Programming concepts

Chris Paciorek

2022-09-06

Table of contents

PDF

Note This Unit will be posted in phases as I get the material ready. 2022-09-15: Currently only Sections 1-7 are available.

References:

- Books on R listed on the syllabus: Adler, Chambers, Wickham

- R intro manual and R language manual (R-lang), both on CRAN.
- Murrell, Introduction to Data Technologies

(Optional) Videos

There are various videos from 2020 in the bCourses Media Gallery that you can use for reference if you want to. Note that I've reorganized the material in this Unit relative to 2020, so the section numbers and ordering in the videos may differ from that in the current Unit, but you should be able to match things up fairly easily.

1. Strings and regular expressions
2. Type/class coercion
3. Object-oriented programming - S3 classes
4. Object-oriented programming - R6 classes
5. Nested function calls and the call stack
6. Operators in R
7. Unexpected functions and replacement functions

Overview

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

I'll likely refer to R syntax as *statements* or *expressions*, meaning any code that is a valid, complete R expression. But note that the term *expression* also means a specific type of object within the R language, as seen late in this Unit when we discuss computing on the language.

1. 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 an R character vector.

String processing and regular expressions in R

For details of string processing in R, including use of regular expressions, see the string processing tutorial. (You can ignore the sections on Python if you wish.) That tutorial then refers to the bash shell tutorial for details on regular expressions.

In class we'll work through some problems in the string processing tutorial, focusing in particular on the use of regular expressions with the *stringr* package. This will augment our consideration of regular expressions in the shell, in particular by seeing how we can replace patterns in addition to finding them.

Regex/string processing challenges

We'll work on these challenges (and perhaps one or two others) in class in the process of working through the string processing tutorial.

1. What regex would I use to find any number with or without a decimal place.
2. 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?)

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`).

Here are some examples of using special characters.

> **Note** It is hard to compile the Rmd file correctly for these R chunks, so I am just pasting in the output from running in R 'manually' in some cases.)

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

```
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)   ## search for a tab or a 'z'
```

```
[1] 1 3
```

As a result, in R we often need two backslashes when working with regular expressions. In the next 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 in the regular expression syntax.

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

```
[1] 1 2 4 5
```

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

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

```
## This fails (and the Rmd won't compile) because
## '\^' is not an escape sequence (i.e., a special character):
## 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", "def"))
```

```
[1] 1 3
```

```
## Search for a period (as a regular character)
grep("\\.", c("3.9", "27", "4.2"))
```

```
[1] 1 3
```

```
## This fails (and the Rmd won't compile) because
## '\.' is not an escape sequence (i.e., a special character):
## 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:

```
## Suppose 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
```

```
  try(grep("\\", tmp))
```

```
Warning in grep("\\", tmp): TRE pattern compilation error 'Trailing backslash'
```

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

> **Warning** 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
```

2. Interacting with the operating system and external code and configuring R

Interacting with the operating system

Scripting languages allow one to interact with the operating system in various ways. Most allow you to call out to the shell to run arbitrary shell code and save results within your session.

I'll assume everyone knows about the following functions/functionality for interacting with the filesystem and file 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")   ## results apparently not shown when compiled...
files <- system("ls", intern = TRUE)
files[1:5]
```

```
[1] "0-bash-shell.sh" "badCode.R"       "cache"           "calc_mean.py"
[5] "convert.sh"
```

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

```
file.exists("unit2-dataTech.Rmd")
```

```
[1] TRUE
```

```
list.files("../data")
```

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

- There are some tools for dealing with differences between operating systems. *file.path* is a nice example:

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

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

  It's best if you can to write your code in a way that is *agnostic* to the underlying operating system.

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

```
Sys.info()
```

```
                                    sysname
                                    "Linux"
                                    release
                            "5.4.0-120-generic"
                                    version
"#136-Ubuntu SMP Fri Jun 10 13:40:48 UTC 2022"
                                   nodename
                                  "smeagol"
                                    machine
                                   "x86_64"
                                      login
                                 "paciorek"
                                       user
                                 "paciorek"
                              effective_user
                                 "paciorek"
```

Controlling the behavior of R

Scripting languages generally allow you to control/customize their behavior in various ways by setting options.

- 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 *max.print* revents too much of a large object from being printed to the screen.

```
## options()  # this would print out a long list of options
options()[1:4]
```

```
$add.smooth
[1] TRUE

$bitmapType
[1] "cairo"

$browser
[1] "xdg-open"

$browserNLdisabled
[1] FALSE
```

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

```
$width
[1] 80
```

```
$digits
[1] 7
```

```
## Often it's nice to have more characters in each line on the screen,
## but that would cause overly lines in the compiled file.
## options(width = 120)

options(max.print = 5000)
```

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(digits = 3)
a <- 0.123456; b <- 0.1234561
a; b; a == b
```

```
[1] 0.123
```

```
[1] 0.123
```

```
[1] FALSE
```

More on how to (and how not to) compare real-valued numbers on a computer in Unit 8.

- 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 the amount of memory available, 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.
- *sessionInfo* gives information on the current R session and can be very helpful for recording the state of your session (including package versions) to allow for reproducibility.

```
sessionInfo()
```

```
R version 4.2.0 (2022-04-22)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 20.04.3 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       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
```

```
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] SCF_4.1.0

loaded via a namespace (and not attached):
 [1] compiler_4.2.0  magrittr_2.0.3  fastmap_1.1.0   cli_3.3.0
 [5] tools_4.2.0     htmltools_0.5.3 yaml_2.3.5      stringi_1.7.8
 [9] rmarkdown_2.14  knitr_1.39      stringr_1.4.0   xfun_0.31
[13] digest_0.6.29   jsonlite_1.8.0  rlang_1.0.4     evaluate_0.15
```

- Any code that you wanted executed automatically when starting R can be placed in `~/.Rprofile` (or in individual, project-specific `.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")
```

  Now we can run it as follows in the shell:

```
./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 .
```

Interacting with external code

Scripting languages such as R, Python, and Julia allow you to call out to "external code", which often means C or C++ (but also Fortran, Java and other languages).

In fact, the predecessor language to R, which was called 'S' was developed specifically (at AT&T's Bell Labs in the 1970s and 1980s) as an interactive wrapper around Fortran, the numerical programming language most commonly used at the time (and still widely relied on today in various legacy codes).

Calling out to external code is particularly important in languages like R and Python that are often much slower than compiled code and less important in a fast language like Julia (which uses Just-In-Time compilation – more on that later).

In R, one can call directly out to C or C++ code using *.Call* or one can use the Rcpp package. *Rcpp* is specifically designed to be able to write C++ code that feels somewhat like writing R code and where it is very easy to pass data between R and C++.

In Python, one can directly call out to C or C++ code or one can use *Cython* to interact with C. With Cython, one can: - Have Cython automatically translate Python code to C, if you provide type definitions for your variables. - Define C functions that can be called from your Python code.

3. Packages and namespaces

Scripting languages that become popular generally have an extensive collection of add-on packages available online (the causal relationship of the popularity and the extensive add-on packages goes in both directions). Packages need to be *installed* (once) on your computer and *loaded* (every time you start a new session).

A big part of R's popularity is indeed the extensive collection of add-on packages on CRAN (and GitHub and elsewhere) 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*.

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)              # load the package
library(help = dplyr)       # get some help info about the package
```

Packages in R (and in Python, Julia, etc.) may be installed in various places on the filesystem, and it sometimes it is helpful (e.g., if you end up with multiple versions of a package installed on your system) to be able to figure out where on the filesystem the package is being loaded from. If you run `library()`, you'll notice that some of the packages are in a system directory and some are in your home directory.

`.libPaths()` shows where R looks for packages on your system and `searchpaths()` shows where individual packages currently loaded in your session have been loaded from. The help information for *.libPaths* gives some information about how R decides what locations to look in for packages (and how you can modify that).

```
.libPaths()
```

```
[1] "/accounts/vis/paciorek/R/x86_64-pc-linux-gnu-library/4.2"
[2] "/system/linux/lib/R-20.04/4.2.0/x86_64/site-library"
[3] "/usr/lib/R/site-library"
[4] "/usr/lib/R/library"
```

```
searchpaths()
```

```
 [1] ".GlobalEnv"
 [2] "tools:quarto"
 [3] "/usr/lib/R/library/stats"
 [4] "/usr/lib/R/library/graphics"
 [5] "/usr/lib/R/library/grDevices"
 [6] "/usr/lib/R/library/utils"
 [7] "/usr/lib/R/library/datasets"
 [8] "/system/linux/lib/R-20.04/4.2.0/x86_64/site-library/SCF"
 [9] "/usr/lib/R/library/methods"
[10] "Autoloads"
[11] "/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.

Packages often depend on other packages. In general, if one package depends on another, R will install the dependency automatically, but sometimes you'll need to install a dependency yourself. In general, package dependencies are handled very cleanly in R without you having having to worry much about it; this is less the case in Python.

Note that R will generally install the package in a reasonable place by default but you can control where it is installed using the *lib* argument.

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

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

This can be handy if you need to install an older version of a package for reproducibility or because of some dependency incompatibility.

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.

Source vs. binary packages

The difference between a *source* package and a *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 that any C and Fortran code will have already 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.

Managing packages using package managers

For reproducibility, it's important to know the versions of the packages you use (and the version of R). Package managers make it easy to do this. Some useful packages that do package management in R are *checkpoint*, *renv*, and *packrat*. The basic commonality is that they try to make it easy to 'freeze' the versions of hte packages you are using, record that information, and restore the versions (potentially on some other machine and by some user other than yourself). The package manager may tell you where the packages are installed, but you can always verify things with `.libPaths()`.

In Python, you can set up and manage isolated environments in which you can control the package versions using virtualenvs or Conda environments.

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. If we want to see the objects in a package's namespace, we can do the following:

```
search()
```

```
 [1] ".GlobalEnv"        "tools:quarto"      "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:SCF"       "package:methods"
[10] "Autoloads"         "package:base"
```

```
## ls(pos = 4) # for the stats package
ls(pos = 4)[1:5] # just show the first few
```

```
[1] "abline"    "arrows"    "assocplot" "axis"      "Axis"
```

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

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

```
ls("package:stats", pattern = "^lm")
```

```
[1] "lm"           "lm.fit"       "lm.influence" "lm.wfit"
```

Why have namespaces?

We'll talk more about namespaces when we talk about variable scope and environments. But as some motivation for why this is useful, consider the following.

The *lm* function calls the *lm.fit* function to calculate the least squares solution in regression.

Suppose we write our own *lm.fit* function that does something else:

```
lm.fit <- function(x)
    print('hi')

x <- 7
lm.fit(x)
```

```
[1] "hi"
```

One might expect that if one now uses `lm()` to fit a regression, that it wouldn't work correctly because we have an *lm.fit* function in our workspace that doesn't calculate the least squares solution. But it works just fine (see below), because *lm* and *lm.fit* are in the *stats* package namespace (see above) and R's scoping rules (more later) ensure that the *lm.fit* that is found when I run *lm* is the *lm.fit* needed to run the regression and not my silly *lm.fit* function in current workspace.

```r
n <- 10
x <- runif(n)
y <- runif(n)
mod <- lm(y ~ x)
mod
```

```
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
      0.720       -0.618
```

Namespace resolution

Standard practice in R has generally been to load a package and then use any of the items in the package namespace directly, e.g.,

```r
library(stringr)
str_detect("hello there", "hello")
```

```
[1] TRUE
```

However, particularly if you're using the package in only a limited way, it can be a nice idea to not load the entire package and instead use the namespace resolution operator in a style that might remind you of Python and some other languages:

```r
stringr::str_detect("hello there", "hello")
```

```
[1] TRUE
```

```python
import numpy as np
x = np.ndarray([0,3,5])
```

Of course in Python you could also load the entire package (i.e., import the entire namespace), though it's not standard practice:

```python
from numpy import *
## OR: from numpy import ndarray
x = ndarray([0,3,5])
```

Loading entire packages often causes 'name collisions' where there are multiple functions (or variables, more genreally) that have the same name. This can be confusing. We'll see how R determines what function to use later in the Unit.

4. Types and data structures

Data structures

Please see the data structures section of Unit 2 for some general discussion of data structures.

We'll also see more complicated data structures when we consider objects in the next section on object-oriented programming.

Types and classes

Overview and static vs. dynamic typing

The term 'type' refers to how a given piece of information is stored and what operations can be done with the information. 'Primitive' types are the most basic types that often relate directly to how data are stored in memory or on disk (e.g., boolean, integer, numeric (real-valued, aka *double* or *floating point*), character, pointer (aka *address*, *reference*).

In compiled languages like C and C++, one has to define the type of each variable. Such languages are *statically* typed. Interpreted (or scripting) languages such as Python and R have *dynamic* types. One can associate different types of information with a given variable name at different times and without declaring the type of the variable:

```
x <- 'hello'
print(x)
```

```
[1] "hello"
```

```
x <- 7
x*3
```

```
[1] 21
```

In contrast in a language like C, one has to declare a variable based on its type before using it:

```
double y;
double x = 3.1;
y = x * 7.1;
```

Dynamic typing can be quite helpful from the perspective of quick implementation and avoiding tedious type definitions and problems from minor inconsistencies between types (e.g., multiplying an integer by a real-valued number). But static typing has some critical advantages from the perspective of software development, including:

- protecting against errors from mismatched values and unexpected user inputs, and
- generally much faster execution because the type of a variable does not need to be checked when the code is run.

More complex types in R (and in Python) often use references (*pointers*, aka *addresses*) to the actual locations of the data. We'll see this in detail later in the Unit.

Types and classes in R

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* (i.e., *numeric*), and *character*.

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. As we'll see later in the Unit, 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.

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

Let's look at the type and class of various data structures in R. We'll first see that real-valued are stored as double-precision (8 byte) floating point numbers internally in R (as 'doubles' in C, as the R interpreter is a program written in C).

```
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"
```

In most cases integer-valued numbers are stored as numeric values in R, but there are exceptions such as the result of using the sequence operater, `:`, above. 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"
```

Attributes

We saw the notion of attributes when looking at HTML and XML, where the information was stored as key-value pairs that in many cases had additional information in the form of attributes.

In R, *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 when in subsequent operations the *names* attribute is carried along:

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

```
NULL
```

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

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

```r
qs
```

```
 2.5% 97.5%
-1.91  1.98
```

```r
qs[1] + 3
```

```
2.5%
1.09
```

```r
object.size(qs)
```

```
352 bytes
```

We can get rid of the attribute:

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

```
[1] -1.91  1.98
```

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

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

$class
[1] "data.frame"

$row.names
[1] 1 2
```

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

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

```r
attributes(df)
```

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

$class
[1] "data.frame"

$row.names
[1] "first"  "second"
```

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

```
first
    7
```

```r
attributes(vec)
```

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

Converting between types

This also goes by the term *coercion* and *casting*. Casting often needs to be done explicitly in compiled languages and somewhat less so in interpreted languages like R.

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

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

```
[1] "1" "2" "3"
```

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

```
[1] 1.00 2.73
```

```r
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
```

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(label = rep('a', n), val1 = rnorm(n), val2 = rnorm(n))
df
```

```
  label   val1   val2
1     a  0.832  0.789
2     a -0.334  0.273
3     a -0.557 -1.407
4     a -0.336  0.102
5     a  0.646  1.809
```

```
## Why does the following not work?
try( apply(df, 1, function(x) x[2] + x[3]) )
```

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

```
## Instead, this will work. Why?
apply(df[ , 2:3], 1, function(x) x[1] + x[2])
```

```
[1]  1.6205 -0.0612 -1.9647 -0.2342  2.4550
```

Be careful of using factors as indices:

```
students <- factor(c("basic", "proficient", "advanced",
                     "basic", "advanced", "minimal"))
score <- c(minimal = 65, basic = 75, proficient = 85, advanced = 95)
```

```
  score["advanced"]
```

```
advanced
      95
```

```
  students[3]
```

```
[1] advanced
Levels: advanced basic minimal proficient
```

```
  score[students[3]]
```

```
minimal
     65
```

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

```
advanced
      95
```

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

Data frames and related concepts

Some notes on data frames and operations on data frames

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 way. Module 6 of the R bootcamp describes some of these new tools in more details, but I'll touch on some aspects of this here, without showing much of the tidyverse syntax.

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 *lapply* family of functions such as *by*, *tapply*, and *aggregate*, but the dplyr-based tools are often nicer to use.

split-apply-combine is also closely related to the famous Map-Reduce framework underlying big data tools such as Hadoop and Spark.

It's also very similar to standard SQL queries involving filtering, grouping, and aggregation.

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.

```
long <- data.frame(id = c(1, 1, 1, 2, 2, 2),
                   time = c(1980, 1990, 2000, 1980, 1990, 2000),
                   value = c(5, 8, 9, 7, 4, 7))
wide <- data.frame(id = c(1, 2),
                   value_1980 = c(5, 7), value_1990 = c(8, 4), value_2000 = c(9, 7))
long
```

```
  id time value
1  1 1980     5
2  1 1990     8
3  1 2000     9
4  2 1980     7
5  2 1990     4
6  2 2000     7
```

```
wide
```

```
  id value_1980 value_1990 value_2000
1  1          5          8          9
2  2          7          4          7
```

The wide format can be useful in some situations for treating each row as a (multivariate observation), but the long formatwhile the long format is often what is needed for analyses such as mixed models. ANOVA, or for plotting, such as with *ggplot2*.

There are a variety of functions for converting between wide and long formats. I recommend *pivot_longer* and *pivot_wider* in 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.

Piping

Piping was introduced into R in conjuction with *dplyr* and the *tidyverse*.

The tidyverse pipe is `%>%` while the new base R pipe is `|>`. These are based on the UNIX pipe, which we saw in Unit 3, though they behave somewhat differently in that the output of the previous function is passed in as the *first* argument of the next function. In the shell, the pipe connects *stdout* from the previous command to *stdin* for the next command.

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 using scoping (discussed later in the Unit).

```
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
```

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

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 mean_unemp
  <int> <chr>     <dbl>   <dbl>     <dbl> <dbl>      <dbl>
1  1960 Australia  95.5    NA       NA     1.42       5.52
2  1961 Australia  95.3    NA       -0.07  2.79       5.52
3  1962 Australia  95.3    23.2      5.71  2.63       5.52
4  1963 Australia  95.7    23.0      6.1   2.12       5.52
5  1964 Australia  95.7    22.9      6.28  1.15       5.52
6  1965 Australia  95.7    24.9      4.97  1.15       5.52
```

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 later in the Unit), 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 in group_by(., group_var) :
  Must group by variables found in `.data`.
x Column `group_var` is not found.
```

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

```
Error in group_by(., group_var) :
  Must group by variables found in `.data`.
x Column `group_var` is not found.
```

For more details on how to avoid this problem when writing functions that involve tidyverse manipulations, see this tidyverse programming guide.

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)
```

Challenge Where is the non-standard evaluation there?

5. Programming paradigms: object-oriented and functional programming

Object-oriented and functional programming are two important approaches to programming.

Functional programming focuses on writing functions that take inputs and produce outputs. Ideally those functions don't change the state (i.e., the values) of any variables and can be treated as black boxes. Functions can be treated like other variables, such as passing functions as arguments (as one does with *lapply* in R, for example).

Object-oriented programming revolves around objects that belong to classes. The class of an object defines the fields (the data objects) holding information and (often) methods that can be applied to those fields. When one calls a method, it may modify the value of the fields. A statistical analogy is that an object of a class is like the realization (the object) of a random variable (the class).

One can think of functional programming as being focused on actions (or *verbs* to make an analogy with human language). One carries out a computation as a sequence of function calls. One can think of OOP as being focused on the objects (or *nouns*). One carries out a computation as a sequence of operations with the objects, using the class methods.

Many languages are multi-paradigm, containing aspects of both approaches and allowing programmers to use either approach. Both R and Python are like this, though some might consider R to be more functional and Python to be more object-oriented. That said, in R everything is an object and has a class, while there are plenty of function-based operations in Python.

```
import numpy as np
x = np.array([1.2, 3.5, 4.2])
x.shape  # field (or attribute) of the numpy array class
x.sum()  # method of the class
len(x)   # function
```

Different people have different preferences, but which is better depends on what you are trying to do. If your computation is a data analysis pipeline that involves a series of transformations of some data, a functional approach might make more sense, since the focus is on a series of actions rather than the state of objects. If your computation involves various operations on fixed objects whose state needs to change, OOP might make more sense. For example, if you were writing code to keep track of student information, it would probably make sense to have each student as an object of a Student class with methods such as 'register' and 'assign_grade'.

6. Object-oriented programming (OOP)

Principles

Some of the standard concepts in object-oriented programming include *encapsulation*, *inheritance*, *polymorphism*, and *abstraction*.

Encapsulation involves preventing direct access to internal data in an object from outside the object. Instead the class is designed so that access (reading or writing) happens through the interface set up by the programmer (e.g., 'getter' and 'setter' methods). We'll see this in our R6 class example below.

Inheritance allows one class to be based on another class, adding more specialized features. An example in R's S3 system is that the *glm* class inherits from the *lm* class.

Polymorphism allows for different behavior of an object or function depending on the context. A polymorphic function behaves differently depending on the input types. A polymorphic object is one that can belong to different classes (e.g., based on inheritance), and a given method name can be used with any of the classes. An example would be having a base or super class called 'algorithm' and various specific machine learning algorithms inheriting from that class. All of the classes might have a 'predict' method.

Abstraction involves hiding the details of how something is done (e.g., via the method of a class), giving the user an interface to provide inputs and get outputs. By making the actual computation a black box, the programmer can modify the internals without changing how a user uses the system.

Classes generally have *constructors* that initialize objects of the class and *destructors* that remove objects.

Generic function OOP

Much of the object-oriented programming in R uses *generic function OOP*, also known as *functional OOP*. In this style, classes don't have methods. Instead there are *generic functions* (also known as *generic methods*) that change their behavior based on the type of the input(s). Another way to put it is that the nouns and the verbs are separate, unliked in standard OOP.

The use of generic functions is similar in spirit to function or method *overloading* in C++ and Java.

Generic function OOP is how the (very) old S3 system in R works. It's also a key part of the (fairly) new Julia language.

S3 classes in R

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.

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(smoke) <- "celebrity"
```

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

Generic methods

The real power of the S3 system comes from defining *class-specific methods*. For example,

```
x <- rnorm(10)
summary(x)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -1.053  -0.595   0.107   0.011   0.562   1.035
```

```
y <- rnorm(10)
mod <- lm(y ~ x)
summary(mod)
```

```
Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-1.4509 -0.3021  0.0994  0.6331  1.2968

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.578      0.306    1.89    0.095 .
x              0.315      0.425    0.74    0.480
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.966 on 8 degrees of freedom
Multiple R-squared:  0.0642,    Adjusted R-squared:  -0.0528
F-statistic: 0.548 on 1 and 8 DF,  p-value: 0.48
```

Here *summary* is a generic function (or generic method) that, based on the type of object given to it (the first argument), dispatches a class-specific function (method) that operates on the object.

The above is equivalent to directly calling the class-specific methods:

```
identical(summary(x), summary.default(x))
```

```
[1] TRUE
```

```
identical(summary(mod), summary.lm(mod))
```

```
[1] TRUE
```

This use of generic functions is convenient in that it allows us to work with a variety of kinds of 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.

```
summary
```

```
function (object, ...)
UseMethod("summary")
<bytecode: 0x5610dcfabd18>
<environment: namespace:base>
```

The `UseMethod` syntax is what causes the dispatching of the class-specific method associated with `object` and calls that method. 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 also see what classes have methods for a given generic function.

```
methods(summary)
```

```
 [1] summary,ANY-method                    summary,DBIObject-method
 [3] summary,diagonalMatrix-method         summary,sparseMatrix-method
 [5] summary.aov                           summary.aovlist*
 [7] summary.aspell*                       summary.check_packages_in_dir*
 [9] summary.connection                    summary.data.frame
[11] summary.Date                          summary.default
[13] summary.ecdf*                         summary.factor
[15] summary.glm                           summary.infl*
[17] summary.lm                            summary.loess*
[19] summary.manova                        summary.matrix
[21] summary.mlm*                          summary.nls*
[23] summary.packageStatus*                summary.pandas.core.frame.DataFrame*
[25] summary.pandas.core.series.Series*    summary.POSIXct
[27] summary.POSIXlt                       summary.ppr*
[29] summary.prcomp*                       summary.princomp*
[31] summary.proc_time                     summary.python.builtin.object*
[33] summary.rlang_error*                  summary.rlang_message*
[35] summary.rlang_trace*                  summary.rlang_warning*
[37] summary.rlang:::list_of_conditions*   summary.shingle*
[39] summary.srcfile                       summary.srcref
[41] summary.stepfun                       summary.stl*
[43] summary.table                         summary.trellis*
[45] summary.tukeysmooth*                  summary.vctrs_sclr*
[47] summary.vctrs_vctr*                   summary.warnings
see '?methods' for accessing help and source code
```

Or from a different angle we can see what specific methods are available for a given class.

```
methods(class = 'lm')
```

```
 [1] add1           alias          anova          case.names     coerce
 [6] confint        cooks.distance deviance       dfbeta         dfbetas
[11] drop1          dummy.coef     effects        extractAIC     family
```

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

Let's try this functionality out on our *bear* class.

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

summary(yog)
```

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

We can also define a new generic function.

Let's do this for the *bear* class as an illustration, though this won't provide any functionality beyond what we did with *summary*

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

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

summarize(yog)
```

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

Why use generic functions?

We could have written *summary* as a regular function with a bunch of if statements or if-else clauses (or *switch*) so that it can handle different kinds of input objects.

This has two disadvantages:

1. We need to write the code that does the checking (and all the code for the different cases all lives inside one potentially very long function, unless we create class-specific helper functions).
2. Much more importantly, *summary* will only work for existing classes. And users can't easily extend it for new classes that they create because they don't control the *summary* function. So

a user could not add the additional conditions/classes in a big if-else statement. The generic function approach makes the system *extensible* – we can build our own new functionality on what is already in R. For example, we could have written *summary.bear*.

The print method

Like *summary*, *print* is a generic method, with various class-specific methods, such as *print.lm*. We could write our own *print.bear* specific method.

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. Thus, 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.

```
mod
```

```
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
      0.578        0.315
```

```
print(mod)
```

```
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
      0.578        0.315
```

```
stats:::print.lm(mod)  ## print.lm is private to the stats namespace
```

```
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
      0.578        0.315
```

```
# print.default(mod)   ## lots of output, so don't print in document...
```

```
stats:::print.lm
```

```
function (x, digits = max(3L, getOption("digits") - 3L), ...)
{
```

```
    cat("\nCall:\n", paste(deparse(x$call), sep = "\n", collapse = "\n"),
        "\n\n", sep = "")
    if (length(coef(x))) {
        cat("Coefficients:\n")
        print.default(format(coef(x), digits = digits), print.gap = 2L,
            quote = FALSE)
    }
    else cat("No coefficients\n")
    cat("\n")
    invisible(x)
}
<bytecode: 0x5610d9197098>
<environment: namespace:stats>
```

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 (e.g., using *print.summary.lm*), unless one assigns it to a new object. Note that *print.summary.lm* is hidden from user view (it's a private object in the *stats* namespace).

```
out <- summary(mod)
class(out)
```

```
[1] "summary.lm"
```

```
out
```

```
Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-1.4509 -0.3021  0.0994  0.6331  1.2968

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.578      0.306    1.89    0.095 .
x              0.315      0.425    0.74    0.480
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.966 on 8 degrees of freedom
Multiple R-squared:  0.0642,    Adjusted R-squared:  -0.0528
F-statistic: 0.548 on 1 and 8 DF,  p-value: 0.48
```

```
print(out)
```

```
Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-1.4509 -0.3021  0.0994  0.6331  1.2968

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.578      0.306    1.89    0.095 .
x              0.315      0.425    0.74    0.480
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.966 on 8 degrees of freedom
Multiple R-squared:  0.0642,    Adjusted R-squared:  -0.0528
F-statistic: 0.548 on 1 and 8 DF,  p-value: 0.48
```

```
## One can look at the code for the method (not shown):
## getS3method(f = "print", class = "summary.lm")
```

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*.

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

```
[1] "lm"
```

```
class(mod2)
```

```
[1] "glm" "lm"
```

```
is.list(mod1)
```

```
[1] TRUE
```

```
names(mod1)
```

```
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
```

```
is(mod2, "lm")
```

```
[1] TRUE
```

Here's an example of why this is useful. We don't have to define methods for the *glm* class if the given method for the *lm* class would work fine:

```
model.matrix(mod1)
```

```
   (Intercept)       x
1            1  0.3534
2            1 -0.4556
3            1  0.4121
4            1  0.1413
5            1  1.4450
6            1  0.1064
7            1  0.2701
8            1 -0.0327
9            1  0.6669
10           1 -0.2898
attr(,"assign")
[1] 0 1
```

```
model.matrix(mod2)
```

```
   (Intercept)       x
1            1  0.3534
2            1 -0.4556
3            1  0.4121
4            1  0.1413
5            1  1.4450
6            1  0.1064
7            1  0.2701
8            1 -0.0327
9            1  0.6669
10           1 -0.2898
attr(,"assign")
[1] 0 1
```

```
methods(model.matrix)
```

```
[1] model.matrix.default model.matrix.lm
```

```
see '?methods' for accessing help and source code
```

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 fields on top of those of a *bear*, perhaps *number_of_people_killed* (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).

The above is an example of polymorphism. `yog` is a polymorphic object and the various methods are polymorphic in that *print* can be used with the *bear* class, the *grizzly_bear* class, and other classes beyond that.

> **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()`? (Hint: you can do this by understanding what happens when you type `k` and how to exploit the S3 system.)

Multiple dispatch OOP

S3 method dispatch involves only the first argument to the function. In contrast, Julia emphasizes the importance of multiple dispatch as particularly important for mathematical computation. With multiple dispatch, the specific method can be chosen based on more than one argument.

The old (but still used in some contexts) S4 system in R and the (very) new R7 system both provide for multiple dispatch.

As a very simple example unrelated to any specific language, multiple dispatch would allow one to do the following with the addition operator:

```
3 + 7    # 10
3 + 'a'  # '3a'
'hi' +  ' there'  # 'hi there'
```

The idea of having the behavior of an operator or function adapt to the type of the input(s) is one aspect of *polymorphism*.

Both S4 and R7 are designed to be more formal than the S3 system (recall how we could just 'create' an S3 class by giving a class name to an existing list). With S4 and R7, you need to define your classes.

'Standard' OOP

What I'm calling 'standard' object-oriented programming is the style of OOP used in languages such as Python, C++, and Java. In R, one can use this style via the R6 system (or the older *referenceClass* system).

In this style, objects belong to classes. A class is made up of fields (the data objects) that store information and methods that operate on the fields. Thus, unlike generic function OOP, the verbs are part of the nouns.

We'll illustrate this style of OOP using an example with an R6 class.

R6 classes

R6 classes are a somewhat new construct in R, with a class-based approach fairly similar to Python and C++. Importantly, they behave like pointers. We'll discuss pointers in detail later. Let's work through an example where we set up the fields of the class and class methods, including a constructor.

Example

Our example is to create a class for working with random time series. Each object of the class has specific parameter values that control the stochastic behavior of the time series. With a given object we can simulate one or more time series (realizations).

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()
        },

        setTimes = 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)
        },

        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)))
        }
    ),

    ## 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
    )
)
```

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

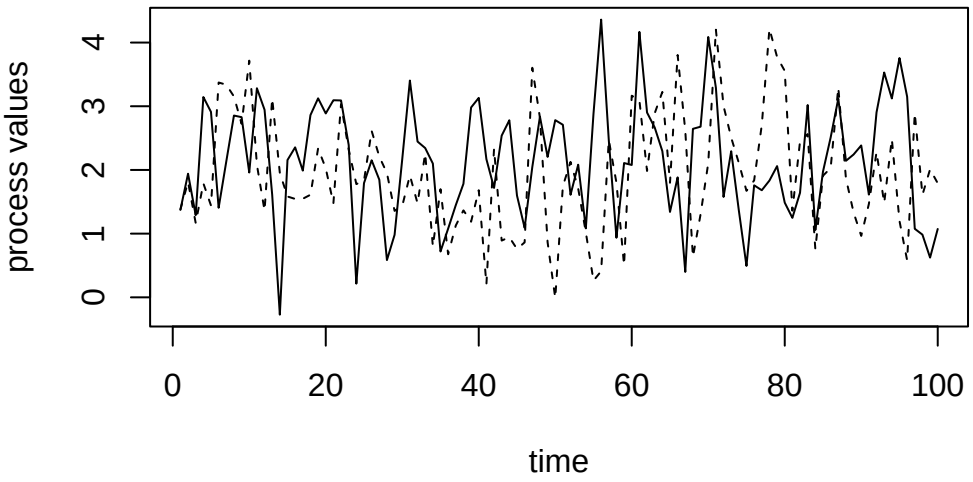
```
myts <- tsSimClass$new(1:100, 2, 1)
```

```
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
y1 <- myts$simulate()
plot(myts$getTimes(), y1, type = 'l', xlab = 'time',
     ylab = 'process values')
## simulate a second series
y2 <- myts$simulate()
lines(myts$getTimes(), y2, lty = 2)
```



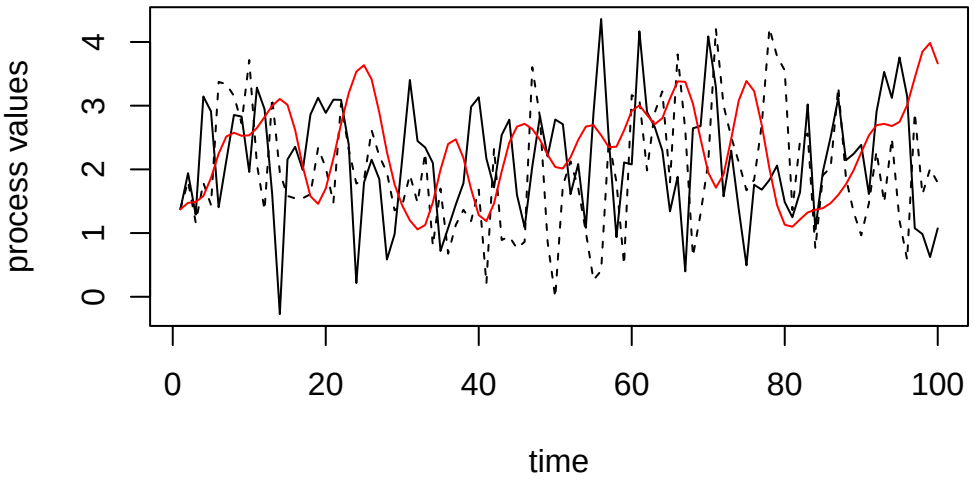
We could set up a different object that has different parameter values. That new simulated time series is less wiggly because the `corParam` value is larger than before.

```
myts2 <- tsSimClass$new(1:100, 2, 4)
```

```
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)
y3 <- myts2$simulate()

plot(myts$getTimes(), y1, type = 'l', xlab = 'time',
     ylab = 'process values')
lines(myts$getTimes(), y2, lty = 2)
lines(myts2$getTimes(), y3, col = 'red')
```



Copies and references

Next let's think about when copies are made. In the next example `mytsRef` is a copy of `myts` in the sense that both names point to the same underlying object. But no data were copied when the assignment to `mytsRef` was done.

```
mytsRef <- myts
## 'mytsRef' and 'myts' are names for the same underlying object
mytsFullCopy <- myts$clone()

## Now let's change the values of a field
myts$setTimes(seq(0,1000, length = 100))
```

```
Done updating correlation matrix and Cholesky factor.
```

```
myts$getTimes()[1:5]
```

```
[1]  0.0 10.1 20.2 30.3 40.4
```

```
mytsRef$getTimes()[1:5] # the same as `myts`
```

```
[1]  0.0 10.1 20.2 30.3 40.4
```

```
mytsFullCopy$getTimes()[1:5] # different from `myts`
```

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

In contrast `mytsFullCopy` is a reference to a different object, and all the data from `myts` had to be copied over to `mytsFullCopy`. This takes additional memory (and time), but is also safer, as it avoids the possibility that the user might modify `myts` and not realize that they were also affecting `mytsRef`.

Encapsulation

Why have private fields (i.e., encapsulation)? The use of private fields shields them from modification by users. In this case, that prevent users from modifying the *times* field. Why is this important? 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*. If they did, it would leave the fields of the object in inconsistent states. Instead we force them to use *setTimes*, which correctly keeps all the fields in the object internally consistent (by calling *calcMats*). It also allows us to improve efficiency by controlling when computationally expensive operations are carried out.

```
try(myts$times <- 1:10)
```

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

Final comments

- As we saw above, a copy of an object is just a pointer to the original object, unless we explicitly invoke the *clone* method.
- 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, ...)
  ```

- If you need to refer to methods and fields you refer to the entire object as either *self* or *private*.

More details on R6 classes can be found in the Advanced R book.

7. Functional programming

Overview of functional programming

Functional programming is an approach to programming that emphasizes the use of modular, self-contained functions. Such functions should operate only on arguments provided to them (avoiding global variables), and **produce no side effects**, although in some cases there are good reasons for making an exception. Another aspect of functional programming is that functions are considered 'first-class' citizens in that they can be passed as arguments to another function, returned as the result of a function, and assigned to variables. In other words, a function can be treated as any other variable.

In many cases (including R and Python), anonymous functions (also called 'lambda functions') can be created on-the-fly for use in various circumstances.

Functional programming in R

R is a language that has strong functional programming aspects to it, including:

- All operations are carried out by functions.
- Functions are first class citizens.
- Functions (generally) do not have side effects.
- *Map* operations (e.g., *lapply*) are central to programming in R.

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

```
typeof(mean)
```

```
[1] "closure"
```

```
typeof(lm)
```

```
[1] "closure"
```

```
typeof(length)
```

```
[1] "builtin"
```

No side effects

Most functions available in R (and ideally functions that you write as well) operate by taking in arguments and producing output that is then (presumably) used subsequently. The functions generally don't have any effect on the state of your R environment/session other than the output they produce.

An important reason for this (plus for not using global variables) is that it means that it is easy for people using the language to understand what code does. Every function can be treated a black box – you don't need to understand what happens in the function or worry that the function might do something unexpected (such as changing the value of one of your variables). The result of running code is simply the result of a composition of functions, as in mathematical function composition.

One aspect of this is that R uses a *pass-by-value* approach to function arguments (as opposed to a *pass-by-reference* approach). We'll talk about function arguments and when copies are made in much more detail later, but briefly, when you pass an object in as an argument and then modify it in the function, you are modifying a local copy of the variable that exists in the context (the *frame*) of the function and is deleted when the function call finishes:

```
x <- 1:3
myfun <- function(x) {
      x[2] <- 7
      print(x)
      return(x)
}

new_x <- myfun(x)
```

```
[1] 1 7 3
```

```
x   # unmodified
```

```
[1] 1 2 3
```

In contrast, let's see what happens in Python

```
x = [1,2,3]
def myfun(x):
  x[1] = 7
  print(x)
  return(x)

new_x = myfun(x)
```

```
[1, 7, 3]
```

```
x   # modified!
```

```
[1, 7, 3]
```

There are some (necessary) exceptions to the idea of no side effects in R. 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.

> **Challenge** What are some other functions that are called for the purpose of the side effects they produce? (For example, which functions change the state of your R session in some way?

Functions are first-class 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"
```

```
  try(x(2))    # x is not a function (yet)
```

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

```
  x <- function(z) z^2  # now it is a function
  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 an important aspect of R being a functional programming language.

```
  x <- rnorm(10)
  sapply(x, abs)
```

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

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

```
[1] -0.12
```

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
```

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 the final section of this Unit. For now, just realize that the parsed code itself is treated as an object(s) with certain types and certain classes.

The *do.call* function is another example of a function that takes a function as an argument. It 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 when the arguments you want to pass are part of a list.

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

```
[1] 5.5
```

All operations are functions

All operations in R are actually function calls, even things that don't look like function calls, including various operators (such as addition, subtraction, etc.), printing to the screen, etc.

Operators

Operators, such as `+` and `[` 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.,

```
`%*%`
```

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

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))

## note that the index "2" is the additional argument to the [[ function
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
```

```
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] +,dgTMatrix,dgTMatrix-method +,Matrix,missing-method
 [3] +,matrix,spam-method         +,spam,matrix-method
 [5] +,spam,missing-method        +,spam,spam-method
 [7] +.Date                       +.gg*
 [9] +.glue*                      +.POSIXt
[11] +.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"
```

Other operations that are functions

Even beyond operators, all code in R can be viewed as a function call, including if statements and for and while loops.

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")
  }
```

Replacement functions

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
```

```
     [,1]   [,2]
[1,] 3.00 -0.215
[2,] 1.34  2.000
```

```
mat <- `diag<-`(mat, c(10, 21))
mat
```

```
      [,1]   [,2]
[1,] 10.00 -0.215
[2,]  1.34 21.000
```

```
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: 0x5610dd8c2608>
```

```
<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*. This can cause memory use to increase temporarily (but generally very briefly). So it's something to keep in mind if you're doing replacements on large objects.

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 functional OOP. 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"
```

Map operations

A *map* operation takes a function and runs the function on each element of some collection of items, analogous to a mathematical map. This kind of operation is very commonly used in programming,

particularly functional programming, and often makes for clean, concise, and readable code.

Base R provides a variety of map-type functions: *lapply* and *sapply* and their variants, as well as *apply*. In addition, the *purrr* package for functional programming provides `purrr::map`. In R, often the map-type function is run on the elements of a list, but they can also generally be run on elements of a vector and in other ways. In other languages, map-type functions are run on a variety of data structures. These are examples of higher-order functions – functions that take a function as an argument.

Let's compare using *lapply* to using a for loop to run a stratified analysis for a generic example (this code won't run because the variables don't exist):

```
# stratification
subsets <- split(df, grouping_variable)

# lapply: one line, easy to understand
results <- lapply(subsets, analysis_function)

# for loop: needs storage set up and multiple lines
results <- list()
length(results) <- length(subsets)
for(i in seq_along(subsets))
  results[[i]] <- analysis_function(subsets[[i]])
```

Map operations are also at the heart of the famous map-reduce paradigm, used in Hadoop and Spark for big data processing.

Function evaluation, frames, and the call stack

Overview

When we run code, we end up calling functions inside of other function calls. This leads to a nested series of function calls. The series of calls is the *call stack*. 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).

Understanding the series of calls is important when reading error messages and debugging. In Python, when an error occurs, the call stack is shown, which has the advantage of giving the complete history of what led to the error and the disadvantage of producing often very verbose output that can be hard to understand. In R, only the function in which the error occurs is shown, but you can see the full call stack by invoking `traceback()` (see the debugging tutorial).

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 yet. We'll see all the details as we proceed through this Unit.

Frames and the call stack

R keeps track of the call stack. 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 later in this Unit, 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.

R provides some functions that allow you to query the call stack and its frames. *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 need to manually insert the output here because the R Markdown processing up the frame counting somehow.

```
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()
```

```
in f: Frame number is 1; parent frame number is 0.
in f: Frame (i.e., environment) is: <environment: 0x55a4d71beb88>
in f: Parent is <environment: R_GlobalEnv>
in f: Two frames up is Error in sys.frame(-2) : not that many frames on the stack
```

```
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()
```

```
in ff: Frame (i.e., environment) is: <environment: 0x55a4d7391700>
in ff: Parent is <environment: R_GlobalEnv>
in f: Frame number is 2; parent frame number is 1.
in f: Frame (i.e., environment) is: <environment: 0x55a4d7393b38>
in f: Parent is <environment: 0x55a4d7391700>
in f: Two frames up is <environment: R_GlobalEnv>
```

Next we'll use a recursive function to illustrate what information we can gather about the call stack using *sys.status*. *sys.status* gives extensive information about the call stack and the frames involved (*sys.status* uses *sys.calls*, *sys.parents* and *sys.frames*).

```
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)
```

```
$sys.calls
$sys.calls[[1]]
g(3)

$sys.calls[[2]]
if(y > 0) g(y-1) else gg()

$sys.calls[[3]]
if(y > 0) g(y-1) else gg()
```

```
$sys.calls[[4]]
if(y > 0) g(y-1) else gg()

$sys.calls[[5]]
if(y > 0) g(y-1) else gg()

$sys.calls[[6]]
tmp <- sys.status()


$sys.parents
[1] 0 1 2 3 4 5

$sys.frames
$sys.frames[[1]]
<environment: 0x55a4d63479e8>

$sys.frames[[2]]
<environment: 0x55a4d6347ba8>

$sys.frames[[3]]
<environment: 0x55a4d5736638>

$sys.frames[[4]]
<environment: 0x55a4d5732028>

$sys.frames[[5]]
<environment: 0x55a4d5732098>

$sys.frames[[6]]
<environment: 0x55a4d5732488>
```

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.

Function inputs and outputs

Arguments

Arguments can be specified by position (based on the order of the inputs) or by name, using `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. For example, `lm()` requires `formula` but not `data`, `subset`, etc., even though none of them have default arguments.

R will error out if it is expecting an argument, rather than looking for that argument elsewhere.

```
print(sum)
```

```
function (..., na.rm = FALSE)  .Primitive("sum")
```

```
sum()
```

```
[1] 0
```

```
print(quantile)
```

```
function (x, ...)
UseMethod("quantile")
<bytecode: 0x5610d75ce1b8>
<environment: namespace:stats>
```

```
try(quantile())
```

```
Error in is.factor(x) : argument "x" is missing, with no default
```

```
x <- 1
y <- 2
myfun <- function(x) {
    z <- y+3
    w <- x+3
}
try(myfun())
```

```
Error in myfun() : argument "x" is missing, with no default
```

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

```
args(dgamma)
```

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

Functions 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 (such as `xlab` and `ylab`) will get passed along to plot:

```
pplot <- function(x, y, pch = 16, cex = 0.4, ...) {
    plot(x, y, pch = pch, cex = cex, ...)
}
pplot(rnorm(10), rnorm(10), xlab = 'x', ylab = 'y')
```

If you want to manipulate what the user passed in as the `...` args, rather than just passing them along, you can extract them:

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

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

As we've seen, functions can be passed in as arguments (e.g., see the variants of *apply* and *lapply*). 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):

```
mylist <- list(rnorm(2), rnorm(3), rnorm(5))
sapply(mylist, length)
```

```
[1] 2 3 5
```

```
lapply(mylist, function(x) x[x < 0])
```

```
[[1]]
[1] -0.1

[[2]]
[1] -0.0736 -0.0376 -0.6817

[[3]]
```

```
[1] -0.324 -0.589 -1.518
```

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
```

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)
```

Challenge In the above code, what do you think `quote()`does? Why is it needed?

Where are arguments evaluated?

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

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

```
[1] 60
```

Here, when `f()` is called and the code is evaluated, `z` is evaluated in the calling frame and `z*5` is assigned to `x` in the frame of the function, while `x*3` is evaluated in the frame of the function (using the local `x` that was just created) and assigned to `y`.

Function 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)
a <- f(3)
a
```

```
[1] 9
```

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

```
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 occurs with many functions, such as *lm*. (Of course `lm()` actually returns an object of the S3 `lm` class, which inherits from the list class.)

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

```
[1] "lm"
```

```
is.list(mod)
```

```
[1] TRUE
```

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 (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. Thus pass-by-value is directly related to functional programming.

Arrays in Python are pass-by-reference (but note that tuples are immutable, so one could not modify a tuple that is passed as an argument).

```
def myfun(x):
    x[1] = 99

y = [0, 1, 2]
z = myfun(y)
y
```

```
[0, 99, 2]
```

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
```

- The `int*` declares `ptr` to be a pointer to (the address of) the integer `x`.
- The `&x` gets the address where `x` is stored.

- `*ptr` dereferences `ptr`, returning the value in that address (which is 3 since `ptr` is the address of `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 object, and inside the function, one can modify the original object, with the new value persisting on exit from the function. For example in the following example one passes in the address of an object and that object is then modified in place, affecting its value when the function call finishes.

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

myCal(&x)  # x itself will be modified
```

> **Note** When calling C or C++ from R, one (implicitly) passes pointers to the vectors into C.

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.
- Environments behave like pointers and are passed by reference rather than by copy.
- R6 objects behave like pointers and are passed by reference, as seen earlier.

We'll see more on these ideas later in the Unit.

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 computation and memory use that that causes, 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 later.
3. We can access the object in the enclosing environment as a 'global variable', as we'll see 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 enclosing environment (provided an object of that name exists in the enclosing environment). We'll discuss enclosing environments when we talk about scoping.
   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 generally bad practice for more formal code development.
4. We can use replacement functions, 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.

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 more detail later. 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.

What's strange about this?

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

```
   user  system elapsed
  0.069   0.000   0.069
```

```
system.time(f(3))
```

```
[1] "hi"
```

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

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

```
[1] "hi"
```

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

Here's an even stranger situation. Do you think the following code will run?

```
f <- function(a, b = d) {
    d <- a*3;
    return(a*b)
}

b <- 100
f(5)
```

Lazy evaluation is not just an R thing. It also occurs in Tensorflow (particularly version 1), the Python Dask package, and in Spark. 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.

Variable scope and lookup

Lexical scoping

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, discussed when we were talking about the call stack.)

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.**

This approach is called *lexical scoping*. Python and many other languages also use lexical scoping.

Why is the enclosing environment defined in this way? Recall our example where I tried to break the usage of the *lm* function by redefining *lm.fit*.

```
lm.fit <- function(x) print('hi')
y <- rnorm(10)
x <- rnorm(10)
```

```
mod <- lm(y~x)  # this still works!
```

When R looks for `lm.fit` when it is called within `lm`, it looks in the enclosing environment of `lm`. That is where `lm` is defined, which is the stats package namespace. It finds `lm.fit` there. All is well! In contrast, if the scoping rules looked for `lm.fit` where `lm` was called from, then the user-defined `lm.fit` would be found and `lm()` would not work until that `lm.fit` was removed. That would be a very fragile system!

Let's dig deeper to understand where R looks for non-local variables, illustrating lexical scoping:

```
x <- 3
f2 <- function() print(x)
f <- function() {
    x <- 7
    f2()
}
f() # what will happen?

x <- 3
f2 <- function() print(x)
f <- function() {
    x <- 7
    f2()
}
x <- 100
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 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()*?

The following 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: 0x5610e1ce9b38>
```

```
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: 0x5610e15b0248>
```

```
environment(myfun)
```

```
<environment: 0x5610e15b0248>
```

```
myfun(3)
```

```
[1] 13
```

```
environment(myfun)$y
```

```
[1] 10
```

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

```
function(x) x + y
<environment: 0x5610e15b0248>
```

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
```

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.

```
f <- function() {
    y <- 3
    print(x + y)
}
codetools::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 a general functional programming idea and not specific to R.

Using a closure 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  4.535  1.170 -1.864
```

So calling `scaler(3)` multiplies 3 by the value of `data` stored in the closure (the enclosing environment) of the function `scaler`.

It turns out that it can be hard to see the memory used involved in the closure.

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

```
3800 bytes
```

```
object.size(environment(scaler)$data)
```

```
80000048 bytes
```

```
library(pryr)
```

```
Attaching package: 'pryr'
```

```
The following object is masked _by_ '.GlobalEnv':

    f
```

```
object_size(scaler) # that's better!
```

```
80,012,560 B
```

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)))
bootmeans()
```

```
 [1] 3.59 3.41 3.47 3.46 3.43 3.48 3.51 3.48 3.50 3.46 3.41 3.62 3.46 3.46 3.49
[16] 3.50 3.56 3.50 3.58 3.60 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 3.53 3.58 3.40 3.50 3.50
[46] 3.56 3.41 3.45 3.50 3.53 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 3.39 3.48 3.62 3.54 3.51
[76] 3.52 3.47 3.49 3.43 3.45 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
```

The closure stores the bootstrapped values and

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.

Then, 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, mainly the namespaces of packages loaded in the R session.

```
search()
```

```
 [1] ".GlobalEnv"          "package:pryr"        "package:R6"
 [4] "package:fields"      "package:viridis"     "package:viridisLite"
 [7] "package:spam"        "package:dplyr"       "package:stringr"
[10] "tools:quarto"        "package:stats"       "package:graphics"
[13] "package:grDevices"   "package:utils"       "package:datasets"
[16] "package:SCF"         "package:methods"     "Autoloads"
```

```
[19] "package:base"
```

We can see the full set of environments in which R looks using code such as the following. This illustrates that in looking for a local variable used in *lm* the search process would go through the stats namespace, the base R namespace, the global environment and then the various packages loaded in the current R session.

```
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:pryr"
[1] "package:R6"
[1] "package:fields"
[1] "package:viridis"
[1] "package:viridisLite"
[1] "package:spam"
[1] "package:dplyr"
[1] "package:stringr"
[1] "tools:quarto"
[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"
```

That code uses `environmentName()`, which prints out a nice-looking version of the environment name.

Here's an alternative way using *pryr*:

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

```
   label                              name
1  <environment: namespace:stats>     ""
2  <environment: 0x5610d5d96f68>      "imports:stats"
```

```
3  <environment: namespace:base>      ""
4  <environment: R_GlobalEnv>         ""
5  <environment: package:pryr>        "package:pryr"
6  <environment: package:R6>          "package:R6"
7  <environment: package:fields>      "package:fields"
8  <environment: package:viridis>     "package:viridis"
9  <environment: package:viridisLite> "package:viridisLite"
10 <environment: package:spam>        "package:spam"
11 <environment: package:dplyr>       "package:dplyr"
12 <environment: package:stringr>     "package:stringr"
13 <environment: 0x5610d6353450>      "tools:quarto"
14 <environment: package:stats>       "package:stats"
15 <environment: package:graphics>    "package:graphics"
16 <environment: package:grDevices>   "package:grDevices"
17 <environment: package:utils>       "package:utils"
18 <environment: package:datasets>    "package:datasets"
19 <environment: package:SCF>         "package:SCF"
20 <environment: package:methods>     "package:methods"
21 <environment: 0x5610d5af9510>      "Autoloads"
22 <environment: base>                ""
23 <environment: R_EmptyEnv>          ""
```

Note that eventually the global environment and the environments of the packages are nested within the base environment (of the base package) and the empty environment.

8. Memory and copies

Under construction.

9. Efficiency

Under construction.

10. Computing on the language (optional)

Under construction.