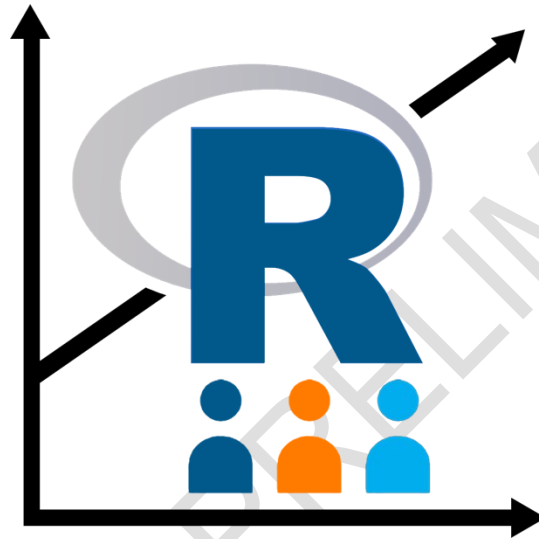


```
# Introducción al uso de "R" en emergencias de salud  
# Módulo 1 – Introducción a R / Clase 4  
# Conceptos fundamentales para comenzar a programar en R
```



```
modulo_1 <- ("Clase 4")
```

```
> Operadores  
> Objetos  
> Funciones  
> Paquetes  
> Clases  
> Scripts
```

FICHA_TÉCNICA

Introducción al uso de "R" en emergencias de salud
Módulo 1 – Introducción a R / Clase 4
Conceptos fundamentales para comenzar a programar en R

ORGANIZACIÓN PANAMERICANA DE LA SALUD/ORGANIZACIÓN MUNDIAL DE LA SALUD (OPS/OMS)

Coordinación General

Dra. Socorro Gross Galiano – Representante de la OPS/OMS – Brasília/Brasil

Dra. Maria Almiron – Advisor, Detection, Verification and Risk Assessment/Health

Emergency Information & Risk Assessment (HIM)/ PAHO Health Emergencies Department (PHE) – Washington/United States

Coordinación Ejecutiva

Unidad Técnica de Vigilancia, Preparación y Respuesta a Emergencias y Desastres/ OPS/OMS – Brasília/Brasil

Walter Massa Ramalho

Juan Cortez-Escalante

Laís de Almeida Relvas-Brandt

Mábia Milhomem Bastos

Amanda Coutinho de Souza

Health Emergency Information & Risk Assessment (HIM)/ PAHO Health Emergencies Department (PHE) – Washington/United States

Cristian Hertlein

Wildo Navegantes de Araújo

Equipo Técnico

Unidad Técnica de Vigilancia, Preparación y Respuesta a Emergencias y Desastres/ OPS/OMS
Contenido

Laís de Almeida Relvas-Brandt

Pedro Amparo Leite

Revisión

Flávia Reis de Andrade

Laís de Almeida Relvas-Brandt

Mábia Milhomem Bastos

Equipo Pedagógico

Mônica Diniz Durães – Colaboradora – Consultora Nacional de Capacidades Humanas para la Salud/ OPS/OMS

Creación Digital

Pedro Augusto Jorge de Queiroz

Flávia Reis de Andrade

> 1

> Para seguir la clase

En esta clase comenzaremos a escribir códigos en R.

Usted puede seguir el contenido de dos formas. La primera es abrir el *script* de la clase, donde encontrará todos los ejemplos que utilizaremos en este material didáctico listos para ejecutarse. Para hacerlo, abra RStudio, haga clic en el menú *Files* ubicado en la parte superior izquierda y luego en *Open File* (Figura 1). Encuentre el *script* de la clase en la carpeta *r_opas/scripts*, que usted descargó en el material del curso (archivo “*r_opas_mod1_clase4_conceptos.R*”). Al final de esta clase, usted comprenderá mejor el concepto de *scripts*.

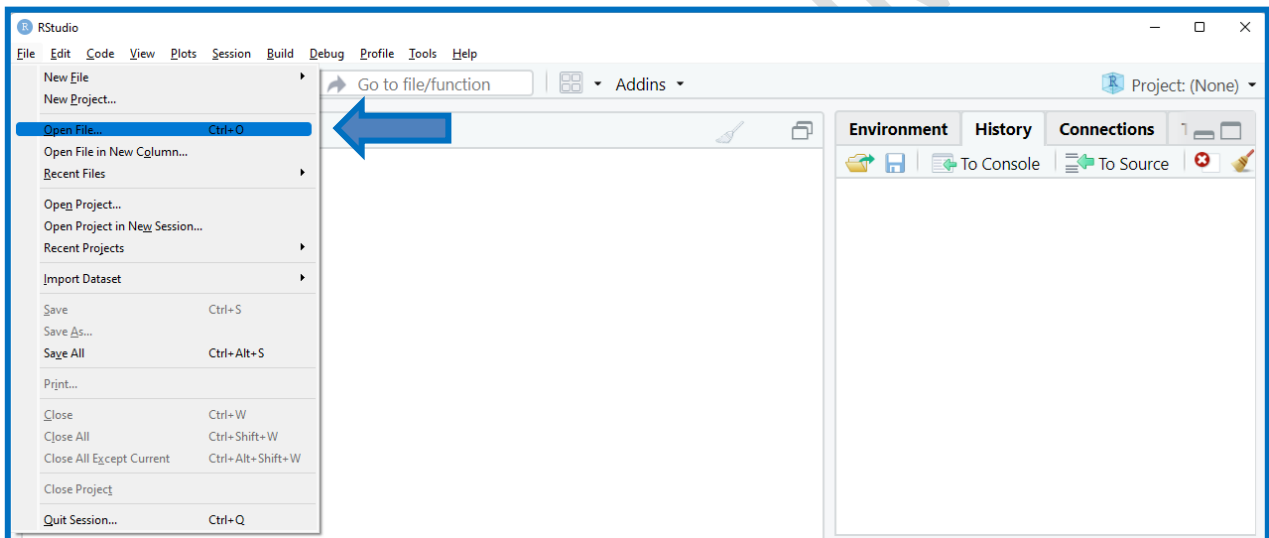



Figura 1: Vista de la pantalla de inicio para abrir un *script* desde el menú *Files*.

Usted también puede intentar escribir los ejemplos que vamos a usar en clase y crear su propio *script*. Esta alternativa le permitirá practicarlo más y puede ser una estrategia para todas las siguientes clases. Para ello, abra un nuevo *script*, como hicimos en la última clase, y reproduzca en él los códigos de la clase. Recuerde guardar su *script*. Usted puede hacer clic en el menú *Files*, luego en *Save As* y guardar el *script* con su nombre en la carpeta *r_opas/scripts* (por ejemplo: “*r_opas_mod1_clase4_minombre.R*”). **¡ATENCIÓN!** No utilice puntos ni otros caracteres especiales que no sea *underline* (“_”) para nombrar sus *scripts*.

Ahora que usted ha abierto el *script* de la clase, debe aprender a ejecutar un comando, es decir, instruir a R para que realice una tarea. Por favor, escriba "2+2" (sin comillas) en la *console*, como se muestra en la Figura 2. Si usted presiona la tecla *Enter* en el teclado, inmediatamente verá el resultado 4 en la misma *console*. Ahora haga lo mismo en el editor y observe lo que sucede. Tenga en cuenta que para ejecutar el mismo comando escrito en el editor debemos enviar el comando a la *console*, lo que se puede hacer usando el símbolo  con la parte del código que desea ejecutar resaltada. También observe que no es necesario seleccionar todo el código, ya que el comando se ejecutará si usted hace clic en cualquier parte de la línea. Usted también tiene la opción de presionar *Control* + *Enter* en su teclado como un atajo para ejecutar el código.

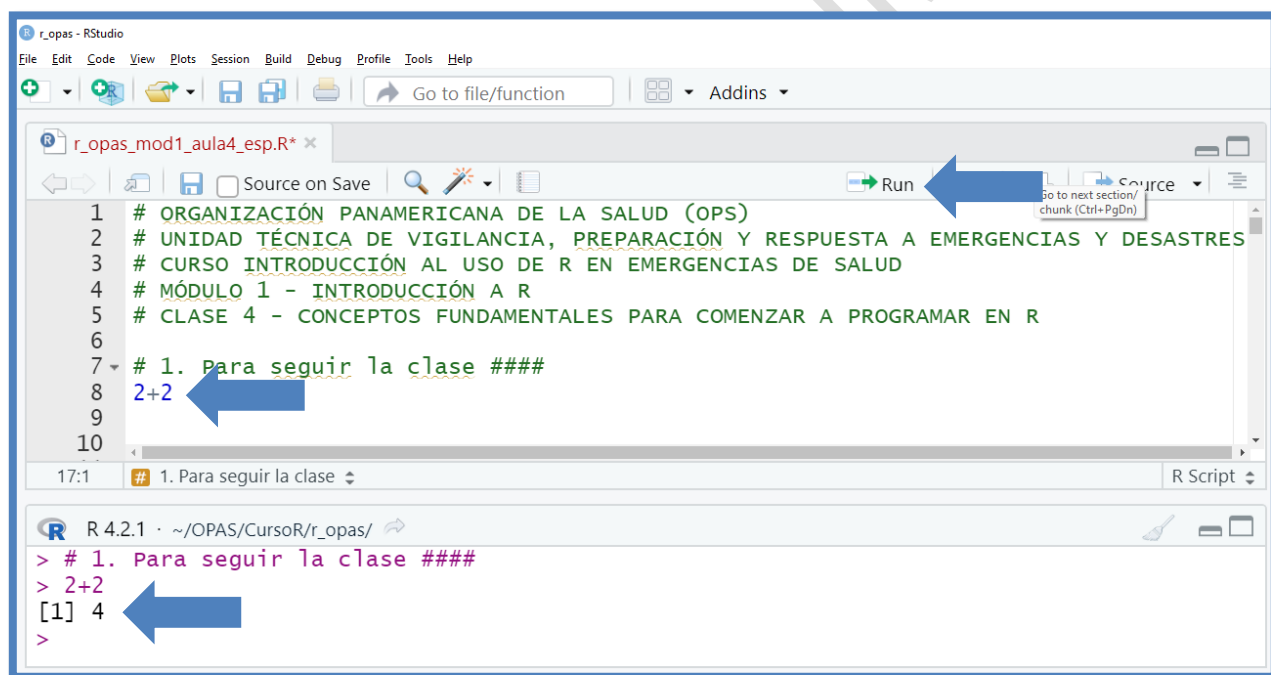


Figura 2: Vista de la pantalla para usar el botón *Run* o *Ctrl* + *Enter* (en el teclado) para ejecutar el código.

Para evitar crear numerosas figuras con pantallas de RStudio en el texto de las clases, a partir de la siguiente sección presentaremos los códigos como en el cuadro sombreado a continuación. El área resaltada en gris reproduce el *script* de la clase en pequeñas partes. Para ayudarle a ubicarse, los códigos siempre corresponderán al

número de la respectiva sección en el texto, los comentarios estarán en negrita y los resultados en gris más claro. Por ejemplo, en el siguiente código vemos el encabezado de la clase y el nombre de esta sección en negrita, la expresión 2+2 en negro y la salida de la operación ([1] 4) en gris más claro.


```
# ORGANIZACIÓN PANAMERICANA DE LA SALUD (OPS)
# UNIDAD TÉCNICA DE VIGILANCIA, PREPARACIÓN Y RESPUESTA A EMERGENCIAS Y DESASTRES
# CURSO INTRODUCCIÓN AL USO DE R EN EMERGENCIAS DE SALUD
# MÓDULO 1 - INTRODUCCIÓN A R
# CLASE 4 - CONCEPTOS FUNDAMENTALES PARA COMENZAR A PROGRAMAR EN R

# 1. Para seguir la clase #####
2 + 2
[1] 4
```

Antes de continuar, también le dejamos algunos consejos para que usted empiece a programar en R:

- R distingue entre mayúsculas y minúsculas, es decir, "A" no es lo mismo que "a" y "datos" no es lo mismo que "Datos".
- Todo lo que viene después de un símbolo # se interpreta como un comentario y se ignora al ejecutar el *script*. Los comentarios deben usarse libremente en todo su código, tanto para sus propias anotaciones como para ayudar a sus colaboradores(as).
- Si usted incluye cuatro signos de almohadilla (####) al final de un comentario, él se convertirá en el título de una sección de su *script*. Observe en el *script* de esta clase que todas las secciones tienen una flecha en el lado izquierdo de la línea del título, que usted puede usar para abrir y cerrar la sección.
- Los comandos suelen estar separados por una nueva línea. Usted también puede usar un punto y coma (" ; ") para separar sus comandos, pero esto rara vez se usa.
- Si un *prompt* de continuación + aparece en la *console* después de ejecutar su código, significa que usted no completó su código correctamente. Esto suele suceder si olvida cerrar un paréntesis y es especialmente común cuando se usan

paréntesis anidados (uno dentro del otro). Por ejemplo, en el código `mean(sum(x))` falta un segundo paréntesis para cerrar la función `mean()`.

- En general, R es bastante tolerante con los espacios adicionales insertados en su código. De hecho, se recomienda el uso de espacios para ayudarle a visualizar mejor las diferentes instrucciones del código. Sin embargo, no se deben insertar espacios en operadores. Por ejemplo, debemos usar `"<-"` y no `"< -"` (vea el espacio).
- Si su *console* "se cuelga" y no responde después de ejecutar un comando, muchas veces se puede solucionar los problemas presionando la tecla *Esc* en el teclado o haciendo clic en el icono de parada  (*stop*) en la esquina superior derecha de la *console*. Esto finalizará la mayoría de las operaciones.

> 2

> operadores

Operadores aritméticos

Una buena manera de comenzar a escribir códigos en R es usar RStudio como una calculadora. Podemos, por ejemplo, escribir una expresión aritmética en nuestro *script* y luego ejecutarla en la *console* para recibir un resultado. Por ejemplo, si escribimos la expresión `2 + 3` y luego ejecutamos esa línea de código, obtendremos la respuesta 5. El `[1]` delante del resultado informa que el número de la observación al principio de la línea es la primera observación. Quizás esto no sea muy útil en este ejemplo, pero puede ser de gran ayuda para leer los resultados con varias líneas en el futuro.

```
# 2. Operadores ####  
# Operadores aritméticos  
# Suma  
2 + 3  
[1] 5
```

Ahora ejecute el siguiente código y compare sus resultados. Aquí presentamos ejemplos de los siguientes operadores aritméticos en R: sustracción (-), multiplicación (*), división (/) y exponencial (^). Usted también puede escribir sus propios ejemplos para practicarlos. Cabe señalar que R sigue la convención matemática habitual del orden de las operaciones. Por ejemplo, la expresión $2 + 3 * 4$ es interpretada para obtener el valor $2 + (3 * 4) = 14$, no $(2 + 3) * 4 = 20$.

```
# 2. Operadores ####  
# Operadores aritméticos  
# Sustracción  
2 - 3  
[1] -1  
  
# Multiplicación  
2 * 3  
[1] 6  
  
# División  
30 / 5  
[1] 6  
  
# Exponencial  
2^3  
[1] 8
```

Operadores relacionales (de comparación)

R también tiene operadores que nos ayudan a conocer cómo un objeto o valor se relaciona con otro. Por ejemplo, en una base de casos notificados de una enfermedad de notificación obligatoria, podemos verificar si la fecha de notificación es posterior a la fecha de inicio de los síntomas con operadores relacionales y, así, explorar posibles inconsistencias en los registros.

Cuando usamos un operador relacional, el resultado (salida, *output*) presentado en la *console* es TRUE o FALSE, que significa verdadero y falso en español, respectivamente. Ejecute los siguientes códigos en su computadora y compare las respuestas. En él presentamos los operadores iguales a (`==`), diferente de (`!=`), más grande que (`>`), menor que (`<`), mayor o igual (`>=`) y menor o igual (`<=`). ¿Los resultados tienen sentido para usted?

Para todos los operadores relacionales, usted puede pensar como si estuviera preguntándole algo a R. Por ejemplo, el resultado de la expresión `"A" == "a"` es FALSE, porque R distingue entre mayúsculas y minúsculas, por lo que las letras se entienden como diferentes. ¡ATENCIÓN! En este operador, tenga en cuenta el doble signo de igual `'=='`, que no debe confundirse con el signo de `"="`, utilizado para la asignación, como veremos más adelante.

```
# 2. Operadores ####
# Operadores relacionales (de comparación)

# Igual a
"A" == "a"
[1] FALSE

# Diferente de
2 != 0
[1] TRUE

# Mayor que
4 > 2
[1] TRUE

# Menor que
4 < 2
[1] FALSE

# Mayor o igual
6 >= 4
[1] TRUE
```



```
# Menor o igual
4 <= 6
[1] TRUE
```

Operadores lógicos

R también tiene operadores que permiten trabajar con múltiples condiciones relacionales en la misma expresión. Dichos operadores también devuelven valores lógicos (TRUE o FALSE). Vea en el código a continuación los siguientes operadores: y (&), que devuelve el valor verdadero solo cuando se cumplen ambas condiciones; o (|), que devuelve el valor verdadero cuando se cumple al menos una de las dos condiciones; y un operador de negación (!), que devuelve el resultado lógico opuesto de la expresión.

Por ejemplo, el resultado de la expresión `2<3 & 2<1` es FALSE, ya que solo la primera parte de la expresión es verdadera (2 es menor que 3, pero 2 no es menor que 1). En el caso de la expresión `2<3 | 2<1`, el resultado es TRUE, porque el operador “o” busca que al menos una de las condiciones sea verdadera (en el ejemplo, 2 es menor que tres y esto es suficiente, aunque la segunda parte de la expresión es falsa). Finalmente, observe que el resultado de la expresión `!(1 == 1)`, es FALSE. Esto porque el resultado de la expresión `1 == 1` sería verdadero (1 es igual a 1), pero el operador devuelve el resultado lógico opuesto de la expresión.

```
# 2. Operadores ####
# Operadores lógicos
# Y
2<3 & 2>1
[1] TRUE

2<3 & 2<1
[1] FALSE

# O
2<3 | 2<1
[1] TRUE
```

```
# Negación
```

```
1 == 1
```

```
[1] TRUE
```

```
!(1 == 1)
```

```
[1] FALSE
```

Hasta ahora, cuando ejecutamos los códigos de la clase, como hicimos anteriormente, la salida (el resultado) del código solo se mostró en la *console*. A partir de ahora, introduciremos también el concepto de objeto, que puede ser útil para almacenar las salidas de su código de numerosas maneras.

```
> 3
```

```
> Objetos
```

En el corazón de casi todos los comandos que usted ejecuta en R está el concepto de que todo es un objeto. Estos objetos pueden ser de diferentes naturalezas, desde un solo número o una cadena de caracteres (como una palabra), hasta estructuras altamente complejas, como la salida de un gráfico, un resumen de su análisis estadístico o un conjunto de comandos R que realizan una tarea específica. Por ejemplo, podemos importar una lista de casos de una investigación epidemiológica y almacenar este conjunto de datos como un objeto en R.

Crear objetos

Comprender cómo crear y asignar valores a los objetos es una de las claves más importantes para comprender cómo funciona el lenguaje R. Para crear un objeto, simplemente escriba un nombre para él (de acuerdo con su preferencia) y luego asígnele un valor a ese objeto utilizando el operador de asignación `<-`. En el siguiente código, creamos un objeto llamado "a" y le asignamos la operación $2 + 2$ usando el operador `<-`. Podemos leer este código como: "el objeto a recibe el valor de la suma 2 más 2" o "el valor de la suma 2 más 2 se asigna al objeto a" o incluso "a se define como 2+2".

Al ejecutar el comando `a <- 2 + 2`, vea que el objeto llamado "a" apareció como un valor en el panel *environment* (Figura 3), que es el entorno de trabajo. Sin embargo, no se presentaron resultados en la *console*, ya que no pedimos que se presentara. Para visualizar (o imprimir) el valor del objeto, simplemente escriba su nombre en el *script* (panel editor) o *console* y ejecute como un comando. De esa manera, al recibir un nombre, se podrá hacer referencia al objeto en comandos posteriores.

```
# 3. Objetos ####  
# Crear un objeto con operador <-  
a <- 2 + 2  
  
# Ejecutar el objeto creado  
a  
[1] 4
```

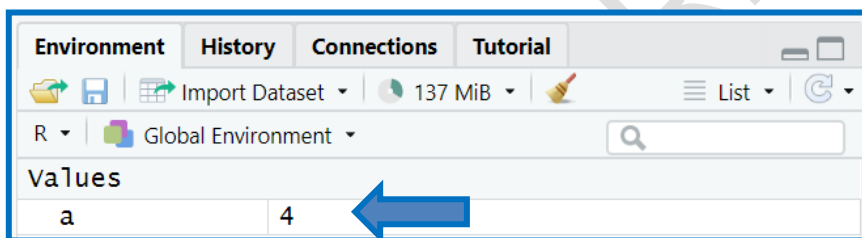



Figura 3: Objeto a de tipo valor, creado con la expresión `a <- 2 + 2`.

Hay muchos tipos diferentes de valores que podemos asignar a un objeto. Por ejemplo, en el siguiente comando creamos un objeto llamado b y le asignamos la expresión "Quiero aprender a usar R", que es una cadena de caracteres. Tenga en cuenta que colocamos el texto entre comillas, ya que así es como R entiende los textos. Si olvida usar las comillas, recibirá un mensaje de error, además de ver el símbolo  a la izquierda del número de la línea del código, como se muestra en la Figura 4.

```
# 3. Objetos ####  
# Asignar una cadena de caracteres y ejecutar el objeto  
b <- "Quiero aprender a usar R"
```

```
b <- Quiero aprender a usar R
b
[1] "Quiero aprender a usar R"
```

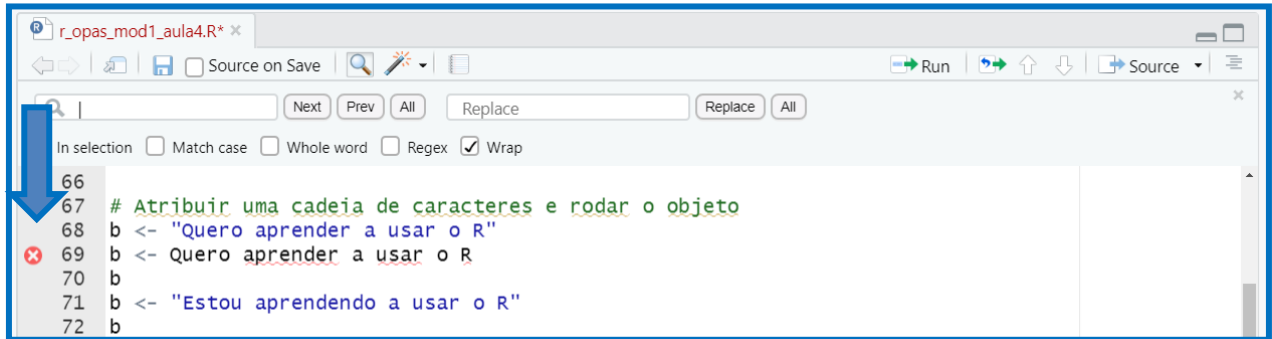


Figura 4: Use comillas para asignar cadenas de caracteres (textos) a un objeto.

El orden en que se crean los objetos marca la diferencia, ya que un objeto se sobrescribe fácilmente. Por ejemplo, con el siguiente código sobrescribimos el objeto `b`, que ahora almacena el valor "Estoy aprendiendo a usar R". Es decir, para cambiar el valor de un objeto existente, simplemente le asignamos un nuevo valor.

```
# 3. Objetos ####
# Asignar una cadena de caracteres y ejecutar el objeto
b <- "Estoy aprendiendo a usar R"
b
[1] "Estoy aprendiendo a usar R"
```

Todos los objetos creados se almacenan en el panel *environment* y se muestran como una lista. En ese momento, nuestro *environment* contiene los dos objetos que hemos creado hasta ahora: `a` y `b`, como en la Figura 5. Si usted hace clic en la flecha hacia abajo junto al botón *List* en el mismo panel y cambia a la vista *Grid*, RStudio mostrará un resumen de los objetos (Figura 6), incluido el tipo (clase) del objeto (*type*), la longitud (*length*), su tamaño "físico" (*size*, que se refiere a la ocupación de la memoria de la computadora) y su valor (*value*).

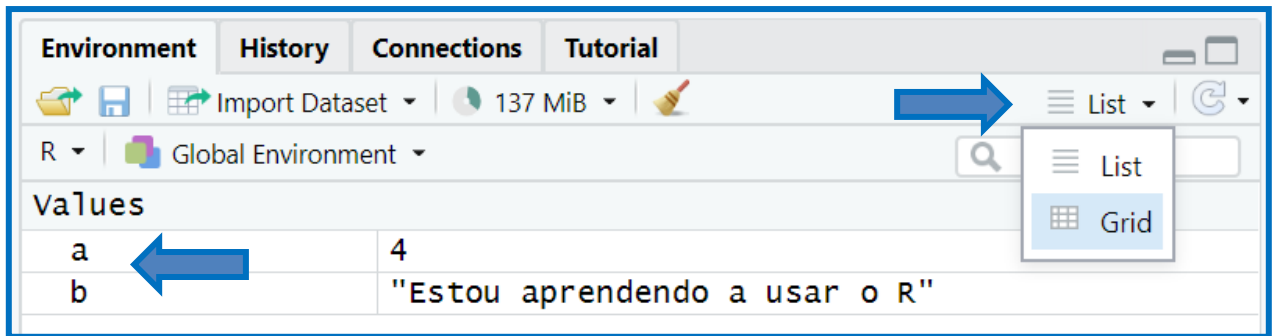


Figura 5: Todos los objetos creados se almacenan en el panel *environment*.

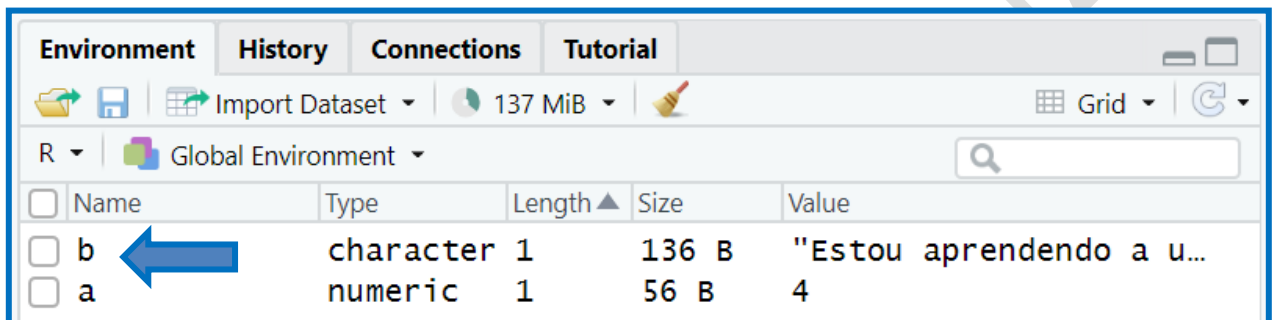


Figura 6: Panel *environment* en modo de vista *Grid*.

Una vez que creamos objetos, podemos realizar operaciones con ellos. Por ejemplo, el siguiente código crea un objeto llamado `c` y le asigna la suma del objeto `a` con el número 5. El código funciona porque ambos son numéricos. Si usted intenta hacerlo con objetos con texto (clase de caracteres), se mostrará un mensaje de error. El mensaje de error informa que uno o ambos objetos no son un número y, por lo tanto, no se pueden sumar.

3. Objetos

Realizar operaciones con objetos

```
c <- a + 5
```

```
texto1 = "Buenos"
```

```
texto2 = "días"
```

```
texto3 <- texto1 + texto2
```

```
## Error in texto1 + texto2: non-numeric argument to binary operator
```

En el código anterior creamos los objetos texto1 y texto2 sin usar el símbolo <-. De hecho, la creación de objetos también se puede realizar con el símbolo = como operador de asignación. Pero desaconsejamos el uso de esta notación, especialmente para evitar confusiones con el operador == (igual a) que vimos en la sección de operadores.

Nombrar objetos

Al analizar datos, es importante tener una visión muy clara de los datos que estamos utilizando. Entender a qué se refiere cada elemento es, por lo tanto, fundamental.

Por ello, se recomienda que los nombres de los objetos sean cortos e informativos, lo que no siempre es fácil. Cuando sea necesario crear objetos con más de una palabra en sus nombres, le sugerimos que use un subrayado (*underline* "_") prioritariamente, pero también es posible utilizar un punto o una letra mayúscula entre las palabras, entre otros. Vea algunos ejemplos en el código a continuación.

```
# 3. Objetos ####  
# Cómo nombrar un objeto  
casos_confirmados <- 569  
casos.notificados <- 1058  
casosDescartados <- 67
```

También existen algunas limitaciones y recomendaciones a la hora de nombrar los objetos. Ejecute los códigos a continuación y vea qué sucede.

```
# 3. Objetos ####  
# Cómo no podemos nombrar objetos  
5 <- 50  
Error in 5 <- 50 : invalid (do_set) left-hand side to assignment  
  
5 <- "cinco"
```

```
Error in 5 <- "cinco" : invalid (do_set) left-hand side to assignment

NA <- 2+2
Error in NA <- 2 + 2 : invalid (do_set) left-hand side to assignment

TRUE <- 2+2
Error in TRUE <- 2 + 2 : invalid (do_set) left-hand side to assignment

FALSE <- 2+2
Error in FALSE <- 2 + 2 : invalid (do_set) left-hand side to assignment
```

Estos errores ocurrieron porque los nombres que se crearon no están permitidos. Para elegir los nombres de sus objetos, considere las siguientes sugerencias y reglas:

- evite usar letras mayúsculas para no tener que cambiar entre mayúsculas y minúsculas numerosas veces mientras esté programando;
- evite usar punto, ya que en el futuro puede confundirse con extensiones (por ejemplo, si exporta el objeto a un archivo en su computadora usando un nombre con punto);
- no se permite comenzar con un número o un punto seguido de un número;
- evite el uso de caracteres no alfanuméricos (&, ^, /, !, etc.);
- asegúrese de no nombrar sus objetos con palabras reservadas (por ejemplo: TRUE, NA);
- nunca es una buena idea darle a su objeto el mismo nombre que una función de R.

Eliminar objetos

Para eliminar un objeto creado, usted puede utilizar la presentación de *Grid* del *environment* que aprendimos antes (Figura 6), marcar las casillas de selección de los

objetos que desea eliminar y hacer clic en la escoba en este panel (Figura 7). También puede utilizar la función `rm()`, del paquete `base`, para eliminar uno o más objetos al mismo tiempo, como en el código siguiente. Las letras “r” y “m” son una referencia al término *remove*, que en español significa "eliminar". En la siguiente sección de esta clase, profundizaremos en el concepto de funciones.

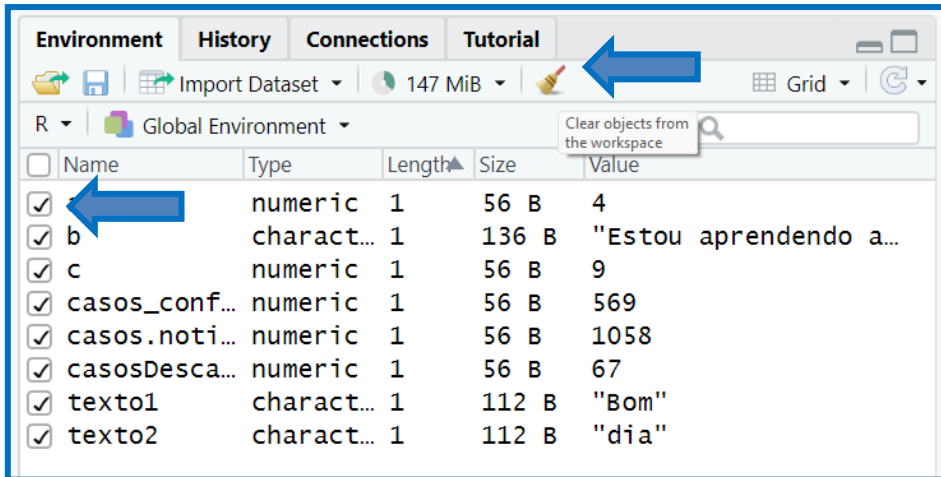


Figura 7: Eliminación de objetos del panel *environment* con modo de vista *Grid*.

3. Objetos

Para eliminar un objeto

```
rm(a)
```

Para eliminar más de un objeto

```
rm(b, c, texto1, texto2)
```

> 4

> Funciones

Hasta ahora hemos creado objetos simples asignándoles directamente un valor único. Es muy probable que pronto usted desee avanzar hacia la creación de objetos más complicados a medida que aumente su experiencia y la complejidad de sus tareas. Afortunadamente, R tiene una gran cantidad de funciones para ayudarle a hacerlo.

Podemos pensar en las funciones como instrucciones para realizar acciones con nuestros datos. Ellas son el corazón del trabajo en R: cómo realizamos tareas y

operaciones específicas. En otras palabras, son códigos que aceptan *inputs* (entradas) y devuelven un *output* (salida) transformado.

Al final de la última sección (3. Objetos), vimos la función `rm()` como una alternativa para eliminar objetos creados. Ahora presentamos la función `c()` del paquete `base`, como una referencia al término concatenar. Se utiliza para recopilar una serie de valores y almacenarlos en una estructura de datos llamada vector, que es un tipo de objeto en R. En el código a continuación, creamos un objeto llamado `vector` y le asignamos la función `c()` con diversos números. Ejecute las dos líneas de comando a continuación y observe la salida que se muestra en la *console*.

```
# 4. Funciones ####  
# Funcione c() para concatenar  
vector <- c(10,12,15,18,14,12,16,18,17,19,18,15,16,14,18)  
  
vector  
[1] 10 12 15 18 14 12 16 18 17 19 18 15 16 14 18
```

Hay algunos puntos realmente importantes para tener en cuenta aquí:

- Cuando usted usa una función en R, el nombre de la función siempre va seguido de dos paréntesis, incluso si no hay nada entre los paréntesis;
- Los argumentos de una función se colocan entre los paréntesis y se separan por comas. Usted puede pensar en un argumento como partes de la instrucción realizada por la función, una forma de personalizar su uso o comportamiento. En el ejemplo anterior, los argumentos son los números que queremos concatenar.
- Una de las cosas complicadas al comenzar a usar R es saber qué función usar para una tarea específica y cómo usarla. Afortunadamente, cada función siempre tendrá un documento de ayuda asociado, que explicará cómo usar la función. Para acceder a él, abra el panel *Help* e ingrese el nombre de la función (Figura 8). *Help* significa ayuda en español. También puede acceder a la documentación de la función escribiendo su nombre precedido por un signo de interrogación (intente escribir `?c` en la *console* para abrir la documentación de `c()`). Además, una búsqueda rápida en Internet siempre ayuda.

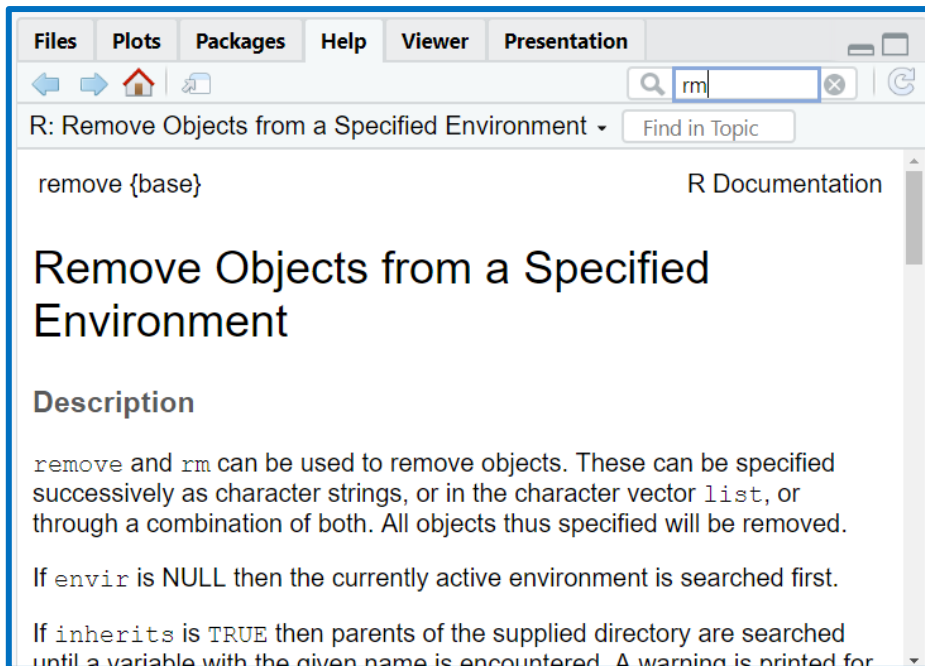


Figura 8: Panel Help con documentación de la función `rm()`.

Funciones estadísticas

Ahora que hemos creado un vector, podemos usar otras funciones para realizar tareas con ese objeto. Por ejemplo, podemos calcular la suma de los elementos de este vector. En el siguiente código presentamos la función `sum()` del paquete base, destinada a calcular la suma de datos numéricos. Ejecute los comandos y observe que al usar la función `sum()` llegamos al mismo resultado que cuando usamos el operador de suma que vimos arriba. Tenga en cuenta que, al escribir funciones, de alguna manera reducimos nuestros códigos, porque usamos las instrucciones que trajeron en sus configuraciones.

```
# 4. Funciones ####  
# Funciones estadísticas  
# Suma  
sum(vector)  
[1] 232
```

```
10 + 12 + 15 + 18 + 14 + 12 + 16 + 18 + 17 + 19 + 18 + 15 + 16 + 14 + 18
```

```
[1] 232
```

Existen numerosas funciones de estadística descriptiva e inferencial que podemos utilizar en R en el paquete base. En el código a continuación, presentamos algunas que seguramente serán útiles para explorar, describir o modelar sus datos. Son ellas: función `length()` para número de elementos; `min()` para el valor mínimo de los datos; `max()` para el valor máximo; `range()` para el intervalo (amplitud) de los datos; `mean()` para calcular la media; `var()` para la varianza; `sd()` para desviación estándar; `sqrt()` para raíz cuadrada; y `quantile()` para cuantiles / percentiles.

En el futuro, usted podrá usar estas funciones para describir sus datos, como mostrar la edad mediana de los casos notificados o el tiempo máximo de hospitalización de los casos que evolucionaron a muerte.

4. Funciones

Número de elementos

```
length(vector)
```

```
[1] 15
```

Mínimo

```
min(vector)
```

```
[1] 10
```

Máximo

```
max(vector)
```

```
[1] 19
```

Intervalo

```
range(vector)
```

```
[1] 10 19
```

Media

```
mean(vector)
```

```
[1] 15.46667
```

```

# Varianza
var(vector)
[1] 7.12381

# Desviación estándar
sd(vector)
[1] 2.669047

# Cuantiles / Percentiles
quantile(c(10,12,15,18,14,12,16,18,17,19,18,15,16,14,18),
        probs = c(0.1, 0.3, 0.5, 0.8))
10%  30%  50%  80%
12.0 14.2 16.0 18.0

quantile(vector,
        probs = c(0.1, 0.3, 0.5, 0.8))
10%  30%  50%  80%
12.0 14.2 16.0 18.0

```

Observe en el código anterior que informamos el argumento `probs =` para informar qué percentiles nos gustaría presentar con la función `quantile()`. Observe también cómo usamos la función `c()` dentro de la función `quantile()`. Las funciones de anidamiento (una dentro de la otra) nos permiten construir comandos bastante complejos dentro de una sola línea de código y es una práctica muy común cuando se usa R.

Sin embargo, se debe tener cuidado, ya que demasiadas funciones anidadas pueden hacer que su código sea bastante difícil de entender. Para mejorar la legibilidad, podemos escribir el código separando explícitamente cada paso diferente del proceso, como utilizar el vector creado previamente. Otro consejo para que su código sea más fácil de leer es usar una línea para cada nuevo argumento, como hicimos en la función `quantile()`. Todos los enfoques arrojarán el mismo resultado, usted solo necesita usar su propio juicio para determinar cuál es más legible.

Funciones de redondeo

A continuación, presentamos algunas funciones de redondeo del paquete base. Son ellas: `round()` para redondear al número deseado de decimales; `ceiling()` para redondear al número entero más cercano y mayor; y `floor()` para redondear al número entero más cercano y menor. En todas ellas informamos un argumento numérico y en la función `round()` también informamos el argumento `digits =` para definir el número deseado de decimales. En el futuro, usted puede utilizar estas funciones para redondear los indicadores calculados, como coeficientes de incidencia y prevalencia.

```
# 4. Funciones ####  
# Redondear a dos decimales  
round(10.258, digits = 2)  
[1] 10.26  
  
# Entero hacia arriba  
ceiling(10.258)  
[1] 11  
  
# Entero hacia abajo  
floor(10.258)  
[1] 10
```

> 5

> Paquetes

La instalación básica de R viene con muchos paquetes útiles predeterminados, que contienen muchas de las funciones que usamos a diario, lo que denominamos paquete base. Sin embargo, a medida que usted comience a usar R para proyectos más diversos (y a medida que evolucione su propio uso de R), llegará un momento en el que necesitará ampliar los recursos disponibles.

Afortunadamente, miles de usuarios de R han desarrollado códigos útiles y los han compartido como paquetes instalables. Usted puede pensar en un paquete como una colección de funciones, datos y archivos de ayuda reunidos en una estructura estándar bien definida que puede descargar e instalar en R. Por ejemplo, a lo largo de este curso usaremos el paquete `tidyverse`, que es un paquete general con numerosas herramientas para la ciencia de datos, y el paquete `lubridate` para el manejo de variables de tipo fecha.

Estos paquetes se pueden descargar de varias fuentes, pero las más populares son:

- CRAN (The Comprehensive R Archive Network) - repositorio central de R;
- GitHub: plataforma de alojamiento de código fuente y archivos con control de versión;
- Bioconductor: proporciona paquetes orientados a la bioinformática, más específicamente para el análisis de datos genómicos;
- RForge: plataforma central para el desarrollo colaborativo de paquetes.

Instalar paquetes de CRAN

Para usar un paquete, primero debe descargarlo a su biblioteca de paquetes. La biblioteca es una carpeta en su computadora, pero usted puede manejarla a través de RStudio. Abra el panel *Packages* (Figura 9) y haga clic en el botón *Install*, en la parte superior izquierda. Se abrirá una nueva ventana, donde usted podrá buscar el nombre del paquete que desea instalar. Necesitará una conexión a Internet para instalar paquetes y actualizarlos.



Figura 9: Botón *Install* para instalar paquetes desde el panel *Packages*.

Como en la Figura 10, escriba `pacman`, seleccione el paquete, mantenga activada la casilla de selección *Install dependencies* y haga clic en el botón *Install*. "Pacman" es una abreviatura de "*package manager*". Es un paquete que ofrece funciones útiles para trabajar con otros paquetes, como veremos a continuación. Luego vea el código que se ejecutó en la *console*.

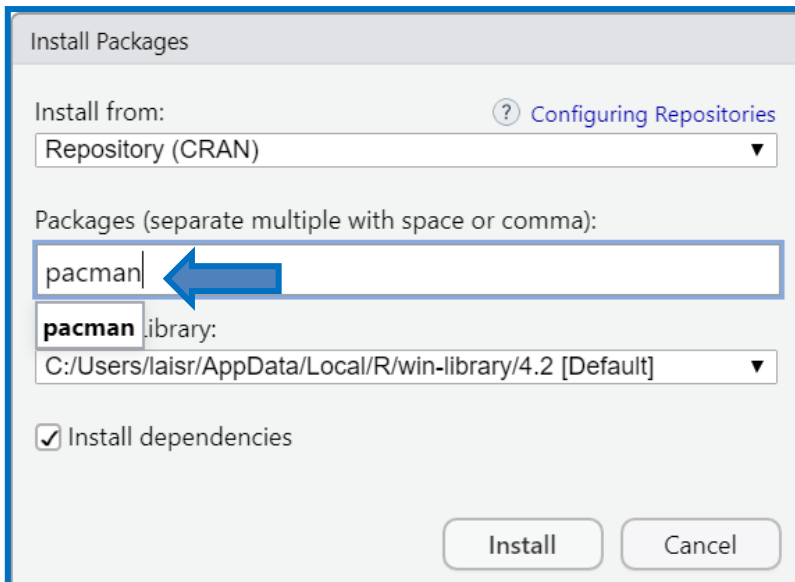


Figura 10: Ventana *Install Packages* para especificar los paquetes que se instalarán.

5. Paquetes

Instalar un paquete de CRAN

```
install.packages("pacman")
```

```
Installing package into 'C:/Users/laisr/AppData/Local/R/win-library/4.2'  
(as 'lib' is unspecified)
```

```
trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.2/pacman_0.5.1.zip'
```

```
Content type 'application/zip' length 389140 bytes (380 KB)
```

```
downloaded 380 KB
```

```
package 'pacman' successfully unpacked and MD5 sums checked
```

```
The downloaded binary packages are in
```

```
C:\Users\laisr\AppData\Local\Temp\RtmpGuE5Nd\downloaded_packages
```

En el código anterior, observe que usted puede instalar un paquete de CRAN usando la función `install.packages()` del paquete base. La función permite instalar

diferentes paquetes al mismo tiempo y el argumento `dependencies = TRUE` asegura que también se instalarán los paquetes adicionales necesarios para usar el paquete de interés. Por ejemplo, en el siguiente código instalamos al mismo tiempo los paquetes `lubridate` y `tidyverse`, mencionados anteriormente, con sus respectivas dependencias.

5. Paquetes

Instalación de múltiples paquetes:

```
install.packages('lubridate', 'tidyverse', dependencies = TRUE)
```

also installing the dependencies 'rex', 'covr'

```
trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.2/rex_1.2.1.zip'
Content type 'application/zip' length 126476 bytes (123 KB)
downloaded 123 KB
```

```
trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.2/covr_3.5.1.zip'
Content type 'application/zip' length 297893 bytes (290 KB)
downloaded 290 KB
```

```
trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.2/lubridate_1.8.0.zip'
Content type 'application/zip' length 1303823 bytes (1.2 MB)
downloaded 1.2 MB
```

package 'rex' successfully unpacked and MD5 sums checked

package 'covr' successfully unpacked and MD5 sums checked

package 'lubridate' successfully unpacked and MD5 sums checked

The downloaded binary packages are in

C:\Users\laisr\AppData\Local\Temp\RtmpiU17yw\downloaded_packages

Instalar paquetes de GitHub

Hay varias opciones para instalar paquetes alojados en GitHub. Quizás el método más eficiente es usar la función `instalar_github()` del paquete `remotes`. Antes de

usar la función, usted deberá conocer el nombre de usuario de GitHub del propietario del repositorio y el nombre del repositorio. Por ejemplo, acceda al sitio del paquete geobr en GitHub: <https://github.com/ipeaGIT/geobr>. El paquete fue desarrollado por el Instituto de Investigación Económica Aplicada e incluye una amplia gama de datos geoespaciales que trataremos en el último módulo de este curso. Identifique el nombre de usuario y el nombre del paquete en la página (ipeaGIT/Geobr). Para instalar esta versión del paquete desde GitHub, use:

5. Paquetes

Instalar paquetes de GitHub

Instalar el paquete geobr

```
remotes::install_github('ipeaGIT/geobr', subdir = "r-package")
```

```
Downloading GitHub repo ipeaGIT/geobr@HEAD
```

```
✓ checking for file
```

```
'C:\Users\laisr\AppData\Local\Temp\RtmpiU17yw\remotes55e865751cec\ipeaGIT-geobr-c52e73d\r-package/DESCRIPTION' ...
```

```
– preparing 'geobr': (986ms)
```

```
✓ checking DESCRIPTION meta-information ...
```

```
– checking for LF line-endings in source and make files and shell scripts
```

```
– checking for empty or unneeded directories
```

```
– building 'geobr_1.6.5999.tar.gz'
```

```
Installing package into 'C:/Users/laisr/AppData/Local/R/win-library/4.2'  
(as 'lib' is unspecified)
```

```
* installing *source* package 'geobr' ...
```

```
** using staged installation
```

```
** R
```

```
** data
```

```
*** moving datasets to lazyload DB
```

```
** inst
```

```
** byte-compile and prepare package for lazy loading
```

```
** help
```

```
*** installing help indices
```

```
*** copying figures
```

```
** building package indices
** installing vignettes
** testing if installed package can be loaded from temporary location
** testing if installed package can be loaded from final location
** testing if installed package keeps a record of temporary installation path
* DONE (geobr)
```

En el código anterior, usamos la función `install_github()` del paquete `remotes` sin cargarlo de forma previa. A veces puede ser útil usar una función sin cargar su paquete, especialmente cuando las bibliotecas tienen los mismos nombres de función. Si usted, por ejemplo, sólo está utilizando una o dos funciones en su *script* y no desea cargar todas las demás funciones en un paquete, puede acceder a la función deseada especificando el nombre del paquete seguido de dos puntos y luego el nombre de la función, como en el ejemplo anterior. Vea también que el argumento `subdir` = fue especificado para encontrar la carpeta en Github referida al paquete `geobr`.

Cargar paquetes

Una vez que instale un paquete en su computadora, él no estará inmediatamente disponible para su uso. Para usar un paquete, usted debe cargarlo. Usted puede pensar en esta acción como si fuera un enchufe: aunque estén instalados, los paquetes deben cargarse para que se activen sus funcionalidades.

Si intenta usar una función sin cargar primero su paquete, recibirá un mensaje de error que le informará que R no pudo encontrar la función. Por ejemplo, si intenta utilizar la función `install_github()` sin cargar el paquete `remotes` primero, recibirá el siguiente mensaje de error:

```
# 5. Paquetes ####
# Recuerde cargar los paquetes
install_github('ipeaGIT/geobr', subdir = "r-package")
Error in install_github("ipeaGIT/geobr", subdir = "r-package") :
  could not find function "install_github"
```

Existen numerosas formas de cargar (activar) sus paquetes. Le sugerimos que utilice la función `pacman::p_load()`, pues ella instala y carga paquetes al mismo

tiempo. Con ella, su código será más ágil, ya que no será necesario instalar y luego cargar sus paquetes. También puede utilizar la función `library()` del paquete base para ejecutar solo la función de cargar, pero no permite cargar múltiples paquetes y su código será más largo.


Por ejemplo, en el siguiente código, la función `p_load()` reemplaza las cuatro líneas de comando para instalar y activar los paquetes `rio` y `ggspatial`. Los paquetes se utilizarán en las próximas clases. Están destinados respectivamente a la importación y exportación de bases de datos y el manejo y confección de mapas.

```
# 5. Paquetes ####  
# Cargar paquetes  
# Instalar y cargar con pacman  
pacman:: p_load(rio, ggspatial)  
  
# Comparar con pasos secuenciales  
install.packages("rio")  
install.packages("ggspatial")  
library(rio)  
library(ggspatial)
```

Es importante observar que cada vez que inicia una nueva sesión de R (o restaura una sesión guardada previamente), usted debe cargar los paquetes que usará. Por ello, a partir de ahora todas las clases del curso incluirán un código inicial con los paquetes utilizados en cada clase. Usualmente ponemos el código con la carga de los paquetes necesarios para nuestro análisis al comienzo de nuestros *scripts*, para que sean fáciles de acceder y agregar a medida que se desarrolla nuestro código. Nótese también que usaremos dos puntos siempre que sea interesante especificar el paquete al que pertenece la función utilizada, ya sea en el código o en el texto de las clases.

Actualizar paquetes

Es una buena práctica actualizar ocasionalmente sus paquetes instalados para obtener acceso a nuevas funcionalidades y correcciones de *bugs*. Para actualizar los paquetes de CRAN, usted puede acceder al panel *Packages*, marcar los paquetes que

desea actualizar y hacer clic en el botón  **Update**. También puede utilizar la función `update.packages()` del paquete base para actualizarlos todos a la vez, como en el código a continuación. El argumento `ask = FALSE` evita que usted tenga que confirmar cada descarga de paquete, pues esto puede ser un problema si hay muchos paquetes instalados.

```
# 5. Paquetes ####  
# Actualizar paquetes  
update.packages(ask = FALSE)
```

La forma más segura (que conocemos hasta ahora) de actualizar un paquete instalado de GitHub es simplemente reinstalarlo usando el comando de instalación que vimos anteriormente.

```
> 6  
> Clases
```

Comprender los diferentes tipos de datos y cómo R los maneja es importante para comprender cómo él funciona e identificar posibles errores. A continuación, presentamos las principales clases de datos y objetos en R.

Numérico (numeric)

Representa todos los números reales con o sin valores decimales. Por ejemplo, en el siguiente código asignamos el valor `63.5` (con decimal) al objeto `peso` y luego usamos la función `class()` del paquete base para identificar la clase del objeto. Hicimos lo mismo con el objeto `altura`, esta vez asignando un valor entero. En ambos ejemplos, recibimos como *output* el término `numeric`. ¡Atención! Tenga en cuenta que R utiliza puntos en lugar de comas para definir decimales.

```
# 6. Clases ####
```

```
# Numérico
peso <- 63.5
peso
[1] 63.5

class(peso)
[1] "numeric"

altura <- 182
altura
[1] 182

class(altura)
[1] "numeric"
```

Entero (integer)

Clase específica para valores reales sin puntos decimales. Necesitamos usar el sufijo `L` para crear explícitamente datos enteros. Observe en el siguiente ejemplo que la clase del objeto `var_int` arrojó como resultado `integer`.

```
# 6. Clases ####
# Entero
var_num <- 42
var_int <- 42L

class(var_num)
[1] "numeric"

class(var_int)
[1] "integer"
```

Carácter (character)

Clase de objetos con valores de caracteres (`character`) o cadena de caracteres (`string`). Tenemos que usar comillas para explícitamente crear la clase de `character`. R

acepta comillas simples (' ') o comillas dobles (" "). En el código a continuación, observe que, si ponemos números entre comillas, ellos también se entenderán como `character`.

```
# 6. Clases ####  
# Carácter  
imc <- c("bajo peso", "sobrepeso", 'obesidad')  
imc  
[1] "bajo peso" "sobrepeso" "obesidad"  
  
class(imc)  
[1] "character"  
  
imc <- c("< 18,5", "18,5 - 24,9", "≥ 30")  
imc  
[1] "< 18,5" "18,5 - 24,9" "≥ 30"  
  
class(imc)  
[1] "character"
```

Fecha (date)

Clase de objetos con valores referidos a fechas, con día, mes y año. Es importante que usted comprenda que R almacena las fechas como números, aunque usted las visualice de la manera como imaginamos tradicionalmente las fechas. Este número representa la cantidad de días desde su fecha de "origen" (1 de enero de 1970). Esto permite que R trate las fechas como variables continuas y, así, realice operaciones especiales, como el cálculo de la distancia entre las fechas.

Observe el código a continuación. Usamos la función `lubridate::today()` para asignar la fecha del sistema al objeto de hoy, que se creó como un objeto de clase `Date`. Luego usamos la función `as.numeric()` del paquete base para mostrar esa fecha en la *console* como un número. Esta y otras funciones de reclasificación se verán en lecciones posteriores del curso. El resultado obtenido se refiere al número de días transcurridos entre el 01/01/1970 y la fecha actual.

```
# 6. Clases ####  
# Fecha  
# Fecha del sistema:  
hoy <- today()  
hoy  
[1] "2022-07-25"  
  
class(hoy)  
[1] "Date"  
  
as.numeric(hoy)  
[1] 19204
```

También tenga en cuenta que R muestra las fechas en formato AAAA-MM-DD (año-mes-día) de forma predeterminada. Esto no significa que sus informes se exportarán con esta configuración, ya que el formato de presentación y almacenamiento son aspectos diferentes para R. Cabe recalcar, además, que existen diferentes formas de almacenar y manejar variables de tipo fecha en R, como `POSIXt`, que almacena fecha y hora al mismo tiempo.

Factor

Un factor es una clase de objeto que se utiliza para trabajar con datos categóricos que tienen un orden. Por ejemplo, cuando trabajamos con los grupos de edad de una lista de casos, sabemos que hay un orden en que los ancianos son mayores que los adultos, quienes a su vez son mayores que los niños.

Supongamos que un vector de nombre `faixet` solo puede contener los siguientes valores de grupo de edad: 0 a 20, 21 a 40, 41 a 60 y 60 o más. En este caso, conocemos de antemano los posibles valores y estos valores predefinidos y distintos se denominan niveles (categorías) de un factor. En R usamos la función `factor()` del paquete base para crear un factor. Una vez que se crea un factor, él solo puede contener niveles predefinidos.

En el siguiente código, observe que creamos el `faixet` tres veces. Observe primero los dos primeros, como factor y como carácter. Tenga en cuenta que invertimos

el orden de los grupos de edad, colocando primero a los más mayores. Vea también que, al definir un factor, R definirá los niveles en orden alfabético o ascendente de forma predeterminada (observe los `levels` cuando ejecutamos el nombre del objeto `faixet` solo). Si está interesado en definirlo de manera diferente, puede usar los argumentos `levels=` y `labels=`, como en el tercer ejemplo.

```
# 6. Clases ####  
# Factor  
# Como factor, sin especificar los niveles  
faixet <- factor(c("21 a 40", "41 a 60", "61 o más"))  
faixet  
[1] 21 a 40    0 a 20    41 a 60    61 o más  
Levels: 0 a 20 21 a 40 41 a 60 61 o más  
  
class(faixet)  
[1] "factor"  
  
# Como carácter  
faixet <- c("61 o más", "0 a 20", "21 a 40", "41 a 60")  
faixet  
[1] "61 o más" "0 a 20"   "21 a 40"   "41 a 60"  
  
class(faixet)  
[1] "character"  
  
# Como factor, definiendo niveles  
faixet <- factor(faixet,  
                 levels = c("61 o más", "41 a 60", "21 a 40", "0 a 20"),  
                 labels = c("61 o más", "41 a 60", "21 a 40", "0 a 20"))  
faixet  
[1] "61 o más" "0 a 20"   "21 a 40"   "41 a 60"
```


Lógico (logical)

Clase que toma solo dos valores: TRUE o FALSE (verdadero o falso). Esta clase se puede utilizar para variables dicotómicas, como la presencia de signos y síntomas o la presencia de algún comportamiento, como en el caso del tabaco. Los valores lógicos también se pueden definir con letras mayúsculas T y F.

```
# 6. Clases ####  
# Lógico  
tabaco <- c(TRUE, FALSE, F, T)  
tabaco  
[1] TRUE FALSE FALSE TRUE  
  
class(tabaco)  
[1] "logical"
```

A veces, una columna se convertirá automáticamente en una clase diferente. Esto se llama coerción, que es la homogeneización de la clase de los elementos (valores) de un vector. Si construimos un vector con valores de diferentes clases, todos los valores serán reprimidos por la clase más dominante entre ellos. En el siguiente ejemplo, todos los valores del vector se han transformado en `character`, porque el carácter es más dominante que las otras clases presentes (numérica y lógica).

```
# 6. Clases ####  
# Cuidado con la coerción  
a <- c(1, -3)  
class(a)  
[1] "numeric"  
  
a <- c(1, -3, "a", TRUE)  
class(a)  
[1] "character"
```

Durante esta clase mencionamos algunas veces la palabra *script* y usted incluso siguió esta clase a través de uno de ellos. ¿Pero, qué son? Los *scripts* son archivos de texto que contienen sus comandos (limpieza de datos, visualizaciones...). Hay algunas ventajas de almacenar y ejecutar sus comandos desde un *script* en lugar de simplemente ejecutarse en el panel *console*. Entre ellas destacamos la portabilidad, la reproducibilidad y el control de versiones. Análisis bien documentados en *scripts* favorecen el trabajo en equipo, ya que todos(as) podrán seguir exactamente cómo se realizó el análisis o los posibles cambios que ocurran.

Un *script* de R tiene la extensión “.R” (final de archivo). En la primera sección de esta clase vimos cómo abrir un *script* aparte del menú *File*, pero también se puede abrirlo directamente desde la carpeta de su computadora en la que se guardó el *script*. Por ejemplo, intente abrir la carpeta `r_opas/scripts`, elegir un *script* diferente de esta clase y hacer doble clic para abrir el archivo. Tenga en cuenta que Windows abrirá este archivo directamente en RStudio.

Al abrir *scripts* esta manera (haciendo doble clic desde la carpeta en su computadora) con RStudio cerrado, automáticamente el directorio de trabajo se definirá como la carpeta del *script* abierto. Dependiendo de la configuración de su computadora, también es posible que Windows abra su *script* en el bloc de notas, si usted lo abre de esta manera. Ninguna de estas situaciones es deseable para el seguimiento del curso. Por esta razón, le sugerimos que siga el flujo de trabajo presentado en la primera sección de esta clase, abriendo primero el proyecto del curso y luego el *script* de la clase.

La mayoría de las veces, lo único que necesitará guardar en una sesión de R es su *script*. Al ser reproducible, en caso de que interrumpamos una sección y tengamos que volver al punto en que la dejamos, solo tenemos que abrir un *script* guardado y ejecutarlo en el panel *console*.

Para ejecutar un *script* en RStudio, haga clic en el icono "Source", en la esquina superior derecha del área del editor de códigos. Hay una flecha que revelará dos opciones

para ejecutar el *script*: *Source* y *Source With Echo*. La primera opción ejecuta su código, pero no muestra las respuestas en la *console*. A su vez, la segunda se ejecuta mostrando las respuestas. No mostrar sus resultados será útil en casos de *scripts* muy grandes, o en situaciones en las que no es muy conveniente mostrar todos los mensajes en la *console*.

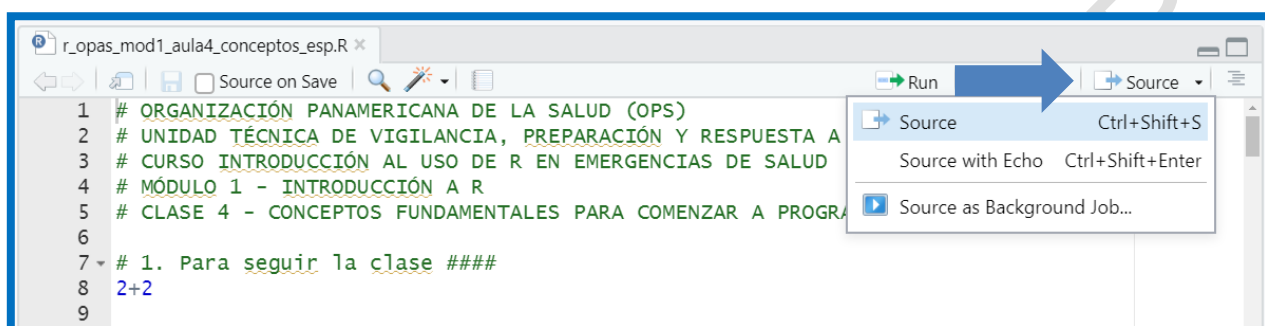


Figura 11: Botón *Source* para ejecutar el *script* completo.

Cuando colocamos el cursor del ratón sobre uno solo de los comandos de su *script* y luego hacemos clic en el icono “Run”, también en la esquina superior en el área del editor de códigos, R ejecuta solo el comando de la línea seleccionada.

Para guardar un *script* vaya al Menú *File > Save*, elija una ubicación y un nombre para su *script* y confirme en el botón *Save*. Es importante almacenar sus archivos de forma ordenada. Utilice carpetas para los diferentes proyectos, eligiendo nombres explicativos para sus trabajos. Para guardarlos más rápido, use el atajo *Ctrl + S* o haga clic en el símbolo del disquete.

Atajos de teclado útiles:

- **Source:** ctrl + shift + enter.
- **Run:** ctrl + enter.
- **Guardar script:** ctrl + S.

Comentarios

Los comentarios son frases escritas en su *script* para explicar lo que él hace o lo que deben hacer determinados códigos. De esta manera, los comentarios ayudan a la legibilidad y reproducibilidad de códigos.

Para añadir comentarios a nuestro *script*, simplemente hay que insertar el símbolo # al principio de la línea. R entiende que debe ignorar esa línea como un comando, ya que es un comentario. Tenga en cuenta que el color del comentario es diferente en RStudio.

El símbolo del comentario también es útil para suprimir líneas de códigos que queremos guardar en nuestro *script*, pero que no deben ejecutarse. Estos códigos pueden, por ejemplo, ser útiles para probar ciertos comportamientos, o simplemente son otras formas de realizar la misma operación, pero sin la necesidad de ejecución.

Si tenemos líneas de códigos que no deben ejecutarse, es decir, queremos dejarlas como comentarios, podemos seleccionar las varias líneas y luego usar el atajo *Ctrl + Shift + C* para comentar o “descomentar” las líneas seleccionadas. Otra forma es en el menú de RStudio: *Code > Comment/Uncomment Lines*.

Ahora que hemos terminado esta clase, recuerde guardar su *script* (en caso de que haya hecho cambios) y reiniciar la sesión, como vimos en la última clase. Esta es una buena práctica, especialmente cuando usted comienza a trabajar con grandes bases de datos.

¡Hasta la próxima clase!

> 8

> Referencias

Batra, Neale, et al. The Epidemiologist R Handbook. 2021. DOI:10.5281/zenodo.475264610.528 Disponible en: <https://epirhandbook.com/en/>.

Silva, Henrique Alvarenga da. Manual Básico da Linguagem R: Introdução à análise de dados com a linguagem R e o RStudio para área da saúde. [recurso electrónico] / Henrique Alvarenga da Silva - 1ª edición – São João del Rei: Editora do Autor, 2018. ISBN: 978-65-900132-0-0 (e-book).

Wickham, H., & Golemund, G. R for data science: import, tidy, transform, visualize, and model data. O'Reilly Media, Inc. 2016. ISBN: 9781491910399. Disponible en: <https://r4ds.had.co.nz/>

VERSIÓN PRELIMINAR