Statistics 360: Advanced R for Data Science

Lecture 4

Becky Lin

R Functions

Scoping

Lazy evaluation and ...

Exiting a function

Function forms

Functional Programming Basics

Digging deeper into functions

- Reading: Text, chapter 5, sections 6.4-6.8
- Topics:
  - more on scoping (finding objects)
  - lazy evaluation and variable arguments with ...
  - exiting a function
  - prefix, infix, replacement and special function forms

R Functions

R function fundamentals

- Reading: text sections 6.1 and 6.2
- In R, functions are objects with three essential components:
  - the code inside the function, or `body`,
  - the list of arguments to the function, or `formals`, and
  - an `environment` that contains all objects defined in the function.
- Functions can have other attributes, but the above three are essential.

Example function

```r
f <- function(x) {
  return(x^2)
}
f
```

```
## function(x) {
##   return(x^2)
## }
```

The function body

- This is the code we want to execute.
- When the end of a function is reached without a call to `return()`, the value of the last line is returned.
  - So in our example function, we could replace `return(x^2)` with just ‘x^2.
- Use `body()` to see the body of a function.

```
body(f)
```

```
## {
##     return(x^2)
## }
```

The function formals

- These are the arguments to the function.
- Function arguments can have default values and/or be defined in terms of other arguments.

```
f <- function(x=0) { x^2}
f <- function(x=0,y=3*x) { x^2 + y^2 }
f()
```

```
## [1] 0
```

```
f(x=1) # missing y, so y=3*x=3, results in: 1^2+3^2
```

```
## [1] 10
```

```
f(y=1) # missing x, default x=0, so it calls f(x=0,y=1): 0^2+1^2
```

```
## [1] 1
```

```
f(x=1,y=2) # 1^2+2^2
```

```
## [1] 5
```

- Use `formals()` to see the formals of a function and their default values.

```
formals(f)
```

```
## $x
## [1] 0
##
## $y
## 3 * x
```

Argument matching when calling a function

- When you call a function, the arguments are matched first by name, then by "prefix" matching and finally by position:

```
f <- function(firstarg,secondarg) {
  firstarg^2 + secondarg
}
f(firstarg=1,secondarg=2) # same as f(secondarg=2,firstarg=1)
```

```
## [1] 3
```

```
f(secondarg=2,firstarg=1)
```

```
## [1] 3
```

```
f(s=2,f=1) # match by prefix
```

```
## [1] 3
```

```
f(2,f=1) # match by prefix first then by position
```

```
## [1] 3
```

```
f(1,2) # match by positions
```

```
## [1] 3
```

The function environment

- R creates an environment (with `rlang::env()`) within each function call to hold its variables.
- Initially includes the formals, but variables created within the function are also stored in this environment

```
f <- function(x) {
  y <- x^2
  ee <- environment() # Returns ID of environment w/in f
  print(ls(ee)) # list objects in ee
  ee
}
# After a function call its environment is usually discarded
my_ee <- f(1:3) # but not if you bind it to a name
```

```
## [1] "ee" "x"  "y"
```

```
my_ee$new_var <- 100
x <- 1:100
ls(my_ee)
```

```
## [1] "ee"      "new_var" "x"       "y"
```

```r
environment(f)
```

```
## <environment: R_GlobalEnv>
```

Enclosing environments

- Our function `f` was defined in the global environment, `.GlobalEnv`, which "encloses" the environment within `f`.
- If `f` needs a variable and can't find it within `f`'s environment, it will look for it in the enclosing environment, and then the enclosing environment of `.GlobalEnv`, and so on.
- The `search()` function lists the heirarchy of environments that enclose `.GlobalEnv`.

```
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics
## [4] "package:grDevices" "package:utils"     "package:datasets
## [7] "package:methods"   "Autoloads"         "package:base"
```

- To facilitate this search, each environment includes a pointer to its enclosing environment.

R packages and the search list

- Use the `library()` command to load packages.
- When we load a package it is inserted in position 2 of the search list, just after `.GlobalEnv`.

```
# install.packages("hapassoc")
library(hapassoc)
search()
```

```
##  [1] ".GlobalEnv"         "package:hapassoc"  "package:stats"
##  [4] "package:graphics"   "package:grDevices" "package:utils"
##  [7] "package:datasets"   "package:methods"   "Autoloads"
## [10] "package:base"
```

Detaching packages

- Detach a package from the search list with `detach()`

```
detach("package:hapassoc")
search()
```

```
## [1] ".GlobalEnv"         "package:stats"      "package:graphics"
## [4] "package:grDevices" "package:utils"      "package:datasets"
## [7] "package:methods"   "Autoloads"          "package:base"
```

Package namespaces

- Package authors create a list of objects that will be visible to users when the package is loaded. This list is called the package namespace.
- You can access functions in a package's namespace without loading the package using the `::` operator.

```
set.seed(321)
n<-30; x<-(1:n)/n; y<-rnorm(n,mean=x); ff<-lm(y~x)
car::sigmaHat(ff)
```

```
## [1] 0.926726
```

- Doing so does not add the package to the search list.

Scoping

Lexical scoping in R

- We have already touched on the essence of scoping in R: When a computation needs an object we start by looking in the current environment, and then search successive enclosing environments.
- More formally R has four rules:
  - Functions
  - Name masking
  - Functions versus variables
  - A fresh start
  - Dynamic lookup

Name masking

- A consequence of the search order for objects is that names defined *inside* a function mask names defined *outside*.

```r
x <- y <- 200
z <- 30 # defined in global environment
f <- function() { # f's env enclosed by global
  x <- 100 # defined in f's environment
  y <- 20
  g <- function() { #g's env enclosed by f's
    x <- 10 # defined in g's environment
    c(x,y,z)
  }
  g()
}
f()
```

```
## [1] 10 20 30
```

Functions *vs* names

- R does the "right thing" when you (stupidly) use the same name for a function and variable.

```
g09 <- function(x) x + 100
g10 <- function() {
  g09 <- 10
  g09(g09)
}
g10()
```

```
## [1] 110
```

Each function call gets a new environment

- As we saw in previous slides, function calls create an environment. On exit, this environment is (typically) unbound and will disappear.

```r
x <- 100
f <- function(){
  print(environment())
  x <- x+1
  x
}
f()
```

```
## <environment: 0x7fbf7e3b4748>
```

```
## [1] 101
```

```r
f()
```

```
## <environment: 0x7fbf7e40cc88>
```

```
## [1] 101
```

Dynamic lookup

- Be aware that functions only look for objects when run (dynamic lookup), not when created (static lookup).
- If a function gets an object from an enclosing environment, it will return different results whenever the object in the enclosing environment changes.
  - This may be what you intend, but it's also a common source of errors. What if I meant to define y in f() but forgot:

```
y <- 100
f <- function(x) {
  x + y
}
f(1)
```

```
## [1] 101
```

```
y <- 200
f(1)
```

```
## [1] 201
```

- consider adding `rm(list=ls())` to the start of your scripts

Lazy evaluation and . . .

Lazy evaluation

- Function arguments are only evaluated when needed.
    - The text describes how lazy evaluation is implemented (Section 6.5), but we will not discuss the details.

```
f<-function(xx,yy) {
  xx
}
f(1) # no value for yy, but OK since yy not used
```

```
## [1] 1
```

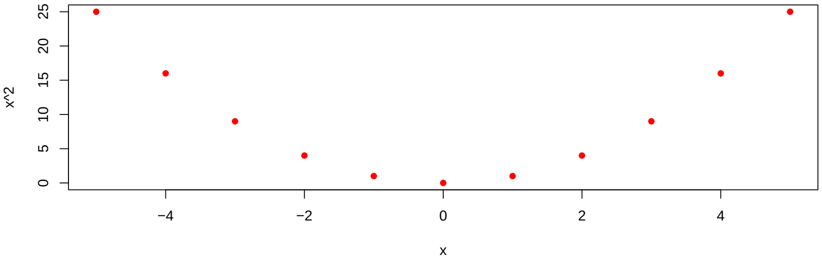
```
try(f(yy=1)) # xx is needed
```

```
## Error in f(yy = 1) : argument "xx" is missing, with no defaul
```

Variable arguments with ...

- The special function argument `...` (dot-dot-dot) allows a function to take any number of arguments.
- A typical use is to pass these to another function, as in the following example.

```
myplot <- function(x,...) {
  plot(x,x^2,...) # pass any args not named x to plot
}
myplot((-5:5),col="red",pch=16)
```



Exiting a function

Exiting a function

- Functions can exit explicitly with `return()` or implicitly, where the last expression in the function is its return value
- When a function returns, explicitly or implicitly, the default is to print the return value.
  - You can suppress this with `invisible()`.

```
ff <- function(x) { x }
ff(1)
```

```
## [1] 1
```

```
ff_invis <- function(x) { invisible(x) }
ff_invis(1) # but x <- ff_invis(1) same as x <- ff(1)
```

Signalling conditions

- Functions can signal error, warning or message conditions with `stop()`, `warning()` and `message()`, respectively.
  - `stop()` stops execution, `warning()` and `message()` don't
- These signals can be "handled" by ignoring them
  - ignore errors with `try()`
  - ignore warnings with `suppressWarnings()`
  - ignore messages with `suppressMessages()`
- or implementing a custom handler that over-rides the default behaviour of a condition
  - see Chapter 8 of the text if you are interested in learning more about handling conditions
- We restrict attention to (i) signalling and (ii) cleaning up any changes to the R session before exiting.

stop()

- If your function encounters an error, use `stop()` to stop and print an error message, also called “throwing” an error.

```
centre <- function(x,method) {
  switch(method,mean=mean(x),median=median(x),
         stop("method ",method," not implemented"))
}
try(centre(1:10,"mymean"))
```

```
## Error in centre(1:10, "mymean") : method mymean not implement
```

warning()

- If you suspect an error but can proceed without stopping, throw a `warning()` instead.

```
centre <- function(x,method) {
  switch(method,mean=mean(x),median=median(x),
         {warning("\nmethod ",method,
                  " not implemented, using mean\n");
          mean(x)})
}
centre(1:10,"mymean")
```

```
## Warning in centre(1:10, "mymean"):
## method mymean not implemented, using mean

## [1] 5.5
```

message()

- If you don't think the condition warrants a warning, you can issue a message.

```
centre <- function(x,method) {
  switch(method,mean=mean(x),median=median(x),
        {message("\nmethod ",method,
                 " not implemented, using mean\n");
          mean(x)})
}
centre(1:10,"mymean")
```

```
##
## method mymean not implemented, using mean
```

```
## [1] 5.5
```

Cleaning up with exit handlers

- An R session has a "global state" of options and parameters that control default behaviour.
  - type `options()` or `par()` to see some of these
- If your function temporarily modifies the global state, you can use an exit handler to re-set, even if your function stops.
  - Use `add=TRUE` to add more than one handler.

```
rplot <- function(y,x){
  opar <- par(mfrow=c(2,2))
  on.exit(par(opar),add=TRUE)
  plot(lm(y~x)) #could throw an error
}
y <- rnorm(100); x <- rnorm(10) # different length
try(rplot(y,x)) # Fails, but re-sets par mfrow
```

```
## Error in model.frame.default(formula = y ~ x, drop.unused.lev
##   variable lengths differ (found for 'x')
```

Function forms

Function forms

- We have been writing “prefix” functions, with a function name followed by arguments.
- Other forms are “infix”, “replacement” and “special”.
- We will cover each form very briefly; see the text, section 6.8 for more details.

Infix functions

- An infix function has two arguments and is called by putting the name between arguments, as in `x+y`.
  - `x+y` calls + as `` `+`(x,y) ``
  - + and - are special infix functions that can be called with only one argument
  - You can define your own infix function by enclosing the function name in %.

```
`%-%` <- function(set1,set2){
  setdiff(set1,set2)
}
s1 <- 1:10; s2 <- 4:6
s1 %-% s2 # same as `%-%`(s1,s2)
```

```
## [1]  1  2  3  7  8  9 10
```

Replacement functions

- Replacement functions are called to change values.
  - For example, change values of attributes of objects
- Must have arguments `x` and `value`, and must return the modified object.
- They are made to look like prefix functions, and may have prefix counterparts.

```
x <- c(a=1,b=2)
names(x)
```

```
## [1] "a" "b"
```

```
names(x) <- c("aa","bb")
x
```

```
## aa bb
##  1  2
```

```
x <- `names<-`(x,c("aaa","bbb"))
x
```

```
## aaa bbb
##   1   2
```

- You can write your own relacement functions if you end the function name with <-

```
`st360names<-` <- function(x,value){
  names(x) <- paste0(value,"360",names(x))
  x
}
st360names(x) <- c("a","b")
x
```

```
## a360aaa b360bbb
##       1       2
```

Special functions

- Examples: subset `[` and extract `[[`, control flow `if`, `for`,etc.
- Key point: These are functions, and it is sometimes useful to know their names so that we can get help or use them like any other prefix function.

```
dd <- data.frame(x=1:2,y=3:4)
`[[`(dd,1) # compare to dd[[1]]
```

```
## [1] 1 2
```

```
dd <- `[[<-`(dd,1,value=5:6) #cf dd[[1]] <- 5:6
dd
```

```
##   x y
## 1 5 3
## 2 6 4
```

- It can be useful to know functions by name so that we can call them in `lapply`-like functions.

```
sapply(dd,`[[<-`,2,value=10)
```

```
##       x  y
## [1,]  5  3
## [2,] 10 10
```

Functional Programming Basics

Functional programming languages

- In a functional language functions are data structures.
  - Can assign them to variables, pass them as arguments to other functions and return them from other functions.
  - This is true of R functions, and is what we'll be focusing on.
- Many functional languages also require functions to be "pure".
  - Function output should only depend on the input, and the function should not have any side-effects.
  - We can see from our brief discussion of scoping that R functions can use global variables, and we know they have side-effects like generating plots.

Functional style

- We will say that functional programming *style* means a top-down approach that breaks big problems into smaller pieces that we solve with small, easy (easier) to understand functions.
- This is the way we are approaching our implementation of MARS.
- Functional languages support this style with “higher-order” functions that take other functions as input .
  - The higher-order function is part of the “big problem”, and its input is the “small function”.

Functionals

- You have already seen higher-order functions in Stat 260: The `map` family of functions from the `purrr` package, or `lapply()` from base R.
- The text calls these functionals and they are discussed in Chapter 9.
- You have already studied the `map` functions, so our discussion here will be brief, with an emphasis on parallelization:
  - Not only can we break a computation into pieces, but we can have the pieces computed on different cores of a computer or nodes of a compute cluster.

Map-like functions such as `lapply()`

- Take a vector and function as input and return a list (or some simplification) whose elements are the function applied to each vector element.
  - We are mapping (in math sense) the vector elements x to $f(x)$.
- Basically a for loop over the input vector, calling the input function at each iteration.

```
simple_map <- function(x, f, ...) { # text, Section 9.2
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
}
```

Examples with `lapply()`

```
rfun <- function(seed,n) { set.seed(seed); return(rnorm(n))}
dat <- lapply(1:10,rfun,n=1e7) # specify non-vectorized arg by name
mfun <- function(x) { return(mean(quantile(x,probs=c(.25,.75)))) }
system.time( unlist(lapply(dat,mfun)) ) # Look at elapsed
```

```
##    user  system elapsed
##   1.975   0.070   2.045
```

Parallel execution with `mclapply()`

- `mclapply()` is the multi-core version of `lapply()`; i.e., function calls are done on separate cores.

```
library(parallel) # comes with R
ncores <- detectCores() ; cat("number of cores=",ncores,"\n")
```

```
## number of cores= 10
```

```
system.time({ unlist(mclapply(dat,mfun,mc.cores=ncores)) })
```

```
##    user  system elapsed
##   1.900   0.581   0.390
```

Another example

```
dat2 <- lapply(1:10,rfun,n=1e5)
mfun2 <- function(x) { return(data.frame(sum=sum(x),n=length(x))) }
sumdat <- lapply(dat2,mfun2)
```

Reduce-like functions

- Reduce functions successively combine vector elements.
  - Can use them to assemble output of a map-like function.
  - When implemented to parallelize over multiple computers, this is the MapReduce programming model you hear about for "big data".
- Reducers combine with a loop:

```
simple_reduce <- function(x, f) { # Text section 9.5
  out <- x[[1]]
  for (i in seq(2, length(x))) {
    out <- f(out, x[[i]])
  }
  out
}
```

Example reduce

```
allres <- simple_reduce(sumdat,rbind)
allres
```

```
##           sum      n
## 1  -224.40833 100000
## 2   307.85570 100000
## 3    36.96750 100000
## 4   -20.44743 100000
## 5  -726.27096 100000
## 6  -191.83899 100000
## 7   -50.00946 100000
## 8   270.86608 100000
## 9   -52.17403 100000
## 10 -595.83810 100000
```

```
sum(allres$sum)/sum(allres$n) # mean
```

```
## [1] -0.001245298
```

Further reading on parallel computing

- For parallel computing on a cluster, see the `doParallel` and `foreach` packages (canonical link to doParallel vignette currently broken):

http://users.iems.northwestern.edu/~nelsonb/Masterclass/getting startedParallel.pdf

Other higher-order functions

- Other chapters discuss function factories (Chapter 10) and function operators (Chapter 11).
- These are used less and will not be discussed.

In \ Out	Vector	Function
Vector	Regular function	Function factory
Function	Functional	Function operator

Figure 1: Higher-order functions