Unit 5: Programming concepts, illustrated with R

September 11, 2020

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:

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

## [1] "badCode.R"             "cache"
## [3] "class1-demo.sh"        "class2-taboo-game.md"
## [5] "figures"
```

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

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

## [1] FALSE

list.files("../data")

## [1] "coop.txt.gz"             "cpds.csv"
## [3] "hivSequ.csv"             "IPs.RData"
## [5] "precip.txt"              "precipData.txt"
## [7] "RTADataSub.csv"          "stackoverflow-2016.db"
```

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

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

## [1] "coop.txt.gz"             "cpds.csv"
## [3] "hivSequ.csv"             "IPs.RData"
## [5] "precip.txt"              "precipData.txt"
## [7] "RTADataSub.csv"          "stackoverflow-2016.db"
```

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

```r
Sys.info()

##                                       sysname
##                                       "Linux"
##                                       release
##                           "4.15.0-74-generic"
##                                       version
## "#84-Ubuntu SMP Thu Dec 19 08:06:28 UTC 2019"
##                                      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).

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

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

```
##
## $browserNLdisabled
## [1] FALSE
##
## $CBoundsCheck
## [1] FALSE

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

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

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

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

- Use `Ctrl-C` to interrupt execution. This will generally back out gracefully, returning you to a state as if the command had not been started. Note that if R is exceeding memory availability, there can be a long delay. This can be frustrating, particularly since a primary reason you would want to interrupt is when R runs out of memory.

- The R mailing list archives are very helpful for getting help - always search the archive before posting a question. More info on where to find R help in Unit 5 on debugging.

  - *sessionInfo()* gives information on the current R session - it's a good idea to include this information (and information on the operating system such as from *Sys.info()*) when you ask for help on a mailing list

```
sessionInfo()
```

```
## R version 3.6.1 (2019-07-05)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 18.04.3 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/openblas/libblas.so.3
## LAPACK: /usr/lib/x86_64-linux-gnu/libopenblasp-r0.2.20.so
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8
##  [2] LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8
##  [4] LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8
##  [6] LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8
##  [8] LC_NAME=C
##  [9] LC_ADDRESS=C
## [10] LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8
## [12] LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets
## [6] methods   base
##
## other attached packages:
## [1] pryr_0.1.4 knitr_1.24 SCF_3.6.1
##
## loaded via a namespace (and not attached):
##  [1] compiler_3.6.1   magrittr_1.5     tools_3.6.1
##  [4] Rcpp_1.0.5       codetools_0.2-16 stringi_1.4.3
##   [7] highr_0.8         stringr_1.4.0    xfun_0.8
##  [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 packages (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

## Error in running command bash
```

2 Packages and namespaces

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

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

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

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

```
library(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/3.6"
## [2] "/system/linux/lib/R-18.04/3.6/x86_64/site-library"
## [3] "/usr/lib/R/site-library"
## [4] "/usr/lib/R/library"

searchpaths()

##  [1] ".GlobalEnv"
##  [2] "/system/linux/lib/R-18.04/3.6/x86_64/site-library/dplyr"
##  [3] "/system/linux/lib/R-18.04/3.6/x86_64/site-library/pryr"
##  [4] "/system/linux/lib/R-18.04/3.6/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-18.04/3.6/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 = 10) # for the stats package
ls(pos = 10)[1:5] # just show the first few

## [1] "library" NA        NA        NA        NA

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

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

3 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.

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. Finally, to test out regular expression syntax see this online tool.

In class we'll discuss various answers to the regex practice below to get started and then we'll work through the string processing tutorial, focusing in particular on the use of regular expressions.

3.1 Regex practice

Write a regular expression that matches the following:

1. Only the strings “cat”, “at”, and “t”.
2. The strings “cat”, “caat”, “caaat”, etc.
3. “dog”, “Dog”, “dOg”, “doG”, “DOg”, etc. (the word dog in any combination of lower and upper case).
4. Any line with exactly two words separated by any amount of whitespace (spaces or tabs). There may or may not be whitespace at the beginning or end of the line.
5. Any positive number with or without a decimal point.

3.2 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?)

3.3 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 2 3 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 work!

```
## 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'
```

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

4 Types, classes, and object-oriented programming

4.1 Types and classes

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

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

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"

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).

```r
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:

```r
x <- rnorm(10 * 365)
attributes(x)

## NULL

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

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

```
qs

##  2.5% 97.5%
## -1.89  1.96

qs[1] + 3

## 2.5%
## 1.11

object.size(qs)

## 352 bytes
```

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

```
names(qs) <- NULL
qs

## [1] -1.89  1.96

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:

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

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

names(mtcars)

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

attributes(mtcars)
```

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

```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: 0x55b22cb57880>
## <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   0.000
```

Here's another example:

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

```
## [1] 0 1

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

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

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

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

## [1] "numeric"

vals <- 1:3
class(vals)

## [1] "integer"

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

## [1] 1 2 3

class(vals)

## [1] "integer"
```

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

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

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

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

## [1] 1.00 2.73

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

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

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

```r
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:

```r
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.

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

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

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

## [1]  1.359 -1.737 -0.387  0.311  0.634

## 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:

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

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

```
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.

```
summary

## function (object, ...)
## UseMethod("summary")
## <bytecode: 0x55b2312f8880>
## <environment: namespace:base>

methods(summary)

##  [1] summary.aov
##  [2] summary.aovlist*
##  [3] summary.aspell*
##  [4] summary.check_packages_in_dir*
##  [5] summary.connection
##  [6] summary.data.frame
##  [7] summary.Date
##  [8] summary.default
##  [9] summary.ecdf*
## [10] summary.factor
## [11] summary.glm
## [12] summary.infl*
## [13] summary.lm
## [14] summary.loess*
## [15] summary.manova
## [16] summary.matrix
## [17] summary.mlm*
## [18] summary.nls*
## [19] summary.packageStatus*
## [20] summary.POSIXct
## [21] summary.POSIXlt
## [22] summary.ppr*
## [23] summary.prcomp*
## [24] summary.princomp*
## [25] summary.proc_time
## [26] summary.rlang_error*
## [27] summary.rlang_trace*
## [28] summary.srcfile
## [29] summary.srcref
## [30] summary.stepfun
## [31] summary.stl*
## [32] summary.table
## [33] summary.tukeysmooth*
## [34] 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 new construct in R. They are classes somewhat similar to S4. Importantly, they behave like pointers (the fields in the objects are ’mutable’). Let’s work through an example where we set up the fields of the class (like S4 slots) and class methods, including a constructor.

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

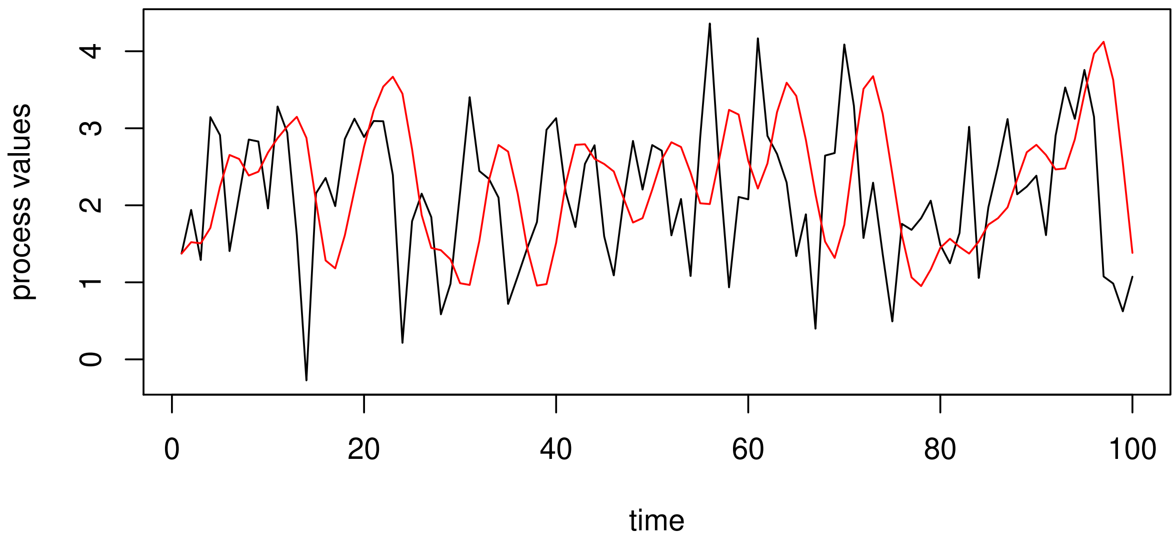
```
library(R6)

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)))
})
```

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

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

## Loading required package:  spam
## Loading required package:  dotCall64
## Loading required package:  grid
## Spam version 2.2-2 (2019-03-07) is loaded.
## Type ’help( Spam)’ or ’demo( spam)’ for a short introduction
## and overview of this package.
## Help for individual functions is also obtained by adding the
## suffix ’.spam’ to the function name, e.g.  ’help( chol.spam)’.
##
## Attaching package:  ’spam’
## The following objects are masked from 'package:base':
##
##     backsolve, forwardsolve
## Loading required package:  maps
## See https://github.com/NCAR/Fields for
##  an extensive vignette, other supplements and source code

## Done updating correlation matrix and Cholesky factor.

master

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

set.seed(1)
devs <- master$simulate()
plot(master$getTimes(), devs, type = 'l', xlab = 'time',
      ylab = 'process values')
master <- tsSimClass$new(1:100, 2, 3)

## Done updating correlation matrix and Cholesky factor.

set.seed(1)
devs <- master$simulate()
lines(master$getTimes(), devs, col = 'red')
```



```
mycopy <- master
myRealCopy <- master$clone()
master$changeTimes(seq(0,1000, length = 100))

## Done updating correlation matrix and Cholesky factor.

mycopy$getTimes()[1:5]

## [1]  0.0 10.1 20.2 30.3 40.4

myRealCopy$getTimes()[1:5]

## [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, which might cause problems:

```
## the next line would be dangerous if 'times' were public, since
## currentU would no longer be accurate
master$times <- 1:10

## Error in master$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 newer 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 6.9).

```
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 Austra~  95.5    NA       NA     1.42
## 2  1961 Austra~  95.3    NA       -0.07  2.79
## 3  1962 Austra~  95.3    23.2      5.71  2.63
## 4  1963 Austra~  95.7    23.0      6.1   2.12
## 5  1964 Austra~  95.7    22.9      6.28  1.15
## 6  1965 Austra~  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 : Column `group_var` is unknown

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

## Error : Column `group_var` is unknown
```

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, variable scope, and frames

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