More on Functions

Stat 133 Feb 19

The first few slides are repeated from last time.

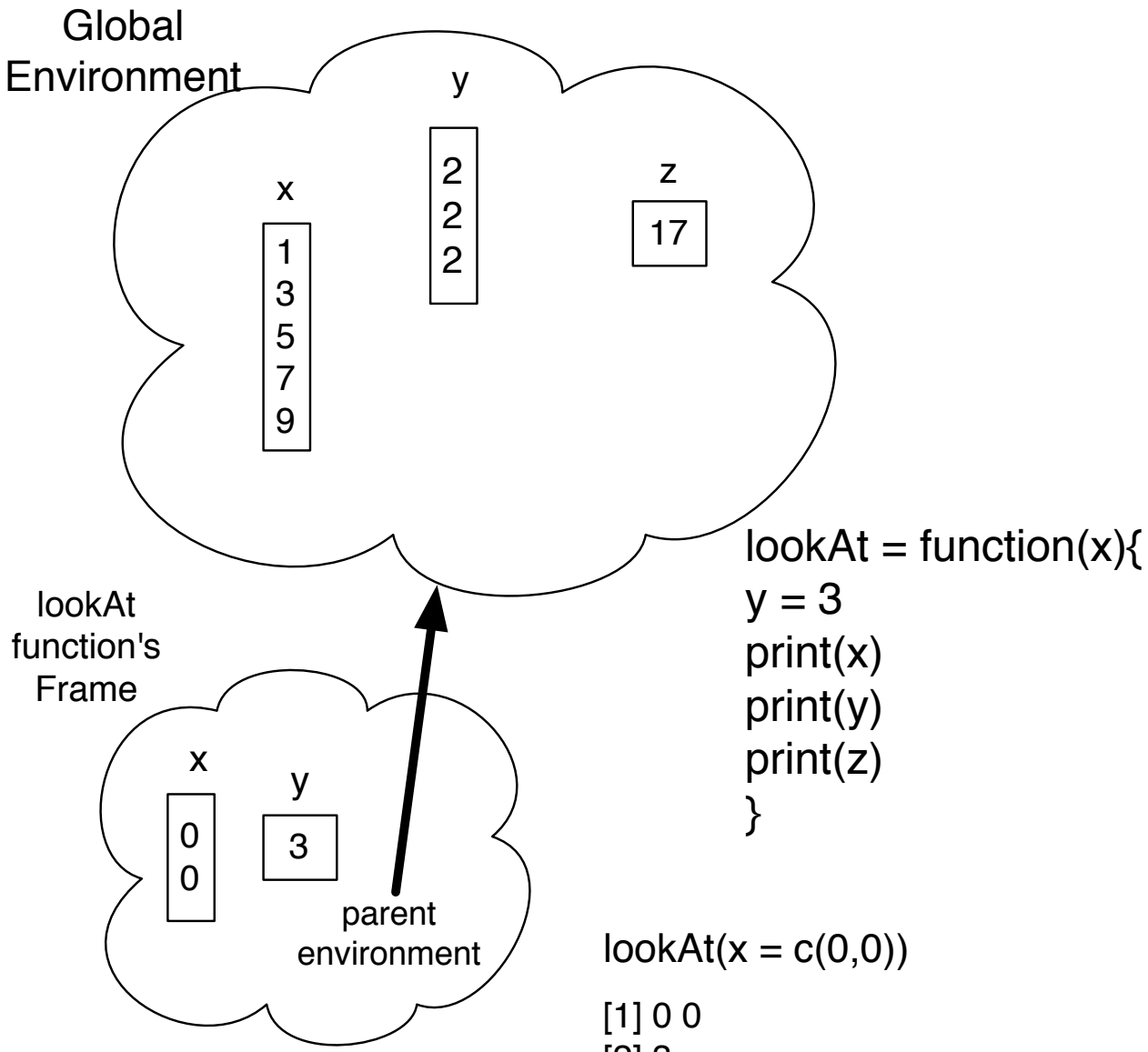
Environments and Scope

Environments and variable scope

R has a special mechanism for allowing you to use the same name in different places in your code and have it refer to different objects.

For example, you want to be able to create new variables in your functions and not worry if there are variables with the same name already in the workspace.

The solution relies on *environments* and the *variable scoping rules*.




```
lookAt = function(x){
y = 3
print(x)
print(y)
print(z)
}
```

```
lookAt(x = c(0,0))

[1] 0 0
[2] 3
[3] 17
```

When you call a function, R creates a new workspace containing just the variables defined by the arguments of that function. This space is called a *frame*.

```
> x = c(1, 3, 5, 7, 9); y = rep(2, 3); z = 17
> lookAt = function(x) {
+  y = 3
+  print(x); print(y); print(z)
}
> lookAt(x = c(0, 0))
```

x exists in the lookAt frame and so does y. However, z is not. R has a way of accessing variables that are not in the frame created by the function.

What is happening is that R is looking for variables with that name in a sequence of environments. An *environment* is just a frame (collection of variables) plus a pointer to the next environment to look in.

In our example, R didn't find the variable z in the environment defined by the function `lookAt`, so it went on to the next one. In this case, this was our main workspace, which is called the *Global Environment*.

The "next environment to look in" is called the parent environment.

If R reaches the Global Environment and still can't find the variable, it looks in something called the *search path*. This is a list of additional environments, which is used for packages of functions and user attached data. You can see the search path by typing `search()`.

This helps explain why we can write over built-in objects in R. What we're really doing is creating that object in the Global Environment, and then when we refer to it by name, R finds it here before it finds the predefined one.

```
> help(pi)
> pi = 3
> pi
> rm(pi)
> pi
```

CRAN: http://cran.r-project.org/

CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

Contributed Packages

Installation of Packages

Please type `help("INSTALL")` or `help("install.packages")` in R for information on how to install packages from this directory. The manual R Installation and Administration (also contained in the R base sources) explains the process in detail.

CRAN Task Views allow you to browse packages by topic and provide tools to automatically install all packages for special areas of interest. Currently, 28 views are available.

Daily Package Check Results

All packages are tested regularly on machines running Debian GNU/Linux, Fedora and Solaris. Packages are also checked under MacOS X and Windows, but typically only at the day the package appears on CRAN.

The results are summarized in the check summary (some timings are also available). Additional details for Windows checking and building can be found in the Windows check summary.

Writing Your Own Packages

The manual Writing R Extensions (also contained in the R base sources) explains how to write new packages and how to contribute them to CRAN.

Available Packages

Currently, the CRAN package repository features 2523 available packages.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

ACCLMA	ACC & LMA Graph Plotting
ADGofTest	Anderson-Darling GoF test
ADaCGH	Analysis of data from aCGH experiments
AER	Applied Econometrics with R
AGSDest	Estimation in adaptive group sequential trials

If a particular package is already installed on your system, you can access its contents by typing

```
> library("nameofpackage")
```

The authors of the package write documentation for the functions and datasets included in it, which you can read as usual using `help()`.

All packages come with a reference manual, which you can access by visiting CRAN. Go to http://cran.r-project.org/, click on "Packages," then scroll down for the particular package. This is just a hard copy of the help pages. A few packages also come with a tutorial.

Search path

- As we load packages into our session, they get added to the search path.

```
> search()
 [1] ".GlobalEnv"         "tools:rstudio"
 [3] "package:stats"      "package:graphics"
 [5] "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"
 [9] "Autoloads"          "package:base"
> library(RColorBrewer)
> search()
 [1] ".GlobalEnv"            "package:RColorBrewer"
 [3] "tools:rstudio"          "package:stats"
 [5] "package:graphics"       "package:grDevices"
 [7] "package:utils"          "package:datasets"
 [9] "package:methods"        "Autoloads"
[11] "package:base"
```

objects()

Show objects in global environment

```
> objects()
 [1] "AP"           "Convert"      "age"
 [4] "annualQuant" "bmi"          "day"
 [7] "deltaWt"      "desiredWt"    "distToPt"
[10] "gender"       "height"       "numInts"
[13] "qqunif"       "rain"         "timesWt"
[16] "weight"       "wifi"
```

Display objects in RColorBrewer package

```
> objects(2)
[1] "brewer.pal"          "brewer.pal.info"
[3] "display.brewer.all" "display.brewer.pal"
```

Catching Errors

Catching errors

1. The function `stop` stops execution of the current expression and prints a specified error message.

```
> showstop = function(x){
+   if(any(x < 0)) stop("x must be >= 0")
+   return("ok")
+ }

> showstop(1)
[1] "ok”

> showstop(c(-1, 1))
Error in showstop(c(-1, 1)) : x must be >= 0
```

2. A similar function is `stopifnot`. It has the advantage of being able to take multiple conditions.

```
> showstopifnot = function(x){
+   stopifnot(x >= 0, x %% 2 == 1)
+   return("ok")
+ }

> showstopifnot(1)
[1] "ok”

> showstopifnot(c(1, -1))
Error: all(x >= 0) is not TRUE

> showstopifnot(c(1,2))
Error: x %%2 == 1 is not all TRUE
```

3. The `warning` function just prints a warning message without stopping the execution of the function.

```
> ratio.warn = function(x, y){
+   if(any(y == 0))
+     warning("Dividing by zero")
+   return(x/y)
+ }

> ratio.warn(x = 1, y = c(1, 0))
[1]   1 Inf
Warning message:
In ratio.warn(x = 1, y = c(1, 0)) : Dividing by zero

> ratio.warn(x = 1:3, y = 1:2)
[1] 1 1 3
Warning message:
In x/y : longer object length is not a multiple of shorter
object length
```

4. Finally, `try` allows you to try the code. If it doesn't produce an error then you proceed. If it does, you decide what to do

```
ratio.try = function(x, y){
  z = try(x/y, silent = TRUE)
  if(inherits(z, "try-error"))
     stop("Error in division x/y")
  return(z)
}

> ratio.try(x = 1, y = c(1, 0))
[1]   1 Inf

> ratio.try(x = 1, y = "r")
Error in ratio.try(x = 1, y = "r") : Error in division x/y
```

Debugging Strategies

- Write code in a plain text file, e.g. in a script in Rstudio
- source() code into R (do not copy and paste)
- Syntax error will be caught and line number given
- Line numbers may not locate the error exactly
- Sometimes the error occurred earlier, but it gives you a starting place

```
> source('…/buggyCode.r')
Error in source("~/…/buggyCode.r") :
unexpected ')'
5: mybug = function(L, lambda) {
6:   numHits = rpois(L*lambda))
```

Can you see the error?

What line is it on?

Let's Debug this Function

```
# Here is a function that I have written to
# generate random Poisson scatter on an interval
of length "L"
# with a rate of lambda per unit

mybug = function(L, lambda) {
  numHits = rpois(L*lambda))
  hitLocs = runif(o, numHits)
  return(numHits, hitLocs)
}
```

Let's remove the extra right parenthesis?

Finding Errors

- Write code in a plain text file, e.g. in a script in Rstudio
- source() code into R (do not copy and paste)
- Syntax error will be caught and line number given
- Line numbers may not locate the error exactly
- Sometimes the error occurred earlier, but it gives you a starting place

Some debugging strategies

1. The `traceback` function prints the sequence of calls that led to the last error. This can show you where in your function something is going wrong.

It may not even be in the function itself, but in another function that is being called within the original function.

```
> cv <- function(x) sd(x/mean(x))
> cv(0)
Error in var(x, na.rm = na.rm) : missing observations in
cov/cor
> traceback()
3: var(x, na.rm = na.rm)
2: sd(x/mean(x))
1: cv(0)
```

2. If you have some idea where the error is occurring, you can use `print` to check that key variables are what you think they are.

3. Consider "commenting out" lines of your code where the error might occur, then adding them back in one by one.

4. To step through the function, expression by expression, and be able to print out any variable at each step, use the `debug` function. Use `undebug` to turn off debugging.

While in the debugger, you can use the following commands:

'n' (or just return) - Advance to the next step.
'c' - continue to the end of the current context: e.g. to the end of the loop if within a loop or to the end of the function.
'Q' - exit the browser and the current evaluation and return to the top-level prompt.

You can also evaluate any valid R expression. For example, you can type the names of variables to see their current values.

Efficient programming

The first rule of efficient programming in R is to make use of vectorized calculations and the apply mechanisms whenever possible.

You can check how much time it takes to evaluate any expression by wrapping it in `system.time()`. Units are in seconds.

```
> system.time(normal.samples <- rnorm(1000000))
   user  system elapsed
  0.196   0.013   0.221
```

wall clock time

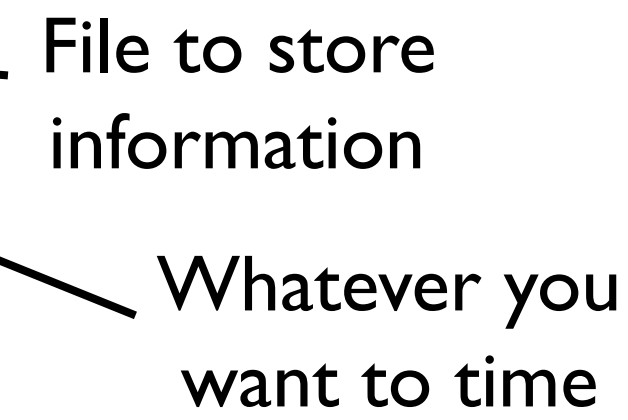
CPU time for R process

CPU time for system on behalf of R

A systematic way to time every part of a function is to use the `Rprof` and `summaryRprof` functions. This can be a handy way to find bottlenecks. The general syntax looks like this:

```
Rprof("profiling")
statements
Rprof(NULL)

summaryRprof("profiling")
```



The for statement

Looping is the repeated evaluation of a statement or block of statements.

Much of what is handled using loops in other languages can be more efficiently handled in R using vectorized calculations or one of the apply mechanisms.

However, certain algorithms, such as those requiring recursion, can only be handled by loops.

There are two main looping constructs in R: `for` and `while`.

For loops

A *for loop* repeats a statement or block of statements a predefined number of times.

The syntax in R is

```
for ( var in vector ){
  statement
}
```

For each element in `vector`, the variable `var` is set to the value of that element and `statement` is evaluated.

`vector` often contains integers, but can be any valid type.

While loops

A *while loop* repeats a statement or block of statements for as many times as a particular condition is `TRUE`.

The syntax in R is

```
while (condition){
  statement
}
```

`condition` is evaluated, and if it is `TRUE`, statement is evaluated. This process continues until condition evaluates to `FALSE`.

The Coin Flip

A Case study

Exercise:

The expression

```
sample(1:0, size = 1, prob = c(p, 1-p))
```

simulates a random coin flip, where the coin has probability `p` of coming up heads, represented by a `1`.

Write a function that simulates flipping a coin until a fixed number of heads are obtained. It should take the probability `p` and the total number of heads `total` and return the trial on which the final head was obtained. This produces a single sample from the *negative binomial distribution*.

The break statement causes a loop to exit. This is particularly useful with while loops, which, if we're not careful, might loop indefinitely (or until we kill R).

```
myRNG = function(total, p = 0.5){
# Simulate number of tosses to get 10 Heads

 x = 0
 steps = 0
 max.iter = 1000

 while(x < total){
  x = x + sample(1:0, size = 1, prob = c(p, 1-p))
  steps = steps + 1
  if(steps >= max.iter){
    warning("Maximum iteration reached")
    break
  }
 }
 return(steps)
}
```