Unit 5: Programming concepts, illustrated with R

September 16, 2019

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] "badCode.R"   "cache"       "figures"     "goodCode.R"
## [5] "tmp.Rda"
```

- 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"
## [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:

```
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:

```
Sys.info()
```

```
##                                       sysname
##                                       "Linux"
##                                       release
##                           "4.15.0-55-generic"
##                                       version
## "#60-Ubuntu SMP Tue Jul 2 18:22:20 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).

```
## 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

```r
sessionInfo()

## R version 3.6.1 (2019-07-05)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 18.04.2 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
## [3] RhpcBLASctl_0.18-205 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_1.0.2       codetools_0.2-15 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/RhpcBLASctl"
## [11] "/system/linux/lib/R-18.04/3.6/x86_64/site-library/SCF"
## [12] "/usr/lib/R/library/methods"
## [13] "Autoloads"
## [14] "/usr/lib/R/library/base"
```

Installing packages If a package is on CRAN but not on your system, you can install it easily (usually). You don't need root permission on a machine to install a package (though sometimes you run into hassles if you are installing it just as a user, so if you have administrative privileges it may help to use them). 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:RhpcBLASctl"
## [11] "package:SCF"         "package:methods"
## [13] "Autoloads"           "package:base"

## ls(pos = 10) # for the stats package
ls(pos = 10)[1:5] # just show the first few

## [1] "blas_get_num_procs"   "blas_set_num_threads"
## [3] "get_num_cores"        "get_num_procs"
## [5] "omp_get_max_threads"
```

```
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.

This can get particularly confusing in R as the backslash is also used to input special characters such as newline (\n) or tab (\t). As a result in R, we often need two backslashes when working with regular expressions. In the second example, the first backslash says to interpret the next backslash literally, with the second backslash being used to indicate that the bracket should be interpreted literally.

```
## for some reason, output from next few lines not printing out in pdf...
tmp <- "Harry said, \"Hi\""
cat(tmp)
tmp <- "Harry said, \"Hi\".\n"
cat(tmp)
## search for characters that are not 'z'
grep("[^z]", c("a^2", "93", "zit", "azar", "zzz"))
## search for either a '^' or a 'z':
grep("[\\^z]", c("a^2", "93", "zit", "azar", "zzz"))
## 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
""[\^"
```

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:

```r
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.

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.

4 Types, classes, and object-oriented programming

4.1 Types and classes

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

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

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

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

## [1] "numeric"

typeof(devs)

## [1] "double"
```

```r
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.

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

##   2.5% 97.5%
## -2.03  1.86

qs[1] + 3

##   2.5%
## 0.972
```

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

```
names(qs) <- NULL
qs

## [1] -2.03  1.86
```

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

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

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

names(mtcars)

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

attributes(mtcars)

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

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

```
## $names
## [1] "x" "y"
##
## $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: 0x5654bc7fe170>
## <environment: namespace:base>

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

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

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

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

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

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

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

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

##    user  system elapsed
##   0.001   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). We saw see implicit conversion (also called coercion) when we read in characters into R using *read.table()* - strings are often automatically coerced to factors. Consider these examples of implicit coercion:

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

## [1] 1 2
```

Be careful of using factors as indices:

```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]  1.011  1.319 -2.572 -0.538  0.873

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