Lecture 6: Functions in R

STAT598z: Intro. to computing for statistics

Vinayak Rao

Department of Statistics, Purdue University

In [ ]:
```
options(repr.plot.width=3, repr.plot.height=3)
```

Why functions?

R comes with its own suite of built-in functions

- An important part of learning R is learning the vocabulary, e.g. http://adv-r.had.co.nz/Vocabulary.html (http://adv-r.had.co.nz/Vocabulary.html)

Non-trivial applications require you build your own functions

- Reuse the same set of commands
- Apply the same commands to different inputs
- Cleaner, more modular code
- Easier testing/debugging

Creating functions

Create functions using function :

```
my_func <- function( formal_arguments ) body
```

The above statement creates a function called `my_func`

formal_arguments: comma separated names

- describe inputs `my_func` expects

function_body: a statement or a block

- describes what `my_func` does with inputs

Example functions

In [ ]:
```
my_add <- function(x,y) x+y
```

In [ ]:
```
gauss_pdf <- function(ip, mn, vr, lg_pr) {
# Calculate the (log)-probability of Gaussian with mean m and variance vr
  rslt <- -((ip-mn)^2)/(2*vr)
  rslt <- rslt - 0.5*log(2*pi*vr)
# Do we want the prob or the log-prob?
  if(lg_pr == F) rslt <- exp(rslt)
  return(rslt)
}
```

In [ ]:
```
print(gauss_pdf(1,0,1,F)); dnorm(1, log=F)
```

In [ ]:
```
normalize_mtrx <- function( ip_mat, row ) {
# Normalizes rows to add up to one if row = TRUE
# Else normalizes columns
  if(row) { # We want the rows to add up to one
    rslt <- ip_mat / rowSums(ip_mat)
  } else { # We want the columns to add up to one
    rslt <- t( t(ip_mat) / colSums(ip_mat))
  }
  return(rslt) # Works even without this
}
```

In [ ]:
```
mtrx <- matrix(runif(9), nrow=3); mtrx
```

In [ ]:
```
n_mtrx <- normalize_mtrx(row = TRUE, ip_mat = mtrx); n_mtrx
```

In [ ]:
```
n_mtrx <- normalize_mtrx(TRUE, mtrx)
```

In [ ]:
```
n_mtrx <- normalize_mtrx(TRUE, ip = mtrx) # Partial matching
```

`gauss_pdf` is an object:

In [ ]:
```
typeof(gauss_pdf)
```

In [ ]:
```
class(gauss_pdf)
```

In [ ]:
```
str(gauss_pdf)
```

Expects three numerics and a boolean input, and returns a numeric

A function can accept/return any object:

- this includes other functions
- multiple return values can be organized into vectors/lists/dataframes

Can add some defaults and checks

In [ ]:
```
normalize_mtrx <- function( ip_mat, row = TRUE ) {
  # Normalizes columns to add up to one if row = FALSE
  # If row = TRUE or row not specified, normalizes columns
  if(!is.matrix(ip_mat)) {
    warning("Expecting a matrix as input");
    return(NULL)
  }
  # You can define objects inside a function
  # You can even define other functions
  rslt <- if(row) ip_mat / rowSums(ip_mat) else
            t( t(ip_mat) / colSums(ip_mat))
}
```

In [ ]:
```
n_mtrx <- normalize_mtrx(mtrx)
```

In [ ]:
```
gauss_pdf <- function(ip, mn=0, vr=1, lg_pr=T) {
# Calculate the (log)-probability of Gaussian with mean m and variance vr
  if(vr <= 0) {
    warning("Expect a positive variance");
    return(NULL)
  }
  rslt <- -((ip-mn)^2)/(2*vr)
  rslt <- rslt - 0.5*log(2*pi*vr)
# Do we want the prob or the log-prob?
  if(lg_pr == F) rslt <- exp(rslt)
  rslt
}
```

In [ ]:
```
pr <- gauss_pdf(1,0,1); print(exp(pr))
```

In [ ]:
```
my_add <- function(x,y) {return(x+y)}
```

In [ ]:
```
my_mul <- function(x,y) x*y
```

In [ ]:
```
my_gen <- function(ip_fun, x) function(z) ip_fun(x,z)
```

In [ ]:
```
inc3 <- my_gen(my_add,3)
```

In [ ]:
```
inc3(5)
```

Argument matching

Proceeds by a three-pass process

- Exact matching on tags
- Partial matching on tags: multiple matches gives error
- Positional matching

Any remaining unmatched arguments triggers an error

In [ ]:
```
mean(,TRUE,x=c(1:10,NA)) # From Advanced R, Hadley Wickham
```

Arguments via '...'

'...' allows any number of arguments

Useful when passing arguments to other functions:

```
pick_func <- function (two_arg, ...) {
   # Function w/ 2 arguments
   if(two_arg) two_arg_fun(...) else
   # Function w/ 3 arguments
     three_arg_fun(...)
 }
```

Example: Recursive addition via functional programming

In [ ]:
```
recurse_sum <- function(x = TRUE, ...) # Cute but inefficient
                    if(isTRUE(x)) 0 else x + recurse_sum(...)
```

In [ ]:
```
recurse_sum(1,2,3,5,6,7) # Don't include TRUE in the input!
```

Note the use of isTRUE() above

Scoping rules

We saw a function `recurse_sum()` that called itself

This raises a few questions:

- what objects are visible to a function?
- what happens when a function makes assignments?

R decides this by following a set of scoping rules

R follows what is called *lexical scoping*

Function objects have attributes

- **formals**: its arguments
- **body**: its code
- **environment**: what objects exist

In [ ]:
```
body(recurse_sum)
```

In [ ]:
```
formals(recurse_sum)
```

In [ ]:
```
environment(recurse_sum)
```

`environment`: data-structure that binds names to values

Determines scoping rules in R

Environments in R

An environment is a kind of named list of symbol-value pairs

In [ ]:
```
x <- 5; env <- environment(); env
```

In [ ]:
```
env$x
```

In [ ]:
```
func1 <- function() {my_local <- 1; environment()}
```

In [ ]:
```
(local_env <- func1())
```

In [ ]:
```
local_env$my_local
```

In [ ]:
```
parent.env(local_env) # Each environment has a parent environment
```

Lexical scoping:

- To evaluate a symbol R checks current environment
- If not present, move to parent environment and repeat
- Value of the variable at the time of calling is used
- Assignments are made in current environment (but see `<<-`, the super-assignment operator)

Here, environments are those at time of definition

Where the function is defined (rather than how it is called) determines which variables to use

Values of these variables at the time of calling are used

Scoping in R

In [ ]:
```
x <- 5
func1 <- function(x) {x + 1}
```

In [ ]:
```
func1(1)
```

In [ ]:
```
x <- 5; func2 <- function() {x + 1}
```

In [ ]:
```
func2(); x
```

In [ ]:
```
x <- 10; func2() # use new x or x at the time of definition?
```

In [ ]:
```
x <- 1; y <- 10
func3 <- function() {x <- x + 1; y <<- y + 1; environment()}
env <- func3()
```

In [ ]:
```
c(x, y, env$x)
```

In [ ]:
```
func1 <- function(x) {x + 1}
func4 <- function(x) {func1(x)}
```

In [ ]:
```
func4(2)
```

In [ ]:
```
x <- 5; func2 <- function() {x + 1}
func5 <- function(x) {func2()}
```

In [ ]:
```
func5(2) # func2 uses x from calling or global environment?
```

Scoping in R

For more on scoping, see (Advanced R, Hadley Wickham)

The bottomline

- Avoid using global variables
- Always define and use clear interfaces to functions
- *Warning*: we're always implicitly using global objects in R

```
'+' <- function(x,y) x*y #Open new RStudio session!
2 + 10
```

Lazy evaluation: R evaluates arguments only when needed

Can also cause confusion

In [ ]:
```
func <- function(x,y) if(x) 2*x else x + 2*y
```

In [ ]:
```
func(1, {print("Hello"); 5})
```

In [ ]:
```
func(0, {print("Hello"); 5})
```

Some comments on the homework

functions: modular blocks of code that map input arguments to output (and sometimes have side-effects e.g. plotting)

Should not use global variables!

- Except in very special situations, this is just laziness
- Will eventually cause you trouble (and cost you points)

Functions should produce same output for same input, irrespective of values of non-input variables

Some exceptions are functions

- that are random (but see `set.seed()`)
- that read files, accept user input
- that read date/time/local system information

```
my_func <- function(mf_arg1, mf_arg2) {
  # Stuff not involving information other than
  # mf_arg's
}
```

- Give your function an informative name
- Needn't prefix all local variables, but helps against typos
- Arguments are placeholders for actual supplied inputs

If you're going to change R datasets, make local copies

Bad:

```
USArrests['Indiana'] <- USArrests['Indiana'] + 1
# What happens next time you run your script?
# What if you want the original value?
```

Good:

```
my_USArrests <- USArrests
my_USArrests['Indiana'] <- USArrests['Indiana'] + 1
```

Bad: Hacking away at the console and later trying to reconstruct how you got your output

Good: Work with a text file in the editor

- This allows you to see the structure of your program
- Encourages use of functions
- Encourages comments
- Maybe start by thinking of your plan of attack, write down a skeleton of your program and then fill it in?
- Bad idea to dive into details without a broader plan

Use the console to explore outcome of one line, check help/syntax, but once successful, add line to editor

RStudio shortcuts

While working at the editor, <Ctrl><Enter>

- executes selected lines
- executes current line if none are selected

<Ctrl><Shift><Enter>: executes all lines

<Ctrl><1> and <Ctrl><2>: Move cursor to editor or console

Also <Tab> autocompletes, <Up> moves through command history, and <Ctrl><Up> autocompletes from command history

For more:

- <ALT><SHIFT><k> (Tools>Keyboard Shortcuts Help)
- https://support.rstudio.com/hc/en-us/articles/200711853-Keyboard-Shortcuts (https://support.rstudio.com/hc/en-us/articles/200711853-Keyboard-Shortcuts)

Ideally, instead of submitting a script, wrap it up in a function Assigning variables won't mangle someone else's namespace

```
# Homework 1A
Lots of variable assignments
result <- ...
```

Better:

```
homework_1a <- function(ip_data) {
  # Helpful comment
  Lots of variable assignments
  result <- ...
}
homework_1a(USArrests)
```