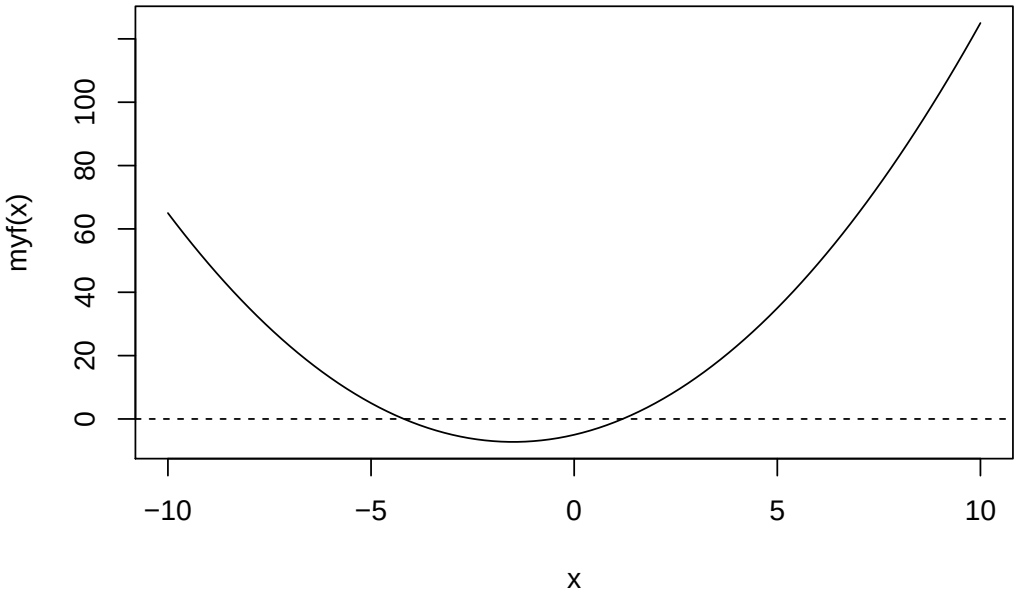
Example: functions as arguments

We'll find the roots of $f(x) = x^2+3x-5$, using Newton-Raphson. We need the derivative of $f(x)$: $f'(x) = 2x + 3$

```
myf      <- function(x){ x^2 + 3*x - 5 }
myfprime <- function(x){ 2*x + 3 }
```

We use the `newton.raphson()` function with initial value of 10 and a convergence threshold of 0.0001 to obtain a root:

```
> newton.raphson(f=myf,fprime=myfprime,x0=10,thresh=0.0001)
[1] 1.192582
```



How did we do?

$$\frac{-b \pm \sqrt{b^2-4ac}}{2a} = \frac{-3 \pm \sqrt{3^2+4\times 5}}{2} \approx -4.19, 1.19$$

(Try other values of `x0` to find the other root)

Tips for writing functions

- Avoid rewriting the same code... use functions!
- Modularize as much as possible: write function that call other functions. (Start with the low-level ones)
- Test your functions: use data/arguments for which you know the results to verify that your functions are working properly
- Later on: provide documentation, including detailed comments describing the procedures being conducted by the functions, especially for large, complex programs
- Use *meaningful* variable and function names

Summary (so far)

- User-defined functions are easy to create in R, with `my.fun <- function(argument list)`
- Arguments of a function are allowed to be practically any R object including lists, numeric vectors, data frames, and functions
- In functions calls, arguments are matched by name or by position
- An object can be returned by a function with `return()`. If `return()` is not invoked, the last evaluated expression in the body of a function will be returned.
- `list()` can be used for returning multiple values

Shiny: a quick look

It's also possible to display data analyses on websites – and have them be interactive. The `shiny` package, by RStudio, builds 'apps' that do this - using function definitions, in scripts.

The syntax is (roughly) a hybrid of R and HTML, so we give just a short example, showing off the `salary` data again*.

To make an app, in a directory named for your app, you need two files;

- **ui.R** This R script controls the layout and appearance of your app
- **server.R** This script contains the instructions that your computer needs to build your app

For more, see Shiny's excellent online tutorial.

Shiny: ui.R

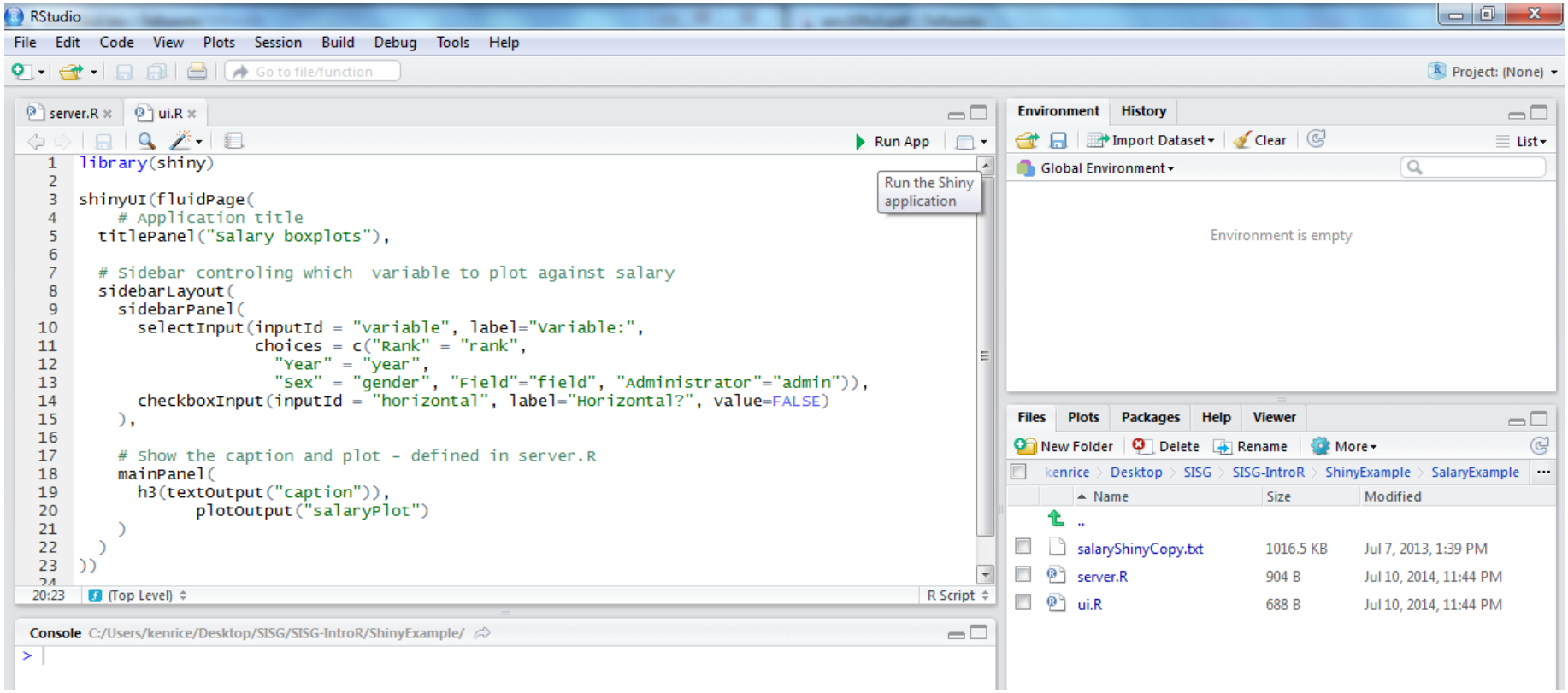
```
library("shiny") # after installing it
shinyUI(fluidPage(
    # Application title
  titlePanel("Salary boxplots"),

  # Sidebar controlling which  variable to plot against salary
  sidebarLayout(
    sidebarPanel(
      selectInput(inputId = "variable", label="Variable:",
                  choices = c("Rank" = "rank", "Year" = "year",
                              "Sex" = "gender", "Field"="field",
                              "Administrator"="admin")
                  ),
      checkboxInput(inputId = "horizontal", label="Horizontal?", value=FALSE)
      ),
    # Show the caption and plot - defined in server.R
    mainPanel(
      h3(textOutput("caption")),
            plotOutput("salaryPlot")
    ) # close main Panel
  )   # close sidebarLayout
))
```

Shiny: server.R

```
library("shiny")
# first, a local copy of salary data sits in same directory
salary <- read.table("salaryShinyCopy.txt", header=T)

# make some variable factors - for prettiness
salary$year  <- factor(salary$year)
salary$admin <- factor(salary$admin)

# Define server "logic" required to plot salary vs various variables
shinyServer(function(input, output) {

  # Compute the formula text in a "reactive expression"
  # it is shared by  output$caption and output$mpgPlot, below
  formulaText <- reactive({ paste("salary ~", input$variable) })

  # Return the formula text for printing as a caption
  output$caption <- renderText({ formulaText() })

  # Do the boxplot, using the formula syntax, and setting horizontal=T/F
  output$salaryPlot <- renderPlot({
    boxplot(as.formula(formulaText()),
            data = salary, horizontal = input$horizontal) })
}) # close function
```

Shiny: making it work in Rstudio

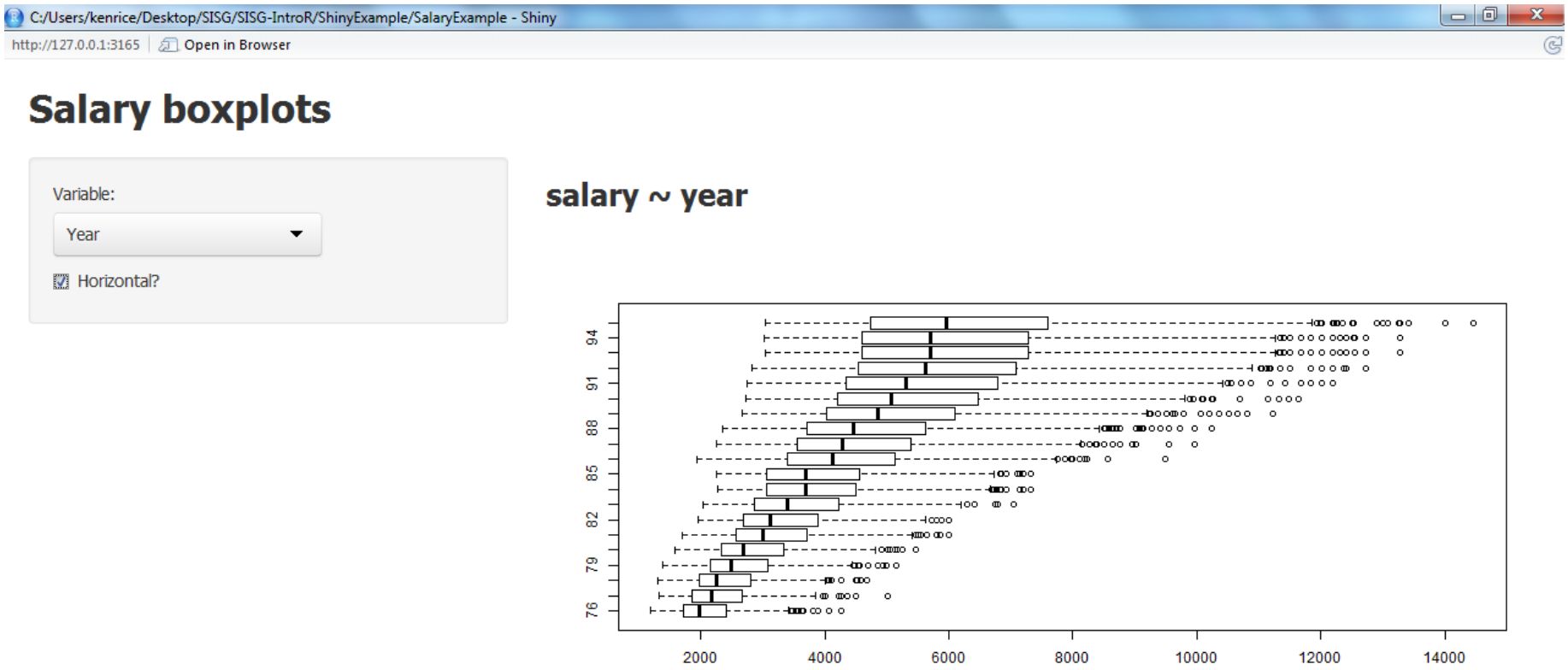
This is remarkably straightforward;



- Hit 'Run App' – and it (should) run
- Note that `ui.R`, `server.R` and the `salaryShinyCopy.txt` data file are *all* in the `SalaryExample` directory

Shiny: making it work in Rstudio

The (interactive) output should look something like this;



- Expect mild differences, across systems
- To share your app online, go to https://www.shinyapps.io/ – registration is needed. [Online example]
- Be careful with personal data!

Shiny: making it work in Rstudio

A final example; first making the familiar mammals plot;

```
mammals <- read.table(
   "http://faculty.washington.edu/kenrice/rintro/mammals.txt", header=TRUE)
plot(log(brain)~log(body), data=mammals) # usual plot
```

Writing a function that locates the nearest point to where you click, adds its name, and then looks up that animal's name on Google images

```
showme <- function(){
   mychoice <- identify(y=log(mammals$brain), x=log(mammals$body),
                        labels=row.names(mammals), n=1)
   myURL <- paste("http://images.google.com/images?q=",
             row.names(mammals)[mychoice], sep="")
   shell.exec(myURL)
   }
```

And putting this in a one-line loop;

```
for(i in 1:10){ showme() }
```

What next?

This concludes our course. To learn more;

- Take another one! Elements of R follows on, with genetics/bioinformatics examples (and lots of programming)
- See the recommended books, on the course site – the course site remains 'up'
- To find simple examples, Google is a good place to start
- There are several R mailing lists; R-help is the main one. *But* contributors expect you to have read the documentation – all of it! CrossValidated is friendlier to beginners
- Emailing package authors may also work
- For questions about *any* software, say;
  - What you did (ideally, with an example)
  - What you expected it to do
  - What it did instead