Lecture 6: Functions in R

STAT598z: Intro. to computing for statistics

Vinayak Rao

Department of Statistics, Purdue University

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

An example function

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)
```

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

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

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

`normalize_mtrx` is an object:

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

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

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

Expects a matrix and boolean input, and returns a matrix

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 [ ]:

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

Each environment has a parent environment

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

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

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

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

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

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

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; y <- 6
func1 <- function(x) {x + 1}
func1(1)
```

In [ ]:

```
func2 <- function() {x + 1}
func2()
x <- 10; func2() # use new x or x at the time of definition?
```

In [ ]:

```
func3 <- function() {x <- x + 1; y <<- y + 1; environment()}
env <- func3()
c(x, y, env$x)
```

In [ ]:

```
func4 <- function(x) {func1(x)}
func4(2)
```

In [ ]:

```
func5 <- function(x) {func2()}
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: you are always implicitly using global objects in R

In [ ]:
```
'+' <- function(x,y) x*y # Open a 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})
```