Using R

September 15, 2013

This will cover some of the more advanced features of R that were not covered, or were treated lightly in the R bootcamp. It's a bit of a mish-mash because I'm assuming you have a working familiarity with the first 8 or so units from the bootcamp.

References:

- Adler
- Chambers
- R intro manual on CRAN (R-intro).
- 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.

One of my goals in our coverage of R is for us to think about why things are the way they are in R. I.e., what principles were used in creating the language.

1 Interacting with the operating system from R

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] "best-practices.png" "branchcommit.png"   "cache"
## [4] "commit_anatomy.png" "example.bashrc"
```

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

```
file.exists("file.txt")

list.files("~/research")
```

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

```
Sys.info()

##                                              sysname
##                                              "Linux"
##                                              release
##                                  "3.2.0-43-generic"
##                                              version
## "#68-Ubuntu SMP Wed May 15 03:33:33 UTC 2013"
##                                             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] 75
```

```
##
## $digits
## [1] 4


# options(width = 120) # often nice to have more characters on screen

options(width = 55)  # for purpose of making the pdf of this document

options(max.print = 5000)

options(digits = 3)

a <- 0.123456
b <- 0.1234561

a

## [1] 0.123

b

## [1] 0.123

a == b

## [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 3.0.1 (2013-05-16)
## Platform: x86_64-pc-linux-gnu (64-bit)
##
## 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=C
##  [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] knitr_1.2 SCF_1.1-1
##
## loaded via a namespace (and not attached):
## [1] digest_0.6.3   evaluate_0.4.3 formatR_0.7
## [4] stringr_0.6.2  tools_3.0.1
```

- 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 packages (see below), sourcing files with user-defined functions (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 command (like running a bash shell script) as follows.

  1. Write your R code in a text file, say *file.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).
  3. Make the R code file executable with *chmod*.
  4. Run the script from the command line: `./file.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])
```

2 Packages

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.

Loading packages You can use *library()* to either (1) make a package available (loading it) or (2) to see all the installed packages.

```
library(fields)

## Loading required package:  methods
## Loading required package:  spam
## Spam version 0.29-3 (2013-04-23) 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 object is masked from 'package:base':
##
##    backsolve, forwardsolve
## Loading required package:  maps

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

Notice that some of the packages are in a system directory and some are in my 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.

```
.libPaths()

## [1] "/accounts/gen/vis/paciorek/R/x86_64-pc-linux-gnu-library/3.0"
## [2] "/server/linux/lib/R/3.0/x86_64/site-library"
## [3] "/usr/lib/R/site-library"
## [4] "/usr/lib/R/library"

searchpaths()

##  [1] ".GlobalEnv"
##  [2] "/accounts/gen/vis/paciorek/R/x86_64-pc-linux-gnu-library/3.0/field
##  [3] "/accounts/gen/vis/paciorek/R/x86_64-pc-linux-gnu-library/3.0/maps"
##  [4] "/accounts/gen/vis/paciorek/R/x86_64-pc-linux-gnu-library/3.0/spam"
##  [5] "/usr/lib/R/library/methods"
##  [6] "/server/linux/lib/R/3.0/x86_64/site-library/knitr"
##  [7] "/usr/lib/R/library/stats"
##  [8] "/usr/lib/R/library/graphics"
##  [9] "/usr/lib/R/library/grDevices"
## [10] "/usr/lib/R/library/utils"
## [11] "/usr/lib/R/library/datasets"
## [12] "/server/linux/lib/R/3.0/x86_64/site-library/SCF"
## [13] "Autoloads"
## [14] "/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).

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

You can also download the zipped source file from CRAN and install from the file; see the help page for *install.packages()*.

Accessing objects from packages 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()*. We'll talk more about this when we talk about scope and environments. If we want to see the objects in one of the other workspaces, we can do the following:

```
search()

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

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

## [1] "abline"    "arrows"    "assocplot" "axis"
## [5] "Axis"

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

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

3 Objects in R

3.1 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 '<-'.

```
out = mean(rnorm(7))  # OK
system.time(out = rnorm(10000))

## Error:  unused argument (out = rnorm(10000))

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

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

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

```
mean

## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x32d6c10>
## <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:  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.:

```
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 here?
apply(mat, 1, sum.isna = function(vec) {
    return(sum(is.na(vec)))
})  # NOPE

## Error:  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
```

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:  non-numeric argument to binary operator

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

## [1] -1.334  1.009  0.541 -1.549  0.295

df2 <- data.frame(cbind(rep("a", n), rnorm(n), rnorm(5)))
apply(df2[, 2:3], 1, function(x) x[1] + x[2])

## Error:  non-numeric argument to binary operator
```

3.2 Type vs. class

You should be familiar with vectors as the basic data structure in R, with character, integer, numeric, etc. classes. Objects also 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"

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 in the R programming unit.

We can create objects with our own defined class.

```
me <- list(firstname = "Chris", surname = "Paciorek")
class(me) <- "personClass"  # it turns out R already has a 'person' class
class(me)

## [1] "personClass"

is.list(me)

## [1] TRUE

typeof(me)

## [1] "list"

typeof(me$firstname)

## [1] "character"
```

3.3 Information about objects

Some functions that give information about objects are:

```
is(me, "personClass")

## [1] TRUE

str(me)

## List of 2
##  $ firstname: chr "Chris"
##  $ surname  : chr "Paciorek"
##  - attr(*, "class")= chr "personClass"

attributes(me)

## $names
## [1] "firstname" "surname"
##
## $class
## [1] "personClass"

mat <- matrix(1:4, 2)
class(mat)

## [1] "matrix"

typeof(mat)

## [1] "integer"

length(mat)

## [1] 4

# recall that a matrix can be thought of as a vector
# with dimensions
attributes(mat)

## $dim
## [1] 2 2

dim(mat)

## [1] 2 2
```

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(0.025, 0.975))
qs

##  2.5% 97.5%
## -1.97  1.97

qs[1] + 3

## 2.5%
## 1.03
```

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] -1.97  1.97
```

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"

mat <- data.frame(x = 1:2, y = 3:4)
row.names(mat) <- c("first", "second")
mat
```

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

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

## first
##     7
```

3.4 The workspace

Objects exist in a workspace, which in R is called an environment.

```
# objects() # what objects are in my workspace
identical(ls(), objects())  # synonymous

## [1] TRUE

dat <- 7
dat2 <- 9
subdat <- 3
obj <- 5
obj2 <- 7
objects(pattern = "^dat")

## [1] "dat"  "dat2"

rm(dat2, subdat)
rm(list = c("obj", "obj2"))
# a bit confusing - the 'list' argument should be a
# character vector
rm(list = ls(pattern = "^dat"))
exists("dat")  # can be helpful when programming

## [1] FALSE

dat <- rnorm(5e+05)
object.size(dat)
```

```
## 4000040 bytes

print(object.size(dat), units = "Mb")  # this seems pretty clunky!

## 3.8 Mb

# but we'll understand why it's clunky when we see S3
# classes in detail
rm(list = ls())  # what does this do?
```

3.5 Some other details

Special objects There are also some special objects, which often begin with a period, like hidden files in UNIX. One is *.Last.value*, which stores the last result.

```
rnorm(10)

##  [1]  0.6184  0.5876  0.4175 -0.1859  0.0891  0.1729
##  [7] -0.8441  0.7767  0.8534 -1.3917

# .Last.value # this should return the 10 random
# normals but knitr is messing things up, commented
# out here
```

Scientific notation R uses the syntax *"xep"* to mean $x * 10^p$.

```
x <- 1e+05
log10(x)
y <- 1e+05
x <- 1e-08
```

Information about functions To get help on functions (I'm having trouble evaluating these with knitr, so just putting these in as text here):

```
?lm # or help(lm)
help.search('lm')
```

```
apropos('lm')
help('[[') # operators are functions too
args(lm)
```

Strings and quotation Working with strings and quotes (see `?Quotes` in R). Generally one uses double quotes to denote text. If we want a quotation symbol in text, we can do something like the following, either combining single and double quotes or escaping the quotation:

```
ch1 <- "Chris's\n"
ch2 <- 'He said, "hello."\n'
ch3 <- "He said, \"hello.\"\n"
```

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.

4 Working with data structures

4.1 Lists and dataframes

Extraction You extract from lists with "*[[*" or with "*[*"

```
x <- list(a = 1:2, b = 3:4, sam = rnorm(4))
x[[2]]  # extracts the indicated component, which can be anything,

## [1] 3 4

# in this case just an integer vector
x[c(1, 3)]  # extracts subvectors, which since it is a list,

## $a
## [1] 1 2
##
## $sam
## [1] -0.801 -0.729  1.050  1.897

# will also be a list
```

When working with lists, it's handy to be able to use the same function on each element of the list:

```
lapply(x, length)

## $a
## [1] 2
##
## $b
## [1] 2
##
## $sam
## [1] 4

sapply(x, length)  # returns things in a user-friendly way

##   a   b sam
##   2   2   4
```

Note that to operate on a data frame, which is a list, we'll generally want to use *lapply()* or *sapply()*, as *apply()* is really designed for working with elements that are all of the same type:

```
apply(CO2, 2, class)  # hmmm

##       Plant        Type   Treatment        conc
## "character" "character" "character" "character"
##      uptake
## "character"

sapply(CO2, class)

## $Plant
## [1] "ordered" "factor"
##
## $Type
## [1] "factor"
##
## $Treatment
## [1] "factor"
```

```
##
## $conc
## [1] "numeric"
##
## $uptake
## [1] "numeric"
```

Here's a nice trick to pull out a specific component from each element of a list. (Note the use of the additional argument(s) to *sapply()* - this can also be done in the other *apply()* variants.)

```
params <- list(a = list(mn = 7, sd = 3), b = list(mn = 6,
    sd = 1), c = list(mn = 2, sd = 1))
sapply(params, "[[", 1)

## a b c
## 7 6 2
```

Finally, we can flatten a list with *unlist()*.

```
unlist(x)

##     a1     a2     b1     b2   sam1   sam2   sam3
##  1.000  2.000  3.000  4.000 -0.801 -0.729  1.050
##   sam4
##  1.897
```

Calculations in the context of stratification Note that some of the basic R functionality for doing stratified analysis is mentioned here. For a new way to do such split-apply-combine operations see the *plyr* package.

We can also use an *apply()* variant to do calculations on subgroups, defined based on a factor or factors.

```
tapply(mtcars$mpg, mtcars$cyl, mean)

##    4    6    8
## 26.7 19.7 15.1

tapply(mtcars$mpg, list(mtcars$cyl, mtcars$gear), mean)
```

```
##      3    4    5
## 4 21.5 26.9 28.2
## 6 19.8 19.8 19.7
## 8 15.1   NA 15.4
```

Check out *aggregate()* and *by()* for nice wrappers to *tapply()* when working with data frames. *aggregate()* returns a data frame and works when the output of the function is univariate, while *by()* returns a list, so can return multivariate output:

```
aggregate(mtcars, by = list(cyl = mtcars$cyl), mean)

##   cyl  mpg cyl disp    hp drat   wt qsec    vs    am
## 1   4 26.7   4  105  82.6 4.07 2.29 19.1 0.909 0.727
## 2   6 19.7   6  183 122.3 3.59 3.12 18.0 0.571 0.429
## 3   8 15.1   8  353 209.2 3.23 4.00 16.8 0.000 0.143
##   gear carb
## 1 4.09 1.55
## 2 3.86 3.43
## 3 3.29 3.50

# this uses the function on each column of mtcars,
# split by the 'by' argument
by(warpbreaks, warpbreaks$tension, function(x) {
    lm(breaks ~ wool, data = x)
})

## warpbreaks$tension: L
##
## Call:
## lm(formula = breaks ~ wool, data = x)
##
## Coefficients:
## (Intercept)        woolB
##        44.6        -16.3
##
## ------------------------------------------
## warpbreaks$tension: M
```

```
##
## Call:
## lm(formula = breaks ~ wool, data = x)
##
## Coefficients:
## (Intercept)        woolB
##       24.00         4.78
##
## -----------------------------------------
## warpbreaks$tension: H
##
## Call:
## lm(formula = breaks ~ wool, data = x)
##
## Coefficients:
## (Intercept)        woolB
##       24.56        -5.78
```

In some cases you may actually need an object containing the subsets of the data, for which you can use *split()*:

```
split(mtcars, mtcars$cyl)
```

To stratify based on a continuous variable, you can create a factor with the *cut()* function. By default the levels are not formally ordered, but we can manipulate them with *relevel()*, or make sure they're ordered by using the *ordered_result* argument to *cut()*.

```
x <- rnorm(100)
f <- cut(x, breaks = c(-Inf, -1, 1, Inf), labels = c("low",
    "medium", "high"))
levels(f)  # note that f is not explicitly ordered

## [1] "low"    "medium" "high"

f <- relevel(f, "high")  # puts high as first level
f <- cut(x, breaks = c(-Inf, -1, 1, Inf), labels = c("low",
    "medium", "high"), ordered_result = TRUE)
```

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).

4.2 Vectors and matrices

Column-major vs. row-major matrix storage Matrices in R are column-major ordered, which means they are stored by column as a vector of concatenated columns.

```
mat <- matrix(rnorm(500), nr = 50)
identical(mat[1:50], mat[, 1])

## [1] TRUE

identical(mat[1:10], mat[1, ])

## [1] FALSE

vec <- c(mat)
mat2 <- matrix(vec, nr = 50)
identical(mat, mat2)

## [1] TRUE
```

If you want to fill a matrix row-wise:

```
matrix(1:4, 2, byrow = TRUE)

##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

Column-major ordering is also used in Matlab and Fortran, while row-major ordering is used in C.

Identifying elements by index You can figure out the indices of elements having a given characteristic using *which()*:

```
x <- c(1, 10, 2, 9, 3, 8)
which(x < 3)

## [1] 1 3

x <- matrix(1:6, nrow = 2)
which(x < 3, arr.ind = TRUE)

##      row col
## [1,]   1   1
## [2,]   2   1
```

which.max() and *which.min()* have similar sort of functionality.

We can determine which elements match those in another set with *%in%* (to return logicals) or *match()* (to return indices that describe the mapping):

```
set <- c("Mazda RX4", "Merc 240D", "Fiat 128")
row.names(mtcars) %in% set

##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
##  [9] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [17] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

which(row.names(mtcars) %in% set)

## [1]  1  8 18

match(row.names(mtcars), set)

##  [1]  1 NA NA NA NA NA NA  2 NA NA NA NA NA NA NA NA NA
## [18]  3 NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```

Vectorized subsetting We can subset vectors, matrices, and rows of data frames by index or by logical vectors.

```
set.seed(0)
vec <- rnorm(8)
mat <- matrix(rnorm(9), 3)
vec

## [1]  1.263 -0.326  1.330  1.272  0.415 -1.540 -0.929
## [8] -0.295

mat

##          [,1]   [,2]   [,3]
## [1,] -0.00577 -0.799 -0.299
## [2,]  2.40465 -1.148 -0.412
## [3,]  0.76359 -0.289  0.252
```

```
vec[vec < 0]

## [1] -0.326 -1.540 -0.929 -0.295

vec[vec < 0] <- 0
vec

## [1] 1.263 0.000 1.330 1.272 0.415 0.000 0.000 0.000

mat[mat[, 1] < 0, ]  # similarly for data frames

## [1] -0.00577 -0.79901 -0.29922

mat[mat[, 1] < 0, 2:3]  # similarly for data frames

## [1] -0.799 -0.299

mat[, mat[1, ] < 0]

##          [,1]   [,2]   [,3]
## [1,] -0.00577 -0.799 -0.299
## [2,]  2.40465 -1.148 -0.412
## [3,]  0.76359 -0.289  0.252

mat[mat[, 1] < 0, 2:3] <- 0
set.seed(0)  # so we get the same vec as we had before
vec <- rnorm(8)
wh <- which(vec < 0)
logicals <- vec < 0
logicals

## [1] FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE

wh

## [1] 2 6 7 8

identical(vec[wh], vec[logicals])

## [1] TRUE
```

```
vec <- c(1L, 2L, 1L)
is.integer(vec)

## [1] TRUE

vec[vec == 1L]  # in general, not safe with numeric vectors

## [1] 1 1

vec[vec != 3L]  # nor this

## [1] 1 2 1
```

Finally, we can also subset a matrix with a two-column matrix of {row,column} indices.

```
mat <- matrix(rnorm(25), 5)
rowInd <- c(1, 3, 5)
colInd <- c(1, 1, 4)
mat[cbind(rowInd, colInd)]

## [1] -0.00577  0.76359 -0.69095
```

Indexing and factors Be careful of using factors as indices for subsetting:

```
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?

Apply() The *apply()* function will apply a given function to either the rows or columns of a matrix or a set of dimensions of an array:

```
x <- matrix(1:6, nr = 2)
x

##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6

apply(x, 1, min)  # by row

## [1] 1 2

apply(x, 2, min)  # by column

## [1] 1 3 5

x <- array(1:24, c(2, 3, 4))
apply(x, 2, min)  # for(j in 1:3) print(min(x[ , j, ]))

## [1] 1 3 5

apply(x, c(2, 3), min)

##      [,1] [,2] [,3] [,4]
## [1,]    1    7   13   19
## [2,]    3    9   15   21
## [3,]    5   11   17   23

# equivalent to: for(j in 1:3) { for(k in 1:4) {
# print(min(x[ , j, k])) }}
```

This can get confusing, but can be very powerful. Basically, I'm calculating the min for each element of the second dimension in the second-to-last example and for each pair of elements in the first and third dimensions in the final example. Caution: if the result of each subcalculation is a vector, these will be *cbind()*'ed together (recall column-major order), so if you used *apply()* on the row margin, you'll need to transpose the result.

Why use apply(), lapply(), etc.? The various *apply()* functions (apply, lapply, sapply, tapply, etc.) may be faster than a loop (when a substantial part of the work lies in the overhead of the looping), but if the dominant part of the calculation lies in the time required by the function on each of the elements, then the main reason for using an *apply()* variant is code clarity.

Here's an example where *apply()* is not faster than a loop.

```
n <- 5e+05
nr <- 10000
nCalcs <- n/nr
mat <- matrix(rnorm(n), nrow = nr)
times <- 1:nr
system.time(out1 <- apply(mat, 2, function(vec) {
    mod = lm(vec ~ times)
    return(mod$coef[2])
}))

##    user  system elapsed
##   1.324   0.004   0.587

system.time({
    out2 = rep(NA, nCalcs)
    for (i in 1:nCalcs) {
        out2[i] = lm(mat[, i] ~ times)$coef[2]
    }
})

##    user  system elapsed
##   1.252   0.008   0.456
```

4.3 Long and wide formats

Finally, we may want to convert between so-called 'long' and 'wide' formats, which are motivated by working with longitudinal data (multiple observations per subject). 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. *stack()* converts from wide to long while *unstack()* does the reverse. *reshape()* is similar but more flexible and it can go in either direction. 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. Let's use the precipitation data as an example.

```
load("../data/prec.RData")
prec <- prec[1:1000, ]  # just to make the example code run faster
precVars <- 5:ncol(prec)
precStacked <- stack(prec, select = precVars)
out <- unstack(precStacked)
# to use reshape, we need a unique id for each row
# since reshape considers each row in the wide format
# as a subject
prec <- cbind(unique = 1:nrow(prec), prec)
precVars <- precVars + 1
precLong <- reshape(prec, varying = names(prec)[precVars],
    idvar = "unique", direction = "long", sep = "")
precLong <- precLong[!is.na(precLong$prec), ]
precWide <- reshape(precLong, v.names = "prec", idvar = "unique",
    direction = "wide", sep = "")
```

Check out *melt()* and *cast()* in the *reshape2* package for easier argument formats than *reshape()*.

4.4 Linear algebra

We'll focus on matrices here. A few helpful functions are *nrow()* and *ncol()*, which tell the dimensions of the matrix. The *row()* and *col()* functions will return matrices of the same size as the original, but filled with the row or column number of each element. So to get the upper triangle of a matrix, *X*, we can do:

```
X <- matrix(rnorm(9), 3)
X

##        [,1]   [,2]   [,3]
## [1,] -1.525 -0.514 -0.225
## [2,]  1.600 -0.953  1.804
## [3,] -0.685 -0.338  0.354

X[col(X) >= row(X)]

## [1] -1.525 -0.514 -0.953 -0.225  1.804  0.354
```

See also the *upper.tri()* and *lower.tri()* functions, as well as the *diag()* function. *diag()* is quite handy - you can extract the diagonals, assign into the diagonals, or create a diagonal matrix:

```
diag(X)

## [1] -1.525 -0.953  0.354

diag(X) <- 1
X

##        [,1]   [,2]   [,3]
## [1,]  1.000 -0.514 -0.225
## [2,]  1.600  1.000  1.804
## [3,] -0.685 -0.338  1.000

d <- diag(c(rep(1, 2), rep(2, 2)))
d

##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    2    0
## [4,]    0    0    0    2
```

To transpose a matrix, use *t()*, e.g., `t(X)`.

Basic operations The basic matrix-vector operations are:

```
X %*% Y  # matrix multiplication
X * Y  # direct product
x %o% y  # outer product of vectors x, y: x times t(y)
outer(x, y)  # same thing
# evaluation of f(x,y) for all pairs of x,y values:
outer(x, y, function(x, y) cos(y)/(1 + x^2))
crossprod(X, Y)  # same as but faster than t(X) %*% Y!
```

For inverses (X^{-1}) and solutions of systems of linear equations ($X^{-1}y$):

```
solve(X)  # inverse of X
solve(X, y)  # (inverse of X) %*% y
```

Note that if all you need to do is solve the linear system, you should never explicitly find the inverse, UNLESS you need the actual matrix, e.g., to get a covariance matrix of parameters.

Otherwise, to find many solutions, all with the same matrix, X, you can use *solve()* with the second argument being a matrix with each column a different 'y' vector for which you want the solution.

solve() is an example of using a matrix decomposition to solve a system of equations (in particular the LU decomposition). We'll defer matrix decompositions (LU, Cholesky, eigendecomposition, SVD, QR) until the numerical linear algebra unit.

5 Functions, variable scoping, and frames

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 offically as *closures* (this is their *type*) - this terminology sometimes comes up in error messages.

5.1 Inputs

Arguments can be specifed in the correct order, or given out of order by specifying *name* = *value*. 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 is 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
```

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*:

```r
mat <- matrix(1:9, 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"
```

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

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

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

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

Pass by value vs. pass by reference Note that R makes a copy of all objects that are arguments to a function, with the copy residing in the frame (the environment) of the function (we'll see more about frames just below). This is a case of *pass by value*. In other languages it is also possible to *pass by reference*, in which case, changes to the object made within the function affect the value of the argument in the calling environment. 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 in the R programming unit.

An important exception is *par()*. If you change graphics parameters by calling *par()* in a user-defined function, they are changed permanently. 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.

5.2 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.

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

## [1] 9
```

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

```r
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.

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

## [1] "lm"

is.list(mod)

## [1] TRUE
```

5.3 Variable scope

To consider variable scope, we need to define the terms *environment* and *frame*.

Environments and frames are closely related. An environment is a collection of named objects (a frame), with a pointer to the 'enclosing environment', i.e., the next environment to look for something in, also called the parent. (Be careful as this is different than the parent frame of a function.)

Variables in the enclosing environment (the environment in which a function is defined, also called the parent environment) are available within a function. This is the analog of *global variables* in other languages. Note that enclosing/parent environment is NOT the environment from which the function was called. This is called *lexical scoping*.

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 it some cases it can be useful.

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

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()*?

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 assigned to *x* in the frame of the function, while `y = x*3` is evaluated in the frame of the function.

5.4 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 start it (it actually continues further up; see below). In general, as we've seen, these are not the frames on the stack.

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 this is next place that is searched if an object can't be found in the frame of the function call. This is called lexical scoping (and differs from the S language on which R was based). 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 (also called the namespace) of the stats package (since this is the environment where *lm()* is defined), 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()*. Data frames or list that you attach using *attach()* generally are placed just after the global environment.

```
search()

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

searchpaths()

##  [1] ".GlobalEnv"
##  [2] "/accounts/gen/vis/paciorek/R/x86_64-pc-linux-gnu-library/3.0/fields
##  [3] "/accounts/gen/vis/paciorek/R/x86_64-pc-linux-gnu-library/3.0/maps"
##  [4] "/accounts/gen/vis/paciorek/R/x86_64-pc-linux-gnu-library/3.0/spam"
##  [5] "/usr/lib/R/library/methods"
##  [6] "/server/linux/lib/R/3.0/x86_64/site-library/knitr"
##  [7] "/usr/lib/R/library/stats"
##  [8] "/usr/lib/R/library/graphics"
##  [9] "/usr/lib/R/library/grDevices"
## [10] "/usr/lib/R/library/utils"
## [11] "/usr/lib/R/library/datasets"
## [12] "/server/linux/lib/R/3.0/x86_64/site-library/SCF"
## [13] "Autoloads"
## [14] "/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 <- .GlobalEnv
parent.env(x)

## <environment: package:fields>
## attr(,"name")
## [1] "package:fields"
## attr(,"path")
## [1] "/accounts/gen/vis/paciorek/R/x86_64-pc-linux-gnu-library/3.0/fields
```

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

## [1] "package:fields"
## [1] "package:maps"
## [1] "package:spam"
## [1] "package:methods"
## [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"
## [1] "R_EmptyEnv"
```

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.

We can look at the objects of an environment as follows:

```
ls(pos = 7)[1:5]  # what does this do?

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

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

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

environment(lm)

## <environment: namespace:stats>
```

The enclosing environments for a function from an attached package are a bit more complicated:

```
x <- environment(lm)
parent.env(x)

## <environment: 0x16615a8>
## attr(,"name")
## [1] "imports:stats"

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

## [1] "imports:stats"
## [1] "base"
## [1] "R_GlobalEnv"
## [1] "package:fields"
## [1] "package:maps"
## [1] "package:spam"
## [1] "package:methods"
## [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"
## [1] "R_EmptyEnv"
```

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

```
lm <- function() {
    return(NULL)
}  # this seems dangerous but isn't
```

```
x <- 1:3
y <- rnorm(3)
mod <- lm(y ~ x)

## Error:  unused argument (y ~ x)

mod <- get("lm", pos = "package:stats")(y ~ x)
mod <- stats::lm(y ~ x)  # an alternative
```

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.

5.5 Frames and the call stack

R keeps track of the call stack, which is the set 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 parent environment. Careful: here, *parent* now refers to the parent in terms of the call stack, not the enclosing environment. *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.

```
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)
```

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.

5.6 with() and within()

with() provides a clean way to use a function (or any R code, specified as R statements enclosed within {}, unless you are evaluating a single expression as in the demo here) within the context of a data frame (or an environment). *within()* is similar, evaluating within the context of a data frame or a list, but it allows you to modify the data frame (or list) and returns the result.

```
with(mtcars, cyl * mpg)

##  [1] 126.0 126.0  91.2 128.4 149.6 108.6 114.4  97.6
##  [9]  91.2 115.2 106.8 131.2 138.4 121.6  83.2  83.2
## [17] 117.6 129.6 121.6 135.6  86.0 124.0 121.6 106.4
## [25] 153.6 109.2 104.0 121.6 126.4 118.2 120.0  85.6

new.mtcars <- within(mtcars, crazy <- cyl * mpg)
names(new.mtcars)

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

6 Flow control and logical operations

6.1 Logical operators

Everyone should be familiar with the comparison operators, <, <=, >, >=, ==, !=. Logical operators are slightly trickier:

Logical operators for vectors and subsetting *&* and | are the "AND" and "OR" operators used when subsetting - they act in a vectorized way:

```
x <- rnorm(10)
x[x > 1 | x < -1]

## [1] -1.35 -2.07 -1.34 -1.45

x <- 1:10
y <- c(rep(10, 9), NA)
x > 5 | y > 5  # note that TRUE | NA evaluates to TRUE

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

Logical operators for *if* statements *&&* and || use only the first element of a vector and also proceed from left to right, returning the result as soon as possible and then ignoring the remaining comparisons (this can be handy because in some cases the second condition may give an error if the first condition is not passed). They are used in flow control (i.e., with *if* statements). Let's consider how the single and double operators differ:

```
a <- 7
b <- NULL
a < 8 | b > 3

## logical(0)

a < 8 || b > 3

## [1] TRUE

a <- c(0, 3)
b <- c(4, 2)
if (a < 7 & b < 7) print("this is buggy code")

## Warning:  the condition has length > 1 and only the first element
will be used

## [1] "this is buggy code"

if (a < 7 && b < 7) print("this is buggy code too, but runs w/o warnings")

## [1] "this is buggy code too, but runs w/o warnings"

if (a[1] < 7 && b[1] < 7) print("this code is correct and the condition is 

## [1] "this code is correct and the condition is TRUE"
```

You can use ! to indicate negation:

```
a <- 7
b <- 5
!(a < 8 & b < 6)

## [1] FALSE
```

6.2 If statements

If statements are at the core of programming. In R, the syntax is **`if(condition) statement else other_statement`**, e.g.,

```
x <- 5
if (x > 7) {
    x <- x + 3
} else {
    x <- x - 3
}
```

When one of the statements is a single statement, you don't need the curly braces around that statement.

```
if (x > 7) x <- x + 3 else x <- x - 3
```

An extension of *if* looks like:

```
x <- -3
if (x > 7) {
    x <- x + 3
    print(x)
} else if (x > 4) {
    x <- x + 1
    print(x)
} else if (x > 0) {
    x <- x - 3
    print(x)
} else {
    x <- x - 7
    print(x)
}

## [1] -10
```

Finally, be careful that *else* should not start its own line, unless it is preceded by a closing brace on the same line. Why?

```
if(x > 7) {
statement1 } # what happens at this point?
else{ # what happens now?
statement2
}
```

There's also the *ifelse()* function, which operates in a vectorized fashion:

```
x <- rnorm(6)
truncx <- ifelse(x > 0, x, 0)
truncx

## [1] 0.769 1.698 0.000 0.000 0.590 0.000
```

Common bugs in the condition of an if statement include the following:

1. Only the first element of *condition* is evaluated. You should be careful that *condition* is a single logical value and does not evaluate to a vector as this would generally be a bug. [see p. 152 of Chambers]

2. Use *identical()* or *all.equal()* rather than "==" to ensure that you deal properly with vectors and always get a single logical value back. We'll talk more about issues that can arise when comparing decimal numbers on a computer later in the course.

3. If *condition* includes some R code, it can fail and produce something that is neither TRUE nor FALSE. Defensive programming practice is to check the condition for validity.

```
vals <- c(1, 2, NA); eps <- 1e-9

# now pretend vals comes from some other chunk of code that we don't co

if(min(vals) > eps) # not good practice

   { print(vals) }

## Error:  missing value where TRUE/FALSE needed
```

```
# better practice:

minval <- min(vals)

if(!is.na(minval) && minval > eps) { print(vals) }
```

6.3 switch()

switch() is handy for choosing amongst multiple outcomes depending on an input, avoiding a long set of if-else syntax. The first argument is a statement that determines what choice is made and the second is a list of the outcomes, in order or by name:

```
x <- 2; y <- 10
switch(x, log(y), sqrt(y), y)

## [1] 3.16

center <- function(x, type){
switch(type,
mean = mean(x), # make sure to use = and not <-
median = median(x),
trimmed = mean(x, trim = .1))
}
x <- rgamma(100, 1)
center(x, 'median')

## [1] 0.625

center(x, 'mean')

## [1] 0.771
```

6.4 Loops

Loops are at the core of programming in other functional languages, but in R, we often try to avoid them as they're often (but not always) slow. In many cases looping can be avoided by using vectorized calculations, versions of *apply()*, and other tricks. But sometimes they're unavoidable and for quick and dirty coding and small problems, they're fine. And in some cases they may be faster than other alternatives. One case we've already seen is that in working with lists they may be faster than their *lapply*-style counterpart, though often they will not be.

The workhorse loop is the *for* loop, which as the syntax: **`for(var in sequence) statement`**, where, as with the *if* statement, we need curly braces around the body of the loop if it contains more than one valid R statement:

```
nIts <- 500
means <- rep(NA, nIts)
for (it in 1:nIts) {
    means[it] <- mean(rnorm(100))
    if (identical(it%%100, 0))
        cat("Iteration", it, date(), "\n")
}

## Iteration 100 Sun Sep 15 13:07:53 2013
## Iteration 200 Sun Sep 15 13:07:53 2013
## Iteration 300 Sun Sep 15 13:07:53 2013
## Iteration 400 Sun Sep 15 13:07:53 2013
## Iteration 500 Sun Sep 15 13:07:53 2013
```

Challenge: how do I do this much faster?

You can also loop over a non-numeric vector of values.

```
for (state in c("Ohio", "Iowa", "Georgia")) {
    sub <- row.names(state.x77) == state
    print(state.x77[sub, "Income"])
}

## [1] 4561
## [1] 4628
## [1] 4091
```

Challenge: how can I do this faster?

Note that to print to the screen in a loop you explicitly need to use *print()* or *cat()*; just writing the name of the object will not work. This is similar to *if* statements and functions.

```
for (i in 1:10) i
```

You can use the commands *break* (to end the looping) and *next* (to go to the next iteration) to control the flow:

```
for (i in 1:10) {
    if (i == 5)
        break
    print(i)
}

## [1] 1
## [1] 2
## [1] 3
## [1] 4

for (i in 1:5) {
    if (i == 2)
        next
    print(i)
}

## [1] 1
## [1] 3
## [1] 4
## [1] 5
```

while loops are used less frequently, but can be handy: **while(condition) statement**, e.g. in optimization. See p. 59 of Venables and Ripley, 4th ed., whose code I've included in the demo code file.

A common cause of bugs in for loops is when the range ends at zero or a missing value:

```
mat <- matrix(1:4, 2)
submat <- mat[mat[1, ] > 5]
for (i in 1:nrow(submat)) print(i)

## Error:  argument of length 0
```

7 Formulas

Formulas were initially introduced into R to specify linear models, but are now used more generally.

Here are some examples of formulas in R, used to specify a model structure:

- Additive model:
  `y ~ x1 + x2 + x3`
- Additive model without the intercept:
  `y ~ x1 + x2 + x3 -1`
- All the other variables in the data frame are used as covariates:
  `y ~ .`
- All possible interactions:
  `y ~ x1 * x2 * x3`
- Only specified interactions (in this case *x1* by *x2*) (of course, you'd rarely want to fit this without *x2*):
  `y ~ x1 + x3 + x1:x2`
- Creating a factor on the fly:
  `y ~ x1 + factor(x2)`
- Protecting arithmetic expressions:
  `y ~ x1 + I(x1^2) + I(x1^3)`
- Using functions of variables
  `y ~ x1 + log(x2) + sin(x3)`

In some contexts, such as *lattice* package graphics, the "|" indicates conditioning, so `y ~ x | z` would mean to plot *y* on *x* within groups of *z*. In the context of lme-related packages (e.g., *lme4*, *nlme*, etc.), variables after "|" are *grouping* variables (e.g., if you have a random effect for each hospital, hospital would be the grouping variable) and multiple grouping variables are separated by "/".

We can manipulate formulae as objects, allowing automation. Consider how this sort of thing could be used to write code for automated model selection.

```
resp <- "y ~"
covTerms <- "x1"
for (i in 2:5) {
    covTerms <- paste(covTerms, "+ x", i, sep = "")
}
form <- as.formula(paste(resp, covTerms, sep = ""))
# lm(form, data = dat)
form

## y ~ x1 + x2 + x3 + x4 + x5

class(form)

## [1] "formula"
```

The for loop is a bit clunky/inefficient - let's do better:

```
resp <- "y ~"
covTerms <- paste("x", 1:5, sep = "", collapse = " + ")
form <- as.formula(paste(resp, covTerms))
form

## y ~ x1 + x2 + x3 + x4 + x5

# lm(form, data = dat)
```

Standard arguments in model fitting functions, in addition to the formula are *weights*, *data* (indicating the data frame in which to interpret the variable names), *subset* (for using a subset of data), and *na.action*. Note that the default *na.action* in R is set in *options()$na.action* and is *na.omit*, so be wary in fitting models in that you are dropping cases with NAs and may not be aware of it.

There is some more specialized syntax given in *R-intro.pdf* on CRAN.

8 Text manipulations and regular expressions

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 refering 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.

8.1 Basic text manipulation

A few of the basic R functions for manipulating strings are *paste()*, *strsplit()*, and *substring()*. *paste()* and *strsplit()* are basically inverses of each other: *paste()* concatenates together an arbitrary set of strings (or a vector, if using the *collapse* argument) with a user-specified separator character, while *strsplit()* splits apart based on a delimiter/separator. *substring()* splits apart the elements of a character vector based on fixed widths. Note that all of these operate in a vectorized fashion.

```
out <- paste("My", "name", "is", "Chris", ".", sep = " ")
paste(c("My", "name", "is", "Chris", "."), collapse = " ")  # equivalent

## [1] "My name is Chris ."

strsplit(out, split = " ")

## [[1]]
## [1] "My"    "name"  "is"    "Chris" "."
```

Note that *strsplit()* returns a list because it can operate on a character vector (i.e., on multiple strings).

nchar() tells the number of characters in a string.

To identify particular subsequences in strings, there are several related R functions. *grep()* will look for a specified string within an R character vector and report back indices identifying the elements of the vector in which the string was found in (using the *fixed=TRUE* argument ensures that regular expressions are NOT used). *gregexpr()* will indicate the position in each string that the specified string is found (use *regexpr()* if you only want the first occurrence). *gsub()* can be used to replace a specified string with a replacement string (use *sub()* if you only want to replace only the first occurrence).

```
vars <- c("P", "HCA24", "SOH02")
substring(vars, 2, 3)

## [1] ""   "CA" "OH"

vars <- c("date98", "size98", "x98weights98", "sdfsd")
grep("98", vars)

## [1] 1 2 3

gregexpr("98", vars)
```

```
## [[1]]
## [1] 5
## attr(,"match.length")
## [1] 2
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1] 5
## attr(,"match.length")
## [1] 2
## attr(,"useBytes")
## [1] TRUE
##
## [[3]]
## [1]  2 11
## attr(,"match.length")
## [1] 2 2
## attr(,"useBytes")
## [1] TRUE
##
## [[4]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
```

```
gsub("98", "04", vars)
```

```
## [1] "date04"       "size04"       "x04weights04"
## [4] "sdfsd"
```

8.2 Regular expressions (regexp/regex)

Overview and core syntax The *grep()*, *gregexpr()* and *gsub()* functions are more powerful when used with regular expressions. Regular expressions are a domain-specific language for finding patterns and are one of the key functionalities in scripting languages such as Perl and Python, as well as the UNIX utilities *sed*, *awk* and *grep*. Duncan Temple Lang (UC Davis Statistics) has written a nice tutorial that I've put in the repository (*regexpr-Lang.pdf*) or check out Sections 9.9 and 11 of Murrell. We'll just cover the use of regular expressions in R, but once you know that, it would be easy to use them elsewhere (Python, grep and other UNIX commands, etc.). What I describe here is the "extended regular expression" syntax (POSIX 1003.2), but with the argument `Perl=TRUE`, you can get Perl-style regular expressions. At the level we'll consider them, the syntax is quite similar.

The basic idea of regular expressions is that they allow us to find matches of strings or patterns in strings, as well as do substitution. Regular expressions are good for tasks such as:

- extracting pieces of text - for example finding all the links in an html document;
- creating variables from information found in text;
- cleaning and transforming text into a uniform format;
- mining text by treating documents as data; and
- scraping the web for data.

Regular expressions are constructed from three things:

Literal characters are matched only by the characters themselves,

Character classes are matched by any single member in the class, and

Modifiers operate on either of the above or combinations of them.

Note that the syntax is very concise, so it's helpful to break down individual regular expressions into the component parts to understand them. As Murrell notes, since regexp are their own language, it's a good idea to build up a regexp in pieces as a way of avoiding errors just as we would with any computer code. *gregexpr()* is particularly useful in seeing **what** was matched to help in understanding and learning regular expression syntax and debugging your regexp.

The special characters (meta-characters) used for defining regular expressions are: * . ^ *$ + ? ( ) [ ] { }* | \ . To use these characters literally as characters, we have to 'escape' them. In R, we have to use two backslashes insstead of a single backslash because R uses a single backslash to symbolize certain control characters, such as *\n* for newline. Outside of R, one would only need a single backslash.

Character sets and character classes If we want to search for any one of a set of characters, we use a character set, such as `[13579]` or `[abcd]` or `[0-9]` (where the dash indicates a sequence) or `[0-9a-z]` or `[ \t]`. To indicate any character not in a set, we place a ^ just inside the first bracket: `[^abcd]`. The period stands for any character.

There are a bunch of named character classes so that we don't have write out common sets of characters. The syntax is `[:class:]` where *class* is the name of the class. The classes include the *digit*, *alpha*, *alnum*, *lower*, *upper*, *punct*, *blank*, *space* (see `?regexp` in R for formal definitions of all of these, but most are fairly self-explanatory). To make a character set with a character class you need two square brackets, e.g. the digit class: `[[:digit:]]`. Or we can make a combined character set such as `[[:alnum]_]`. E.g., the latter would be useful in looking for email addresses. If you use the `[:class:]` syntax, you need to use the `perl=TRUE` argument to the relevant function to make use of Perl-style regular expressions.

```
addresses <- c("john@att.com", "stat243@bspace.berkeley.edu",
    "john_smith@att.com")
grep("[[:digit:]_]", addresses, perl = TRUE)

## [1] 2 3
```

Some synonyms for the various classes are: `\\w` = `[:alnum:]`, `\\W` = `^[:alnum:]`, `\\d` = `[:digit]`, `\\D` = `^[:digit:]`, `\\s` = `[:space:]`, `\\S` = `^[:space:]`.

Challenge: how would we 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".

Location-specific matches To find a pattern at the beginning of the string, we use ^ (note this was also used for negation, but in that case occurs only inside square brackets) and to find it at the end we use $.

```
text <- c("john", "jennifer pierce", "Juan carlos rey")
grep("^[[:upper:]]", text)  # finds text starting with upper case letter

## [1] 3

grep("[[:digit:]]$", text)  # finds text with a number at the end

## integer(0)
```

What does this match: `^[^[:lower:]]$` ?

Here are some more examples, illustrating the use of regexp in *grep()*, *gregexpr(),* and *gsub()*.

```
text <- c("john", "jennifer pierce", "Juan carlos rey")
grep("[ \t]", text)

## [1] 2 3

gregexpr("[ \t]", text)

## [[1]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1] 9
## attr(,"match.length")
## [1] 1
## attr(,"useBytes")
## [1] TRUE
##
## [[3]]
## [1]  5 12
## attr(,"match.length")
## [1] 1 1
## attr(,"useBytes")
## [1] TRUE

gsub("^j", "J", text)

## [1] "John"            "Jennifer pierce"
## [3] "Juan carlos rey"
```

To extract the actual matching sequences and not just which elements match and where they match, you can use *regmatches()*:

```
text <- c("john", "jennifer pierce", "Juan carlos rey")
matches <- gregexpr("^[[:upper:]][[:lower:]]+ ", text)
regmatches(text, matches)

## [[1]]
## character(0)
##
## [[2]]
## character(0)
##
## [[3]]
## [1] "Juan "
```

Repetitions Now suppose I wanted to be able to detect "V1@gra" as well. I need to be able to deal with repetitions of character sets.

I can indicate repetitions as indicated in these examples:

- [[:digit:]]* – any number of digits (zero or more)
- [[:digit:]]+ – at least one digit
- [[:digit:]]? – zero or one digits
- [[:digit:]]{1,3} – at least one and no more than three digits
- [[:digit:]]{2,} – two or more digits

An example is that `\\[.*\\]` is the pattern of any number of characters (.*) separated by square brackets.

So the spam search might become:

```
text <- c("hi John", "V1@gra", "here's the problem set")
grep("[[:alpha:]]+[[:digit:][:punct:]]+[[:alpha:]]*", text)

## [1] 2 3

# ok, so that doesn't quite work...
```

Grouping and references We often want to be able to look for multi-character patterns and to be able to refer back to the patterns that are found. Both are accomplished with parentheses. We can search for a group of characters as follows by putting the group in parentheses, such as `([[:digit:]]{1,3}\\.)` to find a 1 to 3 digit number followed by a period. Here's an example of searching for an IP number:

```
grep("([[:digit:]]{1,3}\\.){3}[[:digit:]]{1,3}", text)
```

It's often helpful to be able to save a pattern as a variable and refer back to it. Here's an example that might have been helpful in dealing with the extra commas in the comma-delimited FAO file in PS1:

```
text <- ('"H4NY07011","ACKERMAN, GARY L.","H","$13,242",,,')
gsub("([^\",]),", "\\1", text)

## [1] "\"H4NY07011\",\"ACKERMAN GARY L.\",\"H\",\"$13242\",,,"
```

We can have multiple sets of parentheses, referred to using `\\1`, `\\2`, etc.

We can indicate any one of a set of multi-character sequences as: `(http|ftp)`.

```
gregexpr("(http|ftp):\\/\\/", c("at the site http://www.ibm.com",
    "other text", "ftp://ibm.com"))

## [[1]]
## [1] 13
## attr(,"match.length")
## [1] 7
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
##
## [[3]]
## [1] 1
## attr(,"match.length")
## [1] 6
## attr(,"useBytes")
## [1] TRUE
```

Challenge: How would I extract an email address from an arbitrary text string? Comment: if you want to just return the address using `gsub(pattern, "\\1", text)` will not do the trick. Why not?

Challenge: 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/replace?

gsub("(Jan|Feb|Mar|...|Dec)-([0-9]{1,2})","\\2 \\1",text)

Greedy matching It turns out the pattern matching is 'greedy' - it looks for the longest match possible.

Suppose we want to strip out html tags as follows:

```
text <- "Students may participate in an internship <b> in place\n</b> of <b
gsub("<.*>", "", text)

## [1] "Students may participate in an internship  of their courses."
```

One solution is to append a ? to the repetition syntax to cause the matching to be non-greedy. Here's an example.

```
gsub("<.*?>", "", text)

## [1] "Students may participate in an internship  in place\n of  one  of th
```

use <[^>]*> to get around the greediness without using the ?

However, one can also avoid greedy matching by being more clever. **Challenge**: How could we change our regexp to avoid the greedy matching without using the "?"?

Regular expressions in other contexts Regular expression can be used in a variety of places. E.g., to split by any number of white space characters

```
line <- "a dog\tjumped\nover \tthe moon."
cat(line)

## a dog jumped
## over  the moon.
```

Table 1. Regular expression syntax.

Syntax	What it matches
^ab	match 'ab' at the beginning of the string
ab$	match 'ab' at the end of the string
[abc]	match a or b or c anywhere (this is a character class)
[\t]	match a space or a tab
`(ab|cd|def)`	match any of the strings in the set
`(ab){2,9}`	match 'ab' repeated at least 2 and no more than 9 times
(ab){2,}	match 'ab' repeated 2 or more times
[0-9a-z]	match a single digit or lower-case alphabetical
[^0-9]	match any single character except a digit
`a.b`	match a and b separated by a single character
*a.*b*	match a and b separated by any number of (or no) characters
a.+b	like a.*b but must have at least one character in between
[[:digit:]]	match *digit* class; other classes are *alpha*, *alnum*, *lower*, *upper*, *punct*, *blank*, *space* (see ?regexp)
\\[	double backslashes are used if we want to search for a meta-character used in regexp syntax

```
strsplit(line, split = "[[:space:]]+")

## [[1]]
## [1] "a"      "dog"    "jumped" "over"   "the"
## [6] "moon."

strsplit(line, split = "[[:blank:]]+")

## [[1]]
## [1] "a"            "dog"          "jumped\nover"
## [4] "the"          "moon."
```

Summary Table 1 summarizes the key syntax in regular expressions.