Introducción a R

Luis Alejandro C.

Marzo, 2020

Presentación

R es un lenguaje de programación orientado al análisis estadístico. Eso significa que a través de instrucciones y algoritmos, **R** nos permite crear programas para manipular datos, generar gráficos y visualizaciones y realizar inferencia estadística. Con **R** podemos también llevar a cabo análisis econométrico e implementar algoritmos de aprendizaje automático, entre otras.

La funcionalidad de este lenguaje se da a través de los llamados **"paquetes"** o **"librerías"**. **R** cuenta con una **librería "base"**, y un número creciente de **librerías "adicionales"**. Las librerías suelen ser proyectos colaborativos y se consideran como la "unidad fundamental de código que se puede compartir". Un paquete agrupa código, datos, documentación y pruebas y facilita las tareas que podrían desarrollarse con las funciones base del lenguaje. Las librerías se ponen a disposición de la comunidad a través de servidores centrales como The Comprehensive R Archive Network o **CRAN** aunque pueden ser compartidas a través de repositorios como **Github** o **Bioconductor**.

R es un lenguaje orientado a objetos pero también un lenguaje funcional. John Chambers, uno de los miembros clave del proyecto para crear **R**, lo plantea de esta forma:

> *"Todo lo que existe es un objeto. Todo lo que sucede es una llamada a una función".*

Lo anterior significa, en términos generales, que las variables, datos, resultados, etc., que guardamos en la memoria de nuestras computadoras se convierten en objetos con nombres específicos. Sin embargo, para modificar o manipular los objetos, recurrimos a funciones. Aunque la comprensión de estos paradigmas escapa de los alcances de este curso, es importante tener en mente esta distinción general entre objetos y funciones.

1. Ejecución de código

Ya sea que trabajemos con **R** o **RStudio**, el código suele generarse y guardarse en "scripts". En el primer caso, al abrir un nuevo script nos aparecerá un editor de texto. En el segundo, se desplegará una pestaña dentro de la ventana principal de trabajo ubicada en la parte superior izquierda de nuestra pantalla. Cuando "corremos" o ejecutamos un código, el resultado se mostrará en la consola, que por default se encuentra en la parte superior izquierda en **R** y en la parte inferior izquierda en **RStudio**.

Un código puede ejecutarse por línea, por dos o más líneas o por completo.

1. En **R**:

- Para correr una línea, nos posicionamos al final del código y presionamos `Ctrl+R`
- Para ejecutar dos líneas o más, seleccionamos el segmento de código y presionamos `Ctrl+R`
- Para correr todo un script, presionamos `Ctrl+A` y posteriormente `Ctrl+R`

2. En **RStudio**:

- Para correr una línea, nos posicionamos al final del código y presionamos `Run` (parte superior derecha de la ventana del script)
- Para ejecutar dos líneas o más, seleccionamos el segmento de código y presionamos `Run`
- Para correr todo un script, hacemos clic derecho, `Select All` y posteriormente `Run`

Ejercicio: Copiar y ejecutar el siguiente código. ¿Cuáles son los resultados?

```
Este es un segmento de código
print("¡Hola, mundo!")
2+2
mean(1:10)
```

En **R**, el código se comenta con el signo # y no se evalúa nada que se encuentre después de él. ¿Qué resultados arrojan las siguientes líneas de código cuando son ejecutadas?

```
# mean(1:10)
mean(1:10) # La función mean() calcula la media
```

2. Acceder a la documentación

En **R** y **RStudio**, existen al menos dos formas de obtener información sobre funciones. La primera es con `?` y un criterio de búsqueda para rastrear coincidencias exactas de la función especificada y la segunda es con `??` más un criterio de búsqueda que trabaja sobre coincidencias aproximadas. Con la segunda forma es posible también acceder a información sobre funciones de librerías que no tenemos instaladas. En **RStudio** contamos además con el atajo `<F1>` que podemos activar tras teclear el nombre de alguna función reconocida como válida.

```
# Buscar información para la función filter()
?filter
??filter

# Buscar información para la función mean()
?mean
??mean
```

Para acceder a información relativa a paquetes o librerías, tenemos `package?nombre_paquete`, `help(package="nombre_paquete")` y `??nombre_paquete`. El tipo de documentación al que accesan esos comandos no se encuentra estandarizada por lo que cada opción mostrará resultados diferentes para cada librería y entre distintas librerías. Accedamos, por ejemplo, a la información de dos de las librerías más importantes en **R**: `tidyverse` y `data.table`:

```
# Documentación para tidyverse
package?tidyverse
help(package="tidyverse")
??tidyverse

# Documentación para data.table
package?tidyverse
help(package="tidyverse")
??tidyverse
```

*Nota: Mientras que en **R** se abrirá un navegador, en **RStudio** la documentación aparece en la ventana inferior derecha.*

3. Directorio de trabajo, rutas relativas y rutas absolutas

El directorio de trabajo (*"working directory"*) es el lugar dentro de nuestra computadora donde se guardan por defecto los archivos (bases de datos y scripts, por ejemplo) que generamos en **R**. Para conocer la ubicación de este directorio, ingresamos el siguiente comando en nuestra consola:

```
getwd()
```

El directorio de trabajo establece un "punto inicial" y nos permite trabajar "hacia arriba" o "hacia abajo" con **"rutas relativas"** (*relative paths*) respecto a esa ubicación. Para ejemplificar la noción de rutas relativas haremos lo siguiente:

1. Crear en el escritorio una carpeta llamada **Rtest**
2. Dentro de **Rtest** guardamos el archivo **prueba_uno.R** y generamos la subcarpeta **Rtestuno**
3. Finalmente, dentro de **Rtestuno** creamos la subcarpeta **Rtestdos** y en esta última carpeta alojamos el archivo **prueba_dos.R**

A continuación, cambiaremos la ubicación de nuestro directorio de trabajo. Para ello, tomaremos como punto inicial la ruta de la carpeta **Rtestdos**, la cual en Windows suele mostrarse con una barra hacia atrás (*"backslash"*) ****:

```
# Ojo: La ruta varía según la configuración de cada usuario
C:\Users\Alex\Desktop\Rtest\Rtestuno\Rtestdos
```

Para efectos de simplicidad y de compatibilidad con otros entornos, emplearemos la barra hacia adelante (*"forward slash"*) **/**:

`C:/Users/Alex/Desktop/Rtest/Rtestuno/Rtestdos`

Esa ruta se incorpora en la función `setwd()`:

```
setwd("C:/Users/Alex/Desktop/Rtest/Rtestuno/Rtestdos")
```

Podemos verificar que la ubicación de nuestro directorio de trabajo ha cambiado:

```
getwd()
```

Ahora emplearemos la función `list.files()` y veremos que el archivo **prueba_dos** se encuentra en la carpeta que hemos elegido como directorio de trabajo:

```
list.files()
```

Con la misma función y un path relativo "hacia arriba" accederemos al contenido de nuestra carpeta **Rtest**

```
list.files(path="../../")
```

La expresión **"../"** indica que la búsqueda se hace en un directorio "por encima". En este caso, **"../../"** se interpreta como "subir dos niveles respecto al directorio de trabajo". Si queremos que `list.files()` nos indique si específicamente **prueba_uno** se encuentra en el directorio deseado, incorporamos el argumento `pattern=` con una cadena de caracteres:

```
list.files(path="../../", pattern="prueba")
```

Para demostrar cómo funcionan los paths relativos "hacia abajo", cambiaremos nuestro directorio de trabajo:

```
# Fijar "Rtest" como el directorio de trabajo
setwd("C:/Users/Alex/Desktop/Rtest/")
```

Si queremos acceder a la carpeta **Rtestdos** en la que se encuentra el archivo **prueba_dos**, indicamos de izquierda a derecha los nombres de las carpetas **Rtestuno** y **Rtestdos** pues jerárquicamente son las que se hallan por debajo de **Rtest**, que es nuestro directorio de trabajo:

```r
list.files("Rtestuno/Rtestdos")
```

El uso de paths relativos favorece el ahorro de código, ya que si recurriéramos a un "path absoluto", tendríamos que llamar a `list.files()` en la forma:

```r
# Para buscar prueba_uno en la carpeta Rtest
list.files(path="C:/Users/Alex/Desktop/Rtest", pattern="prueba")

# Para buscar prueba_dos en la carpeta Rtestdos
list.files(path="C:/Users/Alex/Desktop/Rtest/Rtestuno/Rtestdos")
```

Por otra parte, si llegaramos a cambiar el sitio de nuestro directorio de trabajo, bastará utilizar `setwd()` para establecer la nueva ubicación (nuestros paths relativos continuarán funcionando correctamente). Si nos valiéramos de los paths absolutos, podríamos cometer errores especificando las rutas.

4. Instalación, carga y actualización de librerías

Las funciones para instalar y cargar librerías en nuestro entorno de trabajo (es decir, en nuestro proyecto actual) son `install.packages()` y `library()`. Con la primera podemos descargar librerías de forma individual o múltiple:

```r
# Para descargar librerías individualmente
install.packages("libreria")

# Para descargar múltiples librerías
install.packages(c("libreria_uno", "libreria_dos", "libreria_tres"))
```

Por su parte, la carga de librerías solo puede hacerse una a una. Sin embargo, la librería `pacman` cuenta con una función pensada en esa necesidad:

```r
# Para cargar librerías individualmente
library("libreria")

# Para cargar múltiples librerías con pacman
# Descargamos la librería
install.packages("pacman")
# Cargamos la librería en nuestro entorno de trabajo
library("pacman")
# Utilizamos la función p_load()
p_load(libreria_uno, libreria_dos, libreria_n)
```

La actualización de **R** se hace a través de la función `updateR()` de la librería `installr`:

```r
# Uso de installr para actualizar nuestra versión de R
install.packages("installr")
library("installr")
updateR()
```

Si contamos con la versión más reciente, en consola se mostrará el mensaje `[1] FALSE`. En caso contrario, el sistema nos indicará la existencia de una nueva versión y nos guiará en el proceso de instalación.

En el caso de las librerías, la actualización no se realiza de forma automática cuando cambiamos de versión de **R** sino que es un proceso llevado a cabo por el usuario. Eso significa que **no siempre es necesario actualizar las librerías si instalamos la versión más reciente de R** a menos que que el paquete tenga nuevas funcionalidades (o se deprecien funcionalidades actuales), dependa de una versión más nueva o sea recomendado por los autores de las librerías. Debemos ser cuidadosos de que la actualización de librerías no se encuentre en conflicto con la estructura de nuestros códigos y scripts en uso.

Para actualizar las librerías, podemos revisar primero cuáles son las que tienen actualizaciones disponibles con `(old.packages()` y obtener las versiones recientes con `update.packages()`.

```r
# Librerías actualizables en formato data.frame
View(old.packages())

# Para actualizar las librerías
# El argumento ask=FALSE suprime el cuadro de diálogo que solicita permisos
update.packages(ask=FALSE)
```

4.1 Notas importantes sobre librerías

1. Las librerías instaladas vía `install.packages()` no se cargan de forma automática sino que es necesario llamarlas a través de `library()`. Veamos un ejemplo:

```r
# Instalemos las librerías tidyverse y data.table
install.packages(c("tidyverse", "data.table"))

# La primera sentencia nos mostrará "TRUE" y la segunda "FALSE"
is.element(c("tidyverse","data.table"), installed.packages())
is.element(c("tidyverse","data.table"), loadedNamespaces())
```

2. **En R distintas librerías suelen tener funciones con nombres similares**. Cuando cargamos paquetes que se encuentran en ese supuesto, se "enmascara" (oculta) una función ya existente y recibimos advertencias en la consola. Por ejemplo, cuando cargamos `tidyverse` se podría observar una lista como la siguiente:

```
library(tidyverse)
-- Conflicts ---------------------------------------- tidyverse_conflicts()
x tidyr::expand() masks Matrix::expand()
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
x tidyr::pack()   masks Matrix::pack()
x tidyr::unpack() masks Matrix::unpack()
```

3. Para evitar resultados no deseados al aplicar una función que se encuentra "enmascarada", lo correcto es usar el esquema `libreria::funcion`. Por ejemplo, si queremos usar la función `lag()` de la librería `stats`, la llamamos como `stats::lag()`. En cambio, si nos interesa esa misma función pero de `dplyr`, lo correcto es emplear `dplyr::lag()`. **Es importante estar atentos a los mensajes que se despliegan en consola cuando cargamos librerías.**

5. La función ls()

Es una función que nos devuelve la lista de objetos en un entorno especificado. Cuando se utiliza sin argumentos, la función regresa las bases de datos, variables y funciones definidas por el usuario. En **RStudio**, la pestaña "Environment", situada en la parte superior derecha, muestra el listado de todos los objetos en el entorno global. Por su parte, la función `ls.str()` es una variación de `ls()` a la cual se le aplica la función `str()`. Con el argumento "mode" podemos obtener listados para nombres específicos:

```
# Listado de funciones
ls.str(mode="function")

# Listado de data.frames
ls.str(mode="list")
```

Nota: En la sección 2 del curso se verá por qué una data.frame es una lista

Se suele emplear `ls()` dentro de la función `rm()` (*"remove"*) para generar un espacio de trabajo en blanco de la siguiente manera:

```
rm(list=ls())
```

Sin embargo, esta acción es desaconsejada y se sugiere más bien trabajar por "proyectos", en particular si nuestros códigos serán utilizados y/o replicados por terceros. En la sección final del curso se abordarán algunas buenas prácticas de trabajo en **R** entre las cuales se encuentra la noción de "proyectos".

6. La función str()

En la sección anterior se hizo mención de `str()`. Se trata de una función denominada "genérica", lo que significa básicamente que su "salida" (resultado en consola) se "adapta" a cada tipo de dato (es decir, al objeto que se ingresa como argumento). Como se verá a lo largo de este curso, `str()` es la función de uso común para realizar el primer acercamiento a una base de datos.

¿Qué nos regresa `str()` para cada tipo de objeto? (Correr cada línea por separado)

```
x <- 5; str(x)
y <- "5"; str(x)
xy <- NA; str(xy)
yx <- factor(c("A","B","C")); str(yx)
str(mtcars)
```

7. El uso de source()

`source()` es una función que ejecuta el contenido de un script. Es útil cuando requerimos cargar librerías y funciones que empleamos de orma recurrente. Su principal agumento es la ruta del archivo, los cuales suelen tener la extensión **.R**. Debe recordarse que si el archivo se encuentra en subdirectorios cercanos a nuestro directorio de trabajo (ya sea "hacia arriba" o "hacia abajo"), podemos hacer uso de los paths relativos para la ejecución.

```
source("script.R")
```

Codificaciones.

Otro punto importante que es bueno aclarar es el tema de la codificación, ya que suele traer dolores de cabeza. Los archivos de texto, suelen tener codificaciones, que son formas en que cada caracter se escribe, lo habitual es la codificación ANSI o más bien llamada Latin1 y por supuesto el UTF-8. De hecho esto es algo que deberíamos configurar inicialmente en el entorno, para asegurarnos de no tener problemas a la hora de compartir los scripts o de moverlos de instalación en instalación. Lo Ideal es tener todo en UTF-8, en Rstudio se configura en la opciones globales. Con source si tuviéramos que cargar un Script que no sea nuestro, conociendo esta codificación podremos hacer

```
source("script_a_ejecutar.R", encoding="Latin1")
```

Recordar que Latin1 es la antigua codificación dónde cada caracter es un único byte, esto hoy en día no es lo ideal, compartir un script o un archivo de texto de este tipo puede traernos problemas, ciertos caracteres cambian según ciertas configuraciones regionales, es preferible adoptar el estándar utf-8.