Unit 5: Programming concepts, illustrated with R

September 18, 2021

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"              "exampleRscript.R"
## [3] "exampleRscript.R~" "figures"
## [5] "subset_unit5.sh"
```

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

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

## [1] FALSE

list.files("../data")

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

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

```
Sys.info()

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

```
##                              "5.4.0-74-generic"
##                                         version
## "#83-Ubuntu SMP Sat May 8 02:35:39 UTC 2021"
##                                        nodename
##                                       "smeagol"
##                                         machine
##                                        "x86_64"
##                                           login
##                                       "paciorek"
##                                            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 4.1.0 (2021-05-18)
```

```
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 20.04.2 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3
## LAPACK: /usr/lib/x86_64-linux-gnu/openblas-pthread/liblapack.so.3
##
## 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] methods   base
##
## other attached packages:
## [1] pryr_0.1.4 knitr_1.33 SCF_4.1.0
##
## loaded via a namespace (and not attached):
##  [1] compiler_4.1.0   magrittr_2.0.1   tools_4.1.0
##  [4] Rcpp_1.0.7       codetools_0.2-18 stringi_1.7.3
##  [7] highr_0.9        stringr_1.4.0    xfun_0.25
## [10] evaluate_0.14
```

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

Text manipulations in R have a number of things in common with Python, Perl, 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.

2.1 String processing and regular expressions in R

For material on string processing in R, see the tutorial, String processing in R and Python. (You can ignore the sections on Python.) That tutorial then refers to the *Using the bash shell* tutorial for details on regular expressions, which we discussed as part of the end of Unit 3. Finally, to test out regular expression syntax see this online tool.

In class we'll work through the string processing tutorial, focusing in particular on the use of regular expressions with the *stringr* package.

2.2 Regex/string processing challenges

We'll work on these challenges in class in the process of working through the string processing tutorial.

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?)

2.3 Side notes on special characters in R

Recall that when characters are used for special purposes, we need to escape them if we want them interpreted as the actual character. In what follows, I show this in R, but similar manipulations are sometimes needed in the shell and in Python.

This can get particularly confusing in R as the backslash is also used to input special characters such as newline (\n) or tab (\t). (Note that it is hard to get the PDF to compile correctly for these R chunks, so I am just pasting in the output from running in R 'manually'.)

```
tmp <- "Harry said, \"Hi\""
## cat(tmp)   ## prints out without a newline (It's hard to show in the pdf
tmp <- "Harry said, \"Hi\".\n"
cat(tmp)
## Harry said, "Hi".

tmp <- c("azar", "foo", "hello\tthere\n")
cat(tmp)
## azar foo hello there
print(tmp)
## [1] "azar"           "foo"            "hello\tthere\n"
grep("[\tz]", tmp)
## [1] 1 3
```

As a result in R, we often need two backslashes when working with regular expressions. In these examples, the first backslash says to interpret the next backslash literally, with the second backslash being used to indicate that the caret (^) should be interpreted literally and not as a special character used for specifying regular expressions.

```
## Search for characters that are not 'z'
## (using ^ as regular expression syntax)
grep("[^z]", c("a^2", "93", "zit", "azar", "zzz"))
# [1] 1 2 3 4

## Search for either a '^' (as a regular charcter) or a 'z':
grep("[\\^z]", c("a^2", "93", "zit", "azar", "zzz"))
# [1] 1 3 4 5

## This fails because '\^' is not an escape sequence:
grep("[\^z]", c("a^2", "93", "zit", "azar", "zzz"))
# Error: '\^' is an unrecognized escape in character string starting ""[\^"

## Search for exactly three characters
```

```
## (using . as regular expression syntax)
grep("^.{3}$", c("abc", "1234"))
# [1] 1

## Search for a period (as a regular character)
grep("\\.", c("3.9", "27"))
# [1] 1

## This fails because '\.' is not an escape sequence
grep("\.", c("3.9", "27"))
# 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:

```
## Suupose we want to use a \ in our string:
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 and the subsections of the string processing tutorial that discuss backslashes and escaping.

Advanced note: Searching for an actual backslash gets even more complicated, because we need to pass two backslashes as the regular expression, so that a literal backslash is searched for. However, to pass two backslashes, we need to escape each of them with a backslash so R doesn't treat each backslash as part of a special character. So that's four backslashes to search for a single backslash. Yikes. One rule of thumb is just to keep entering backslashes until things works!

```r
## Search for an actual backslash
tmp <- "something \\ other\n"
cat(tmp)
# something \ other

grep("\\\\", tmp)
# [1] 1
grep("\\", tmp)
# Error in grep("\\", tmp) :
#  invalid regular expression '\', reason 'Trailing backslash'
```

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.

Similar things come up in the shell and in Python, but in the shell you often don't need two backslashes. E.g. you could do this to look for a literal ^ character.

```
grep '\^' file.txt
```

3 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(dplyr)

##
## Attaching package:  'dplyr'
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union

library(help = dplyr)
## 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/4.1"
## [2] "/system/linux/lib/R-20.04/4.1/x86_64/site-library"
## [3] "/usr/lib/R/site-library"
## [4] "/usr/lib/R/library"

searchpaths()

##  [1] ".GlobalEnv"
##  [2] "/system/linux/lib/R-20.04/4.1/x86_64/site-library/dplyr"
##  [3] "/system/linux/lib/R-20.04/4.1/x86_64/site-library/pryr"
##  [4] "/system/linux/lib/R-20.04/4.1/x86_64/site-library/knitr"
##  [5] "/usr/lib/R/library/stats"
##  [6] "/usr/lib/R/library/graphics"
```

```
##  [7] "/usr/lib/R/library/grDevices"
##  [8] "/usr/lib/R/library/utils"
##  [9] "/usr/lib/R/library/datasets"
## [10] "/system/linux/lib/R-20.04/4.1/x86_64/site-library/SCF"
## [11] "/usr/lib/R/library/methods"
## [12] "Autoloads"
## [13] "/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('dplyr', 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('dplyr_VERSION.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:dplyr"
##  [3] "package:pryr"      "package:knitr"
##  [5] "package:stats"     "package:graphics"
##  [7] "package:grDevices" "package:utils"
##  [9] "package:datasets"  "package:SCF"
## [11] "package:methods"   "Autoloads"
## [13] "package:base"

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

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

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

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

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).

You can look at Table 7.1 in the Adler book to see some other types.

```
devs <- rnorm(5)
class(devs)

## [1] "numeric"

typeof(devs)

## [1] "double"

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" "array"

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 (an S3 class in this simple example - we'll discuss S3 classes in Section 4.4.1).

```
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)
attributes(x)

## NULL
```

```r
qs <- quantile(x, c(.025, .975))
attributes(qs)

## $names
## [1] "2.5%"  "97.5%"

qs

##  2.5% 97.5%
## -1.93  2.00

qs[1] + 3

## 2.5%
## 1.07

object.size(qs)

## 352 bytes
```

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

```r
names(qs) <- NULL
qs

## [1] -1.93  2.00

object.size(qs)

## 64 bytes
```

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

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

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

```r
names(mtcars)
```

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

```r
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"
```

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

```
## $names
## [1] "x" "y"
```

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

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

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

attributes(mat)

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

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: 0x55729290c818>
## <environment: namespace:base>

x <- 0; y <- 0
out <- mean(x = c(3,7)) # usual way to pass an argument to a function by na
out <- mean(c(3,7))     # or by position
## what does the following do?
out <- mean(x <- c(3,7)) # this is allowable, but confusing
out <- mean(y = c(3,7))  # why doesn't this work?

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

out <- mean(y <- c(3,7)) # again, allowable, but confusing
```

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
##   0.001   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). 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] -0.463  2.638 -1.206 -0.182  2.445

## 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 (i.e., class-specific functions) 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:

```r
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

## function (object, ...)
## UseMethod("summary")
## <bytecode: 0x5572967ea100>
## <environment: namespace:base>

methods(summary)

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

```
## [13] summary.glm
## [14] summary.infl*
## [15] summary.lm
## [16] summary.loess*
## [17] summary.manova
## [18] summary.matrix
## [19] summary.mlm*
## [20] summary.nls*
## [21] summary.packageStatus*
## [22] summary.POSIXct
## [23] summary.POSIXlt
## [24] summary.ppr*
## [25] summary.prcomp*
## [26] summary.princomp*
## [27] summary.proc_time
## [28] summary.rlang_error*
## [29] summary.rlang_trace*
## [30] summary.srcfile
## [31] summary.srcref
## [32] summary.stepfun
## [33] summary.stl*
## [34] summary.table
## [35] summary.tukeysmooth*
## [36] summary.vctrs_sclr*
## [37] summary.vctrs_vctr*
## [38] summary.warnings
## see '?methods' for accessing help and source code
```

In many cases there will be a default method (here, *summary.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.
```

The print method Like *summary()*, *print()* is a generic method, with various class-specific methods, such as *print.lm()*.

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..

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 'k' (no parentheses!) instead of '*quit()*'?

What we've just discussed are the old-style R (and S) object orientation, called S3 methods. An old, but somewhat newer style is called S4 and we'll discuss it next. S3 is still commonly used, in part because S4 can be slow. S4 is more structured than S3.

4.4.2 S4 approach (optional)

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)
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 R6 classes

R6 classes are a somewhat new construct in R. They are classes somewhat similar to S4. Importantly, they behave like pointers (the fields in the objects are 'mutable'). We'll discuss pointers in Section 6.5. Let's work through an example where we set up the fields of the class (like S4 slots) and class methods, including a constructor.

Here's the initial definition of the class, with both public (user-facing) and private (internal use only) methods and fields.

```
library(R6)

tsSimClass <- R6Class("tsSimClass",
    ## class for holding time series simulators
    public = list(
        initialize = function(times, mean = 0, corParam = 1){
            library(fields)
            stopifnot(is.numeric(corParam), length(corParam) == 1)
            stopifnot(is.numeric(times))
            private$times <- times
            private$n <- length(times)
            private$mean <- mean
            private$corParam <- corParam
            private$currentU <- FALSE
            private$calcMats()
        },

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

```
        getTimes = function(){
            return(private$times)
        },

        print = function(){ # 'print' method
            cat("R6 Object of class 'tsSimClass' with ",
                private$n, " time points.\n", sep = '')
            invisible(self)
        }
    ),

    ## private methods and functions not accessible externally
    private = list(
        calcMats = function() {
            ## calculates correlation matrix and Cholesky factor
            lagMat <- fields::rdist(private$times) # local variable
            corMat <- exp(-lagMat^2 / private$corParam^2)
            private$U <- chol(corMat) # square root matrix
            cat("Done updating correlation matrix and Cholesky factor.\n")
            private$currentU <- TRUE
            invisible(self)
        },
        n = NULL,
        times = NULL,
        mean = NULL,
        corParam = NULL,
        U = NULL,
        currentU = FALSE
    )
)
```

We can add methods after defining the class (but those methods wouldn't be accessible to objects of the class that have already been created.

```
tsSimClass$set("public", "simulate", function() {
    if(!private$currentU)
```

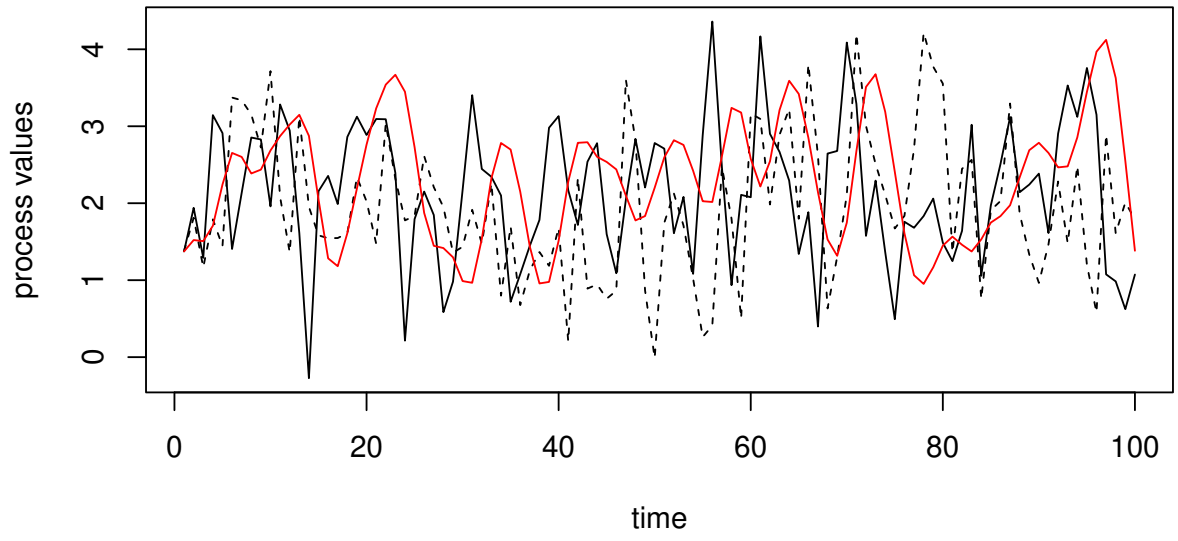
```
        private$calcMats()
    ## analogous to mu+sigma*z for generating N(mu, sigma^2)
    return(private$mean + crossprod(private$U, rnorm(private$n)))
})
```

That's just for demonstration. In general we would define *simulate* when we define the class originally.

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

```
myts <- tsSimClass$new(1:100, 2, 1)

## Loading required package:  spam
## Loading required package:  dotCall64
## Loading required package:  grid
## Spam version 2.6-0 (2020-12-14) 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:  viridis
## Loading required package:  viridisLite
## See https://github.com/NCAR/Fields for
##  an extensive vignette, other supplements and source code

## Done updating correlation matrix and Cholesky factor.

myts

## R6 Object of class 'tsSimClass' with 100 time points.

set.seed(1)
## here's a simulated time series
y <- myts$simulate()
```

```
plot(myts$getTimes(), y, type = 'l', xlab = 'time',
     ylab = 'process values')
## here's a second simulated time series
y2 <- myts$simulate()
lines(myts$getTimes(), y2, lty = 2)

myts2 <- tsSimClass$new(1:100, 2, 3)

## Done updating correlation matrix and Cholesky factor.

set.seed(1)
## here's a simulated time series with a different value of
## the correlation parameter (corParam)
y <- myts2$simulate()
lines(myts2$getTimes(), y, col = 'red')
```




```
## That simulated time series is less wiggly because the corParam value
## is larger than before.

mytsRef <- myts
## 'mytsRef' and 'myts' are names for the same underlying object
```

```
mytsFullCopy <- myts$clone()  # mytsFullCopy is a true copy and a new objec
myts$changeTimes(seq(0,1000, length = 100))

## Done updating correlation matrix and Cholesky factor.

myts$getTimes()[1:5] # this and

## [1]  0.0 10.1 20.2 30.3 40.4

mytsRef$getTimes()[1:5] # this are the same

## [1]  0.0 10.1 20.2 30.3 40.4

mytsFullCopy$getTimes()[1:5] # this is different

## [1] 1 2 3 4 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 *clone()* 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:

  ```
  R6Class("tsSimClass", inherit = "simClass", ...)
  ```

- Here we see that use of private fields shields them from modification by users. In this example, the correlation matrix and the Cholesky factor U are both functions of the vector of times. So we don't want to allow a user to directly modify *times*. Instead we force them to use *setTimes()*, which correctly keeps all the fields in the object internally consistent (by calling *calcMats()*):

```
## the next line would be dangerous if 'times' were public, since
## changing 'times' should result in changing the correlation matrix and
## changing U.
myts$times <- 1:10

## Error in myts$times <- 1:10:  cannot add bindings to a locked environment
```

- If you need to refer to methods and fields you refer to the entire object as either *self* or *private*.
- There is a older, more complicated, slower variation on R6 classes called ReferenceClasses. See the *help(ReferenceClasses)*.

More details on R6 classes can be found in the Advanced R book: https://adv-r.hadley.nz/r6.html.

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 as part of a group of packages called the *tidyverse*) 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 *dplyr* package implements this framework (as does the *pandas* package for Python). 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 recommend *pivot_longer()* and *pivot_wider()* from recent versions of the *tidyr* package. There are also older *tidyr* functions called *gather()* and *spread()*. 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.

5.3 Non-standard evaluation and the tidyverse

Many tidyverse packages use non-standard evaluation to make it easier to code. For example in the following dplyr example, you can refer directly to *country* and *unemp*, which are variables in the data frame, without using `data$country` or `data$unemp` and without using quotes around the variable names, as in "`country`" or "`unemp`". Referring directly to the variables in the data frame is not standard R usage, hence the term "non-standard evaluation". One reason it is not standard is that country and unemp are not themselves independent R variables so R can't find them in the usual way (see Section 6.6).

```
library(dplyr)

cpds <- read.csv(file.path('..', 'data', 'cpds.csv'),
```

```
                  stringsAsFactors = FALSE)

cpds2 <- cpds %>% group_by(country) %>%
                  mutate(mean_unemp = mean(unemp))

head(cpds2)

## # A tibble: 6 x 7
## # Groups:   country [1]
##    year country   vturn outlays realgdpgr unemp
##   <int> <chr>     <dbl>   <dbl>     <dbl> <dbl>
## 1  1960 Australia  95.5    NA       NA     1.42
## 2  1961 Australia  95.3    NA       -0.07  2.79
## 3  1962 Australia  95.3    23.2      5.71  2.63
## 4  1963 Australia  95.7    23.0      6.1   2.12
## 5  1964 Australia  95.7    22.9      6.28  1.15
## 6  1965 Australia  95.7    24.9      4.97  1.15
## # ... with 1 more variable: mean_unemp <dbl>
```

This 'magic' is done by capturing the code expression you write and evaluating it in a special way in the context of the data frame. I believe this uses R's environment class (discussed in Section 6), but haven't looked more deeply.

While this has benefits, this so-called non-standard evaluation makes it harder to program functions in the usual way, as illustrated in the following code chunk, where neither attempt to use the function works.

```
add_mean <- function(data, group_var, summarize_var) {
    data %>% group_by(group_var) %>%
             mutate(mean_of_var = mean(summarize_var))
}

try(cpds2 <- add_mean(cpds, country, unemp))

## Error : Must group by variables found in `.data`.
## * Column `group_var` is not found.

try(cpds2 <- add_mean(cpds, 'country', 'unemp'))
```

```
## Error : Must group by variables found in `.data`.
## * Column `group_var` is not found.
```

For more details on how to avoid this problem when writing functions that involve tidyverse manipulations, see https://dplyr.tidyverse.org/articles/programming.html.

Note that the tidyverse is not the only place where non-standard evaluation is used. Consider this *lm()* call:

```
lm(y ~ x, weights = w, data = mydf)
```

Where is the non-standard evaluation there?

6 Functions, frames, and variable scope

R is a functional programming language. All operations are carried out by functions including assignment, various operators (such as addition, subtraction, etc.), 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.

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 function frame is taken off the stack. The environment is removed, unless the environment serves as the enclosing environment of another environment.

I'm not expecting you to fully understand that previous paragraph and all the terms in it. We'll see all the details in this section.

6.1 Functions as objects

Everything in R is an object, including functions. We can assign functions to variables in the same way we assign numeric and other values.

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

## [1] "numeric"
## [1] "double"

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] 0.347
```

We can also pass a function into another function 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] -0.12

## lapply/sapply operate similarly
sapply(x, abs)

##  [1] 0.636 0.462 1.432 0.651 0.207 0.393 0.320 0.279
##  [9] 0.494 0.177
```

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] -0.12

f(mean, x)

## [1] -0.12
```

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
```

You can't generally tell directly which arguments are required; in general you'd need to look at the documentation. R will error out if it is expecting an argument, rather than looking for that argument elsewhere.

```
x <- 1
y <- 2
myfun <- function(x) {
    print(y)
    print(x)
}
myfun()

## [1] 2

## Error in h(simpleError(msg, call)): error in evaluating the argument
'x' in selecting a method for function 'print': argument "x" is missing,
with no default
```

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.

```
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.

```
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 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). Each function call is associated with a *frame* that contains the local variables for that function call.

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 frame from which a function was called.

Some terminology: for our purposes we'll use the terms *frame* and *environment* somewhat interchangeably for the moment. A *frame* or *environment* is a collection of named objects. (Note that when we talk about variable scope in Section 6.6, we'll have to be more careful with our terminology.) So in the context of a function call, the frame is the set of local variables available in the function, including arguments passed to the function.

sys.nframe() returns the number of the current frame/environment and *sys.parent()* the number of the parent, while *parent.frame()* gives the name of the frame/environment of the parent (i.e., the calling) frame. *sys.frame()* gives the name of the frame/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 frame/environment. I won't print the results here because *knitr* messes up the frame counting somehow.

```
## NOTE: run this chunk outside RStudio as RStudio seems to
##       inject additional frames
sys.nframe()
f <- function() {
        cat('in f: Frame number is ', sys.nframe(),
            '; parent frame number is ', sys.parent(), '.\n', sep = '')
        cat('in f: Frame (i.e., environment) is: ')
        print(sys.frame(sys.nframe()))
        cat('in f: Parent is ')
        print(parent.frame())
        cat('in f: Two frames up is ')
        print(sys.frame(-2))
}
f()
ff <- function() {
        cat('in ff: Frame (i.e., environment) is: ')
        print(sys.frame(sys.nframe()))
        cat('in ff: Parent is ')
        print(parent.frame())
```

```
        f()
}
ff()
```

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

## Here's a recursive function, so we'll get a lot of function calls on the
g <- function(y) {
    if(y > 0) g(y-1) else gg()
}

## Ultimately, gg() is called, and it prints out info about the call stack
gg <- function() {
    ## this gives us the information from sys.calls(),
    ##   sys.parents() and sys.frames() as one object
    ## Rather than running print(sys.status()),
    ## which would involve adding print() to the call stack,
    ## we'll run sys.status and then print the result out.
    tmp <- sys.status()
    print(tmp)
}

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.

6.5 Approaches to passing arguments to functions

6.5.1 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 R6 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. For example, using the old .C syntax, here's 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)
```

Pointers in R? Are there pointers in R? From a user perspective, one might say 'no', because an R programmer can't use pointers explicitly. But pointer-like behavior is occurring behind the scenes in lots of ways:

- Lists in R are essentially vectors of pointers to the elements of the list
- Character vectors in R are essentially pointers to the individual character strings.
- R6 objects behave like pointers and are passed by reference rather than by copy, as seen in Section 4.4.3.
- Environments (see Section 6.9) behave like pointers and are passed by reference rather than by copy.

6.5.2 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)

d <- 100
f(7)
```

What's strange about that?

Another example:

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

##    user  system elapsed
##    0.07    0.00    0.07

system.time(f(3))

## [1] "hi"
##    user  system elapsed
##   0.001   0.000   0.000

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

## [1] "hi"
##    user  system elapsed
##   0.002   0.000   0.001
```

Lazy evaluation is not just an R thing. It also occurs in Spark and in the Python Dask package. The basic idea is to delay executation until it's really needed, with the goal that if one does so, the system may be able to better optimize a series of multiple steps as a joint operation relative to executing them one by one.

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?

6.6 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] 103
```

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

x <- 3
f <- function() {
    f2 <- function() { print(x) }
    x <- 7
```

```
    f2()
}
f() # what will happen?

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

Here's a somewhat tricky example:

```
y <- 100
fun_constructor <- function(){
        y <- 10
        g <- function(x) {
            return(x + y)
        }
        return(g)
}
## fun_constructor() creates functions
myfun <- fun_constructor()
myfun(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 *fun_constructor()* finishes, does its environment disappear? What would happen if it did?
4. What is the enclosing environment of *myfun()*?

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.

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

## <environment: 0x557298b95030>

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

## [1] "g" "y"

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

## <environment: 0x557299d79718>

environment(myfun)

## <environment: 0x557299d79718>

myfun(3)

## [1] 13

environment(myfun)$y

## [1] 10

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

## function(x) x + y
## <environment: 0x557299d79718>
```

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?

Detecting non-local variables We can use *codetools::findGlobals* to detect non-local variables when we are programming.

```
library(codetools)
f <- function() {
    y <- 3
    print(x + y)
}
findGlobals(f)

## [1] "{"     "+"     "<-"    "print" "x"
```

Is that result what you would expect? What does it say about my statement that using non-local variables is a bad idea?

Closures One way to avoid passing data by value is to associate data with a function, using a *closure*. This is a functional programming way to achieve something like an OOP class. 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 function(s) is created and returned and one can then call that new function(s). The new function 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 the function (or the multiple functions) have access. The nice thing about this compared to using a global variable is that the data in the closure is bound up with the function(s) and is protected from being changed by the user of the closure. Chambers provides an example of this in Sec. 5.4.

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

##  [1]  4.786  0.989 -2.461  1.462  2.215  1.727 -0.916
##  [8]  4.535  1.170 -1.864

x <- rnorm(1e7)
scaler <- scaler_constructor(x)
object.size(scaler) # hmmm

## 3800 bytes

object.size(environment(scaler)$data)

## 80000048 bytes

library(pryr)
object_size(scaler) # that's better!

## 80 MB
```

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] 3.59 3.41 3.47 3.46 3.43 3.48 3.51 3.48 3.50 3.46
##  [11] 3.41 3.62 3.46 3.46 3.49 3.50 3.56 3.50 3.58 3.60
##  [21] 3.46 3.45 3.50 3.41 3.46 3.59 3.35 3.50 3.51 3.37
##  [31] 3.46 3.38 3.58 3.52 3.45 3.58 3.50 3.47 3.54 3.57
##  [41] 3.53 3.58 3.40 3.50 3.50 3.56 3.41 3.45 3.50 3.53
##  [51] 3.49 3.57 3.46 3.50 3.43 3.48 3.54 3.45 3.53 3.53
##  [61] 3.46 3.36 3.41 3.58 3.58 3.47 3.51 3.50 3.56 3.48
##  [71] 3.39 3.48 3.62 3.54 3.51 3.52 3.47 3.49 3.43 3.45
##  [81] 3.40 3.52 3.43 3.49 3.51 3.56 3.55 3.46 3.30 3.56
##  [91] 3.47 3.49 3.41 3.40 3.46 3.43 3.43 3.44 3.45 3.42
```

6.7 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:codetools"
##  [3] "package:fields"      "package:viridis"
##  [5] "package:viridisLite" "package:spam"
##  [7] "package:grid"        "package:dotCall64"
##  [9] "package:R6"          "package:dplyr"
## [11] "package:pryr"        "package:knitr"
## [13] "package:stats"       "package:graphics"
## [15] "package:grDevices"   "package:utils"
## [17] "package:datasets"    "package:SCF"
## [19] "package:methods"     "Autoloads"
## [21] "package:base"
```

```
searchpaths()
```

```
##  [1] ".GlobalEnv"
##  [2] "/usr/lib/R/library/codetools"
##  [3] "/system/linux/lib/R-20.04/4.1/x86_64/site-library/fields"
##  [4] "/system/linux/lib/R-20.04/4.1/x86_64/site-library/viridis"
##  [5] "/system/linux/lib/R-20.04/4.1/x86_64/site-library/viridisLite"
##  [6] "/system/linux/lib/R-20.04/4.1/x86_64/site-library/spam"
##  [7] "/usr/lib/R/library/grid"
##  [8] "/system/linux/lib/R-20.04/4.1/x86_64/site-library/dotCall64"
##  [9] "/system/linux/lib/R-20.04/4.1/x86_64/site-library/R6"
## [10] "/system/linux/lib/R-20.04/4.1/x86_64/site-library/dplyr"
## [11] "/system/linux/lib/R-20.04/4.1/x86_64/site-library/pryr"
## [12] "/system/linux/lib/R-20.04/4.1/x86_64/site-library/knitr"
## [13] "/usr/lib/R/library/stats"
## [14] "/usr/lib/R/library/graphics"
## [15] "/usr/lib/R/library/grDevices"
## [16] "/usr/lib/R/library/utils"
## [17] "/usr/lib/R/library/datasets"
## [18] "/system/linux/lib/R-20.04/4.1/x86_64/site-library/SCF"
## [19] "/usr/lib/R/library/methods"
## [20] "Autoloads"
## [21] "/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) # enclosing env't, NOT parent frame!
}

## [1] "stats"
## [1] "imports:stats"
## [1] "base"
## [1] "R_GlobalEnv"
## [1] "package:codetools"
## [1] "package:fields"
```

```
## [1] "package:viridis"
## [1] "package:viridisLite"
## [1] "package:spam"
## [1] "package:grid"
## [1] "package:dotCall64"
## [1] "package:R6"
## [1] "package:dplyr"
## [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] "package:methods"
## [1] "Autoloads"
## [1] "base"
```

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

```
##    label
## 1  <environment: namespace:stats>
## 2  <environment: 0x5572932bc830>
## 3  <environment: namespace:base>
## 4  <environment: R_GlobalEnv>
## 5  <environment: package:codetools>
## 6  <environment: package:fields>
## 7  <environment: package:viridis>
## 8  <environment: package:viridisLite>
## 9  <environment: package:spam>
## 10 <environment: package:grid>
## 11 <environment: package:dotCall64>
## 12 <environment: package:R6>
## 13 <environment: package:dplyr>
```

```
## 14 <environment: package:pryr>
## 15 <environment: package:knitr>
## 16 <environment: package:stats>
## 17 <environment: package:graphics>
## 18 <environment: package:grDevices>
## 19 <environment: package:utils>
## 20 <environment: package:datasets>
## 21 <environment: package:SCF>
## 22 <environment: package:methods>
## 23 <environment: 0x557291bee598>
## 24 <environment: base>
## 25 <environment: R_EmptyEnv>
##    name
## 1  ""
## 2  "imports:stats"
## 3  ""
## 4  ""
## 5  "package:codetools"
## 6  "package:fields"
## 7  "package:viridis"
## 8  "package:viridisLite"
## 9  "package:spam"
## 10 "package:grid"
## 11 "package:dotCall64"
## 12 "package:R6"
## 13 "package:dplyr"
## 14 "package:pryr"
## 15 "package:knitr"
## 16 "package:stats"
## 17 "package:graphics"
## 18 "package:grDevices"
## 19 "package:utils"
## 20 "package:datasets"
## 21 "package:SCF"
## 22 "package:methods"
## 23 "Autoloads"
```

```
## 24 ""
## 25 ""
```

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.

6.8 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 R6 (or Reference Class) objects.
2. We can use a *closure*, as discussed previously.
3. 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.

4. We can use replacement functions (Section 6.11), 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.

- 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] -1.414  3.411  5.041 -2.310 -0.182  2.426 -1.747
##  [8]  0.283  0.123 -2.021

x <- rnorm(1e7)
myFun2 <- with(list(data = x), function(param) return(param * data))
object_size(myFun2)

## 80 MB
```

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

6.9 Creating and working in an environment (optional)

We've already talked extensively about the environments that R creates. Occasionally you may want to create your own 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 R6 and Reference Classes, using an environment was one way to pass objects by reference, avoiding having to re-assign the output (and in fact R6 classes are just a wrapper around the use of environments). Here's an example where we iteratively update a random walk (but note that if I were actually doing this I would use an R6 class and not an environment).

```
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)
```

6.10 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(state = 'new york', value = 1:5),
               list(state = 'california', value = 6:10),
               list(state = 'delaware', value = 11:15))

result <- lapply(myList, `[[`, 2)
result

## [[1]]
## [1] 1 2 3 4 5
##
```

```
## [[2]]
## [1]  6  7  8  9 10
##
## [[3]]
## [1] 11 12 13 14 15

## note that the index "2" is the additional argument to the [[ function
myMat <- sapply(myList, `[[`, 2)
myMat

##      [,1] [,2] [,3]
## [1,]    1    6   11
## [2,]    2    7   12
## [3,]    3    8   13
## [4,]    4    9   14
## [5,]    5   10   15

cbind(myList[[1]][[2]], myList[[2]][[2]])  ## 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
```

We can also use operators with our S3 classes. Picking up our example from our discussion of S3 OOP, 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.

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

methods(`+`)

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

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

older_yog

## $firstname
## [1] "Yogi"
##
```

```
## $surname
## [1] "the Bear"
##
## $age
## [1] 35
##
## attr(,"class")
## [1] "bear"
```

6.11 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: 0x557295bd4768>
## <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'
```

We can use replacement functions with S3 classes we define. Again, picking up our example from our discussion of S3 OOP, 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

older_yog

## $firstname
## [1] "Yogi"
##
## $surname
## [1] "the Bear"
##
## $age
## [1] 60
##
## attr(,"class")
## [1] "bear"
```

6.12 Summing up

Now that we've seen all that, this summary that I included at the start of Section 6 should make sense.

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 function frame is taken off the stack. The environment is removed, unless the environment serves as the enclosing environment of another environment.