Introduction to R

PS239T

03 February, 2019

- Many of the materials for this session are adapted from Hadley Wickam's Advanced R. I also mention other books and papers I referred to at the outset of each markdown file.

1. What's R?

R is a versatile, open source programming/scripting language that's useful both for statistics but also data science. Inspired by the programming language `S`.

- Open source software under GPL.
- Superior (if not just comparable) to commercial alternatives. As of January 2019, R ranks 12th in the TIOBE index, which measures the popularity of programming languages. It's widely used both in academia and industry especially in the circle of data scientists.
- Available on all platforms (Unix, Windows, Linux).
- As a result, if you do your analysis in R, anyone can easily replicate it.
- Not just for statistics, but also **general purpose** programming.
- Is **object oriented** (= R has objects) and **functional** (= You can write functions).
- Large and growing community of peers. (Take advantage of git repositories.)

- The power of objects. I will explain about what the below code does at the end of the session.

```
library(dplyr)
library(gapminder) # load gapminder package

gapminder %>%
  filter(continent == "Europe") %>% # filter by Europe
  group_by(country) %>% # group by country
  summarize(Mean = mean(gdpPercap)) %>% # collapse data by mean
  top_n(5, Mean) %>% # count only top 5 by mean
  arrange(desc(Mean)) # arrange by descending order
```

- The power of functions. We're going to learn about them next week. Functions help you Don't Repeat Yourself. I love the following Wickam's example.

```
df$a[df$a == -99] <- NA
df$b[df$b == -99] <- NA
df$c[df$c == -99] <- NA
df$d[df$d == -99] <- NA
df$e[df$e == -99] <- NA
df$f[df$f == -99] <- NA

# very tedious

fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
```

```
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
df$f <- fix_missing(df$e)

# beautiful
```

2. RStudio

There are two main ways of interacting with R: using the console or by using script files (plain text files that contain your code).

The console window (in RStudio, the bottom left panel) is the place where R is waiting for you to tell it what to do, and where it will show the results of a command. You can type commands directly into the console, but they will be forgotten when you close the session. It is better to enter the commands in the script editor, and save the script. This way, you have a complete record of what you did, you can easily show others how you did it and you can do it again later on if needed. You can copy-paste into the R console, but the RStudio script editor allows you to 'send' the current line or the currently selected text to the R console using the `Ctrl-Enter` shortcut.

At some point in your analysis, you may want to check the content of variable or the structure of an object, without necessarily keep a record of it in your script. You can type these commands directly in the console. RStudio provides the `Ctrl-1` and `Ctrl-2` shortcuts allow you to jump between the script and the console windows.

If R is ready to accept commands, the R console shows a `>` prompt. If it receives a command (by typing, copy-pasting or sent from the script editor using `Ctrl-Enter`; `Command-Enter` will also work on Macs), R will try to execute it, and when ready, show the results and come back with a new `>`-prompt to wait for new commands. This is the equivalent of the `$` in your terminal.

If R is still waiting for you to enter more data because it isn't complete yet, the console will show a `+` prompt. It means that you haven't finished entering a complete command. This is because you have not 'closed' a parenthesis or quotation. If you're in RStudio and this happens, click inside the console window and press `Esc`; this should help you out of trouble.

3. R Environment

R is a functionalized language, like Python 3; that means that unlike in bash, you have to enclose the object of the command inside of the command using `( )` parentheses. Let's take a look at some familiar-looking commands:

Viewing objects in your global environment and how to clean them up

List objects in your current environment

```
ls()
```

Remove objects from your current environment

```
x <- 5
rm(x)
```

Remove all objects from your current environment

```
a <- 7
b <- 3
rm(list = ls())
```

Force memory release

```
gc()
```

Notice that we have nested one function inside another.

Package management

- `install.packages(package-name)` will download a package from one of the CRAN mirrors assuming that a binary is available for your operating system. If you have not set a preferred CRAN mirror in your `options()`, then a menu will pop up asking you to choose a location.
- `library(package-name)` will load a package so you can use it. It is required at the beginning of each R session.

```
library(stats)
```

Tips

If you have multiple packages to install, then please consider using pacman package. The following is the example. First, you install pacman. Then, you load several libraries by using p_load method.

```
install.packages("pacman")

pacman::p_load(
  ggplot2,
  dplyr,
  broom
)
```

If you don't like to use pacman, then the other option is to create a list. (We're going to learn what is list soon.)

```
pkgs <- c("ggplot2", "dplyr", "broom")

install.packages(pkgs)
```

Still, we have to write two lines. The simpler, the better, right? Here's another approach that can simplify the code further.

Note that lapply applies (there's a family of apply functions) a function to a list. In this case, library to pkgs. apply is an advanced concept, which is related to anonymous functions. We will learn about it later when we study functions.

```
inst = lapply(pkgs, library, character.only = TRUE)
```

4. Basic Syntax

Separing commands

Use `#`.

Whitespace

Meaningless.

Comments

Use `#` signs to comment. Comment liberally in your R scripts. Anything to the right of a `#` is ignored by R. For those of you familiar with other languages, there is no doc string, or equivalent to `"""` in R.

Assignment operator

`<-` is the assignment operator. It assigns values on the right to objects on the left. So, after executing `x <- 3`, the value of `x` is `3`. The arrow can be read as 3 **goes into** `x`. You can also use `=` for assignments.

```
USwerid <- "Why use lb for pound!" # Use this
"Why use lb for pound!" -> USweird
assign("Why use lb for pound", USweird)
"Why use lb for pound!" = USweird
```

Nonetheless, can does not mean you should. It is good practice to use `<-` for assignments. `=` should only be used to specify the values of arguments inside of functions. This is what Google and Hadley Wickham recommend, too. If they don't convince you enough, here's a real example.

```
mean(x = 1:10) # Does it save x?
```

```
## [1] 5.5
```

```
rm(x)
```

```
## Warning in rm(x): object 'x' not found
```

```
mean(x <- 1:10) # Does it save x?
```

```
## [1] 5.5
```

```
rm(x)
```

Variable Names

Variable names can only contain letters, numbers, the underscore character, and (unlike Python) the period character. Whereas an object name like `myobject.thing` would point to the subclass or method `thing` of `myobject` in Python, R treats `myobject.thing` as its own entity.

Printing

In R, the contents of an object can be printed by either simply executing the the object name or calling the `print()` function.

Help

- `?` + object opens a help page for that specific object
- `??` + object searches help pages containing the name of the object

```
?mean
??mean
```

Miminal reproducible example

Chances are you're going to use StackOverFlow a lot to solve a pressing problem you face. However, other can't understand/be interested in your problem unless you can provide an example which they can understand with minimal efforts. Such example is called a minimal reproducible example.

Read this StackOverFlow post to understand the concept and best practices.

Simply put, a MRE consists of the following items:

- A minimal dataset
- the minimal burnable code
- the necessary information on package, R version, system (use sessionInfo())
- (for random process) a seed for reproducibility (set.seed())

Bonus: More about R

From Hadley Wickham: # Note - this is the guy who wrote ggplot. We'll see his work again soon!

If you are new to R, you might wonder what makes learning such a quirky language worthwhile. In my opinion, R's some of the best features are:

- It's free, open source, and available on every major platform. As a result, if you do your analysis in R, anyone can easily replicate it.
- A massive set of packages for statistical modeling, machine learning, visualization, and importing and manipulating data. Whatever model or graphic you're trying to do, the chances are that someone has already tried to do it. At a minimum, you can learn from their efforts.
- Cutting edge tools. Researchers in statistics and machine learning will often publish an R package to accompany their articles. This means immediate access to the very latest statistical techniques and implementations.
- Deep-seated language support for data analysis. This includes features likes missing values, data frames, and subsetting.
- A fantastic community. It is easy to get help from experts on the R-help mailing list, stackoverflow, or subject-specific mailing lists like R-SIG-mixed-models or ggplot2. You can also connect with other R learners via twitter, linkedin, and through many local user groups.
- Powerful tools for communicating your results. R packages make it easy to produce HTML or PDF reports, or create interactive websites.
- A strong foundation in functional programming. The ideas of functional programming are well suited to solving many of the challenges of data analysis. R provides a powerful and flexible toolkit which allows you to write concise yet descriptive code.
- An IDE tailored to the needs of interactive data analysis and statistical programming.
- Powerful metaprogramming facilities. R is not just a programming language; it is also an environment for interactive data analysis. Its metaprogramming capabilities allow you to write magically succinct and concise functions and provide an excellent environment for designing domain-specific languages.
- Designed to connect to high-performance programming languages like C, FORTRAN, and C++.

Of course, R is not perfect. R's biggest challenge is that most R users are not programmers. This means that:

- Much of the R code you'll see in the wild is written in haste to solve a pressing problem. As a result, code is not very elegant, fast, or easy to understand. Most users do not revise their code to address these shortcomings.
- Compared to other programming languages, the R community tends to be more focused on results instead of processes. Knowledge of software engineering best practices is patchy: for instance, not enough R programmers use source code control or automated testing.
- Metaprogramming is a double-edged sword. Too many R functions use tricks to reduce the amount of typing at the cost of making code that is hard to understand and that can fail in unexpected ways.
- Inconsistency is rife across contributed packages, even within base R. You are confronted with over 20 years of evolution every time you use R. Learning R can be tough because there are many special cases to remember.
- R is not a particularly fast programming language, and poorly written R code can be terribly slow. R is also a profligate user of memory.