Unit 4: Programming concepts, illustrated with R

September 29, 2017

This unit covers a variety of programming concepts, illustrated in the context of R. So it also serves as a way to teach advanced features of R. In general the concepts are relevant in other languages, though other languages may implement things differently. One of my goals here for us to think about why things are the way they are in R. I.e., what principles were used in creating the language and what choices were made? While other languages use different principles and made difference choices, understanding what R does in detail will be helpful when you are learning another language.

References:

- Books on R listed on the syllabus: Adler, Chambers, Wickham
- R intro manual and R language manual (R-lang), both on CRAN.
- Venables and Ripley, Modern Applied Statistics with S
- Murrell, Introduction to Data Technologies

I'm going to try to refer to R syntax as *statements*, where a statement is any code that is a valid, complete R expression. I'll try not to use the term *expression*, as this actually means a specific type of object within the R language, as seen in Section 9.

1 Interacting with the operating system from R and controlling R's behavior

I'll assume everyone knows about the following functions/functionality in R:

getwd(), setwd(), source(), pdf(), save(), save.image(), load()

- To run UNIX commands from within R, use *system()*, as follows, noting that we can save the result of a system call to an R object:

```
system("ls -al")
## knitr/Sweave doesn't seem to show the output of system()
files <- system("ls", intern = TRUE)
files[1:5]

## [1] "cache"            "class2.log"
## [3] "class3.log"       "class4.log"
## [5] "exampleRscript.R"
```

- There are also a bunch of functions that will do specific queries of the filesystem, including

```
file.exists("unit2-bash.sh")

## [1] TRUE

list.files("../data")

## [1] "coop.txt.gz"    "cpds.csv"       "IPs.RData"
## [4] "precip.txt"     "RTADataSub.csv"
```

- There are some tools for dealing with differences between operating systems. Here's an example:

```
list.files(file.path("..", "data"))

## [1] "coop.txt.gz"    "cpds.csv"       "IPs.RData"
## [4] "precip.txt"     "RTADataSub.csv"
```

- To get some info on the system you're running on:

```
Sys.info()

##                                     sysname
##                                     "Linux"
##                                     release
```

```
##                                "4.4.0-93-generic"
##                                           version
## "#116-Ubuntu SMP Fri Aug 11 21:17:51 UTC 2017"
##                                          nodename
##                                         "smeagol"
##                                           machine
##                                          "x86_64"
##                                             login
##                                         "unknown"
##                                              user
##                                        "paciorek"
##                                    effective_user
##                                        "paciorek"
```

- To see some of the options that control how R behaves, try the *options()* function. The *width* option changes the number of characters of width printed to the screen, while the *max.print* option prevents too much of a large object from being printed to the screen. The *digits* option changes the number of digits of numbers printed to the screen (but be careful as this can be deceptive if you then try to compare two numbers based on what you see on the screen).

```
## options()  # this would print out a long list of options
options()[1:5]

## $add.smooth
## [1] TRUE
##
## $bitmapType
## [1] "cairo"
##
## $browser
## [1] "xdg-open"
##
## $browserNLdisabled
## [1] FALSE
##
## $CBoundsCheck
```

```
## [1] FALSE

options()[c('width', 'digits')]

## $width
## [1] 55
##
## $digits
## [1] 7

## options(width = 120)
## often nice to have more characters on screen
options(width = 55)  # for purpose of making pdf of this document
options(max.print = 5000)
options(digits = 3)
a <- 0.123456; b <- 0.1234561
a; b; a == b

## [1] 0.123
## [1] 0.123
## [1] FALSE
```

- Use `Ctrl-C` to interrupt execution. This will generally back out gracefully, returning you to a state as if the command had not been started. Note that if R is exceeding memory availability, there can be a long delay. This can be frustrating, particularly since a primary reason you would want to interrupt is when R runs out of memory.
- The R mailing list archives are very helpful for getting help - always search the archive before posting a question. More info on where to find R help in Unit 5 on debugging.
  - *sessionInfo()* gives information on the current R session - it's a good idea to include this information (and information on the operating system such as from *Sys.info()*) when you ask for help on a mailing list

```
sessionInfo()

## R version 3.4.1 (2017-06-30)
```

```
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 16.04.2 LTS
##
## Matrix products: default
## BLAS: /usr/lib/openblas-base/libblas.so.3
## LAPACK: /usr/lib/libopenblasp-r0.2.18.so
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8
##  [2] LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8
##  [4] LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8
##  [6] LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8
##  [8] LC_NAME=C
##  [9] LC_ADDRESS=C
## [10] LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8
## [12] LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets
## [6] base
##
## other attached packages:
## [1] pryr_0.1.2   knitr_1.15.1 SCF_3.4.0
##
## loaded via a namespace (and not attached):
##  [1] compiler_3.4.1   magrittr_1.5     tools_3.4.1
##  [4] Rcpp_0.12.10     codetools_0.2-15 stringi_1.1.5
##  [7] highr_0.6        methods_3.4.1    stringr_1.2.0
## [10] evaluate_0.10
```

- Any code that you wanted executed automatically when starting R can be placed in *~/.Rprofile* (or in individual *.Rprofile* files in specific directories). This could include loading pack-

ages (see below), sourcing files that contain user-defined functions that you commonly use (you can also put the function code itself in *.Rprofile*), assigning variables, and specifying options via *options()*.

- You can have an R script act as a shell script (like running a bash shell script) as follows. This will probably on work on Linux and Mac.

  1. Write your R code in a text file, say *exampleRscript.R*.
  2. As the first line of the file, include `#!/usr/bin/Rscript` (like `#!/bin/bash` in a bash shell file, as seen in Unit 2) or (for more portability across machines, include `#!/usr/bin/env Rscript`.
  3. Make the R code file executable with *chmod*: `chmod ugo+x exampleRscript.R`.
  4. Run the script from the command line: `./exampleRscript.R`

  If you want to pass arguments into your script, you can do so as long as you set up the R code to interpret the incoming arguments:

```
args <- commandArgs(TRUE)
## Now args is a character vector containing the arguments.
## Suppose the first argument should be interpreted as a number
# and the second as a character string and the third as a boolean:
numericArg <- as.numeric(args[1])
charArg <- args[2]
logicalArg <- as.logical(args[3]
cat("First arg is: ", numericArg, "; second is: ",
   charArg, "; third is: ", logicalArg, ".\n")
```

```
./exampleRscript.R 53 blah T
./exampleRscript.R blah 22.5 t

## First arg is,  53 ; second is:  blah ; third is:  TRUE .
## Warning message:
## NAs introduced by coercion
## First arg is,  NA ; second is:  22.5 ; third is:  NA .
```

2 Packages and namespaces

One of the killer apps of R is the extensive collection of add-on packages on CRAN (www.cran.r-project.org) that provide much of R's functionality. To make use of a package it needs to be installed on your system (using *install.packages()* once only) and loaded into R (using *library()* every time you start R).

Some packages are *installed* by default with R and of these, some are *loaded* by default, while others require a call to *library()*. For packages I use a lot, I install them once and then load them automatically every time I start R using my *~/.Rprofile* file.

If you want to sound like an R expert, make sure to call them *packages* and not *libraries*. A *library* is the location in the directory structure where the packages are installed/stored.

Loading packages You can use *library()* to either (1) make a package available (loading it), (2) get an overview of the package, or (3) (if called without arguments) to see all the installed packages.

```
library(fields)

## Loading required package:  methods
## Loading required package:  spam
## Loading required package:  grid
## Spam version 1.4-0 (2016-08-29) is loaded.
## Type 'help( Spam)' or 'demo( spam)' for a short introduction
## and overview of this package.
## Help for individual functions is also obtained by adding the
## suffix '.spam' to the function name, e.g.  'help( chol.spam)'.
##
## Attaching package:  'spam'
## The following objects are masked from 'package:base':
##
##    backsolve, forwardsolve
## Loading required package:  maps

library(help = fields)
## library()  # I don't want to run this on my SCF machine
##  because so many are installed
```

If you run `library()`, you'll notice that some of the packages are in a system directory and some are in your home directory. Packages often depend on other packages. In general, if one package depends on another, R will load the dependency, but if the dependency is installed locally (see below), R may not find it automatically and you may have to use *library()* to load the dependency first. *.libPaths()* shows where R looks for packages on your system and *searchpaths()* shows where individual packages are loaded from. Looking the help info for *.libPaths()* gives some information about how R decides what locations to look in for packages.

```
.libPaths()
```

```
## [1] "/accounts/gen/vis/paciorek/R/x86_64-pc-linux-gnu-library/3.4"
## [2] "/system/linux/lib/R-16.04/3.4/x86_64/site-library"
## [3] "/usr/lib/R/site-library"
## [4] "/usr/lib/R/library"
```

```
searchpaths()
```

```
##  [1] ".GlobalEnv"
##  [2] "/system/linux/lib/R-16.04/3.4/x86_64/site-library/fields"
##  [3] "/system/linux/lib/R-16.04/3.4/x86_64/site-library/maps"
##  [4] "/system/linux/lib/R-16.04/3.4/x86_64/site-library/spam"
##  [5] "/usr/lib/R/library/grid"
##  [6] "/usr/lib/R/library/methods"
##  [7] "/system/linux/lib/R-16.04/3.4/x86_64/site-library/pryr"
##  [8] "/system/linux/lib/R-16.04/3.4/x86_64/site-library/knitr"
##  [9] "/usr/lib/R/library/stats"
## [10] "/usr/lib/R/library/graphics"
## [11] "/usr/lib/R/library/grDevices"
## [12] "/usr/lib/R/library/utils"
## [13] "/usr/lib/R/library/datasets"
## [14] "/system/linux/lib/R-16.04/3.4/x86_64/site-library/SCF"
## [15] "Autoloads"
## [16] "/usr/lib/R/library/base"
```

Installing packages If a package is on CRAN but not on your system, you can install it easily (usually). You don't need root permission on a machine to install a package (though sometimes you run into hassles if you are installing it just as a user, so if you have administrative privileges

it may help to use them). Of course in RStudio, you can install via the GUI. If you are installing by specifying the *lib* argument, you'd generally want to use whatever user-owned directory (i.e., library) is specified by the output of *.libPaths()*. If none of them are user-owned, you may need to add a library via .libPaths() (e.g., by putting something like `.libPaths('~/Rlibs')` in your *.Rprofile*).

```
install.packages('fields', lib = '~/Rlibs') # ~/Rlibs needs to exist!
```

Note that R will generally install the package in a reasonable place if you omit the *lib* argument.

You can also download the zipped source file from CRAN and install from the file; see the help page for *install.packages()*. This is called "installing from source. On Windows and Mac, you'll need to do something like this:

```
install.packages('fields.tar.gz', repos = NULL, type = 'source')
```

If you've downloaded the binary package (files ending in .tgz for Mac and .zip for Windows) and want to install the package directly from the file, use the syntax above but omit the `type='source'` argument.

The difference between the source package and the binary package is that the source package has the raw R (and C and Fortran, in some cases) code as text files while the binary package has all the code in a binary/non-text format, including any C and Fortran code having been compiled. To install a source package with C or Fortran code in it, you'll need to have developer/command-line tools (e.g., *XCode* on Mac or *Rtools.exe* on Windows) installed on your system so that you have a compiler.

Package namespaces The objects in a package (primarily functions, but also data) are in their own workspaces, and are accessible after you load the package using *library()*, but are not directly visible when you use *ls()*. In other words, each package has its own *namespace*. Namespaces help achieve modularity and avoid having zillions of objects all reside in your workspace. We'll talk more about this when we talk about scope and environments. If we want to see the objects in a package's namespace, we can do the following:

```
search()
```

```
##  [1] ".GlobalEnv"        "package:fields"
##  [3] "package:maps"      "package:spam"
##  [5] "package:grid"      "package:methods"
```

```
##  [7] "package:pryr"      "package:knitr"
##  [9] "package:stats"     "package:graphics"
## [11] "package:grDevices" "package:utils"
## [13] "package:datasets"  "package:SCF"
## [15] "Autoloads"         "package:base"

## ls(pos = 8) # for the stats package
ls(pos = 8)[1:5] # just show the first few

## [1] "all_labels"      "all_patterns"
## [3] "all_rcpp_labels" "asis_output"
## [5] "clean_cache"

ls("package:stats")[1:5] # equivalent

## [1] "acf"        "acf2AR"     "add1"       "addmargins"
## [5] "add.scope"
```

3 Text manipulation, string processing and regular expressions (regex)

Text manipulations in R have a number of things in common with Perl, Python and UNIX, as many of these evolved from UNIX. When I use the term *string* here, I'll be referring to any sequence of characters that may include numbers, white space, and special characters, rather than to the character class of R objects. The string or strings will generally be stored as R character vectors.

For material on string processing in R, see the tutorial, *String processing in R and Python*. (You can ignore the sections on Python for now.) That tutorial then refers to the *Using the bash shell* tutorial for details on regular expressions. Finally, to test out regular expression syntax see this online tool.

Recall that when characters are used for special purposes, we need to escape them if we want them interpreted as the actual character. In the second example, the first backslash says to interpret the next backslash literally, with the second backslash being used to indicate that the bracket should be interpreted literally.

```
## for some reason, output from next few lines not printing out in pdf...
tmp <- "Harry said, \"Hi\""
cat(tmp)
tmp <- "Harry said, \"Hi\".\n"
cat(tmp)
## search for either a '^' or a 'z':
grep("[\\^z]", c("a^2", "93"))
## fails because '\^' is not an escape sequence:
grep("[\^z]", c("a^2", "93"))

## Error:  '\^' is an unrecognized escape in character string starting
""[\^"
```

Challenge: explain why we use a single backslash to get a newline and double backslash to write out a Windows path in the examples here:

```
cat("hello\nagain")

## hello
## again

cat("hello\\nagain")

## hello\nagain

cat("My Windows path is: C:\\Users\\My Documents.")

## My Windows path is: C:\Users\My Documents.
```

For more information, see `?Quotes` in R.

Be careful when cutting and pasting from documents that are not text files as you may paste in something that looks like a single or double quote, but which R cannot interpret as a quote because it's some other ASCII quote character. If you paste in a " from PDF, it will not be interpreted as a standard R double quote mark.

3.1 Regex practice

Write a regular expression that matches the following:

1. Only the strings "cat", "at", and "t".

2. The strings “cat”, “caat”, “caaat”, etc.

3. “dog”, “Dog”, “dOg”, “doG”, “DOg”, etc. (the word dog in any combination of lower and upper case).

4. Any positive number with or without a decimal point.

5. Any line with exactly two words separated by any amount of whitespace (spaces or tabs). There may or may not be whitespace at the beginning or end of the line.

3.2 Regex/string processing challenges

We’ll work on these challenges in class.

1. What regex would I use to find a spam-like pattern with digits or non-letters inside a word? E.g., I want to find "V1agra" or "Fancy repl!c@ted watches".

2. How would I extract email addresses from lines of text using regular expressions and R string processing?

3. Suppose a text string has dates in the form “Aug-3”, “May-9”, etc. and I want them in the form “3 Aug”, “9 May”, etc. How would I do this search and replace operation? (Alternatively, how could I do this without using regular expressions at all?)

4. How would I search for numeric URLs such as in the file *urls.txt*.

4 Types, classes, and object-oriented programming

4.1 Types and classes

You should be familiar with vectors as the basic data structure in R, with character, integer, numeric, etc. classes. Vectors are either *atomic vectors* or *lists*. Atomic vectors generally contain one of the four following types: *logical*, *integer*, *double/numeric*, and *character*.

Objects in general have a type, which relates to what kind of values are in the objects and how objects are stored internally in R (i.e., in C).

Let’s look at Adler’s Table 7.1 to see some other types.

```
a <- data.frame(x = 1:2)
class(a)
```

```
## [1] "data.frame"

typeof(a)

## [1] "list"

is.data.frame(a)

## [1] TRUE

is.matrix(a)

## [1] FALSE

is(a, "matrix")

## [1] FALSE

m <- matrix(1:4, nrow = 2)
class(m)

## [1] "matrix"

typeof(m)

## [1] "integer"
```

Everything in R is an object and all objects have a class. For simple objects class and type are often closely related, but this is not the case for more complicated objects. The class describes what the object contains and standard functions associated with it. In general, you mainly need to know what class an object is rather than its type. Classes can *inherit* from other classes; for example, the *glm* class inherits characteristics from the *lm* class. We'll see more on the details of object-oriented programming shortly.

We can create objects with our own defined class.

```
bart <- list(firstname = 'Bart', surname = 'Simpson',
             hometown = "Springfield")
class(bart) <- 'personClass'
## it turns out R already has a 'person' class
class(bart)
```

```
## [1] "personClass"

is.list(bart)

## [1] TRUE

typeof(bart)

## [1] "list"

typeof(bart$firstname)

## [1] "character"
```

4.2 Attributes

Attributes are information about an object attached to an object as something that looks like a named list. Attributes are often copied when operating on an object. This can lead to some weird-looking formatting:

```
x <- rnorm(10 * 365)
qs <- quantile(x, c(.025, .975))
qs

##  2.5% 97.5%
## -2.00  1.94

qs[1] + 3

## 2.5%
##    1
```

Thus in an subsequent operations with *qs*, the *names* attribute will often get carried along. We can get rid of it:

```
names(qs) <- NULL
qs

## [1] -2.00  1.94
```

A common use of attributes is that rows and columns may be named in matrices and data frames, and elements in vectors:

```
row.names(mtcars)[1:6]

## [1] "Mazda RX4"          "Mazda RX4 Wag"
## [3] "Datsun 710"         "Hornet 4 Drive"
## [5] "Hornet Sportabout" "Valiant"

names(mtcars)

##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec"
##  [8] "vs"   "am"   "gear" "carb"

attributes(mtcars)

## $names
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec"
##  [8] "vs"   "am"   "gear" "carb"
##
## $row.names
##  [1] "Mazda RX4"           "Mazda RX4 Wag"
##  [3] "Datsun 710"          "Hornet 4 Drive"
##  [5] "Hornet Sportabout"   "Valiant"
##  [7] "Duster 360"          "Merc 240D"
##  [9] "Merc 230"            "Merc 280"
## [11] "Merc 280C"           "Merc 450SE"
## [13] "Merc 450SL"          "Merc 450SLC"
## [15] "Cadillac Fleetwood"  "Lincoln Continental"
## [17] "Chrysler Imperial"   "Fiat 128"
## [19] "Honda Civic"         "Toyota Corolla"
## [21] "Toyota Corona"       "Dodge Challenger"
## [23] "AMC Javelin"         "Camaro Z28"
## [25] "Pontiac Firebird"    "Fiat X1-9"
## [27] "Porsche 914-2"       "Lotus Europa"
## [29] "Ford Pantera L"      "Ferrari Dino"
## [31] "Maserati Bora"       "Volvo 142E"
##
```

```
## $class
## [1] "data.frame"
```

```
mat <- data.frame(x = 1:2, y = 3:4)
attributes(mat)
```

```
## $names
## [1] "x" "y"
##
## $row.names
## [1] 1 2
##
## $class
## [1] "data.frame"
```

```
row.names(mat) <- c("first", "second")
mat
```

```
##        x y
## first  1 3
## second 2 4
```

```
attributes(mat)
```

```
## $names
## [1] "x" "y"
##
## $row.names
## [1] "first"  "second"
##
## $class
## [1] "data.frame"
```

```
vec <- c(first = 7, second = 1, third = 5)
vec['first']
```

```
## first
##     7
```

```
attributes(vec)
```

```
## $names
## [1] "first"  "second" "third"
```

4.3 Assignment and coercion

We assign into an object using either '=' or '<-'. A rule of thumb is that for basic assignments where you have an object name, then the assignment operator, and then some code, '=' is fine, but otherwise use '<-'.

Let's look at these examples to understand the distinction between '=' and '<-' when passing arguments to a function.

```
mean

## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x44ef410>
## <environment: namespace:base>

x <- 0; y <- 0
out <- mean(x = c(3,7)) # usual way to pass an argument to a function
## what does the following do?
out <- mean(x <- c(3,7)) # this is allowable, though perhaps not useful
out <- mean(y = c(3,7))

## Error in mean.default(y = c(3, 7)): argument "x" is missing, with
no default

out <- mean(y <- c(3,7))
```

What can you tell me about what is going on in each case above?

One situation in which you want to use '<-' is if it is being used as part of an argument to a function, so that R realizes you're not indicating one of the function arguments, e.g.:

```
## NOT OK, system.time() expects its argument to be a complete R expression
system.time(out = rnorm(10000))

## Error in system.time(out = rnorm(10000)): unused argument (out
= rnorm(10000))

# OK:
system.time(out <- rnorm(10000))

##    user  system elapsed
##   0.000   0.000   0.001
```

Here's another example:

```
mat <- matrix(c(1, NA, 2, 3), nrow = 2, ncol = 2)
apply(mat, 1, sum.isna <- function(vec) {return(sum(is.na(vec)))})

## [1] 0 1

## What is the side effect of what I have done just above?
apply(mat, 1, sum.isna = function(vec) {return(sum(is.na(vec)))}) # NOPE

## Error in match.fun(FUN): argument "FUN" is missing, with no default
```

R often treats integers as numerics, but we can force R to store values as integers:

```
vals <- c(1, 2, 3)
class(vals)

## [1] "numeric"

vals <- 1:3
class(vals)

## [1] "integer"

vals <- c(1L, 2L, 3L)
vals

## [1] 1 2 3

class(vals)

## [1] "integer"
```

We convert between classes using variants on *as()*: e.g.,

```
as.character(c(1,2,3))

## [1] "1" "2" "3"

as.numeric(c("1", "2.73"))
```

```
## [1] 1.00 2.73

as.factor(c("a", "b", "c"))

## [1] a b c
## Levels: a b c
```

Some common conversions are converting numbers that are being interpreted as characters into actual numbers, converting between factors and characters, and converting between logical TRUE/FALSE vectors and numeric 1/0 vectors. In some cases R will automatically do conversions behind the scenes in a smart way (or occasionally not so smart way). We saw see implicit conversion (also called coercion) when we read in characters into R using *read.table()* - strings are often automatically coerced to factors. Consider these examples of implicit coercion:

```
x <- rnorm(5)
x[3] <- 'hat' # What do you think is going to happen?
indices <- c(1, 2.73)
myVec <- 1:10
myVec[indices]

## [1] 1 2
```

Be careful of using factors as indices:

```
students <- factor(c("basic", "proficient", "advanced",
                     "basic", "advanced", "minimal"))
score <- c(minimal = 3, basic = 1, advanced = 13, proficient = 7)
score["advanced"]

## advanced
##       13

score[students[3]]

## minimal
##       3

score[as.character(students[3])]

## advanced
##       13
```

What has gone wrong and how does it relate to type coercion?

In other languages, converting between different classes is sometimes called *casting* a variable.

Here's an example we can work through that will help illustrate how type conversions occur behind the scenes in R.

```
n <- 5
df <- data.frame(rep('a', n), rnorm(n), rnorm(n))
apply(df, 1, function(x) x[2] + x[3])

## Error in x[2] + x[3]:  non-numeric argument to binary operator

## why does that not work?
apply(df[ , 2:3], 1, function(x) x[1] + x[2])

## [1]  1.998  3.685 -0.652  0.826  1.810

## let's look at apply() to better understand what is happening
```

4.4 Object-oriented programming

Popular languages that use OOP include C++, Java, and Python. In fact C++ is the object-oriented version of C. Different languages implement OOP in different ways.

The idea of OOP is that all operations are built around objects, which have a class, and methods that operate on objects in the class. Classes are constructed to build on (inherit from) each other, so that one class may be a specialized form of another class, extending the components and methods of the simpler class (e.g., *lm* and *glm* objects).

Note that in more formal OOP languages, all functions are associated with a class, while in R, only some are.

Often when you get to the point of developing OOP code in R, you're doing more serious programming, and you're going to be acting as a software engineer. It's a good idea to think carefully in advance about the design of the classes and methods.

4.4.1 S3 approach

S3 classes are widely-used, in particular for statistical models in the *stats* package. S3 classes are very informal in that there's not a formal definition for an S3 class. Instead, an S3 object is just a primitive R object such as a list or vector with additional attributes including a class name.

Inheritance Let's look at the *lm* class, which builds on lists, and *glm* class, which builds on the *lm* class. Here *mod* is an object (an instance) of class *lm*. An analogy is the difference between a random variable and a realization of that random variable.

```
library(methods)
yb <- sample(c(0, 1), 10, replace = TRUE)
yc <- rnorm(10)
x <- rnorm(10)
mod1 <- lm(yc ~ x)
mod2 <- glm(yb ~ x, family = binomial)
class(mod1)

## [1] "lm"

class(mod2)

## [1] "glm" "lm"

is.list(mod1)

## [1] TRUE

names(mod1)

##  [1] "coefficients"  "residuals"     "effects"
##  [4] "rank"          "fitted.values" "assign"
##  [7] "qr"            "df.residual"   "xlevels"
## [10] "call"          "terms"         "model"

is(mod2, "lm")

## [1] TRUE

methods(class = "lm")

##  [1] add1           alias          anova
##  [4] case.names     coerce         confint
##  [7] cooks.distance deviance       dfbeta
## [10] dfbetas        drop1          dummy.coef
## [13] effects        extractAIC     family
```

```
## [16] formula         hatvalues      influence
## [19] initialize      kappa          labels
## [22] logLik          model.frame    model.matrix
## [25] nobs            plot           predict
## [28] print           proj           qr
## [31] residuals       rstandard      rstudent
## [34] show            simulate       slotsFromS3
## [37] summary         variable.names vcov
## see '?methods' for accessing help and source code
```

Often S3 classes inherit from lists (i.e., are special cases of lists), so you can obtain components of the object using the $ operator.

Creating our own class We can create an object with a new class as follows:

```
yog <- list(firstname = 'Yogi', surname = 'the Bear', age = 20)
class(yog) <- 'bear'
```

Actually, if we want to create a new class that we'll use again, we want to create a *constructor* function that initializes new bears:

```
bear <- function(firstname = NA, surname = NA, age = NA){
        # constructor for 'indiv' class
        obj <- list(firstname = firstname, surname = surname,
                    age = age)
        class(obj) <- 'bear'
        return(obj)
}
smoke <- bear('Smokey','Bear')
```

For those of you used to more formal OOP, the following is probably disconcerting:

```
class(yog) <- "silly"
class(yog) <- "bear"
```

Methods The real power of OOP comes from defining *methods*. For example,

```r
mod <- lm(yc ~ x)
summary(mod)
gmod <- glm(yb ~ x, family = 'binomial')
summary(gmod)
```

Here *summary()* is a generic method (or generic function) that, based on the type of object given to it (the first argument), dispatches a class-specific function (method) that operates on the object. This is convenient for working with objects using familiar functions. Consider the generic methods *plot()*, *print()*, *summary()*, *'['*, and others. We can look at a function and easily see that it is a generic method. We can also see what classes have methods for a given generic method.

```r
summary
```

```
## standardGeneric for "summary" defined from package "base"
##
## function (object, ...)
## standardGeneric("summary")
## <environment: 0x4abdf00>
## Methods may be defined for arguments: object
## Use  showMethods("summary")  for currently available ones.
```

```r
methods(summary)
```

```
##  [1] summary,ANY-method
##  [2] summary.aov
##  [3] summary.aovlist*
##  [4] summary.aspell*
##  [5] summary.check_packages_in_dir*
##  [6] summary.connection
##  [7] summary.data.frame
##  [8] summary.Date
##  [9] summary.default
## [10] summary.ecdf*
## [11] summary.factor
## [12] summary.glm
## [13] summary.infl*
## [14] summary.Krig
```

```
## [15] summary.lm
## [16] summary.loess*
## [17] summary.manova
## [18] summary.matrix
## [19] summary.mKrig
## [20] summary.mlm*
## [21] summary.ncdf
## [22] summary.nls*
## [23] summary.packageStatus*
## [24] summary.PDF_Dictionary*
## [25] summary.PDF_Stream*
## [26] summary.POSIXct
## [27] summary.POSIXlt
## [28] summary.ppr*
## [29] summary.prcomp*
## [30] summary.princomp*
## [31] summary.proc_time
## [32] summary.qsreg
## [33] summary.spam
## [34] summary.spam.chol.NgPeyton
## [35] summary,spam.chol.NgPeyton-method
## [36] summary,spam-method
## [37] summary.spatial.design
## [38] summary.spatialProcess
## [39] summary.srcfile
## [40] summary.srcref
## [41] summary.sreg
## [42] summary.stepfun
## [43] summary.stl*
## [44] summary.table
## [45] summary.tukeysmooth*
## see '?methods' for accessing help and source code
```

In many cases there will be a default method (here, *mean.default()*), so if no method is defined for the class, R uses the default. Sidenote: arguments to a generic method are passed along to the selected method by passing along the calling environment.

We can define new generic methods:

```
summarize <- function(object, ...)
        UseMethod("summarize")
```

Once *UseMethod()* is called, R searches for the specific method associated with the class of *object* and calls that method, without ever returning to the generic method. Let's try this out on our *bear* class. In reality, we'd write either *summary.bear()* or *print.bear()* (and of course the generics for *summary* and *print* already exist) but for illustration, I wanted to show how we would write both the generic and the specific method, so I'll write a *summarize* method.

```
summarize.bear <- function(object)
        return(with(object, cat("Bear of age ", age,
        " whose name is ", firstname, " ", surname, ".\n",
    sep = "")))
summarize(yog)

## Bear of age 20 whose name is Yogi the Bear.
```

Note that the *print()* function is what is called when you simply type the name of the object, so we can have object information printed out in a structured way. Recall that the output when we type the name of an *lm* object is NOT simply a regurgitation of the elements of the list - rather *print.lm()* is called.

Similarly, when we used `print(object.size(x))` we were invoking the *object_size*-specific print method which gets the value of the size and then formats it. So there's actually a fair amount going on behind the scenes.

Surprisingly, the *summary()* method generally doesn't actually print out information; rather it computes things not stored in the original object and returns it as a new class (e.g., class *summary.lm*), which is then automatically printed, per my comment above, using *print.summary.lm()*, unless one assigns it to a new object. Note that *print.summary.lm()* is hidden from user view.

```
out <- summary(mod)
out
print(out)
getS3method(f="print",class="summary.lm")
```

More on inheritance As noted with *lm* and *glm* objects, we can assign more than one class to an object. Here *summarize()* still works, even though the primary class is *grizzly_bear*.

```
class(yog) <- c('grizzly_bear', 'bear')
summarize(yog)

## Bear of age 20 whose name is Yogi the Bear.
```

The classes should nest within one another with the more specific classes to the left, e.g., here a *grizzly_bear* would have some additional objects on top of those of a *bear*, perhaps *number_of_people_eaten* (since grizzly bears are much more dangerous than some other kinds of bears), and perhaps additional or modified methods. *grizzly_bear* inherits from *bear*, and R uses methods for the first class before methods for the next class(es), unless no such method is defined for the first class. If no methods are defined for any of the classes, R looks for *method.default()*, e.g., *print.default()*, *plot.default()*, etc..

Class-specific operators We can also use operators with our classes. The following example will be a bit silly (it would make more sense with a class that is a mathematical object) but indicates the power of having methods.

```
methods(`+`)

## [1] +.Date                  +,matrix,spam-method
## [3] +.POSIXt                +,spam,matrix-method
## [5] +,spam,missing-method +,spam,spam-method
## see '?methods' for accessing help and source code

`+.bear` <- function(object, incr) {
        object$age <- object$age + incr
        return(object)
}
older_yog <- yog + 15
```

Class-specific replacement functions We can use replacement functions with our classes. For more on what a replacement function is, see Section 6.6.

This is again a bit silly but we could do the following. We need to define the generic replacement function and then the class-specific one.

```
`age<-` <- function(x, ...) UseMethod("age<-")
`age<-.bear` <- function(object, value){
        object$age <- value
        return(object)
}
age(older_yog) <- 60
```

Why use class-specific methods? We could have implemented different functionality (e.g., for *summary()*) for different objects using a bunch of *if* statements (or *switch()*) to figure out what class of object is the input, but then we need to have all that checking. Furthermore, we don't control the *summary()* function, so we would have no way of adding the additional conditions in a big if-else statement. The OOP framework makes things *extensible*, so we can build our own new functionality on what is already in R.

Final thoughts Consider the *Date* class discussed in the R bootcamp. This is another example of an S3 class, with methods such as *julian()*, *weekdays()*, etc.

Challenge: how would you get R to quit immediately, without asking for any more information, when you simply type '*q*' (no parentheses!)?

What we've just discussed are the old-style R (and S) object orientation, called S3 methods. The new style is called S4 and we'll discuss it next. S3 is still commonly used, in part because S4 can be slow (or at least it was when I last looked into this a few years ago). S4 is more structured than S3.

4.4.2 S4 approach

S4 methods are used a lot in *bioconductor*, a project that provides a lot of bioinformatics-related code. They're also used in *lme4*, among other packages. Tools for working with S4 classes are in the *methods* package.

Note that components of S4 objects are obtained as `object@component` so they do not use the usual list syntax. The components are called *slots*, and there is careful checking that the slots are specified and valid when a new object of a class is created. You can use the *prototype* argument to *setClass()* to set default values for the slots. There is a default constructor (the method is actually called *initialize()*), but you can modify it. One can create methods for operators and for replacement functions too. For S4 classes, there is a default method invoked when *print()* is called on an object in the class (either explicitly or implicitly) - the method is actually called *show()* and it can also be modified. Let's reconsider our *bear* class example in the S4 context.

```
library(methods)
setClass("bear",
         representation(
                  name = "character",

                  age = "numeric",

                  birthday = "Date"
         )
)
yog <- new("bear", name = 'Yogi', age = 20,
                        birthday = as.Date('91-08-03'))
## next notice the missing age slot
yog <- new("bear", name = 'Yogi',
         birthday = as.Date('91-08-03'))
## finally, apparently there's not a default object of class Date
yog <- new("bear", name = 'Yogi', age = 20)

## Error in validObject(.Object):  invalid class "bear" object:  invalid
object for slot "birthday" in class "bear":  got class "S4", should
be or extend class "Date"

yog

## An object of class "bear"
## Slot "name":
## [1] "Yogi"
##
## Slot "age":
## numeric(0)
##
## Slot "birthday":
## [1] "91-08-03"

yog@age <- 60
```

S4 methods are designed to be more structured than S3, with careful checking of the slots.

```
setValidity("bear",
        function(object) {
                if(!(object@age > 0 && object@age < 130))
                        return("error: age must be between 0 and 130")
                if(length(grep("[0-9]", object@name)))
                        return("error: name contains digits")
                return(TRUE)
        # what other validity check would make sense given the slots?
        }
)

## Class "bear" [in ".GlobalEnv"]
##
## Slots:
##
## Name:        name        age  birthday
## Class: character   numeric      Date

sam <- new("bear", name = "5z%a", age = 20,
        birthday = as.Date('91-08-03'))

## Error in validObject(.Object):  invalid class "bear" object:  error:
name contains digits

sam <- new("bear", name = "Z%a B''*", age = 20,
        birthday = as.Date('91-08-03'))
sam@age <- 150 # so our validity check is not foolproof
```

To deal with this latter issue of the user mucking with the slots, it's recommended when using OOP that slots only be accessible through methods that operate on the object, e.g., a *setAge()* method, and then check the validity of the supplied age within *setAge()*.

Here's how we create generic and class-specific methods. Note that in some cases the generic will already exist.

```
## generic method
setGeneric("isVoter", function(object, ...) {
                standardGeneric("isVoter")
            })
```

```
## [1] "isVoter"

# class-specific method
isVoter.bear <- function(object){
        if(object@age > 17){
                cat(object@name, "is of voting age.\n")
        } else cat(object@name, "is not of voting age.\n")
}
setMethod(isVoter, signature = c("bear"), definition = isVoter.bear)

## [1] "isVoter"
## attr(,"package")
## [1] ".GlobalEnv"

isVoter(yog)

## Yogi is of voting age.
```

We can have method signatures involve multiple objects. Here's some syntax where we'd fill in the function body with appropriate code - perhaps the plus operator would create a child.

```
setMethod(`+`, signature = c("bear", "bear"),
 definition = function(bear1, bear2) {
    ## method code goes here
}
```

As with S3, classes can inherit from one or more other classes. Chambers calls the class that is being inherited from a *superclass*.

```
setClass("grizzly_bear",
        representation(
                number_of_people_eaten = "numeric"
        ),

        contains = "bear"
)
sam <- new("grizzly_bear", name = "Sam", age = 20,
   birthday = as.Date('91-08-03'), number_of_people_eaten = 3)
```

```
isVoter(sam)

## Sam is of voting age.

is(sam, "bear")

## [1] TRUE
```

For a more relevant example suppose we had spatially-indexed time series. We could have a time series class, a spatial location class, and a "location time series" class that inherits from both. Be careful that there are not conflicts in the slots or methods from the multiple classes. For conflicting methods, you can define a method specific to the new class to deal with this. Also, if you define your own *initialize()* method, you'll need to be careful that you account for any initialization of the superclass(es) and for any classes that might inherit from your class (see help on *new()* and Chambers, p. 360).

You can inherit from other S4 classes (which need to be defined or imported into the environment in which your class is created), but not S3 classes. You can inherit (at most one) of the basic R types, but not environments, symbols, or other non-standard types. You can use S3 classes in slots, but this requires that the S3 class be declared as an S4 class. To do this, you create S4 versions of S3 classes use *setOldClass()* - this creates a virtual class. This has been done, for example, for the *data.frame* class:

```
showClass("data.frame")

## Class "data.frame" [package "methods"]
##
## Slots:
##
## Name:                .Data               names
## Class:                list           character
##
## Name:            row.names             .S3Class
## Class: data.frameRowLabels           character
##
## Extends:
## Class "list", from data part
## Class "oldClass", directly
## Class "vector", by class "list", distance 2
```

You can use *setClassUnion()* to create what Adler calls *superclass* and what Chambers calls a *virtual class* that allows for methods that apply to multiple classes. So if you have a person class and a pet class, you could create a "named lifeform" virtual class that has methods for working with name and age slots, since both people and pets would have those slots. You can't directly create an object in the virtual class.

4.4.3 Reference classes

Reference classes are a new construct in R. They are classes somewhat similar to S4 that allow us to access their fields by reference. Importantly, they behave like pointers (the fields in the objects are 'mutable'). Let's work through an example where we set up the fields of the class (like S4 slots) and class methods, including a constructor. Note that one cannot add fields to an already existing class.

Here's the initial definition of the class.

```
tsSimClass <- setRefClass("tsSimClass",
    fields = list(
        n = "numeric",
        times = "numeric",
        corMat = "matrix",
        corParam = "numeric",
        U = "matrix",
        currentU = "logical"),

    methods = list(
        initialize = function(times = 1:10, corParam = 1, ...){
            ## we seem to need default values for the copy() method
            ## to function properly
            require(fields)
            times <<- times # field assignment requires using <<-
            n <<- length(times)
            corParam <<- corParam
            currentU <<- FALSE
            calcMats()
            callSuper(...) # calls initializer of base class (envRefClass)
        },
```

```
        calcMats = function(){
            ## Python-style doc string
            ' calculates correlation matrix and Cholesky factor '
            lagMat <- rdist(times) # local variable
            corMat <<- exp(-lagMat / corParam) # field assignment
            U <<- chol(corMat) # field assignment
            cat("Done updating correlation matrix and Cholesky factor\n")
            currentU <<- TRUE
        },

        changeTimes = function(newTimes){
            times <<- newTimes
            calcMats()
        },

        show = function(){ # 'print' method
            cat("Object of class 'tsSimClass' with ", n, " time points.\n",
                sep = '')
        }
    )
)
```

We can add methods after defining the class.

```
tsSimClass$methods(list(

        simulate = function(){
                  ' simulates random processes from the model '
                   if(!currentU)
                         calcMats()
                   return(crossprod(U, rnorm(n)))
        })
)
```

Now let's see how we would use the class.

```
master <- tsSimClass$new(1:100, 10)
master
tsSimClass$help('calcMats')
devs <- master$simulate()
plot(master$times, devs, type = 'l')
mycopy <- master
myDeepCopy <- master$copy()
master$changeTimes(seq(0,1, length = 100))
mycopy$times[1:5]
myDeepCopy$times[1:5]
```

A few additional points:

- As we just saw, a copy of an object is just a pointer to the original object, unless we explicitly invoke the *copy()* method.
- As with S3 and S4, classes can inherit from other classes. E.g., if we had a *simClass* and we wanted the *tsSimClass* to inherit from it:

  ```
  setRefClass("tsSimClass", contains = "simClass")
  ```

  - We can call a method inherited from the superclass from within a method of the same name with *callSuper(...)*, as we saw for our *initialize()* method.
- If we need to refer to a field or change a field we can do so without hard-coding the field name as:

```
master$field('times')[1:5]
## the next line is dangerous in this case, since
## currentU will no longer be accurate
master$field('times', 1:10)
```

- Note that reference classes have Python style doc strings. We get help on a class with *class$help()*, e.g. `tsSimClass$help()`. This prints out information, including the doc strings.
- If you need to refer to the entire object within an object method, you refer to it as *.self*. E.g., with our *tsSimClass* object, `.self$U` would refer to the Cholesky factor. This is sometimes necessary to distinguish a class field from an argument to a method.

- There is a new, more efficient version of ReferenceClasses call R6 classes. See the *R6* package.

5 Standard dataset manipulations

Base R provides a variety of functions for manipulating data frames, but now many researchers use add-on packages (many written by Hadley Wickham) to do these manipulations in a more elegant, often more efficient way. Module 5 of the R bootcamp describes some of these new tools, but I'll summarize them here.

5.1 split-apply-combine

Often analyses are done in a stratified fashion - the same operation or analysis is done on subsets of the data set. The subsets might be different time points, different locations, different hospitals, different people, etc.

The split-apply-combine framework is intended to operate in this kind of context: first one splits the dataset by one or more variables, then one does something to each subset, and then one combines the results. The *plyr* and *dplyr* packages implement this framework, with *dplyr* newer and faster. One can also do similar operations using various flavors of the *apply()* family of functions such as *by()*, *tapply()*, and *aggregate()*, but the dplyr-based tools are often nicer to use.

5.2 Long and wide formats

Finally, we may want to convert between so-called 'long' and 'wide' formats, which we can motivate in the context of longitudinal data (multiple observations per subject) and panel data (temporal data for each of multiple units such as in econometrics). The wide format has repeated measurements for a subject in separate columns, while the long format has repeated measurements in separate rows, with a column for differentiating the repeated measurements. The wide format is useful for doing separate analyses by group, while the long format is useful for doing a single analysis that makes use of the groups, such as ANOVA or mixed models or for plotting, such as with *ggplot2*.

```
long <- data.frame(id = c(1, 1, 2, 2),
                   time = c(1980, 1990, 1980, 1990),
                   value = c(5, 8, 7, 4))
wide <- data.frame(id = c(1, 2),
                   value_1980 = c(5, 7), value_1990 = c(8, 4))
```

```
long

##   id time value
## 1  1 1980     5
## 2  1 1990     8
## 3  2 1980     7
## 4  2 1990     4

wide

##   id value_1980 value_1990
## 1  1          5          8
## 2  2          7          4
```

There are a variety of functions for converting between wide and long formats. I'd recommend the *gather()* and *spread()* functions in the tidyr package. There are also the *melt()* and *cast()* in the *reshape2* package. These are easier to use than the functions in base R such as *reshape()* or *stack()* and *unstack()* functions.

6 Functions, variable scope, and frames

R is a functional programming language. All operations are carried out by functions including assignment, various operators, printing to the screen, etc.

Functions are at the heart of R. In general, you should try to have functions be self-contained - operating only on arguments provided to them, and producing no side effects, though in some cases there are good reasons for making an exception.

Functions that are not implemented internally in R (i.e., user-defined functions) are also referred to officially as *closures* (this is their *type*) - this terminology sometimes comes up in error messages.

6.1 Functions as objects

Everything in R is an object, including functions.

```
x <- 3
x(2)

## Error in x(2):  could not find function "x"
```

```
x <- function(z) z^2
x(2)

## [1] 4

class(x); typeof(x)

## [1] "function"
## [1] "closure"
```

We can call a function based on the text name of the function.

```
myFun <- 'mean'; x <- rnorm(10)
eval(as.name(myFun))(x)

## [1] -0.159
```

We can also pass a function into another function either as the actual function object. This is one aspect of R being a functional programming language.

```
x <- rnorm(10)

f <- function(fxn, x) {
    fxn(x)
}
f(mean, x)

## [1] -0.163
```

We can also pass in a function based on a a character vector of length one with the name of the function. Here *match.fun()* is a handy function that extracts a function when the function is passed in as an argument of a function. It looks in the calling environment for the function and can handle when the function is passed in as a function object or as a character vector of length 1 giving the function name.

```
f <- function(fxn, x){
        match.fun(fxn)(x)
}
f("mean", x)
```

```
## [1] -0.163

f(mean, x)

## [1] -0.163
```

This allows us to write functions in which the user passes in the function (as an example, this works when using *outer()*). Caution: one may need to think carefully about scoping issues in such contexts.

Function objects contain three components: an argument list, a body (a parsed R statement), and an environment.

```
f1 <- function(x) y <- x^2
f2 <- function(x) {y <- x^2; z <- x^3; return(list(y, z))}
class(f1)

## [1] "function"

body(f2)

## {
##     y <- x^2
##     z <- x^3
##     return(list(y, z))
## }

typeof(body(f1)); class(body(f1))

## [1] "language"
## [1] "<-"

typeof(body(f2)); class(body(f2))

## [1] "language"
## [1] "{"
```

We'll see more about objects relating to the R language and parsed code in Section 9. For now, just realize that the parsed code itself is treated as an object(s) with certain types and certain classes.

do.call()

The *do.call()* function will apply a function to the elements of a list. For example, we can *rbind()* together (if compatible) the elements of a list of vectors instead of having to loop over the elements or manually type them in:

```
myList <- list(a = 1:3, b = 11:13, c = 21:23)
args(rbind)

## function (..., deparse.level = 1)
## NULL

rbind(myList$a, myList$b, myList$c)

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]   11   12   13
## [3,]   21   22   23

rbind(myList)

##         a          b          c
## myList Integer,3 Integer,3 Integer,3

do.call(rbind, myList)

##   [,1] [,2] [,3]
## a    1    2    3
## b   11   12   13
## c   21   22   23
```

Why couldn't we just use *rbind()* directly? Basically we're using *do.call()* to use functions that take "..." as input (i.e., functions accepting an arbitrary number of arguments) and to use the list as the input instead (i.e., to use the list elements).

More generally do.call() is a way to pass arguments to a function where the arguments are a list:

```
do.call(mean, list(1:10, na.rm = TRUE))

## [1] 5.5
```

6.2 Inputs

Arguments can be specifed in the correct order, or given out of order by specifying *name* = *value*. R first tries to match arguments by name and then by position. In general the more important arguments are specified first. You can see the arguments and defaults for a function using *args()*:

```
args(lm)

## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## NULL
```

Functions may have unspecified arguments, which are designated using '...'. Unspecified arguments occurring at the beginning of the argument list are generally a collection of like objects that will be manipulated (consider *paste()*, *c()*, and *rbind()*), while unspecified arguments occurring at the end are often optional arguments (consider *plot()*). These optional arguments are sometimes passed along to a function within the function. For example, here's my own wrapper for plotting, where any additional arguments specified by the user will get passed along to plot:

```
pplot <- function(x, y, pch = 16, cex = 0.4, ...) {
        plot(x, y, pch = pch, cex = cex, ...)
}
```

If you want to manipulate what the user passed in as the ... args, rather than just passing them along, you can extract them (the following code would be used within a function to which '...' is an argument:

```
myFun <- function(...){
  print(..2)
  args <- list(...)
  print(args[[2]])
}
myFun(1,3,5,7)

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

You can check if an argument is missing with *missing()*. Arguments can also have default values, which may be *NULL*. If you are writing a function and designate the default as *argname = NULL*, you can check whether the user provided anything using `is.null(argname)`. The default values can also relate to other arguments. As an example, consider *dgamma()*:

```
args(dgamma)

## function (x, shape, rate = 1, scale = 1/rate, log = FALSE)
## NULL
```

As we've seen, functions can be passed in as arguments (e.g., see the variants of *apply()*). Note that one does not need to pass in a named function - you can create the function on the spot - this is called an *anonymous function* (also called a *lambda function* in some languages such as Python):

```
mat <- matrix(1:9, 3)
apply(mat, 1, min)  # apply() uses match.fun()

## [1] 1 2 3

apply(mat, 2, function(vec) vec - vec[1])

##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    1    1    1
## [3,]    2    2    2

apply(mat, 1, function(vec) vec - vec[1])

##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    3    3    3
## [3,]    6    6    6

## explain why the result of the last expression is transposed
```

We can see the arguments using *args()* and extract the arguments using *formals()*. *formals()* can be helpful if you need to manipulate the arguments.

```r
f <- function(x, y = 2, z = 3 / y) { x + y + z }
args(f)

## function (x, y = 2, z = 3/y)
## NULL

formals(f)

## $x
##
##
## $y
## [1] 2
##
## $z
## 3/y

class(formals(f))

## [1] "pairlist"
```

A *pairlist* is like a list, but with pairing that in this case pairs argument names with default values.

match.call() will show the user-suppled arguments explicitly matched to named arguments.

```r
match.call(definition = mean,
  call = quote(mean(y, na.rm = TRUE)))

## mean(x = y, na.rm = TRUE)

## what do you think quote does? Why is it needed?
```

6.3 Outputs

`return(x)` will specify *x* as the output of the function. By default, if *return()* is not specified, the output is the result of the last evaluated statement. *return()* can occur anywhere in the function, and allows the function to exit as soon as it is done.

```
f <- function(x) {
        if(x < 0) {
     return(-x^2)
    } else res <- x^2
}
f(-3)

## [1] -9

f(3)
```

invisible(x) will return x and the result can be assigned in the calling environment but it will not be printed if not assigned:

```
f <- function(x){ invisible(x^2) }
f(3)
a <- f(3)
a

## [1] 9
```

A function can only return a single object (unlike Matlab, e.g.), but of course we can tack things together as a list and return that, as with *lm()* and many other functions.

```
mod <- lm(mpg ~ cyl, data = mtcars)
class(mod)

## [1] "lm"

is.list(mod)

## [1] TRUE
```

6.4 Approaches to passing arguments to functions

6.4.1 Pass by value vs. pass by reference

When talking about programming languages, one often distinguishes *pass-by-value* and *pass-by-reference*. Pass-by-value means that when a function is called with one or more arguments, a copy

is made of each argument and the function operates on those copies. Pass-by-reference means that the arguments are not copied, but rather that information is passed allowing the function to find and modify the original value of the objects passed into the function. In pass-by-value, changes to an argument made within a function do not affect the value of the argument in the calling environment. In pass-by-reference changes inside a function do affect the object outside of the function. R is (roughly) pass-by-value. R's designers chose not to allow pass-by-reference because they didn't like the idea that a function could have the side effect of changing an object. However, passing by reference can sometimes be very helpful, and we'll see ways of passing by reference later in this Unit (and also note our discussion of Reference Classes).

Pass-by-value is elegant and modular in that functions do not have side effects - the effect of the function occurs only through the return value of the function. However, it can be inefficient in terms of the amount of computation and of memory used. In contrast, pass-by-reference is more efficient, but also more dangerous and less modular. It's more difficult to reason about code that uses pass-by-reference because effects of calling a function can be hidden inside the function.

An important exception is *par()*. If you change graphics parameters by calling *par()* in a user-defined function, they are changed permanently outside of the function. One trick is as follows:

```
f <- function(){
        oldpar <- par()
        par(cex = 2)
        # body of code
        par() <- oldpar
}
```

Note that changing graphics parameters within a specific plotting function - e.g., `plot(x, y, pch = '+')`, doesn't change things except for that particular plot. Can you think of other R functions that have side effects?

Pointers By way of contrast to a pass-by-value system, I want to briefly discuss the idea of a pointer, common in compiled languages such as C.

```
int x = 3;
int* ptr;
ptr = &x;
*ptr * 7; // returns 21
```

Here *ptr* is the address of the integer *x*.

Vectors in C are really pointers to a block of memory:

```
int x[10];
```

In this case x will be the address of the first element of the vector. We can access the first element as `x[0]` or `*x`.

Why have we gone into this? In C, you can pass a pointer as an argument to a function. The result is that only the scalar address is copied and not the entire vector, and inside the function, one can modify the original vector, with the new value persisting on exit from the function. For example:

```
int myCal(int* ptr){
    *ptr = *ptr + *ptr;
}
```

When calling C or C++ from R, one (implicitly) passes pointers to the vectors into C. Let's see an example:

```
out <- rep(0, n)
out <- .C("logLik", out = as.double(out),
                    theta = as.double(theta))$out
```

In C, the function definition looks like this:

```
void logLik(double* out, double* theta)
```

6.4.2 Promises and lazy evaluation

In actuality, R is not quite pass-by-value; rather it is *call-by-value*. Copying of arguments is delayed in two ways. The first is the idea of promises and lazy evaluation, described here. The second is the idea of *copy-on-change*, described in Section 8. Basically, with copy-on-change, copies of arguments are only made if the argument is changed within the function. Until then the object in the function just refers back to the original object.

Let's see what a *promise* object is. In function calls, when R matches user input arguments to formal argument names, it does not (usually) evaluate the arguments until they are needed, which is called *lazy evaluation*. Instead the formal arguments are of a special type called a *promise*. Let's see lazy evaluation in action. Do you think the following code will run?

```
f <- function(a, b = d) {
        d <- log(a);
        return(a*b)
}
f(7)
```

What's strange about that?

Another example:

```
f <- function(x) print("hi")
system.time(mean(rnorm(1000000)))

##    user  system elapsed
##   0.064   0.004   0.071

system.time(f(3))

## [1] "hi"
##    user  system elapsed
##       0       0       0

system.time(f(mean(rnorm(1000000))))

## [1] "hi"
##    user  system elapsed
##       0       0       0
```

Where are arguments evaluated? User-supplied arguments are evaluated in the calling frame, while default arguments are evaluated in the frame of the function:

```
z <- 3
x <- 100
f <- function(x, y = x*3) {x+y}
f(z*5)

## [1] 60
```

Here, when *f()* is called, *z* is evaluated in the calling frame and `z*5` is assigned to *x* in the frame of the function, while `y = x*3` is evaluated in the frame of the function.

Challenge: How could I experimentally determine if the default argument is treated as a promise as well?

6.5 Operators

Operators, such as '*+*', '*[*' are just functions, but their arguments can occur both before and after the function call:

```
a <- 7; b <- 3
# let's think about the following as a mathematical function
#  -- what's the function call?
a + b

## [1] 10

`+`(a, b)

## [1] 10
```

In general, you can use back-ticks to refer to the operators as operators instead of characters. In some cases single or double quotes also work. We can look at the code of an operator as follows using back-ticks to escape out of the standard R parsing, e.g., `` `%*%` ``.

Finally, since an operator is just a function, you can use it as an argument in various places:

```
x <- 1:3; y <- c(100,200,300)
outer(x, y, `+`)

##      [,1] [,2] [,3]
## [1,]  101  201  301
## [2,]  102  202  302
## [3,]  103  203  303

myList <- list(list(a = 1:5, b = "sdf"), list(a = 6:10, b = "wer"))
myMat <- sapply(myList, `[[`, 1)
## note that the index "1" is the additional argument to the [[ function
myMat

##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10

cbind(myList[[1]][[1]], myList[[2]][[1]])  ## equivalent but doesn't scale
```

```
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

You can define your own *binary* operator (an operator taking two arguments) using a string inside % symbols. Here's how we could do Python-style string addition:

```
`%+%` <- function(a, b) paste0(a, b, collapse = '')
"Hi " %+% "there"

## [1] "Hi there"
```

Since operators are just functions, there are cases in which there are optional arguments that we might not expect. Here's how to pass a sometimes useful argument to the bracket operator (in this case avoiding conversion from a matrix to a vector, which can mess up subsequent code).

```
mat <- matrix(1:4, 2, 2)
mat[ , 1]

## [1] 1 2

mat[ , 1, drop = FALSE] # what's the difference?

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

6.6 Unexpected functions and replacement functions

All code in R can be viewed as a function call.

What do you think is the functional version of the following code? What are the arguments?

```
if(x > 27){
        print(x)
} else{
```

```
        print("too small")
}
```

Assignments that involve functions or operators on the left-hand side (LHS) are called *replacement expressions* or *replacement functions*. These can be quite handy. Here are a few examples:

```
diag(mat) <- c(3, 2)
is.na(vec) <- 3
names(df) <- c('var1', 'var2')
```

Replacement expressions are actually function calls. The R interpreter calls the replacement function (which often creates a new object that includes the replacement) and then assigns the result to the name of the original object.

```
mat <- matrix(rnorm(4), 2, 2)
diag(mat) <- c(3, 2)
mat <- `diag<-`(mat, c(10, 21))
base::`diag<-`

## function (x, value)
## {
##     dx <- dim(x)
##     if (length(dx) != 2L)
##         stop("only matrix diagonals can be replaced")
##     len.i <- min(dx)
##     len.v <- length(value)
##     if (len.v != 1L && len.v != len.i)
##         stop("replacement diagonal has wrong length")
##     if (len.i) {
##         i <- seq_len(len.i)
##         x[cbind(i, i)] <- value
##     }
##     x
## }
## <bytecode: 0x3973e30>
## <environment: namespace:base>
```

The old version of *mat* still exists until R's memory management cleans it up, but it's no longer referred to by the symbol '*mat*'. Occasionally this sort of thing might cause memory usage to increase (for example it's possible if you're doing replacements on large objects within a loop), but in general things should be fine.

You can define your own replacement functions like this, with the requirements that the last argument be named '*value*' and that the function return the entire object:

```
yog <- list(firstName = 'Yogi', lastName = 'Bear')
`firstName<-` <- function(obj, value){
  obj$firstName <- value
  return(obj)
}
firstName(yog) <- 'Yogisandra'
```

6.7 Variable scope

In this section, we seek to understand what happens in the following circumstance. Namely, where does R get the value for the object 'x'?

```
f <- function(y) {
  return(x + y)
}
f(3)

## [1] 4 5 6
```

To consider variable scope, we need to define the terms *environment* and *frame*. Environments and frames are closely related.

- A *frame* is a collection of named objects.
- An *environment* is a frame, with a pointer to the 'enclosing environment', i.e., the next environment to look for something in. (Be careful as this is different than the parent frame of a function.)

Variables in the enclosing environment (also called the parent environment) are available within a function. This is the analog of *global variables* in other languages. The enclosing environment is the environment in which a function is defined, not the environment from which a function is called.

Be careful when using variables from the enclosing environment as the value of that variable in the enclosing environment may well not be what you expect it to be. In general it's bad practice to use variables that are taken from environments outside that of a function, but in some cases it can be useful. Here are some examples of using variables outside of the frame of a function.

```
x <- 3
f <- function() {x <- x^2; print(x)}
f()
x # what do you expect?
f <- function() { assign('x', x^2, env = .GlobalEnv) }
## careful: could be dangerous as a variable is changed as a side effect
f()
x
f <- function(x) { x <<- x^2 }
## careful: could be dangerous as a variable is changed as a side effect
f(5)
x
```

Let's dig deeper to understand where R looks for non-local variables. **R looks for variables that are not local to a function in the enclosing environment of the function, where the enclosing environment is the environment in which the function was defined.** Note that the enclosing/parent environment is NOT the environment from which the function was called. This approach is called *lexical scoping*.

Here are some examples to illustrate scope:

```
x <- 3
f <- function() {
    f2 <- function() { print(x) }
    f2()
}
f() # what will happen?

f <- function() {
    f2 <- function() { print(x) }
    x <- 7
    f2()
}
```

```
f() # what will happen?

f2 <- function() print(x)
f <- function() {
    x <- 7
    f2()
}
f() # what will happen?
```

Here's a somewhat tricky example:

```
y <- 100
f <- function(){
        y <- 10
        g <- function(x) x + y
        return(g)
}
## you can think of f() as a function constructor
h <- f()
h(3)

## [1] 13
```

Let's work through this:

1. What is the enclosing environment of the function *g()*?
2. What does *g()* use for *y*?
3. When *f()* finishes, does its environment disappear? What would happen if it did?
4. What is the enclosing environment of *h()*?

This code helps explain things, but it's a bit confusing because *environment()* gives back different results depending on whether it is given a function as its argument. If given a function, it returns the enclosing environment for that function. If given no argument, it returns the current execution environment.

```r
environment(h)  # enclosing environment of h()

## <environment: 0x6741e48>

ls(environment(h)) # objects in that environment

## [1] "g" "y"

f <- function(){
        print(environment()) # execution environment of f()
        y <- 10
        g <- function(x) x + y
        return(g)
}
h <- f()

## <environment: 0x51ca770>

environment(h)

## <environment: 0x51ca770>

h(3)

## [1] 13

environment(h)$y

## [1] 10

## advanced: explain this:
environment(h)$g

## function(x) x + y
## <environment: 0x51ca770>
```

Comprehension problem Here's a case where something I tried failed and I had to think more carefully about scoping to understand why.

```
set.seed(1)
rnorm(1)

## [1] -0.626

save(.Random.seed, file = 'tmp.Rda')
rnorm(1)

## [1] 0.184

tmp <- function() {
  load('tmp.Rda')
  print(rnorm(1))
}
tmp()

## [1] -0.836
```

Question: what was I hoping that code to do, and why didn't it work?

6.8 Environments and the search path

So far we've seen lexical scoping in action primarily in terms of finding variables in a single enclosing environment. But what if the variable is not found in either the frame/environment of the function or the enclosing environment? When R goes looking for an object (in the form of a symbol), it starts in the current environment (e.g., the frame/environment of a function) and then runs up through the enclosing environments, until it reaches the global environment, which is where R starts when you open R (it actually continues further up; see below). In general, as we've seen, these environments are not the environments of the calling function(s) - i.e., they are *not* the frames on the stack (see the next Section).

By default objects are created in the global environment, *.GlobalEnv*. As we've seen, the environment within a function call has as its enclosing environment the environment where the function was defined (not the environment from which it was called), and based on lexical scoping this is next place that is searched if an object can't be found in the frame of the function call. As an example, if an object couldn't be found within the environment of an *lm()* function call, R would first look in the environment (i.e., the *namespace*) of the stats package (since this is the environment where *lm()* is defined and is therefore the enclosing environment for *lm()*), then in packages imported by the stats package, then the base package, and then the global environment.

If R can't find the object when reaching the global environment, it runs through the search path, which you can see with *search()*. The search path is a set of additional environments. Generally packages are created with namespaces, i.e., each has its own environment, as we see based on *search()*.

```
search()
```

```
##  [1] ".GlobalEnv"        "package:fields"
##  [3] "package:maps"      "package:spam"
##  [5] "package:grid"      "package:methods"
##  [7] "package:pryr"      "package:knitr"
##  [9] "package:stats"     "package:graphics"
## [11] "package:grDevices" "package:utils"
## [13] "package:datasets"  "package:SCF"
## [15] "Autoloads"         "package:base"
```

```
searchpaths()
```

```
##  [1] ".GlobalEnv"
##  [2] "/system/linux/lib/R-16.04/3.4/x86_64/site-library/fields"
##  [3] "/system/linux/lib/R-16.04/3.4/x86_64/site-library/maps"
##  [4] "/system/linux/lib/R-16.04/3.4/x86_64/site-library/spam"
##  [5] "/usr/lib/R/library/grid"
##  [6] "/usr/lib/R/library/methods"
##  [7] "/system/linux/lib/R-16.04/3.4/x86_64/site-library/pryr"
##  [8] "/system/linux/lib/R-16.04/3.4/x86_64/site-library/knitr"
##  [9] "/usr/lib/R/library/stats"
## [10] "/usr/lib/R/library/graphics"
## [11] "/usr/lib/R/library/grDevices"
## [12] "/usr/lib/R/library/utils"
## [13] "/usr/lib/R/library/datasets"
## [14] "/system/linux/lib/R-16.04/3.4/x86_64/site-library/SCF"
## [15] "Autoloads"
## [16] "/usr/lib/R/library/base"
```

We can also see the nestedness of environments using the following code, using *environmentName()*, which prints out a nice-looking version of the environment name.

```
x <- environment(lm)
while (environmentName(x) != environmentName(emptyenv())) {
        print(environmentName(x))
        x <- parent.env(x)
}

## [1] "stats"
## [1] "imports:stats"
## [1] "base"
## [1] "R_GlobalEnv"
## [1] "package:fields"
## [1] "package:maps"
## [1] "package:spam"
## [1] "package:grid"
## [1] "package:methods"
## [1] "package:pryr"
## [1] "package:knitr"
## [1] "package:stats"
## [1] "package:graphics"
## [1] "package:grDevices"
## [1] "package:utils"
## [1] "package:datasets"
## [1] "package:SCF"
## [1] "Autoloads"
## [1] "base"

library(pryr)
x <- environment(lm)
parenvs(x, all = TRUE)

##    label
## 1  <environment: namespace:stats>
## 2  <environment: 0x206b1f8>
## 3  <environment: namespace:base>
## 4  <environment: R_GlobalEnv>
## 5  <environment: package:fields>
## 6  <environment: package:maps>
```

```
## 7  <environment: package:spam>
## 8  <environment: package:grid>
## 9  <environment: package:methods>
## 10 <environment: package:pryr>
## 11 <environment: package:knitr>
## 12 <environment: package:stats>
## 13 <environment: package:graphics>
## 14 <environment: package:grDevices>
## 15 <environment: package:utils>
## 16 <environment: package:datasets>
## 17 <environment: package:SCF>
## 18 <environment: 0x110fe20>
## 19 <environment: base>
## 20 <environment: R_EmptyEnv>
##    name
## 1  ""
## 2  "imports:stats"
## 3  ""
## 4  ""
## 5  "package:fields"
## 6  "package:maps"
## 7  "package:spam"
## 8  "package:grid"
## 9  "package:methods"
## 10 "package:pryr"
## 11 "package:knitr"
## 12 "package:stats"
## 13 "package:graphics"
## 14 "package:grDevices"
## 15 "package:utils"
## 16 "package:datasets"
## 17 "package:SCF"
## 18 "Autoloads"
## 19 ""
## 20 ""
```

Note that eventually the global environment and the environments of the packages are nested within the base environment (of the base package) and the empty environment. Note that here *parent* **is** referring to the enclosing environment, even though it is best to talk about *enclosing environment* rather than parent environment.

We can retrieve and assign objects in a particular environment and/or namespace as follows:

```
lm <- function() {return(NULL)} # this seems dangerous but isn't
x <- 1:3; y <- rnorm(3); mod <- lm(y ~ x)

## Error in lm(y ~ x): unused argument (y ~ x)

mod <- get('lm', pos = 'package:stats')(y ~ x)
mod <- stats::lm(y ~ x) # an alternative
## :: is the namespace resolution operator
rm(lm)
mod <- lm(y ~ x)
```

Note that our (bogus) *lm()* function masks but does not overwrite the default function. If we remove ours, then the default one is still there.

6.9 Frames and the call stack

R keeps track of the call stack, which is the series of nested calls to functions. The stack operates like a stack of cafeteria trays - when a function is called, it is added to the stack (pushed) and when it finishes, it is removed (popped). There are a bunch of functions that let us query what frames are on the stack and access objects in particular frames of interest. This gives us the ability to work with objects in the environment(s) from which a function was called.

sys.nframe() returns the number of the current frame and *sys.parent()* the number of the parent, while *parent.frame()* gives the name of the environment of the parent frame. Careful: here, *parent* refers to the parent in terms of the call stack and has nothing to do with enclosing environments. *sys.frame()* gives the name of the environment for a given frame number (for non-negative numbers). For negative numbers, it goes back that many frames in the call stack and returns the name of the associated environment. I won't print the results here because *knitr* messes up the frame counting somehow.

```
## NOTE: run this chunk outside RStudio as it seems to inject additional fr
sys.nframe()
f <- function() {
```

```
        cat('f: Frame number is ', sys.nframe(),
            '; parent frame number is ', sys.parent(), '.\n', sep = '')
        cat('f: Frame (i.e., environment) is: ')
        print(sys.frame(sys.nframe()))
        cat('f: Parent is ')
        print(parent.frame())
        cat('f: Two frames up is ')
        print(sys.frame(-2))
}
f()
f2 <- function() {
        cat('f2: Frame (i.e., environment) is: ')
        print(sys.frame(sys.nframe()))
        cat('f2: Parent is ')
        print(parent.frame())
        f()
}
f2()
```

Now let's look at some code that gets more information about the call stack and the frames involved using *sys.status()*, *sys.calls()*, *sys.parents()* and *sys.frames()*.

```
## exploring functions that give us information the frames in the stack
g <- function(y) {
        gg <- function() {
            ## this gives us the information from sys.calls(),
            ##  sys.parents() and sys.frames() as one object
                ## print(sys.status())
                tmp <- sys.status()
            print(tmp)
        }
        if(y > 0) g(y-1) else gg()
}
g(3)
```

Challenge: why did I not do `print(sys.status())` directly?

If you're interested in parsing a somewhat complicated example of frames in action, Adler provides a user-defined timing function that evaluates statements in the calling frame.

6.10 Alternatives to pass by value in R

There are occasions we do not want to pass by value. In addition to avoiding copies and the attendant computation and memory use, another reason is when we want a function to modify a complicated object without having to return it and re-assign it in the parent environment. There are several work-arounds:

1. We can use Reference Class (or R6) objects.

2. We can access the object in the enclosing environment as a 'global variable', as we've seen when discussing scoping. More generally we can access the object using *get()*, specifying the environment from which we want to obtain the variable. To specify the location of an object when using *get()*, we can generally specify (1) a position in the search path, (2) an explicit environment, or (3) a location in the call stack by using *sys.frame()*. However we cannot change the value of the object in the parent environment without some additional tools:

   (a) We can use the '<<-' operator to assign into an object in the parent environment (provided an object of that name exists in the parent environment).

   (b) We can also use *assign()*, specifying the environment in which we want the assignment to occur.

   While these techniques are possible and ok for exploratory coding, they're bad practice for more formal code development.

3. We can use replacement functions (Section 6.6), which hide the reassignment in the parent environment from the user. Note that a second copy is generally created in this case, but the original copy is quickly removed.

4. We can use a *closure*, which is a function with associated data. This Wikipedia entry nicely summarizes the idea, which is not an R-specific construct. This involves creating one or more functions within a function call and returning the function(s) as the output. When one executes the original function, the new functions are created and returned as a list and one can call the functions in that list. Those functions then can access objects in the enclosing environment (the environment of the original function) and can use '<<-' to assign into the

enclosing environment, to which all the functions have access. Chambers provides an example of this in Sec. 5.4.

```
x <- rnorm(10)
f <- function(input){
        data <- input
        g <- function(param) return(param * data)
        return(g)
}
myFun <- f(x)
rm(x) # to demonstrate we no longer need x
myFun(3)

##  [1]  1.462  2.215  1.727 -0.916  4.535  1.170 -1.864
##  [8] -6.644  3.375 -0.135

x <- rnorm(1e7)
myFun <- f(x)
object.size(myFun) # hmmm

## 3584 bytes

object.size(environment(myFun)$data)

## 80000040 bytes
```

Here's a fun example. You might do this with an *apply()* variant, in particular *replicate()*, but this is slick:

```
make_container <- function(n) {
        x <- numeric(n)
        i <- 1

        function(value = NULL) {
                if (is.null(value)) {
                        return(x)
                } else {
```

```
                        x[i] <<- value
                        i <<- i + 1
                }
        }
}
nboot <- 100
bootmeans <- make_container(nboot)
data <- faithful[ , 1] # Old Faithful geyser eruption lengths
for (i in 1:nboot)
        bootmeans(mean(sample(data, length(data),
      replace=TRUE)))
## this will place results in x in the function env't and you can grab it o
bootmeans()

##   [1] 3.43 3.42 3.37 3.43 3.31 3.57 3.44 3.45 3.41 3.45
##  [11] 3.64 3.60 3.58 3.52 3.61 3.53 3.41 3.44 3.60 3.39
##  [21] 3.54 3.57 3.51 3.49 3.57 3.48 3.43 3.51 3.51 3.48
##  [31] 3.48 3.48 3.51 3.61 3.45 3.52 3.55 3.43 3.50 3.52
##  [41] 3.50 3.46 3.45 3.58 3.36 3.50 3.51 3.56 3.50 3.34
##  [51] 3.39 3.52 3.53 3.43 3.55 3.31 3.54 3.54 3.52 3.51
##  [61] 3.47 3.57 3.47 3.42 3.48 3.39 3.42 3.49 3.53 3.38
##  [71] 3.53 3.60 3.59 3.60 3.45 3.67 3.51 3.60 3.48 3.54
##  [81] 3.44 3.49 3.57 3.54 3.53 3.55 3.50 3.53 3.38 3.46
##  [91] 3.44 3.54 3.34 3.59 3.51 3.55 3.56 3.34 3.42 3.49
```

- A related approach is to wrap data with a function using *with()*.

```
x <- rnorm(10)
myFun2 <- with(list(data = x), function(param) return(param * data))
rm(x)
myFun2(3)

##  [1] -4.614  2.132  2.458  4.069 -3.814  3.707 -3.463
##  [8]  2.041  0.021 -2.204

x <- rnorm(1e7)
```

```
myFun2 <- with(list(data = x), function(param) return(param * data))
object.size(myFun2)

## 1560 bytes
```

Question: When would it be useful to have an object carried along with a function as done here?

6.11 Creating and working in an environment

We've already talked extensively about the environments that R creates. Occasionally you may want to create an environment in which to store objects.

```
e <- new.env()
assign('x', 3, envir = e) # same as e$x <- 3
e$x

## [1] 3

get('x', envir = e, inherits = FALSE)

## [1] 3

## the FALSE avoids looking for x in the enclosing environments
e$y <- 5
ls(e)

## [1] "x" "y"

rm('x', envir = e)
parent.env(e)

## <environment: R_GlobalEnv>
```

Before the existence of Reference Classes, using an environment was one way to pass objects by reference, avoiding having to re-assign the output. Here's an example where we iteratively update a random walk.

```
myWalk <- new.env(); myWalk$pos = 0
nextStep <- function(walk) walk$pos <- walk$pos +
    sample(c(-1, 1), size = 1)
nextStep(myWalk)
```

We can use *eval()* to evaluate some code within a specified environment. By default, it evaluates in the result of *parent.frame()*, which amounts to evaluating in the frame from which *eval()* was called. *evalq()* avoids having to use *quote()*. Here we override the default and evaluate in the *myWalk* environment we created:

```
eval(quote(pos <- pos + sample(c(-1, 1), 1)), envir = myWalk)
evalq(pos <- pos + sample(c(-1, 1), 1), envir = myWalk)
```

6.12 Summing up

What happens when an R function is evaluated? The user-provided function arguments are evaluated in the calling environment and the results are matched to the argument names in the function definition. A new environment with its own frame is created, with the frame on the call stack. Assignment to the argument names is done in the environment, including any default arguments. The body of the function is evaluated in the environment. Any look-up of variables not found in the environment is done using R's lexical scoping rules to look in the series of enclosing environments. When the function finishes, the return value is passed back to the calling frame and the frame is taken off the stack. The environment is removed, unless the environment serves as the enclosing environment of another environment.

7 Efficiency

In part because R is an interpreted language and in part because R is very dynamic (objects can be modified essentially arbitrarily after being created), R can be slow. Hadley Wickham's Advanced R book has a section on Performance that discusses this in detail. However, there are a variety of ways that one can write efficient R code.

In general, make use of R's built-in functions, as these tend to be implemented internally (i.e., via compiled code in C or Fortran). In particular, if R is linked to optimized BLAS and Lapack code (e.g. Intel's *MKL*, *OpenBLAS* [on BCE and the SCF Linux servers], AMD's *ACML* [on the SCF Linux cluster, *vecLib* for Macs [on the SCF Macs]), you should have good performance

(potentially comparable to Matlab and to coding in C). Sometimes you can figure out a trick to take your problem and transform it to make use of the built-in functions.

Note that I run a lot of MCMCs so I pay attention to making sure my calculations are fast as they are done repeatedly. Similarly, one would want to pay attention to speed when doing large simulations and bootstrapping, and in some cases for optimization. And if you're distributing code, it's good to have it be efficient. But in other contexts, it may not be worth your time. Also, it's good practice to code it transparently first to reduce bugs and then to use tricks to speed it up and make sure the fast version works correctly.

Results can vary with with your system setup and version of R, so the best thing to do is figure out where the bottlenecks are in your code (e.g., with *Rprof()* or just some basic use of *system.time()* and *benchmark()*), and then play around with alternative specifications. And as you gain more experience, you'll get some intuition for what approaches might improve speed, but even with experience I find myself often surprised by what matters and what doesn't. It's often worth trying out a bunch of different ideas; *system.time()* and *benchmark()* are your workhorse tools in this context.

For material on efficient R coding, including tools for timing and profiling your code to understand where the bottlenecks are, see the tutorial, *Writing efficient R code.*

7.1 A note on hashing

In the tutorial on efficient R coding, I mention that looking up objects by name in an R environment occurs via hashing, so it can be very fast. I'll briefly describe what hashing is here.

A hash function is a function that takes as input some data and maps it to a fixed-length output that can be used as a shortened reference to the data. We've seen this in the context of git commits where each commit was labeled with a long base-16 number. This also comes up when verifying files on the Internet. You can compute the hash value on the file you get and check that it is the same as the hash value associated with the legitimate copy of the file.

For our purposes here, hashing can allow one to look up values by their name via a hash table. The idea is that you have a set of key-value pairs (sometimes called a dictionary) where the key is the name associated with the value and the value is some arbitrary object. Hashing allows one to quickly determine an index associated with the key and therefore quickly find the relevant value based on the index. For example, one approach is to compute the hash as a function of the key and then take the remainder when dividing by the number of key-value pairs to get the index. Here's the procedure in pseudocode:

```
hash = hashfunc(key)
index = hash %% array_size
```

```
## %% is modulo operator - it gives the remainder
```

In general, there will be collisions, with multiple keys assigned to the same index, but usually there will be a small number of keys associated with a given index or slot, and determining the correct value within a given index/slot (also called a bucket) is fast. Put another way, the hash function distributes the keys amongst an array of buckets and allows one to look up the appropriate bucket quickly based on the computed index value. When the hash table is properly set up, the cost of looking up a value does not depend on the number of key-value pairs stored.

7.2 Other approaches to speeding up R

7.2.1 pqR and other R engines

Radford Neal, a prominent statistician/computer scientist has been working on a project called *pqR* (pretty quick R) to rewrite some aspects of R to make them faster. There are also a few other projects that aim to reimplement the "R engine" such that one could run one's R code with different back ends.

Here are some of the highlights of *pqR* in terms of efficiency, as discussed at http://radfordneal.github.io/pqR/:

1. When R runs code such as just below, it actually creates a vector 1,2,....,n, (which can be computationally and memory intensive for large n) and then iterates over the values in the vector. pqR avoids this vector creation.

   ```
   for(i in 1:n) { }
   vec[1:n]
   ```

2. As we'll see in Section 8.6, R often avoids making copies of objects when it is not necessary. However, the scheme used to do this can be improved so that even fewer copies are made.

3. pqR automatically uses multiple cores for some calculations.

4. pqR avoid some checks for NA and NaN and the like in matrix calculations in which such checking would be slow and it doesn't make sense to check for them anyway.

7.2.2 Byte compiling

R now allows you to compile R code, which goes by the name of byte compiling. Byte-compiled code is a special representation that can be executed more efficiently because it is in the form of compact codes that encode the results of parsing and semantic analysis of scoping and other

complexities of the R source code. This byte code can be executed faster than the original R code because it skips the stage of having to be interpreted by the R interpreter.

The functions in the *base* and *stats* packages are now byte-compiled by default. (If you print out a function that is byte-compiled, you'll see something like <*bytecode: 0x243a368*> at the bottom.

We can byte compile our own functions using *cmpfun()*. Here's an example (silly since we we actually do this calculation using vectorized operations):

```
library(compiler); library(rbenchmark)
f <- function(x){
        for(i in 1:length(x)) x[i] <- x[i] + 1
        return(x)
}
fc <- cmpfun(f)
fc # notice the indication that the function is byte compiled.

## function(x){
## \tfor(i in 1:length(x)) x[i] <- x[i] + 1
## \treturn(x)
## }
## <bytecode: 0x573d398>

x <- rnorm(100000)
benchmark(f(x), fc(x), x <- x + 1, replications = 5)

##         test replications elapsed relative user.self
## 2      fc(x)            5   0.036       NA     0.036
## 1       f(x)            5   0.045       NA     0.044
## 3 x <- x + 1            5   0.000       NA     0.000
##   sys.self user.child sys.child
## 2        0          0         0
## 1        0          0         0
## 3        0          0         0
```

You can compile an entire source file with *cmpfile()*, which produces a *.Rc* file. You then need to use *loadcmp()* to load in the *.Rc* file, which runs the code.

Unfortunately, in my experience, byte compiling doesn't usually speed things up much. As experienced R programmers we would never write the unvectorized code above.

7.3 Challenges

One or more of these challenges may appear on a problem set.

Challenge 1: Here's a calculation of the sort needed in mixture component modeling. I have a vector of n observations. I need to find the likelihood of each observation under each of p mixture components (i.e., what's the likelihood if it came from each of the components). So I should produce a matrix of n rows and p columns where the value in the ith row, jth column is the likelihood of the ith observation under the jth mixture component. The idea is that the likelihoods for a given observation are used in assigning observations to clusters. A naive implementation is:

```
lik <- matrix(as.numeric(NA), nr = n, nc = p)
for(j in 1:p) lik[ , j] <- dnorm(y, mns[j], sds[j])
```

Note that *dnorm()* can handle matrices and vectors as the observations **and** as the means and sds, so there are multiple ways to do this.

Challenge 2: Here's a calculation of the sort needed in a mixed membership model, where each observation is associated with some number of components. Suppose you have

$$y_i \sim \mathcal{N}(\sum_{k=1}^{m_i} w_{i,k}\mu_{ID[i,k]}, \sigma^2)$$

for a large number of observations, n. I give you a vector of μ values and a ragged list of weights (i.e., the number of weights varies by observation) and a ragged list of IDs identifying the cluster corresponding to each weight (note m_i varies by observation). Figure out how to calculate the vector of means, $\sum_k w_{i,k}\mu_{ID[i,k]}$ as fast as possible. Suppose that m_i never gets too big (but μ might have many elements) - could this help you? Part of thinking this through involves thinking about how you want to store the information so that the calculations can be done quickly. The data file *mixed-member.Rda* contains example data for two scenarios: Scenario A has many μ values and Scenario B has few μ values.

Challenge 3: Write code that simulates a random walk in two dimensions for n steps. First write out a straightforward implementation that involves looping. Then try to speed it up. The *cumsum()* function may be helpful.

Challenge 4: Determine if it's faster to subset based on vector of indices or a vector of logicals. Determine if it matters how big the original object is and how large the subset is, as well as whether the vector of indices is ordered.

Challenge 5: Figure out how to improve the efficiency of the following code chunk, which is part of a likelihood calculation for a student's PhD research. Some test data is in *likLoops.Rda*.

```
for (i in 1:n) {
  for (j in 1:n) {
    for (z in 1:K) {
       if (theta.old[i, z]*theta.old[j, z] == 0){
          q[i, j, z] <- 0
       } else {
          q[i, j, z] <- theta.old[i, z]*theta.old[j, z] /
               Theta.old[i, j]
       }
    }
  }
}
theta.new <- theta.old
for (z in 1:K) {
   theta.new[,z] <- rowSums(A*q[,,z])/sqrt(sum(A*q[,,z]))
}
```

Challenge 6: Another problem involving a computation from a student's PhD research. The following is the probability mass function for an overdispersed binomial random variable:

$$\begin{aligned} P(Y &= y) = \frac{f(y; n, p, \phi)}{\sum_{k=0}^{n} f(k; n, p, \phi)} \\ f(k; n, p, \phi) &= \binom{n}{k} \frac{k^k (n-k)^{n-k}}{n^n} \left(\frac{n^n}{k^k (n-k)^{n-k}} \right)^{\phi} p^{k\phi} (1-p)^{(n-k)\phi}, \end{aligned}$$

where the denominator serves as a normalizing constant to ensure this is a valid probability mass function. How would one efficiently code the computation of the denominator? For our purposes here you can take $n = 2000$, $p = 0.3$ and $\phi = 0.5$ when you need to actually run your code.

Challenge 7: Yet another problem from a student's PhD research. This is a very simplified version of a bioinformatics problem.

Fact 1: DNA sequencing involves chopping up the long sequence of DNA on a chromosome into many short *reads*, which may overlap and are of various lengths.

Fact 2: DNA is composed of interspersed chunks of DNA that code for proteins (exons) and chunks that do not code for proteins (introns). The introns are stripped out during creation of the protein.

As part of a workflow, the student needed to determine, for a large number of exons, how many reads each exon overlapped with. The file *exons.Rda* has position information for the reads (in an

object called *reads*) as well as position information for a number of exons (in *exons*). Each element of *reads*, which is a list, is a set of short sequences making up a single read. So the challenge is to figure out, for each exon, how many of the reads there are for which the exon overlaps at least one of the short sequences making up the read.

Challenge 8: And yet another problem from a student's PhD research (this one from a discussion just a few weeks ago).

The goal is to write a very fast function for selecting a random sample of size k from a population of size n.

There are two algorithms:

1. PIKK algorithm: Generate n random numbers and choose the elements of the population corresponding to the k smallest numbers.

2. Fisher-Yates-Knuth-Durstenfeld shuffle algorithm: This involves considering each element in turn and swapping it with a random element later in the set. Then return the first k.

Here are implementations of the two algorithms. Can we speed up either of them without resorting to coding in a faster language?

```
PIKK <- function(n, k) {
    ## return indices of the sample of size k
    sort(runif(n), index.return = TRUE)$ix[1:k]
}

FYKD <- function(n, k) {
    indices <- seq_len(n)
    for(i in 1:n) {
        j = sample(i:n, 1)
        tmp <- indices[i]
        indices[i] <- indices[j]
        indices[j] <- tmp
    }
    return(indices[1:k])
}
```

Challenge 9: Suppose we have a matrix in which each row is a vector of probabilities that add to one, and we want to generate a categorical sample based on each row. E.g., the first row might be (0.9, 0.05, 0.05) and the second row might be (0.1, 0.85, .0.5). When we generate the

first sample, it is very likely to be a 1 and the second sample is very likely to be a 2. We could do this using a for loop over the rows of the matrix, and the *sample()* function, but that is a lot slower than some other ways we might do it. How can we do it faster?

```
n <- 100000
p <- 5  ## number of categories

## way to generate a random matrix of row-normalized probabilities:
tmp <- exp(matrix(rnorm(n*p), nrow = n, ncol = p))
probs <- tmp / rowSums(tmp)

smp <- rep(0, n)

## loop by row and use sample()
set.seed(1)
system.time(for(i in seq_len(n)) smp[i] <- sample(p, 1, prob = probs[i, ]))
```

8 Evaluating memory use

The main things to remember when thinking about memory use are: (1) numeric vectors take 8 bytes per element and (2) we need to keep track of when large objects are created, including in the frames of functions.

In some of our work here we'll use functions from the *pryr* package, which provides functions to help understand what is going on under the hood in R.

In general, don't try to run this code within RStudio, as some of how RStudio works affects memory use.

8.1 Allocating and freeing memory

Unlike compiled languages like C, in R we do not need to explicitly allocate storage for objects. However, we have seen that there are times that we do want to allocate storage in advance, rather than successively concatenating onto a larger object.

R automatically manages memory, releasing memory back to the operating system when it's not needed via garbage collection. Very occasionally you may want to remove large objects as soon as they are not needed. *rm()* does not actually free up memory, it just disassociates the name from the memory used to store the object. In general R will clean up such objects without a

reference (i.e., a name) but you may need to call *gc()* to force the garbage collection. This uses some computation so it's generally not recommended.

In a language like C in which the user allocates and frees up memory, memory leaks are a major cause of bugs. Basically if you are looping and you allocate memory at each iteration and forget to free it, the memory use builds up inexorably and eventually the machine runs out of memory. In R, with automatic garbage collection, this is generally not an issue, but occasionally memory leaks do occur.

8.2 Monitoring overall memory use

8.2.1 Monitoring use within R

There are a number of ways to see how much memory is being used. When R is actively executing statements, you can use *top* from the UNIX shell. In R, *gc()* reports memory use and free memory as *Ncells* and *Vcells*. As far as I know, *Ncells* concerns the overhead of running R and *Vcells* relates to objects created by the user, so you'll want to focus on *Vcells*. You can see the number of Mb currently used (the "*used*" column of the output) and the maximum used in the session (the "*max used*" column)". A newer alternative is *mem_change()* in the *pryr* package.

```
gc()

##            used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   760730  40.7    1442291  77.1  1442291  77.1
## Vcells 21378900 163.2   31182729 238.0 22867199 174.5

x <- rnorm(1e8) # should use about 800 Mb
object.size(x)

## 800000040 bytes

object_size(x)  # from pryr

## 800 MB

gc()

##            used  (Mb) gc trigger   (Mb) max used  (Mb)
## Ncells 7.61e+05  40.7   1.44e+06   77.1 1.44e+06  77.1
## Vcells 1.21e+08 926.1   1.75e+08 1336.6 1.21e+08 926.4
```

```
mem_used() # from pryr

## 1.01 GB

rm(x)
gc() # note the "max used" column

##            used  (Mb) gc trigger   (Mb) max used  (Mb)
## Ncells   761411  40.7   1.44e+06   77.1 1.44e+06  77.1
## Vcells 21380007 163.2   1.40e+08 1069.3 1.21e+08 926.4

mem_change(x <- rnorm(1e8)) # from pryr

## 800 MB

mem_change(x <- rnorm(1e7))

## -720 MB
```

You can reset the value given for max used, with `gc(reset = TRUE)`.

In Windows only, *memory.size()* tells how much memory is being used.

You can check the amount of memory used by individual objects with *object.size()*.

Here is a useful function, *ls.sizes()*, that wraps *object.size()* to report the largest n objects in a given environment:

```
ls.sizes <- function(howMany = 10, minSize = 1){
        pf <- parent.frame()
        obj <- ls(pf) # or ls(sys.frame(-1))
        objSizes <- sapply(obj, function(x) {
                                object.size(get(x, pf))
                        })
        ## or sys.frame(-4) to get out of FUN, lapply(), sapply() and sizes
        objNames <- names(objSizes)
        howmany <- min(howMany, length(objSizes))
        ord <- order(objSizes, decreasing = TRUE)
        objSizes <- objSizes[ord][1:howMany]
        objSizes <- objSizes[objSizes > minSize]
        objSizes <- matrix(objSizes, ncol = 1)
```

```
        rownames(objSizes) <- objNames[ord][1:length(objSizes)]
        colnames(objSizes) <- "bytes"
        cat('object')
        print(format(objSizes, justify = "right", width = 11),
              quote = FALSE)
}
```

Unfortunately with environments, ReferenceClasses, and other such "containers", it can be hard to see how much memory the object is using, including all the components of the object. Here's a trick where we serialize the object, as if to export it, and then see how long the binary representation is.

```
## size of an environment
e <- new.env()
e$x <- rnorm(1e7)
object.size(e)

## 56 bytes

length(serialize(e, NULL))

## [1] 80000183

## size of a closure
x <- rnorm(1e7)
f <- function(input){
        data <- input
        g <- function(param) return(param * data)
        return(g)
}
myFun <- f(x)
rm(x)
object.size(myFun)

## 1560 bytes

length(serialize(myFun, NULL))

## [1] 160009701
```

One frustration with memory management is that if your code bumps up against the memory limits of the machine, it can be very slow to respond even when you're trying to cancel the statement with *Ctrl-C*. You can impose memory limits in Linux by starting R (from the UNIX prompt) in a fashion such as this

```
> R --max-vsize=1000M
```

Then if you try to create an object that will push you over that limit or execute code that involves going over the limit, it will simply fail with the message "*Error: vector memory exhausted (limit reached?)*". So this approach may be a nice way to avoid paging/swapping by setting the maximum in relation to the physical memory of the machine. It might also help in debugging memory leaks because the program would fail at the point that memory use was increasing. I haven't played around with this much, so offer this with a note of caution.

We can use an internal function called *inspect()* to see where in memory an object is stored. We'll see that this can be a handy tool for seeing where copies are made and where they are not.

```
x <- rnorm(5)
.Internal(inspect(x))

## @4bc6358 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 1.16401,-0.712768,-0.1928

obj <- list(a = rnorm(5), b = list(d = "adfs"))
.Internal(inspect(obj$a))

## @4bc5238 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 0.367744,0.103611,0.1543
```

The *pryr* package provides *address()* or *inspect()* as an alternative to *.Internal(inspect())*.

```
obj <- list(a = rnorm(5), b = list(d = "adfs"))
address(x)   # from pryr

## [1] "0x4bc6358"

address(obj$a)

## Error:  x must be the name of an object
```

Apparently there is a memory profiler in R, *Rprofmem*, but it needs to be enabled when R is compiled (i.e., installed on the machine), because it slows R down even when not used. So I've never gotten to the point of playing around with it.

8.2.2 Monitoring overall memory use

To understand how much memory is available on your computer, one needs to have a clear understanding of disk caching. The operating system will generally cache files/data in memory when it reads from disk. Then if that information is still in memory the next time it is needed, it will be much faster to access it the second time around than if it had to read the information from disk. While the cached information is using memory, that same memory is immediately available to other processes, so the memory is available even though it is in use.

We can see this via `free -h` (the "-h" is for 'human-readable', i.e. show in GB (G)) on Linux machine.

```
      total used free shared buff/cache available
Mem:   251G 998M 221G   2.6G        29G      247G
Swap:  7.6G 210M 7.4G
```

You'll generally be interested in the *Memory* row. (See below for some comments on *Swap*.) The *shared* column is complicated and probably won't be of use to you. The *buff/cache* column shows how much space is used for disk caching and related purposes but is actually available. Hence the *available* column is the sum of the *free* and *buff/cache* columns (more or less). In this case only about 1 GB is in use (indicated in the *used* column).

top (Linux or Mac) and *vmstat* (on Linux) both show overall memory use, but remember that the amount actually available to you is the amount free plus any buffer/cache usage. Here is some example output from *vmstat*:

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-
r b   swpd      free   buff    cache si so bi bo in cs us sy id wa st
1 0 215140 231655120 677944 30660296  0  0  1  2  0  0 18  0 82  0  0
```

It shows 232 GB free and 31 GB used for cache and therefore available, for a total of 263 GB available.

Here are some example lines from *top*:

```
KiB Mem : 26413715+total, 23180236+free, 999704 used, 31335072 buff/cache
KiB Swap:  7999484 total,  7784336 free, 215148 used. 25953483+avail Mem
```

We see that this machine has 264 GB RAM (the total column in the *Mem* row), with 259.5 GB available (232 GB free plus 31 GB buff/cache as seen in the *Mem* row). (I realize the numbers don't quite add up for reasons I don't fully understand, but we probably don't need to worry about that degree of exactness.) Only 1 GB is in use.

Swap is essentially the reverse of disk caching. It is disk space that is used for memory when the machine runs out of physical memory. You never want your machine to be using swap for memory because your jobs will slow to a crawl. As seen above, the *swap* line in both *free* and *top* shows 8 GB swap space, with very little in use, as desired.

8.3 The heap and the stack

The *heap* is the memory that is available for dynamically creating new objects while a program is executing, e.g., if you create a new object in R or call *new* in C++. When more memory is needed the program can request more from the operating system. When objects are removed in R, R will handle the garbage collection of releasing that memory.

The *stack* is the memory used for local variables when a function is called.

There's a nice discussion of this on this Stack Overflow thread.

8.4 Hidden uses of memory

- Replacement functions can hide the use of additional memory. How much memory is used here?

```
x <- rnorm(1e7)
gc()
dim(x) <- c(1e4, 1e3)
diag(x) <- 1
gc()
```

- Not all replacement functions actually involve creating a new object and replacing the original object. (However for some reason if I run the code via knitr in creating this PDF a copy IS made.) Here '[<-' is a primitive function, so the modification of the vector can be done without a copy.

```
x <- rnorm(1e7)
address(x)

## [1] "0x7f40dcab9010"

gc()
```

```
##            used  (Mb) gc trigger   (Mb) max used   (Mb)
## Ncells   761785  40.7   1.44e+06  77.1 1.44e+06   77.1
## Vcells 41383145 315.8   1.12e+08 855.5 1.31e+08 1002.4

x[5] <- 7
## when run plainly in R, should be the same address as before
address(x)

## [1] "0x7f40efe06010"

gc()

##            used  (Mb) gc trigger   (Mb) max used   (Mb)
## Ncells   761792  40.7   1.44e+06  77.1 1.44e+06   77.1
## Vcells 41383155 315.8   1.12e+08 855.5 1.31e+08 1002.4
```

- Indexing large subsets can involve a lot of memory use.

```
x <- rnorm(1e7)
gc()
y <- x[1:(length(x) - 1)]
gc()
```

Why was more memory used than just for x and y? Note that this is a limitation of R. Note that R could be designed to avoid this problem (see our discussion of *pqR* earlier in this Unit).

8.5 Passing objects to compiled code

As we've already discussed, when R objects are passed to compiled code (e.g., C or C++), they are passed as pointers and the compiled code uses the memory allocated by R (though it could also allocate additional memory if allocation is part of the code). However, a copy of the object is made, so when calling a C function from R there is some memory overhead.

Furthermore, we need to be aware of any casting that occurs, because the compiled code requires that the R object types match those that the function in the compiled code is expecting.

Here's an example of calling compiled code:

```
res <- .C("fastcount", PACKAGE="GCcorrect", tablex = as.integer(tablex),
tabley = as.integer(tabley), as.integer(xvar), as.integer(yvar),
as.integer(useline), as.integer(length(xvar)))
```

Let's consider when copies are made in casts:

```
f <- function(arg1){
        print(address(arg1))
        return(mean(arg1))
}
x <- rnorm(10)
class(x)
debug(f)
f(x)
f(as.numeric(x))
f(as.integer(x))
```

Next we'll see that C calls do involve a copy, even though it looks like we are just using the same object allocated by R. We'll use the *inline* package to work directly with C code in R and the .C functionality for interfacing with C.

```
library(inline)
src <- '
  for (int i = 0; i < *n; i++) {
    x[i] = exp(x[i]);
  }
'
sillyExp <- cfunction(signature(n = "integer", x = "numeric"),
        src, convention = ".C")
## sillyExp <- cfunction(signature(n = "integer", x = "numeric"),
##    src, convention = ".C")
len <- as.integer(100)  # or 100L
vals <- rnorm(len)
vals[1]

## [1] -0.0867

out1 <- sillyExp(n = len, x = vals)
address(vals)

## [1] "0x41535d0"
```

```
.Internal(inspect(out1))

## @6cfd888 19 VECSXP g0c2 [NAM(2),ATT] (len=2, tl=0)
##   @6002488 13 INTSXP g0c1 [] (len=1, tl=0) 100
##   @41134e0 14 REALSXP g0c7 [] (len=100, tl=0) 0.916936,1.77082,1.65617,2
## ATTRIB:
##   @53c06b8 02 LISTSXP g0c0 []
##     TAG: @f1e520 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x6000] "names" (has
##     @6cfd8c0 16 STRSXP g0c2 [] (len=2, tl=0)
##       @f844f8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "n"
##       @f84948 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "x"

.Internal(inspect(out1$x))

## @41134e0 14 REALSXP g0c7 [NAM(2)] (len=100, tl=0) 0.916936,1.77082,1.656
```

8.6 Delayed copying (copy-on-change)

Next we'll see that something like lazy evaluation occurs outside of functions as well with some functionality called *delayed copying* or *copy-on-change*. When we discussed R as being call-by-value in Section 6.4, copy-on-change was one of the reasons that copies of arguments are not always made.

Let's see what goes on within a function in terms of memory use in different situations. Ignore the *gc()* results in the pdf, as we'll start R fresh to get a clean view of memory use during the class demo.

```
f <- function(x){
        print(gc())
        z <- x[1]
        .Internal(inspect(x))
        return(x)
}
y <- rnorm(1e7)
gc()

##            used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   781779  41.8   1.44e+06  77.1 1.44e+06  77.1
```

```
## Vcells 51419471 392.3   1.12e+08 855.5 1.31e+08 1002.4

.Internal(inspect(y))

## @7f40dc8b7010 14 REALSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) 0.49776

out <- f(y)

##            used  (Mb) gc trigger  (Mb) max used   (Mb)
## Ncells   781847  41.8   1.44e+06  77.1 1.44e+06   77.1
## Vcells 51419490 392.3   1.12e+08 855.5 1.31e+08 1002.4
## @7f40dc8b7010 14 REALSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) 0.49776

.Internal(inspect(y))

## @7f40dc8b7010 14 REALSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) 0.49776

.Internal(inspect(out))

## @7f40dc8b7010 14 REALSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) 0.49776
```

Note that in this example, if you use *address()* instead of *.Internal(inspect())* it's not really clear what is going on.

In fact, this occurs outside function calls as well. Copies of objects are not made until one of the objects is actually modified. Initially, the copy points to the same memory location as the original object.

```
y <- rnorm(1e7)
gc()

##            used  (Mb) gc trigger  (Mb) max used   (Mb)
## Ncells   781895  41.8   1.44e+06  77.1 1.44e+06   77.1
## Vcells 61410626 468.6   1.12e+08 855.5 1.31e+08 1002.4

address(y)

## [1] "0x7f40d7c6b010"

x <- y
gc()
```

```
##            used  (Mb) gc trigger  (Mb) max used   (Mb)
## Ncells   781913  41.8   1.44e+06  77.1 1.44e+06   77.1
## Vcells 51410662 392.3   1.12e+08 855.5 1.31e+08 1002.4
```

```
object_size(x, y)  # from pryr
```

```
## 80 MB
```

```
address(x)
```

```
## [1] "0x7f40d7c6b010"
```

```
x[1] <- 5
gc()
```

```
##            used  (Mb) gc trigger  (Mb) max used   (Mb)
## Ncells   781946  41.8   1.44e+06  77.1 1.44e+06   77.1
## Vcells 61410711 468.6   1.12e+08 855.5 1.31e+08 1002.4
```

```
address(x)
```

```
## [1] "0x7f40efe06010"
```

```
object_size(x, y)
```

```
## 160 MB
```

```
rm(x)
x <- y
address(x)
```

```
## [1] "0x7f40d7c6b010"
```

```
address(y)
```

```
## [1] "0x7f40d7c6b010"
```

```
y[1] <- 5
address(x)
```

```
## [1] "0x7f40d7c6b010"
```

```
address(y)
```

```
## [1] "0x7f40d301f010"
```

Or we can see this using *mem_change()*.

```
library(pryr)
rm(x)
mem_change(x <- rnorm(1e7))

## 80 MB

address(x)

## [1] "0x7f40efe06010"

mem_change(x[3] <- 8)

## 448 B

address(x)

## [1] "0x7f40efe06010"

mem_change(y <- x)

## -80 MB

address(y)

## [1] "0x7f40efe06010"

mem_change(x[3] <- 8)

## 80 MB

address(x)

## [1] "0x7f40d7c6b010"

address(y)

## [1] "0x7f40efe06010"
```

Challenge: explain the results of the example above.

How does copy-on-change work and how can it be fooled? As discussed by Radford Neal, who is working to improve the efficiency of R in a project called pqR, *"So R doesn't copy all the time. Instead, it maintains a count, called NAMED, of how many "names" refer to an object, and copies only when an object that needs to be modified is also referred to by another name. Unfortunately, however, this scheme works rather poorly. Many unnecessary copies are still made, while many bugs have arisen in which copies aren't made when necessary.*" I believe some improvements have been made in recent versions of R in this regard.

Here are examples of how the NAMED count can be fooled into making a copy unnecessarily. Why do I say these copies are unnecessary and why is NAMED fooled?

```
rm(x, y)
f <- function(x) sum(x^2)
y <- rnorm(10)
## result of next line should be 1 if executed in clean R session
refs(y)  # from pryr - reports on the NAMED count

## [1] 2

f(y)

## [1] 12.8

refs(y)

## [1] 2

address(y)

## [1] "0x5bd2138"

y[3] <- 2
address(y)

## [1] "0x4f8bf70"

a <- 1:5
b <- a
address(a)

## [1] "0x55d6888"
```

```
address(b)

## [1] "0x55d6888"

a[2] <- 0
b[2] <- 4
address(a)

## [1] "0x4f89210"

address(b)

## [1] "0x4f88088"
```

We can use the *tracemem()* function to assess what is going on without all those *inspect()* calls. Anything surprising in what you see?

```
a <- 1:10
tracemem(a)

## [1] "<0x6ad6720>"

## b and a share memory
b <- a
b[1] <- 1

## tracemem[0x6ad6720 -> 0x6a7fcb0]: eval eval withVisible withCallingHandle
## tracemem[0x6a7fcb0 -> 0x5cb9e80]: eval eval withVisible withCallingHandle

## result when done through knitr is not as in plain R
untracemem(a)
address(a)

## [1] "0x6ad6720"

address(b)

## [1] "0x5cb9e80"
```

Challenge: How much memory is used in the following calculation?

```
x <- rnorm(1e7)
myfun <- function(y){
        z <- y
        return(mean(z))
}
myfun(x)

## [1] 0.000218
```

How about here? What is going on?

```
x <- rnorm(1e7)
x[1] <- NA
myfun <- function(y){
        return(mean(y, na.rm = TRUE))
}
myfun(x)

## [1] 0.000654
```

This makes sense if we look at *mean.default()*. Consider where additional memory is used.

8.7 Deep copies and lists and character strings

Prior to R 3.1.0, modifying an element of a list caused the entire list to be copied, basically what is called a *deep copy*. In more recent versions of R, only the components that need to get copied are copied.

You can explore this using *.Internal(inspect())* on a list.

R is also clever about saving copying when it works with character strings. We might explore this in a problem set problem.

8.8 Strategies for saving memory

A couple basic strategies for saving memory include:

- Avoiding unnecessary copies.
- Removing objects that are not being used and, if necessary (not generally needed), do a *gc()* call.

If you're really trying to optimize memory use, you may also consider:

- Using reference classes and similar strategies to pass by reference.
- Substituting integer and logical vectors for numeric vectors when possible.

8.9 Example

Let's work through a real example where we keep a running tally of current memory in use and maximum memory used in a function call. We'll want to consider hidden uses of memory, passing objects to compiled code, and lazy evaluation. This code is courtesy of Yuval Benjamini. For our purposes here, let's assume that *xvar* and *yvar* are very long vectors using a lot of memory.

```
fastcount <- function(xvar, yvar) {
        naline <- is.na(xvar)
        naline[is.na(yvar)] = TRUE
        xvar[naline] <- 0
        yvar[naline] <- 0
        useline <- !naline;
        # Table must be initialized for -1's
        tablex <- numeric(max(xvar)+1)
        tabley <- numeric(max(yvar)+1)
        stopifnot(length(xvar) == length(yvar))
        res <- .C("fastcount",PACKAGE="GCcorrect",
                tablex = as.integer(tablex), tabley = as.integer(tabley),
                as.integer(xvar), as.integer(yvar), as.integer(useline),
                as.integer(length(xvar)))
        xuse <- which(res$tablex>0)
        xnames <- xuse - 1
        resb <- rbind(res$tablex[xuse], res$tabley[xuse])
        colnames(resb) <- xnames
        return(resb)
}
```

9 Computing on the language

We won't cover much, if any, of this section and I won't expect you to know this material.

9.1 The R interpreter

Parsing When you run R, the R interpreter takes the code you type or the lines of code that are read in a batch session and parses each statement, translating the text into functional form. It substitutes objects for the symbols (names) that represent those objects and evaluates the statement, returning the resulting object. For complicated R code, this may be recursive.

Since everything in R is an object, the result of parsing is an object that we'll be able to investigate, and the result of evaluating the parsed statement is an object.

We'll see more on parsing in the next section.

***.Primitive()* and *.Internal()* (and *.External*())** Some functionality is implemented internally within the C implementation that lies at the heart of R. If you see *.Internal()* or *.Primitive()* or *.External()*, in the code of a function, you know it's implemented internally (and therefore generally very quickly). Unfortunately, it also means that you don't get to see R code that implements the functionality, though Chambers p. 465 describes how you can look into the C source code. Basically you need to download the source code for the relevant package off of CRAN.

```
plot.xy # plot.xy() is called by plot.default()

## function (xy, type, pch = par("pch"), lty = par("lty"), col = par("col"),
##     bg = NA, cex = 1, lwd = par("lwd"), ...)
## invisible(.External.graphics(C_plotXY, xy, type, pch, lty, col,
##     bg, cex, lwd, ...))
## <bytecode: 0x6148188>
## <environment: namespace:graphics>

print(`%*%`)

## function (x, y)  .Primitive("%*%")
```

9.2 Parsing code and understanding language objects

R code can be manipulated in text form and we can actually write R code that will create or manipulate R code. We can then evaluate that R code using *eval()*.

quote() will parse R code, but not evaluate it. This allows you to work with the code rather than the results of evaluating that code. The *print()* method for language objects is not very helpful! But we can see the parsed code by treating the result as a list.

```
obj <- quote(if (x > 1) "orange" else "apple")
as.list(obj)

## [[1]]
## `if`
##
## [[2]]
## x > 1
##
## [[3]]
## [1] "orange"
##
## [[4]]
## [1] "apple"

class(obj)

## [1] "if"

weirdObj <- quote(`if`(x > 1, 'orange', 'apple'))
identical(obj, weirdObj)

## [1] TRUE
```

Recall that to access symbols that involve special syntax (such as special characters), you use backquotes.

Officially, the name that you assign to an object (including functions) is a *symbol*.

```
x <- 3; typeof(quote(x))

## [1] "symbol"
```

We can create an *expression* object that contains R code as

```
myExpr <- expression(x <- 3)
eval(myExpr)
typeof(myExpr)

## [1] "expression"
```

The difference between *quote()* and *expression()* is basically that *quote()* works with a single statement (including multiple statements inside {...}), while *expression()* can deal with multiple statements, returning a list-like object of parsed statements. Both of them parse R code.

```
a <- quote(x <- 5)
b <- expression(x <- 5, y <- 3)
d <- quote({x <- 5; y <- 3})
class(a)

## [1] "<-"

class(b)

## [1] "expression"

b[[1]]

## x <- 5

class(b[[1]])

## [1] "<-"

identical(a, b[[1]])

## [1] TRUE

identical(d[[2]], b[[1]])

## [1] TRUE
```

The following table shows the *language* objects in R; note that there are three classes of language objects: *expressions*, *calls*, and *names*.

	Example syntax to create	Class	Type
object names	quote(x)	name	symbol (language)
expressions	expression(x <- 3)	expression	expression (language)
function calls	quote(f())	call	language
if statements	quote(if(x < 3) y=5)	if (call)	language
for statement	quote(for(i in 1:5) {})	for (call)	language
assignments	quote(x <- 3)	<- (call)	language
operators	quote(3 + 7)	call	language

Basically any standard function, operator, *if* statement, *for* statement, assignment, etc. are function calls and inherit from the *call* class.

Objects of type language are not officially lists, but they can be queried as such. You can convert between language objects and lists with *as.list()* and *as.call()*.

An official expression is one or more syntactically correct R statements. When we use *quote()*, we're working with a single statement, while *expression()* will create a list of separate statements (essentially separate call objects). I'm trying to use the term *statement* to refer colloquially to R code, rather than using the term *expression*, since that has formal definition in this context.

Let's take a look at some examples of language objects and parsing.

```
e0 <- quote(3)
e1 <- expression(x <- 3)
e1m <- expression({x <- 3; y <- 5})
e2 <- quote(x <- 3)
e3 <- quote(rnorm(3))
print(c(class(e0), typeof(e0)))

## [1] "numeric" "double"

print(c(class(e1), typeof(e1)))

## [1] "expression" "expression"

print(c(class(e1[[1]]), typeof(e1[[1]])))

## [1] "<-"       "language"

print(c(class(e1m), typeof(e1m)))

## [1] "expression" "expression"
```

```
print(c(class(e2), typeof(e2)))

## [1] "<-"       "language"

identical(e1[[1]], e2)

## [1] TRUE

print(c(class(e3), typeof(e3)))

## [1] "call"     "language"

e4 <- quote(-7)
print(c(class(e4), typeof(e4))) # huh? what does this imply?

## [1] "call"     "language"

as.list(e4)

## [[1]]
## `-`
##
## [[2]]
## [1] 7
```

We can evaluate language types using *eval()*:

```
rm(x)
eval(e1)
rm(x)
eval(e2)
e1mlist <- as.list(e1m)
e2list <- as.list(e2)
eval(as.call(e2list))
## here's how to do it if the language object is actually an expression (mu
eval(as.expression(e1mlist))
```

Now let's look in more detail at the components of R expressions. We'll be able to get a sense from this of how R evaluates code. We see that when R evaluates a parse tree, the first element

says what function to use and the remaining elements are the arguments. But in many cases one or more arguments will themselves be call objects, so there's recursion.

```
e1 <- expression(x <- 3)
## e1 is one-element list with the element an object of class '<-'
print(c(class(e1), typeof(e1)))

## [1] "expression" "expression"

e1[[1]]

## x <- 3

as.list(e1[[1]])

## [[1]]
## `<-`
##
## [[2]]
## x
##
## [[3]]
## [1] 3

lapply(e1[[1]], class)

## [[1]]
## [1] "name"
##
## [[2]]
## [1] "name"
##
## [[3]]
## [1] "numeric"

y <- rnorm(5)
e3 <- quote(mean(y))
print(c(class(e3), typeof(e3)))

## [1] "call"     "language"
```

```
e3[[1]]

## mean

print(c(class(e3[[1]]), typeof(e3[[1]])))

## [1] "name"   "symbol"

e3[[2]]

## y

print(c(class(e3[[2]]), typeof(e3[[2]])))

## [1] "name"   "symbol"

## we have recursion
e3 <- quote(mean(c(12,13,15) + rnorm(3)))
as.list(e3)

## [[1]]
## mean
##
## [[2]]
## c(12, 13, 15) + rnorm(3)

as.list(e3[[2]])

## [[1]]
## `+`
##
## [[2]]
## c(12, 13, 15)
##
## [[3]]
## rnorm(3)

as.list(e3[[2]][[3]])
```

```
## [[1]]
## rnorm
##
## [[2]]
## [1] 3
```

```
library(pryr)
call_tree(e3)
```

```
## \- ()
##   \- `mean
##   \- ()
##     \- `+
##     \- ()
##       \- `c
##       \-  12
##       \-  13
##       \-  15
##     \- ()
##       \- `rnorm
##       \-  3
```

9.3 Manipulating the parse tree

Of course since the parsed code is just an object, we can manipulate it, i.e., *compute on the language*:

```
out <- quote(y <- 3)
out[[3]] <- 4
eval(out)
y
```

```
## [1] 4
```

Here's another example:

```
e1 <- quote(4 + 5)
e2 <- quote(plot(x, y))
e2[[1]] <- `+`
eval(e2)

## [1] 7

e1[[3]] <- e2
e1

## 4 + .Primitive("+")(x, y)

class(e1[[3]]) # note the nesting

## [1] "call"

eval(e1) # what should I get?

## [1] 11
```

We can also turn it back into standard R code, as a character, using *deparse()*, which turns the parse tree back into R code as text. *parse()* is like *quote()* but it takes the code in the form of a string rather than an actual expression:

```
codeText <- deparse(out)
parsedCode <- parse(text = codeText)
## parse() works like quote() except on the code in the form of a string
eval(parsedCode)
deparse(quote(if (x > 1) "orange" else "apple"))

## [1] "if (x > 1) \"orange\" else \"apple\""
```

Note that the quotes have been escaped since they're inside a string.

It can be very useful to be able to convert names of objects that are in the form of text to names that R interprets as symbols referring to objects:

```
x3 <- 7
i <- 3
as.name(paste('x', i, sep=''))
```

```
## x3

eval(as.name(paste('x', i, sep='')))

## [1] 7

assign(paste('x', i, sep = ''), 11)
x3

## [1] 11
```

9.4 Parsing replacement expressions

Let's consider replacement expressions.

```
animals <- c('cat', 'dog', 'rat','mouse')
out1 <- quote(animals[4] <- 'rat')
out2 <- quote(`<-`(animals[4], 'rat'))
out3 <- quote('[<-'(animals,4,'rat'))
as.list(out1)

## [[1]]
## `<-`
##
## [[2]]
## animals[4]
##
## [[3]]
## [1] "rat"

as.list(out2)

## [[1]]
## `<-`
##
## [[2]]
## animals[4]
##
```

```
## [[3]]
## [1] "rat"

identical(out1, out2)

## [1] TRUE

as.list(out3)

## [[1]]
## `[<-`
##
## [[2]]
## animals
##
## [[3]]
## [1] 4
##
## [[4]]
## [1] "rat"

identical(out1, out3)

## [1] FALSE

typeof(out1[[2]]) # language

## [1] "language"

class(out1[[2]]) # call

## [1] "call"
```

The parse tree for *out3* is different than those for *out1* and *out2*, but when *out3* is evaluated the result is the same as for *out1* and *out2*:

```
eval(out1)
animals

## [1] "cat" "dog" "rat" "rat"
```

```
animals[4] <- 'mouse'  # reset things to original state
eval(out3)

## [1] "cat" "dog" "rat" "rat"

animals # both do the same thing

## [1] "cat" "dog" "rat" "rat"
```

Why? When R evaluates a call to '<-', if the first argument is a name, then it does the assignment, but if the first argument (i.e. what's on the left-hand side of the "assignment") is a call then it calls the appropriate replacement function. The second argument (the value being assigned) is evaluated first. Ultimately in all of these cases, the replacement function is used.

9.5 substitute()

The substitute function acts like *quote()*:

```
identical(quote(z <- x^2), substitute(z <- x^2))

## [1] TRUE
```

But if you also pass *substitute()* an environment, it will replace symbols with their object values in that environment.

```
e <- new.env(); e$x <- 3
substitute(z <- x^2, e)

## z <- 3^2
```

This can do non-sensical stuff:

```
e$z <- 5
substitute(z <- x^2, e)

## 5 <- 3^2
```

Let's see a practical example of substituting for variables in statements:

`plot(x = rnorm(5), y = rgamma(5, 1)) # how does plot get the axis label names?`

In the *plot()* function, you can see this syntax:

```
  xlabel <- if(!missing(x)) deparse(substitute(x))
```

So what's going on is that within *plot.default()*, it substitutes in for '*x*' with the statement that was passed in as the *x* argument, and then uses *deparse()* to convert to character. The fact that *x* still has *rnorm(5)* associated with it rather than the five numerical values from evaluating *rnorm()* has to do with lazy evaluation and promises. Here's the same idea in action in a stripped down example:

```
f <- function(obj){
objName <- deparse(substitute(obj))
print(objName)
}
f(y)

## [1] "y"
```

More generally, we can substitute into *expression* and *call* objects by providing a named list (or an environment) - the substition happens within the context of this list.

```
substitute(a + b, list(a = 1, b = quote(x)))

## 1 + x
```

Things can get intricate quickly:

```
e1 <- quote(x + y)
e2 <- substitute(e1, list(x = 3))
```

The problem is that *substitute()* doesn't evaluate its first argument, "*e1*", so it can't replace the parsed elements in *e1*. Instead, we'd need to do the following, where we force the evaluation of *e1*:

```
e2 <- substitute(substitute(e, list(x = 3)), list(e = e1))
substitute(substitute(e, list(x = 3)), list(e = e1))

## substitute(x + y, list(x = 3))

## so e1 is substituted as an evaluated object,
## which then allows for substitution for 'x'
e2

## substitute(x + y, list(x = 3))
```

```r
eval(e2)

## 3 + y

substitute_q(e1, list(x = 3))   # from pryr

## 3 + y
```

If this subsection is confusing, let me assure you that it has confused me too. The indirection going on here is very involved.

9.6 Final thoughts

Challenge: figure out how a *for* loop is parsed in R. See how a *for* loop with one statement within the loop differs from one with two or more statements.

We'll see *expression()* again when we talk about inserting mathematical notation in plots.