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. Currently only Sections 1-4 are available.

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 here for us to think about why things are the way they are in R. I.e., what principles were used in creating the language and what choices were made? While other languages use different principles and made 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.

References:

- Books on R listed on the syllabus: Adler, Chambers, Wickham
- R intro manual and R language manual (R-lang), both on CRAN.
- Venables and Ripley, Modern Applied Statistics with S
- Murrell, Introduction to Data Technologies

I'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 in the section on 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 these examples, the first backslash says to interpret the next backslash literally, with the second backslash being used to indicate that the caret (^) should be interpreted literally and not as a special character 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
```

```r
## 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
```

```r
## 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
```

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

```
[1] 1 3
```

```r
## 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:

```r
## Suppose we want to use a \ in our string:
cat("hello\nagain")
```

```
hello
again
```

```r
cat("hello\\nagain")
```

```
hello\nagain
```

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

```
[1] "badCode.R"    "cache"        "calc_mean.py" "convert.sh"   "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.

```r
## 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
```

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

```
$width
[1] 80

$digits
[1] 7
```

```r
## 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 often depend on other packages. In general, if one package depends on another, R will load the dependency, but if the dependency is installed locally (see below), R may not find it automatically and you may have to use *library()* to load the dependency first. 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.

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.

```r
.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"
```

```r
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.

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.

```r
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.

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 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.

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

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

Coefficients:
(Intercept)            x
      0.340        0.344
```

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/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:

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

```
[1] "numeric"
```

```r
vals <- 1:3
class(vals)
```

```
[1] "integer"
```

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

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

```r
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:

```r
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%
-1.95  2.11
```

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

```
2.5%
1.05
```

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

```
352 bytes
```

We can get rid of the attribute:

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

```
[1] -1.95  2.11
```

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

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

```r
names(mtcars)
```

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

```r
attributes(mtcars)
```

```
$names
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"

$row.names
 [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
 [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
 [7] "Duster 360"          "Merc 240D"           "Merc 230"
[10] "Merc 280"            "Merc 280C"           "Merc 450SE"
[13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
[16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
```

```
[19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
[22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
[25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
[28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
[31] "Maserati Bora"       "Volvo 142E"

$class
[1] "data.frame"
```

```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:

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

```
[1] 1 2
```

Be careful of using factors as indices:

```r
students <- factor(c("basic", "proficient", "advanced",
                     "basic", "advanced", "minimal"))
score <- c(minimal = 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?

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  0.557 -0.379
2     a  0.463 -0.273
3     a  0.122 -0.294
4     a  0.514  0.513
5     a -0.896 -1.098
```

```
## 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]  0.178  0.189 -0.172  1.027 -1.994
```

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 *apply()* 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.

```r
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
```

```r
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).

```r
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
```

```r
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.

```r
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.
```

```r
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:

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

Challenge Where is the non-standard evaluation there?