Tutorial introductorio al curso IMSER

Este tutorial es una forma de guiarlo a través de los conceptos y mecanismos básicos que usan R y el curso en sí para funcionar. Se da por sentado que usted ya vió o leyó las lecciones de la unidad 1 antes de empezar con este texto.

A lo largo del mismo usted va a encontrar ejercicios pensados para que usted practique y fije conceptos. Considere que este es el primer repartido de ejercicios del curso, pero a diferencia del resto, no tiene nota. Las soluciones están al final de este documento, puede mirarlas cuando quiera. La idea no es que demore mucho tiempo resolviendo los problemas, si no que entienda bien las soluciones.

> Nota: en este texto usted va a encontrar fragmentos de código (comandos o sentencias escritas en lenguaje R). Estos fragmentos son reconocibles porque se escriben con una fuente de tipo `monospace`. Este es el estándar en programación. Es útil porque asigna el mismo espacio a cada caracter y por lo tanto permite ordenar fácilmente las líneas de código (todo programador serio es muy cuidadoso en el uso de los espacios, por diversas razones).

Dado que busca abarcar muchas cosas, este tutorial es bastante extenso. Esto es compensado en parte por ser muy sencillo de seguir. De todas formas, el contenido está separado en cuatro secciones para dividir el proceso en varias etapas:

1. Interfaz de R/RStudio. Se muestra cómo interaccionar con los principales programas usados en el curso.
2. Elementos básicos de la sintaxis y gramática de R. Todo lenguaje tiene de estos, mejor entenderlos desde el principio.
3. Dinámica de los repartidos. Los repartidos están diseñados para que usted los haga con la mayor independencia posible. Para esto usted debe aprender a usar el sistema de corrección incluido.
4. Interfaz del foro. La comunicación a través del foro es fundamental para plantear y resolver dudas. Vea aquí cómo funciona reddit, la plataforma que usaremos en el curso.
5. Apéndices y soluciones de ejercicios.

---

(1) Interfaz de R y RStudio

La interfaz de RStudio está dividida en 4 regiones, a las que en este curso llamaremos *paneles*. En la figura 1 se muestra la distribución de los paneles y los nombres que usaremos a lo largo del repartido y el curso.

> Nota: si es la primera vez que abre RStudio probablemente esté ausente el panel 1 (el editor de texto plano), ya que no hay ningún archivo abierto. Para ver dicho panel use el menú `File` >> `New...` >> `R script` (o la combinación de teclas: Ctrl+Shift+N).

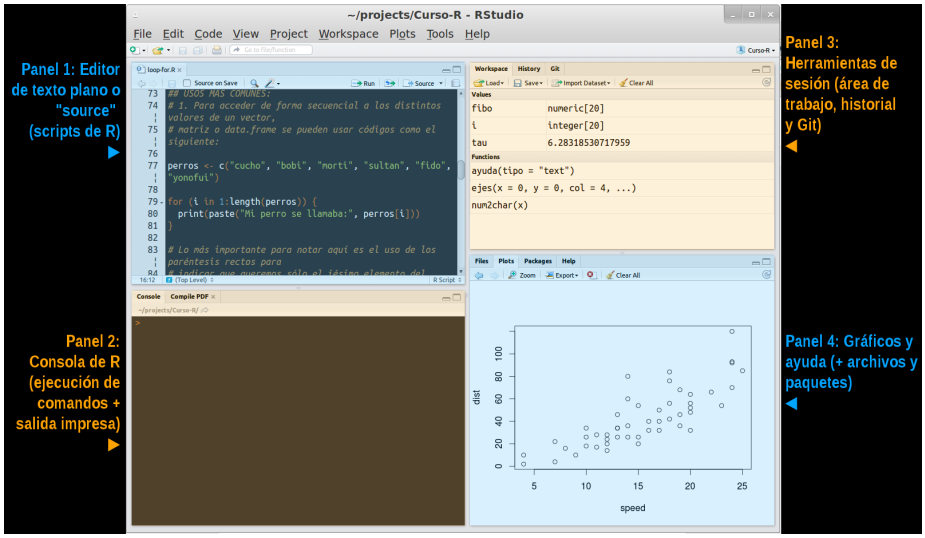

Figure 1: RStudio

Al igual que en este último ejemplo, muchas veces vamos a mencionar combinaciones de teclas, también llamados "shortcuts", atajos, para ejecutar acciones en RStudio. Estos suelen ser muy prácticos. Por lo tanto es conveniente y necesario que usted entienda la notación estándar para estos atajos. Por ejemplo: Ctrl+Shift+K refiere a la acción de presionar **al mismo tiempo** las teclas Ctrl ("Control"), Shift y la letra *k*. A su vez Ctrl++ implica apretar las teclas Ctrl y + al mismo tiempo. Si usa esta combinación como ejemplo notará que RStudio se "recarga" con un aumento generalizado en el tamaño de las letras. Puede volver al tamaño original con Ctrl+-.

1.1 Creación de un proyecto en RStudio

Para organizar los archivos provistos por el curso, se creará un directorio/carpeta en donde se guardarán estas lecciones así como el resto de documentos y datos.

Este directorio contenerá además un proyecto de RStudio. Un proyecto de RStudio es simplemente una forma de recordar nuestro conjunto selecto de archivos cada vez que estamos trabajando en un tema particular. Cuando abrimos RStudio en el proyecto "X", vamos a tener abiertas las mismas pestañas que teníamos la última vez que trabajamos en X. Para crear el directorio y el proyecto al mismo tiempo en RStudio, se puede utilizar el menú `Project >> New Project...`, luego se elije la opción *New Directory* (Fig. 2) y se crea el directorio "CursoR" (aunque usted puede elegir el nombre que más le guste). Al hacer esto el RStudio crea la carpeta y dentro de la misma un archivo llamado *CursoR.Rproj*, el cual se puede usar para abrir RStudio con el proyecto cargado (con doble click). Además, RStudio abrirá cargando ese proyecto por defecto cada vez, a menos que este sea cerrado en la sesión previa (`Project >> Close Project`).

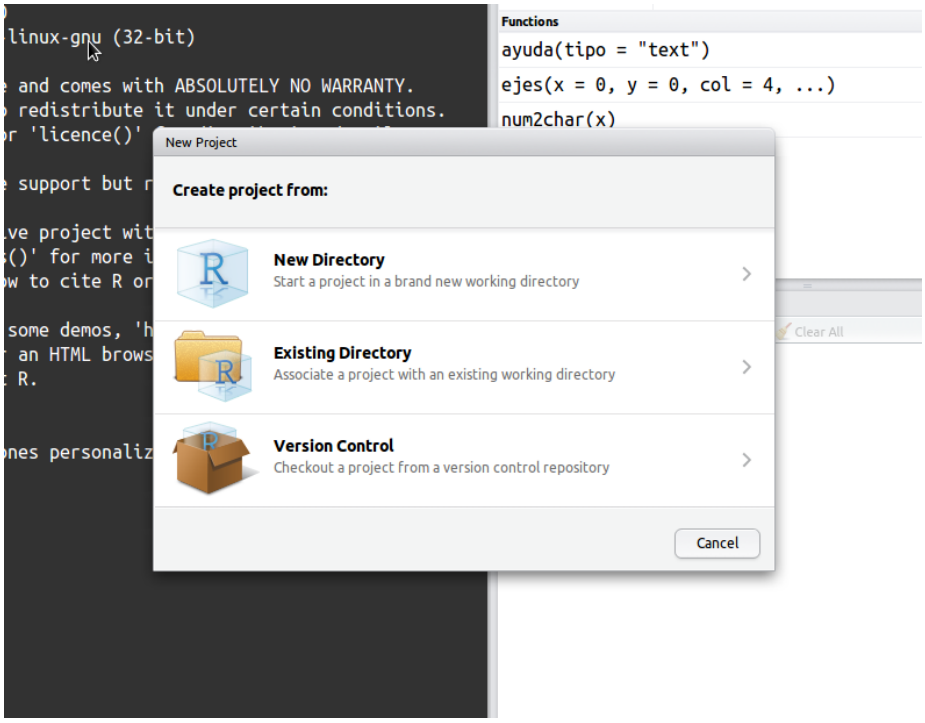

Figure 2: creación de un nuevo proyecto en RStudio

Ejercicio (0): como primer tarea, encárguese de crear el proyecto de RStudio para el curso, con las instrucciones que se dieron más arriba.

1.2 Codificación

El término "codificación" refiere a la forma en que una computadora traduce secuencias de bits, almacenados en el disco duro o la memoria, en caracteres normales (letras, números y otros símbolos). Debido a contingencias históricas de la informática, hay muchas formas de establecer la codificación de un archivo. En verdad hay un subconjunto de caracteres que son codificados de igual manera en cualquier situación: los llamados caracteres ASCII. Es el conjunto de la mayoría de las letras y símbolos, pero no incluye tildes o la letra eñe (en general, nada que no se encuentre en el idioma inglés).

Debido a que los scripts están codificados a través del formato UTF-8, lo recomendable es indicarle a RStudio que esta es la codificación que se va a utilizar por defecto. Esta es la más común entre programadores, independientemente del lenguaje de programación.

Utilizando la codificación correcta, se evitará encontrarse con palabras sin sentido como "programaciÃn" o "mÃ¡s" en los comentarios de los scripts (en lugar de "programación" o "más"). Bajo el menú `Tools >> Options...` (menú "Code Editing") se puede modificar la codificación por defecto, como lo indica la Figura 3.

También puede ver archivos individuales con cualquier codificación a elección, con el menú `File >> Reopen with Encoding...`. Tenga en cuenta que esto *no equivale a cambiarle la codificación al archivo*; para eso está la opción `File >> Save with Encoding...`.

Una opción intermedia es configurar cada proyecto una codificación diferente. Esto se puede hacer en el menú `Project >> Project options...`. De todas formas, recomendamos mantener siempre la misma codificación, en particular, UTF-8.

1.3 Directorio de trabajo

Lo primero que debe hacer es elegir el directorio en el que va a trabajar. Ya recomendamos crear una carpeta llamada "CursoR", aunque puede llamarse como usted quiera. Si aún no la creó puede crearla directamente con R, con el comando:

```
dir.create("CursoR")
```

Escriba esto en la consola de R que se encuentra en RStudio (panel 2, de abajo-izquierda) y presione enter. Esto va a crear una carpeta llamada CursoR dentro del directorio de trabajo actual. Puede confirmar de dos formas que la carpeta efectivamente fue creada: 1. con el navegador de archivos normal de su sistema operativo o 2. con el comando `dir`:

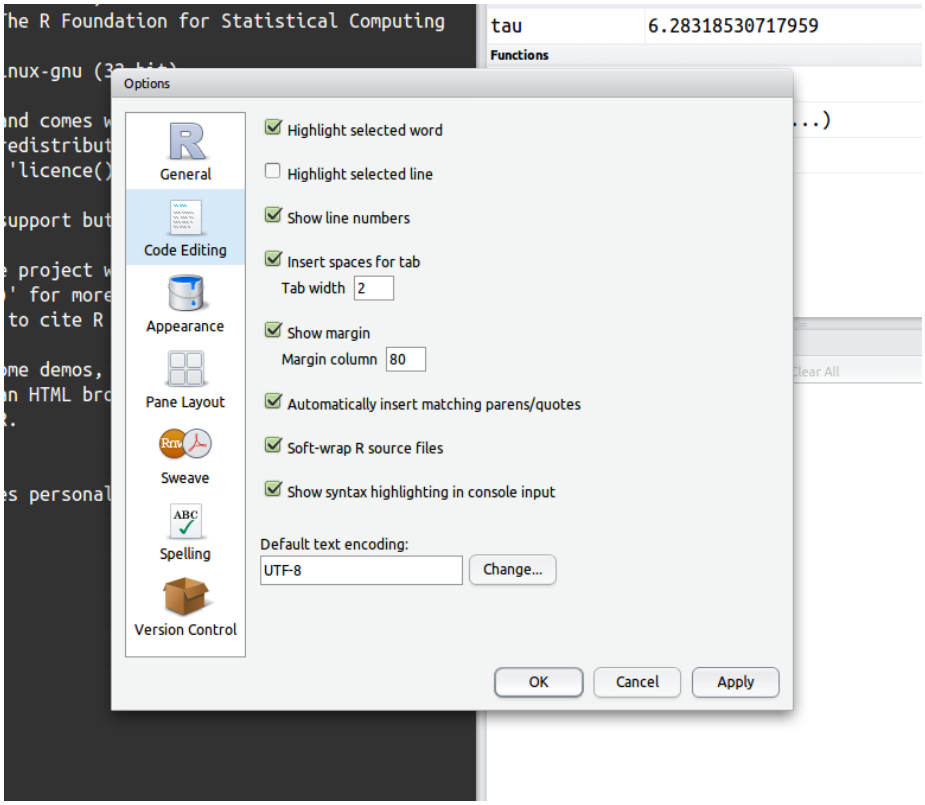

Figure 3: Encoding

```
dir()
```

Escribiendo esto en la consola (y dando enter) vemos una lista de nombres de achivos y carpetas (entre comillas). Allí se debería incluir "CursoR" si hicimos todo bien hasta el momento.

Como recordará de la lección 1.4, usted puede saber cuál es la carpeta de trabajo actual con la función `getwd`. De todas formas lo que nos interesa es "ubicarnos" en la carpeta CursoR, así que vamos a usar `setwd` para hacerlo:

```
setwd("CursoR")
```

Nótese que aquí usamos el camino relativo para ubicarnos. Es útil de momento, pero no siempre es lo ideal. También es posible que este comando no sea el apropiado, si usted creó anteriormente la carpeta "CursoR" en un directorio de trabajo distinto al actual (en Windows, ese directorio probablemente es la carpeta "Mis Documentos"; en sistemas Unix y Mac OS es `/home/user/`), o si el nombre de la carpeta creada no es "CursoR". En cualquiera de estos casos R no podrá realizar la operación y se lo hará saber con un mensaje de error impreso en la consola (abajo-izquierda). Por ejemplo:

```
Error in setwd("CursoR") : cannot change working directory
```

> Nota: podríamos haber hecho lo mismo con los botones de RStudio: `Session` >> `Set Working Directory` >> `Choose Directory` (o el atajo Ctrl+Shift+K). Si usted usa esta alternativa, comando `setwd("camino absoluto")` aparece igualmente en la consola y también en la historia de comandos. Dicha historia se encuentra en el panel de arriba-derecha, en la pestaña "History". Esta historia es muy útil si queremos buscar y recuperar algún comando ejecutado hace un tiempo (aunque no es infinita).

1.4 Consola vs. editor de texto plano

Nuestro primer comando (no escribir aún), será este:

```
mi.objeto <- 4
```

Pero, ¿dónde vamos a escribirlo? Las dos opciones que veremos son válidas, aunque no del todo iguales.

- Opción 1: la consola de RStudio. Sólo hay que dar *enter* al terminar de escribirlo. No hace falta que el cursor esté al final de la sentencia para dar el enter.

- Opción 2: en el editor de texto plano de RStudio (arriba-izquierda). Primero debe iniciar un archivo nuevo (si es que no lo hizo antes): vaya a `File >> New >> R Script` (Ctrl+Shift+N), recién entonces será visible este panel. En RStudio este panel se le llama "Source" (en inglés, fuente); es el "código fuente" con el que trabajamos en un momento dado. Como aún no guardamos el archivo, este figura bajo el nombre "Untitled1" (Sin título 1).

En general preferimos la segunda opción. Escribir los comandos en el editor nos permite guardar todo lo que hacemos, de forma que se puede repetir en el futuro con facilidad. Es cierto que aún si se usa la consola los comandos se guardan en el historial (accesible en el panel de arriba a la derecha, o con el comando `history()`), sin embargo es fácil ver que usar el editor de texto es generalmente más prolijo y ordenado.

De todas formas empezaremos con la opción 1, a fin de ser más ilustrativos. A continuación, escriba nuestro comando en la consola y presione enter:

```
mi.objeto <- 4
```

> Nota: ponga los espacios en blanco también; no afectan al comando, pero facilitan la lectura.

Debido a que usted le dió enter al comando en la consola, en su sesión existe un objeto llamado `mi.objeto`. Puede ver una lista de los objetos que existen en el panel de arriba a la derecha, bajo la pestaña "Workspace" (área de trabajo). Usaremos los términos *sesión*, *workspace* y *área de trabajo* de forma más o menos equivalente. También puede ver una lista de objetos existentes con el comando `ls`: escriba en la consola

```
ls()
```

Ahora veamos como usar la "opción 2". Como se dijo antes, preferimos usar el editor para escribir nuestros comandos, ya que nos permite repetirlos y organizarlos fácilmente. En RStudio se pueden ejecutar directamente los comandos escritos en el editor de texto plano. Hay que ubicar al cursor en la línea que nos interesa y ejecutar el atajo Ctrl+Enter.

Escriba la línea `mi.objeto <- 4` en el editor (arriba-izquierda) y use el atajo mencionado para que se ejecute. Si lo hizo correctamente usted puede ver que aparece la misma línea en la consola y se ve así:

```
> mi.objeto <- 4
>
```

1.5 El command prompt

Al ver la última salida en la consola tenemos la información necesaria para saber que el comando ya fue ejecutado, ¿cómo? gracias a la existencia del "command prompt", el signo de > que aparece al principio de cada línea en la consola.

El solitario command prompt que aparece en la última línea es un indicador de que R ya terminó de ejecutar todo lo que se le pidió anteriormente. Es una forma de decir "estoy listo para recibir órdenes". Considere ahora las diferencias entre encontrar esto en la consola:

```
> mi.objeto <- 4
```

y esto:

```
> mi.objeto <- 4
>
```

En el primer caso el comando `mi.objeto <- 4` aún no se ejecutó (el usuario no presionó enter), mientras que en el segundo sí. A veces el command prompt demora en aparecer, debido a que a R le toma tiempo ejecutar el último comando. En caso de que demore demasiado, usted puede cancelar la operación con la tecla Esc (o el botón de Stop que tiene RStudio, en la consola).

> Nota: en una terminal de Linux, en lugar de Esc se debe usar la combinación *Ctrl+C*; este es el estándar de Unix.

Un error de principiante muy común es el de copiar líneas de comando incluyendo el/los command prompt al principio. Al tratar de ejecutar estas líneas surge un error que difícilmente pueda comprender el usuario, ya justamete es un principiante. Es buena idea ver un ejemplo de este error: en el editor, agregue un command propt al principio de nuestro comando, de forma que quede así: "`> mi.objeto <- 4`". Ahora envíe esta línea a la consola (ponga el cursor en esa línea y aprete Ctrl+Enter). Vea el mensaje de error que devuelve R:

```
> > mi.objeto <- 4
Error: unexpected '>' in ">"
```

> Nota: puede que el mensaje esté en español en su PC, dependiendo del idioma en el que haya instalado R.

Borre el command prompt que acaba de agregar para evitar errores futuros.

El command prompt es entonces una indicación útil, pero también molesta. Muchas veces en libros o páginas web se muestran comandos de R que empiezan con el command prompt, lo cual es desconsiderado, ya que el usuario debe encargarse de borrar manualmente cada uno antes de poder reproducir los ejemplos. Puede ser útil en todo caso si la idea es diferenciar los comandos del usuario de las salidas impresas en la consola. En este texto se usa una aproximación inversa: la salida impresa muchas veces empieza con `##` para distinguirla de los comandos.

El command prompt tiene otra variante, el signo de `+`. El significado es diferente, indica que los comandos anteriores no están completos. Por ejemplo, si escribo solamente `mi.objeto <-` va a faltar algo. Haga el ejemplo: vaya al editor y borre el 4 al final de nuestro comando, luego envíelo a la consola de R con el atajo Ctrl+Enter. Puede ver que en la misma aparece lo siguiente:

```
> mi.objeto <-
+
```

Esta es la forma de R de indicar que el comando no está completo. Le está diciendo al usuario "aún me falta algo para poder ejecutar sus órdenes, dígame ¿qué valor debo asignar a `mi.objeto`?". El usuario, usted, puede completar el comando sin problemas: vaya a la consola y escriba 4. Ahora de enter. Debería ver esto:

```
> mi.objeto <-
+ 4
>
```

También tiene la opción de interrumpir el comando y volver al command prompt normal. Alcanza con ir a la consola y apretar la tecla de escape (Esc). En este caso se ve así en la consola:

```
> mi.objeto <-
+

>
```

1.6 Uso del autocompletar

En la consola de R y en en algunos editores de texto (incluido el de RStudio), es posible usar la función de autocompletar palabras. Esta se activa antes de terminar de escribir una palabra, apretando la tecla *tab*. Siguiendo con el ejemplo anterior, si usted escribe

```
mi.o
```

y apreta la tecla tab, verá como se completa el nombre `mi.objeto`. Esto es muy útil para usar nombres largos e informativos, con bajo riesgo de escribirlos mal. Siempre es mejor el nombre `promedio.valores.positivos` que `p` en términos de información dada al usuario.

1.7 Mensajes de Error y de Advertencia.

Usar R, o programar en general, es en gran medida acostumbrarse a cometer errores. No importa el nivel de experiencia y/o habilidad del programador, cometer errores es una constante a lo largo de la vida. La diferencia entre un programador y un principiante no es tanto la cantidad de errores que cometen, si no la capacidad para entenderlos y solucionarlos. Los errores generalmente producen mensajes impresos en la consola, los cuales suelen ser intimidantes. En realidad el mensaje de error es una gran herramienta que usted debe aprender a usar para ser un usuario exitoso.

En R hay dos tipos de mensajes:

1. Mensajes de **error**. Estos ocurren cuando se da un error tal que R no puede continuar ejecutando los comandos (con la excepción de que se use la función `try`, pero esto escapa a este repartido).
2. Mensajes de **advertencia** ("warnings"). Estos ocurren cuando se da un error que no impide continuar la ejecución de comandos. Debido a esto las advertencias no suelen considerarse como un asunto serio, pero muchas veces lo son, ya que el resultado final de la ejecución posiblemente esté mal.

Puede decirse que hay otro tipo de error en R: el que no deja mensaje alguno. Estos son los errores de lógica o escritura y que solo son detectables siendo precavidos y haciendo pruebas para determinar si el código hace lo que debe.

Haga un errores a propósito, es lo que hacen los profesionales. En muchos casos, esta es una técnica muy efectiva de saber si nuestro código funciona. Esto le beneficiará en al menos dos aspectos:

1. Aprenderá mejor cómo funcionan las cosas.
2. Se acostumbrará a no desesperarse cuando las cosas salen mal (i.e.: formará caracter).

Veamos un ejemplo. Escriba en la consola lo siguiente:

```
lenght(1:6)
```

```
## Error: could not find function "lenght"
```

(Más adelante se explica el significado de `1:6`.)

Como puede ver, el comando genera un error (puede estar en español en su PC: 'Error: no se encontró la función "lenght" '). El mensaje indica que R buscó entre todas las funciones del repertorio y no encontró la llamada "lenght". La enorme mayoría de las veces estos mensajes se deben a que hay un error de escritura; en este caso la h está mal ubicada: es *length* y no lenght. Se trata de un error clásico.

A veces ocurre que R no encuentra una función u otro tipo de objeto, como puede ser un vector o una matriz, debido a que usted no ha cargado el paquete correcto. Por ejemplo, la función `fractions` se encuentra en el paquete "MASS", el cual si bien se instala con R, no está cargado automáticamente en su sesión de trabajo. Si escribimos el nombre de la función y damos enter, R nos va a dar un mensaje de error:

```
> fractions
Error: object 'fractions' not found
```

(Nótese que aquí se usa el término genérico "object", en lugar de "function" como en el ejemplo anterior; en aquel caso, R determinó que lenght debió ser una función, ya que a continuación de dicha palabra seguía la apertura de un paréntesis.)

Ejercicio (1): lea los mensajes de error generados con los siguientes comandos y plantée una hipótesis de qué es lo que está mal (no intente resolverlos si le toma mucho tiempo, mejor vea las respuestas y entienda la solución):

```
Mean(5:7, na.rm = TRUE)
round(8.564432 3)
head(bigcity)
```

Los que hemos visto hasta aquí son errores muy simples. En la práctica, al crear código con más sofisticación, ocurren errores mucho más difíciles de resolver. Para esto existen técnicas, como la depuración de código ("debugging" en inglés), que son de enorme ayuda para solucionarlos.

(2) Elementos básicos de la sintaxis

2.1 Algunas manipulaciones simples

Todas las funciones usan paréntesis luego del nombre de la mismas para ser ejecutadas. Los usuarios más avanzados de R saben que esto no es cierto en última instancia, pero para nosotros este será el paradigma. Algunos ejemplos son:

```
ls()
dir()
sqrt(2)
log(16, 2)
sample(1:8, 3, replace = TRUE)
```

Los paréntesis indican la región en donde el usuario ingresa los *argumentos* de la función, las entradas de la misma. Los argumentos pueden o no ser *nombrados* (el último ejemplo usa esta opción). Por ejemplo, la función `length` sirve para saber la cantidad de elementos de un vector cualquiera, entonces:

```
x <- 3:6
length(x)
```

> Nota: el vector `x` es la secuencia de números enteros 3, 4, 5, y 6.

Escriba este ejemplo. Luego de ejecutar el comando, en la consola debería mostrar impresa la salida de `length(x)`, que es el valor 4. Como dijimos, los paréntesis delimitan el conjunto de lo que son "las entradas" de una función. Aquí hay una sola entrada: el vector `x`. Hay varios lenguajes de programación que no usan este esquema, pero no son la mayoría. Por otro lado, ya vimos que la salida aquí es 4 y que luego de ser impresa en la consola "se pierde". Como usuarios podemos guardar este valor en un objeto, tal como se mostrara en la lección 1.2. Por ejemplo:

```
y <- length(x)
```

Este comando guarda la salida de `length(x)` en un nuevo objeto, `y`. La "flecha" hacia la izquierda, `<-` es el operador que normalmente se usa para hacer asignaciones (hay al menos seis formas de hacer asignaciones, pero en general sólo usaremos la flecha a la izquierda).

Ejercicio (2): haga usted un ensayo con la función `mean` (para calcular promedios). La entrada será otra vez el vector `x` y la salida un objeto llamado `promedio`.

Si usted hizo todo bien, entonces en el panel 3 (arriba-derecha) de RStudio, bajo la pesataña *Workspace*, encontrará al objeto `promedio` en la lista de objetos presentes y su valor será 2.5. También puede ejecutar:

```
exists("promedio")
```

(Si el resultado es `TRUE` entonces `promedio` "existe".)

En todo momento puede usar la consola para inspeccionar el objeto, con sólo escribir el nombre (escriba y de enter):

```
promedio
```

2.2 Secuencias de números

Como habrá notado al crear `x`, el símbolo `:` sirve para crear secuencias de números enteros. Por ejemplo `0:4` son los números 0, 1, 2, 3 y 4.

Ejercicio (3): usando `:` genere las siguientes secuencias de números enteros:

1. Los números de 10 al 10000.
2. Los números del 20 al 10 (sí, es en orden decreciente).
3. Los números del -8 al 6 en orden creciente.
4. Los números del -8 al 6 en orden decreciente.

Pero además del operador `:`, existe la función `seq`, que sirve cuando la secuencia (regular) no es de números *enteros*, o la distancia entre consecutivos es distinta de 1. Por ejemplo, la secuencia 0, 0.2, 0.4, 0.6, 0.8 y 1 se puede crear así:

```
seq(0, 1, by = 0.2)
```

```
## [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

Nótese que entre paréntesis está: el inicio, el final y la distancia entre los valores consecutivos (indicado por el nombre de argumento: "`by`"). Es importante destacar que los tres argumentos están separados por *comas*. Alternativamente, se puede crear una secuencia indicando el inicio, el final y la cantidad de valores del vector de salida:

```
seq(0, 1, length = 11)
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Ejercicio (4): usando `seq` genere las siguientes secuencias de números:

1. Los números pares del 2 al 110.
2. Los números impares del 1 al 110.
3. Un vector de 101 elementos, con valores desde 9 hasta 0 (orden decreciente).

2.3 Pegando cosas

Otra función sumamente útil es la de concatenación: `c`. Sirve para "armar" o "pegar" elementos y así crear un vector. Ejecute el siguiente ejemplo:

```
mi.vector <- c(x, promedio, 14)
```

El resultado es un nuevo vector llamado `mi.vector`, con 6 elementos (vea la salida de `length(mi.vector)` para confirmarlo).

Ejercicio (5): escriba ahora el comando necesario para crear un vector llamado `mi.otro.vector`, el cual tendrá la secuencia de valores 45, -76, 3, 4, 5, 6, 0.333.

> Nota: en R hay tres funciones de concatenación: `c`, `rbind` y `cbind`. Acabamos de ver la primera, las otras dos servián cuando trabajemos con matrices o data.frames.

2.4 Cambiando valores

Hasta ahora hemos visto como crear vectores y hacer asignaciones. Veamos ahora la forma de modificarlos. Supongamos que queremos cambiar el último valor de `mi.vector`; el nuevo valor será 0. Una forma fácil de hacerlo es así:

```
mi.vector[6] <- 0
```

Aquí estamos utilizando los corchetes o paréntesis rectos para *modificar* un vector. En verdad no es más que un caso particular de la operación de asignación. Los paréntesis rectos sirven para indicar la ubicación de ***el o los*** elemento(s) que quiero cambiar.

Se podrían modificar *varios* elementos de `mi.vector` al mismo tiempo también. En lugar de poner un único valor entre corchetes, se puede poner un *vector* con las posiciones que quiere modificar. Por ejemplo, cambiar los valores `mi.vector[1]` y `mi.vector[4]` por -1 se puede hacer con el comando:

```
mi.vector[c(1, 4)] <- -1
```

Nótese que se usa la concatenación `c` para primero formar el vector y luego ponerlo entre los corchetes. Si en cambio escribiéramos `mi.vector[1, 4]` estaríamos cometiendo un error (que usted entenderá cuando aprenda más sobre matrices y data.frames).

Ejercicio (6): modifique las posiciones 2 y 3 de `mi.vector`, sustituyéndolos por 100 y 104 respectivamente *en un sólo paso* (hacerlo en 2 pasos es trivial y para nada elegante). Esta es una situación diferente a los ejemplos anteriores, ya que hay 2 valores reemplazantes. Para hacerlo correctamente tendrá que usar la función `c` del lado derecho de la "flecha" de asignación.

2.5 Extrayendo valores

Los corchetes también sirven para extraer valores. Ejecute el siguiente comando a modo de ejemplo:

```
mi.vector[c(2, 5)]
```

En la consola debería ver los valores 4.0 y 4.5. Al igual que antes, esta salida se puede guardar en un objeto nuevo:

```r
u <- mi.vector[c(3, 4, 5)]
# O también
u <- mi.vector[3:5]
```

> Nota: muchas veces quienes aprenden R escriben expresiones como `c(3:5)`, lo cual es un uso innecesario de `c`. Alcanza con poner `3:5` para hacer lo mismo.

Este método de extraer valores puede ser muy útil para reordenar un vector. Por ejemplo el siguiente comando:

```r
mi.vector[c(4:6, 1:3)]
```

```
## [1] -1.0000  0.7989  0.0000 -1.0000 -0.4869 -0.2107
```

Cambia de lugar las dos mitades del vector original.

Ejercicio (7): busque usted la forma de obtener un vector con los valores de `mi.vector` pero en orden inverso.

> Nota: la extracción o modificación de valores funciona de forma similar con ciertos tipos de objetos: factores, matrices, data.frames o listas. Los detalles vendrán en otras unidades del curso.

2.6 Funciones & operadores

Casi todos los comandos que se ejecutan en la práctica involucran objetos que se califican como *funciones* u *operadores*. La línea que separa entre uno y otro es arbitraria pero útil en la práctica, de la misma manera en que es útil en la cocina diferenciar frutas de verduras. Para los objetivos de este curso, alcanza con decir que:

1. Las funciones son todas aquellas que tienen un nombre escrito (i.e.: una "palabra") y que para ejecutarse requieren de paréntesis, a fin de indicar las "entradas" (los valores de los argumentos). Ejemplos típicos son `mean`, `length`, `print`, `read.table`, `c`, `setwd` y `ls`, así como las funciones creadas por el propio usuario.

2. Los operadores son aquellos que escribimos con uno o pocos símbolos (típicamente 1 o 2) y que sirven para realizar las operaciones más comunes. Ejemplos típicos son `+`, `-`, `*`, `/`, `**`, `^`, `:`, `<-`, `[`, `%*%`, `$` y `==`.

3. Hay algunos objetos que no son funciones ni operadores. Les llamaremos "constructos" generalmente. Afortunadamente no son muchos y todos se encuentran en lo que llamaremos *Estructuras de control*. Estos son: `for`, `while`, `repeat`, `break`, `next`, `if` y `else` (puede consultar la ayuda de R con `?Control`).

En general asociamos a los operadores con operaciones matemáticas (suma, resta, multiplicación), mientras que las funciones pueden asociarse con "programas", ya que pueden realizar procedimientos más complejos. Sin embargo esta división es artificial, ya que muchas operaciones matemáticas se deben realizar con funciones (según la definición anterior). Algunos ejemplos son: `sin` (función seno), `sqrt` (raíz cuadrada), `sum` (sumatoria), `mean` (media aritmética) e `integrate` (integración numérica).

Orden de precedencia. En el caso de los operadores, existe una convención respecto al orden temporal en que deben evaluarse. Esto es necesario para evitar ambigüedades. Por ejemplo:

```
4 + 2 * 9
```

¿Debo sumar 4 al producto de $2 \cdot 9$ o multiplicar por 9 la adición de $4 + 2$? Los resultados serán distintos según cuál es el operador que se ejecute primero. Por convención es la multiplicación la que se ejecutará primero (puede hacer la prueba en R). En caso de duda, siempre puede usar paréntesis para asegurarse una ejecución correcta; ej.: `4 + (2 * 9)`. En este caso es bastante intuitiva la solución a la pregunta, pero en otros no tanto. Para esto tenemos una tabla que puede consultar cuando quiera, que indica el *orden de precedencia* de los operadores. Puede seguir este link para ver dicha tabla.

Ejercicio (8): su tarea es encontrar las soluciones numéricas a los problemas que se plantean, traduciendo correctamente las expresiones matemáticas a expresiones en lenguaje R. Es posible que usted necesite consultar la ayuda para hacer el ejercicio. Si no sabe exactamente cuál es el nombre de la función que quiere consultar, use `help.search`, con cualquiera de sus dos modalidades (ver lección 1.3). Por ejemplo, si quiere averiguar sobre integrales, puede ejecutar:

```
help.search("integral")
??integral
```

Si en cambio usted ya sabe bien a qué página quiere ir, como por ejemplo la ayuda de `integrate`, puede usar la función `help`, que también tiene dos modalidades:

```
help("integrate")
?integrate
```

Estos ejercicios son simples pero con dificultad muy variada, no se espera que usted los pueda hacer todos la primera vez. Recuerde que las respuestas están a la vista; no es la idea que usted se detenga demasiado tiempo en estos ejercicios.

De todas formas, **es importante** que usted entienda las soluciones. Si tiene alguna pregunta relevante, no dude en consultar el foro del curso (ver la sección 4 de este tutorial). Puede que la pregunta ya se haya hecho, o puede ser usted quien la formule.

> Nota: considere usar un traductor en línea (como *Google translate*) para leer la documentación, si representa un obstáculo el idioma.

a. Encuentre la(s) función necesaria para calcular logaritmos. Vaya a la página de ayuda de la función correspondiente y encuentre cuál es el nombre del argumento usado para determinar la base del logaritmo. Luego calcule los siguientes logaritmos: $log_3 8$, $log_5 13$, $log_{10} 1.5$, $ln 7$, $log_2 6$.

b. Encuentre la función que devuelve los extremos (máximo y mínimo) de un vector, el *rango* de valores. Vaya a la página de ayuda de la misma. Debe encontrar un argumento que sirve para cambiar el comportamiento de la función, si el vector de entrada contiene valores no finitos (`Inf` o `-Inf`). Si dicho argumento está "activado", los valores infinitos se omiten. Determine el rango de los vectores (descartando infinitos):

```
x <- 1:100
x <- -100:-1
x <- c(pi, runif(10, 4, 6), exp(8), Inf)
```

c. Calcule los valores que toma el siguiente polinomio para los valores de x: -1, 1.5 y 4

$$5x^3 - 2x^2 + 11$$

d. ¿Cuál es el valor de h para cualquier valor de x entre -1 y 1?

$$h = cos(x)^2 + sen(x)^2$$

e. Calcule los valores de D para $a = 1, 2, 3, 4, ..., 100$:

$$D = a \cdot \sqrt{2}$$

f. Calcule el valor de la siguiente sumatoria para los valores de $n = 1, 5, 10$

$$\sum_{i=0}^{i=n} \frac{1}{i!}$$

g. Calcule el valor de la siguiente integral:

$$\int_0^{20} ln(x)\ dx$$

---

(3) Repartidos

Los repartidos del curso tienen un sistema de corrección "automática" que usted debe aprender a usar. El objetivo de esto es lograr que estudiante no dependa de la supervisión de los profesores para saber si sus ejercicios están correctos. Este sistema consta de dos tipos de archivos: los que debe editar (ejercicios) y los que no. Los archivos se encuentrarán en la carpeta del repartido (ver más abajo).

En general se plantean problemas para los que usted tiene que crear o modificar el código necesario para resolverlo. Siempre que es pertinente, se piden soluciones genéricas a los problemas, es decir, que sirven para una amplia variedad de casos en lugar de uno en particular.

Al tratarse de un curso completamente en línea es imposible controlar por completo el copiado de soluciones. Por esta razón todos los estudiantes deben aceptar el **código de honor** del curso. El mismo se encuentra en el **Apéndice I**, al final de este documento.

A continuación se explica con ejemplos la dinámica de los repartidos.

3.1 Soluciones "genéricas"

> (Nota: si el vector `mi.vector` no está en su sesión de trabajo en este momento, repita los pasos para volver a crearlo y recuerde que debe tener 6 elementos; alternativamente, construya un nuevo `mi.vector` cualquiera con 6 valores.)

Anteriormente quisimos cambiar el último valor de `mi.vector`. Para eso corrimos este comando:

```
mi.vector[6] <- 0
```

Llamémosle a esta la "solución 1" de nuestro problema. Esta solución es perfecta si `mi.vector` tiene 6 elementos. ¿Qué pasa si no sabemos de antemano esta información?

Ya vimos que la función `length` sirve justamente para averiguarlo. Por lo tanto debe haber una manera de usarla para no depender de ese conocimiento. ¿Puede imaginar esta alternativa? Lo invito a que lo intente. De todas formas, una solución es la siguiente:

```
mi.vector[length(mi.vector)] <- 0
```

Es importante que entienda lo que se hizo aquí.

El comando `length(mi.vector)` devuelve la longitud de `mi.vector` (i.e.: la ubicación que queremos modificar); ese valor es enviado a "los corchetes" para definir correctamente el valor que hay que modificar. Esto es lo que podríamos llamar una *composición de funciones* (para quienes quieran profundizar, las dos funciones involucradas son `length` y `[<-`; la página de ayuda de la segunda es accesible con el comando `?"[<-"`).

Podríamos extendernos y encontrar una "solución 3" si dividimos la tarea en dos pasos:

```
ubicacion <- length(mi.vector)
mi.vector[ubicacion] <- 0
```

Esta solución tiene sentido si vamos a volver a utilizar el objeto `ubicacion` en el futuro. Separar los pasos también puede lograr que una secuencia de comandos sea más fácil de leer; en el curso solemos hacer este tipo de separaciones, pero es una estrategia que responde a razones pedagógicas y no a que sea necesariamente la mejor opción del punto de vista práctico.

Lo importante a destacar de las soluciones 2 y 3 es que son *genéricas*: no importa qué tan grande o chico sea el vector `mi.vector`, siempre será una solución correcta. Lo opuesto a una solución genérica es una solución *específica*, como la solución 1.

La utilidad de la programación reside en gran medida en encontrar soluciones genéricas a problemas prácticos, de forma que un mismo código pueda ser aplicable a una variedad de situaciones. Por esto el énfasis en el curso es el de encontrar siempre la solución más genérica posible.

Ejercicio (9): anteriormente se le pidió el código necesario para obtener los valores de `mi.vector` pero en orden invertido. En aquella ocasión la solución específica era suficiente. Ahora debe encontrar una solución *genérica* para el mismo problema. En otras palabras, considere que `mi.vector` puede tener una longitud cualquiera mayor o igual a dos y que usted desconoce.

Durante todo el curso los ejercicios sólo se considerarán correctos si resuelven el problema de forma genérica. Por lo tanto, ante un ejercicio como el anterior, la respuesta `mi.vector[6:1]` será incorrecta; la respuesta genérica `mi.vector[length(mi.vector):1]` será apropiada (pero no necesariamente la *única forma* de obtener una respuesta correcta).

> Nota: en R la función `rev` sirve para "dar vuelta" vectores (ej.: `rev(1:10)`).

3.2 El formato de los ejercicios

Cada repartido se compone de un archivo zip que usted debe descomprimir dentro de la carpeta del curso. Para continuar con el Repartido I, descargue de la web el archivo rep-1.zip. Debe extraer sus contenidos en la carpeta del curso ("CursoR" o el nombre que usted le haya dado), de forma que tendrá una subcarpeta llamada "rep-1". Allí encontrará una serie de archivos. En algunos usted deberá escribir código R para solucionar algún problema, estos son los archivos de los ejercicios. Los otros debe dejarlos como están, sin moverlos ni cambiar su nombre.

Para continuar con los ejemplos, es necesario que usted ya tenga lista la carpeta del repartido, haya abierto una sesión de R y que dicha carpeta sea el directorio de trabajo actual. Puede usar `getwd()` para comprobar que está trabajando en la carpeta correcta, como se indica en la lección 1.4.

El siguiente paso es abrir en el editor de RStudio el archivo del primer ejercicio, "1-varianza.R" (puede usar el atajo Ctrl+O).

Antes de continuar, lea las instrucciones del primer ejercicio, contenidas en el archivo letra.pdf. No se preocupe si no las entiende del todo, iremos paso a paso para resolverlo. Luego vuelva a RStudio y al archivo 1-varianza.R.

Allí encontrará que hay dos tipos de líneas: las que tienen código funcional ("que hace cosas") y líneas con instrucciones que empiezan con `#`. En una *línea* de código R, todo lo que se escribe después del signo `#` es *comentario* y no es evaluado/ejecutado (la idea es dejar mensajes al próximo usuario, muchas veces el propio autor, para que pueda entender mejor lo que hace el código). Lea con tranquilidad las instrucciones de este archivo antes de continuar.

En cada archivo de ejercicio hay unas líneas dedicadas a los objetivos del mismo. Estos generalmente son objetos que su código debe producir. En este caso hay tres: `x_mean`, `s` y `out`. Note que en las descripciones se pone un asterisco a aquellos objetivos que son obligatorios para completar el ejercicio. Si no tiene un asterisco, debe tomarlo como una sugerencia para resolver más fácilmente el problema.

Siguiendo con las líneas del archivo, puede ver también que la región en la que usted debe trabajar está claramente delimitada:

```
# ===== Su código comienza aquí: =====#

x_mean <- 0

s <- 0

out <- 0

# ====== Aquí finaliza su código =====#
```

Es decir, usted puede escribir lo que quiera entre medio de las dos líneas de comentario copiadas aquí. También puede aumentar o disminuir la cantidad de líneas que hay entre medio, sin límites.

Puede escribir código por fuera de este espacio, pero tenga en cuenta que:

1. Esas líneas van a ser descartadas en el proceso de corrección.
2. No es buena idea modificar el código que ya viene en el archivo original.

3.3 Usando la función evaluar

Ahora usted empezará a editar el archivo 1-varianza.R para ir logrando los objetivos que pide el ejercicio.

Nótese que en la línea 2 hay un comando necesario para crear un vector `x` aleatorio. Este comando se puede correr individualmente (por ejemplo, con el atajo Ctrl+Enter) o junto con todo el archivo:

```
source("1-varianza.R")
```

(O en RStudio: Ctrl+Shift+S; note como al hacer este atajo aparece en la consola el comando correspondiente con el camino absoluto al archivo.)

Una vez que tenemos creado `x` podemos empezar a jugar con el archivo para buscar las soluciones correctas al problema.

Lo primero que resolveremos será el objeto `x_mean`. Tenemos que modificar el código para que dicho objeto sea la media aritmética de el vector `x`. En las instrucciones del script se indica que usted debe usar la ayuda de R para encontrar la función que necesita; esto será común a lo largo del curso. De todas formas ya hemos mencionado que la función `mean` es la que precisamos para la tarea. Lo que debe hacer entonces es modificar la línea

```
x_mean <- 0
```

sustituyendo el 0 por `mean(x)`:

```
x_mean <- mean(x)
```

Ahora bien, como `x_mean` es uno de los objetivos del ejercicio, podemos verificar si lo hemos hecho bien o no. Para eso se necesita la función `evaluar`, que es el eje central del mecanismo de corrección. Lo primero es guardar los cambios en el archivo (Ctrl+S).

Ahora corra el archivo evaluar.R con el comando:

```
source("evaluar.R", encoding = "UTF-8")
```

> (Nota: si usted trabaja en Mac OS o Linux, puede omitir el argumento `encoding = "UTF-8"`.)

Cada vez que cargamos este archivo se imprime en la consola un mensaje:

```
Funciones cargadas correctamente: evaluar, verNotas y fecha.datos

Chequeo de encoding:
  Los siguientes caracteres deben ser vocales con tilde:
  á - é - í - ó - ú
  Si *no se ven correctamente* corra el siguiente comando:
  source('evaluar.R', encoding = 'UTF-8')

Para comprobar la fecha de su archivo datos ejecute:
>> fecha.datos()
```

La primer parte del mensaje se explica sola, lo último, referente al "archivo datos", lo explicaremos más adelante.

El resultado que nos importa es que ahora está cargada la función `evaluar`, necesaria para corregir el ejercicio. Para usarla, basta con correr:

```
evaluar()
```

Tras esta acción se imprimirá en la consola un menú para elegir cuál ejercicio desea corregir. En este caso será el 1:

```
Elija el archivo que desea corregir:

1: Ej. (1): 1-varianza.R
2: Ej. (2): 2-zenon.R
3: Ej. (3): 3-extra-dist.R
4: Todos

Selection: 1
```

(Alternativamente puede escribir `evaluar(1)` directamente y así saltear el menú.)

Al hacer esto se imprime una cantidad considerable de texto en la consola:

```
>> Iniciando una nueva semilla:
>> set.seed(444)
>> Creando un nuevo vector x para la corrección:
>> x <- rnorm(sample(100, 200, 1))
>> el nuevo x tiene 20 elementos; su varianza es 0.9023475
valor de x_mean ... OK

==========RESULTADOS==========

El script "1-varianza.R" tiene algún error, lo siento :-(
    -> Verifique que su solución sea genérica y que sigue
       todas las consignas de la letra.

Ejercicios correctos:

==>> Ninguno por ahora

Se generaron los siguientes mensajes de error:

* Al corregir el ej. 1, archivo 1-varianza.R:
Error : la longitud del vector s no es la esperada,
la diferencia observada es la siguiente:

  Obs. Esp. Dif.
     1   20   19
```

```
============================

Total hasta ahora: 0 de 1 ejercicios; NOTA: 0 %
```

Empecemos por las cinco primeras líneas:

```
>> Iniciando una nueva semilla:
>> set.seed(444)
>> Creando un nuevo vector x para la corrección:
>> x <- rnorm(sample(10, 20, 1))
>> el nuevo x tiene 20 elementos; su varianza es 0.9023475
```

Esta información es necesaria si desea reproducir el ejemplo usado en la corrección. En primer lugar se indica que se usa una "semilla", con el comando `set.seed(444)`. Esto sirve para poder reproducir el ejemplo exactamente y es necesario, ya que se usan generadores de números aleatorios como `rnorm` y `sample`. Luego se muestra el comando usado para generar el nuevo `x`. En definitiva, usted puede reproducir exactamente los primeros pasos del mecanismo de corrección con los comandos:

```
set.seed(444)
x <- rnorm(sample(10, 20, 1))
```

La preparación culmina con una indicación de las propiedades básicas de `x`. Note que estas primeras líneas comienzan con >>, para evitar confusiones con comandos que haya ejecutado usted anteriormente.

En la sexta línea ("`valor de x_mean ... OK`") se muestra que nuestro objeto `x_mean` parece estar correcto. Muchas veces las correcciones tienen varios pasos intermedios antes de completarse. Por cada paso superado se imprime una línea como esta, con un "`... OK`" al final. Esté atento a este detalle, ya que puede ser de gran ayuda para entender qué parte del problema no está resolviendo bien.

A continuación se abre un espacio con resultados. Aquí lo más importante son los mensajes de error. Los errores se indican empezando con un asterisco, el número de ejercicio y el archivo en el que se detectaron. Debido a que los errores detienen la ejecución de R, nunca se muestra más de un error *por archivo*.

En este caso el mensaje reza "la longitud del vector s no es la esperada", lo cual es obvio para nosotros, ya que en el archivo sigue estando la línea `s <- 0` (a propósito, el 0 se pone inicialmente para poder correr todo el script sin que ocurran errores, aún si usted no empezó con el ejercicio).

Pero el mensaje de error busca dar un poco más de información, indicando cuál era el valor esperado y comparándolo con lo que se observó. Mire de vuelta el

mensaje de error y fíjese cuál es el valor observado, cuál el esperado y cuánto es la diferencia. ¿Tiene sentido?. Esta información, junto con la que se da en las líneas con >> al principio, es muy útil para determinar en qué parte del ejercicio está el problema.

Por último el mensaje le muestra la nota acumulada hasta el momento. Más adelante iremos más en detalle con este aspecto.

Vayamos entonces al problema del objeto `s`. En las instrucciones del archivo 1-varianza.R se indica claramente que 1) `s` es un vector y que 2) es la entrada de la función `sum`. Específicamente, el vector `s` debe tener todos los términos de la sumatoria necesaria para calcular la varianza de `x`. Sabemos que la fórmula de la varianza es:

$$\sigma^2 = \frac{1}{n-1} \cdot \sum_{i=1}^{i=n} (x_i - \overline{x})^2$$

Por lo tanto el iésimo elemento de `s` debe ser $(x_i - \overline{x})^2$, lo cual se traduce al siguiente código:

```
s <- (x - x_mean)^2
```

Modifique el archivo para que calcule `s` con este comando. Luego vuelva a ejecutar `evaluar(1)` para ver la salida en la consola. Debería encontrar las líneas "`longitud de s ... OK`" y "`valores de s ... OK`" al principio de la salida impresa.

Ejercicio (10): ahora sólo queda obtener el valor de `out` correcto (equivalente a σ^2 en la ecuación). Esta tarea queda para usted.

Recomendación general: muchas veces es buena idea usar un esquema, un dibujo, o cualquier representación válida, antes de empezar a escribir código. Los problemas de programación se pueden separar a groso modo en dos componentes: el problema en sí y la forma de traducirlo a código. Por esto lo recomendable es tratar de resolver el problema de base primero, en el lenguaje que le resulte más cómodo. Esto puede ser en un diagrama de flujo, un texto u otro medio. Posteriormente sólo quedará traducir su solución al lenguaje R.

Esto no es un proceso lineal, ya que muchas veces una parte de su esquema no tiene una traducción "literal" a código y por lo tanto requiere de una revisión

del enfoque. Es decir, se convierte en un subproblema anidado en el problema principal. El esquema general que hizo al principio será muy útil para que los subproblemas no se vuelvan un obstáculo. Servirá de mapa para no perderse en un laberinto de pequeños problemas.

Esta es una recomendación válida para los ejercicios del curso (que esperamos sinceramente que no sean tan intrincados), pero es más válida aún para la práctica de programar en general.

3.4 Mensajes de advertencia

Si hizo el ejercicio anterior, es posible que la corrección devuelva mensajes de *advertencia* además de mensajes de error. Veamos lo que ocurre si usamos el siguiente código para calcular `out`:

```
out <- sum(s)/length(x) - 1
```

Luego de guardar el archivo y ejecutar `evaluar(1)` se debería imprimir estos mensajes de error y de advertencia:

```
* Al corregir el ej. 1, archivo 1-varianza.R:
Error : el valor de out obtenido no es el esperado,
la diferencia observada es la siguiente:

   Obs.  Esp.  Dif.
  85.64 87.38  1.74

Warning message:
ej. 1: no es lo mismo 'a / (b + c)' que 'a / b + c' ...
```

(Los valores exactos no van a ser los mismos, ya que siempre se usa un nuevo `x` generado aleatoriamente.)

("Warning" es la palabra para advertencia en inglés.)

La intención de los dos tipos de mensajes es distinta. Los mensajes de error son siempre objetivos; simplemente indican que no se resolvió correctamente el problema. Usualmente se compara el valor esperado con el obtenido.

Los mensajes de advertencia, cuando los hay, buscan dar una pista sobre cuál puede ser el problema. Estos mensajes son fruto de la experiencia que como profesores hemos tenido, la cual nos permite adelantarnos, a grandes rasgos, a los errores de los estudiantes. No siempre habrá un mensaje de advertencia oportuno para su error específico, ya que la capacidad de predicción siempre es limitada. De todas formas, lo importante es que usted sepa **aprovechar** estos mensajes, ya que pueden ser de gran utilidad.

Ejercicio (11): resuelva por su propia cuenta el ejercicio 2 del repartido: "Paradoja de Zenón". Tenga en cuenta que el ejercicio pide un `n` específico, pero el resto del código debe ser genérico (en función del `n` hallado).

3.5 El puntaje y las notas

Como ya vimos, al evaluar cualquier ejercicio con la función `evaluar` se imprime en la consola un resumen de su performance hasta ese momento. Puede también ver sus notas con la función `verNotas`, la que imprime una tabla con un resumen de su situación en el repartido. Las notas siempre se expresan en porcentaje:

```
> verNotas()
Parte     Nota Script
1            1  1-varianza.R
2            0  2-zenon.R
3            1  3-extra-dist.R
Total (%) 100  --
```

Notará que en este caso se da un 100%, a pesar de que el ejercicio 2 no está completado. Esto se debe a que hay un ejercicio *extra* ya hecho. Los ejercicios extras son problemas más difíciles en general, que sirven para que el estudiante interesado profundice algún aspecto del curso, o simplemente para aumentar la nota total del repartido. Si usted hace *todos* los ejercicios, regulares y extras, entonces tendrá una nota mayor a 100%. Esta nota lo puede favorecer en el promedio de repartidos, al final del curso. Sin embargo ese promedio no podrá superar el 100% (ver el **Apéndice II**).

Las notas se almacenan en el archivo "datos" que se encuentra en la carpeta del repartido. Cada vez que se corre la función `evaluar` dicho archivo se actualiza con sus notas (por esta razón, la carpeta del repartido no puede estar bloqueada a escritura). Nótese que si usted no usa la función `evaluar` para todos los ejercicios, aún si los resolvió bien, su nota no lo reflejará.

Para entregar el repartido usted debe cargar en la página del curso el archivo datos de su carpeta. Este será revisado posteriormente por los profesores.

Este no es el único uso del archivo datos. Verá a continuación por qué es central para el mecanismo de corrección.

3.6 El archivo datos

Además de las notas del estudiante, el archivo datos contiene las funciones que usa `evaluar` para corregir los ejercicios de cada repartido. Cada repartido tiene un archivo datos diferente.

Estas funciones siempre se están perfeccionando, debido a que se descubren desperfectos o se mejoran ciertos aspectos, como el uso de mensajes más informativos. Lamentablemente no siempre se pueden hacer pruebas completas del funcionamiento de dichas funciones. Esto se debe a que es muy difícil predecir todas las formas en que un estudiante puede intentar resolver un ejercicio.

Por esto es importante mantener actualizado el archivo datos. Con este objetivo se creó la función `fecha.datos`. Esta se carga en su sesión cada vez que usted corre `source("evaluar.R")`. Veamos un ejemplo:

```
> fecha.datos()
La fecha de su archivo datos es:
2013-08-28 13:34 CLT
Link para ver fecha de la útima versión:
https://www.dropbox.com/s/3kwtwpa6mvq9lhn/fecha-datos-rep-1.txt
Link para descargar la útima versión:
http://goo.gl/D5aYPW
```

Como puede ver, en la consola se imprime información relevante. Por un lado, se indica la fecha de creación del archivo datos que usted tiene en su carpeta. Por otro, se le da dos links: uno para verificar que la fecha de su archivo coincide con la de la última versión disponible y otro para descargar está última.

> Nota: muchas veces en Windows, al descargar un nuevo archivo datos, ocurre que el sistema le agrega la extensión .txt. Esto es un comportamiento poco deseable, pero no es un problema. Cada vez que usted corre `evaluar` R verifica si datos tiene alguna extensión y en caso de que así sea, se borra (alternativamente, puede usar este comando en R: `file.rename("datos.txt", "datos")`).

(4) Interfaz del foro

Para el curso es muy importante la comunicación fluida a través de la web. Para eso se utiliza un foro, alojado en reddit.com. También puede enviar correos electrónicos a cursosr@gmail.com o simplemente conectarse por chat. De todas formas es mucho más provechoso para todos plantear las dudas en el foro, ya que varios estudiantes pueden compartir una duda y por lo tanto beneficiarse por la respuesta recibida. Por esta razón, siempre se va a insistir en el uso por parte de los profesores.

4.1 Sobre reddit.com

La web reddit.com es un sitio que sirve para crear comunidades online en la que se comparten hipervínculos o textos. Las comunidades reciben el nombre

de “subreddits” o “subs” y tienen algún tema o consigna en particular. Nuestro curso tiene el subreddit “imser”, o como suele referenciarse en estos casos, r/imser.

El único usuario moderador (por el momento) de este foro es u/imser y es una cuenta manejada por los profesores del curso.

Figure 4: subscripción a r/imser.

Ejercicio (12): antes de empezar con este ejercicio, recomendamos continuar leyendo hasta el final de este documento. Posteriormente: hágase un usuario reddit y suscribase al r/imser. Luego comente en el post Ejercicio final del repartido 1..., siguiendo las instrucciones que allí se dan.

4.2 Suscripción.

Si usted ya hizo su usuario en reddit, ahora debe suscribirse al r/imser. Para esto sólo tiene que ir a la url de r/imser (reddit.com/r/imser) y apretar el botón

verde a la derecha que dice “suscribirse” (tiene que haber iniciado sesión con su usuario, por supuesto). En la figura 4 se muestra la interfaz de reddit y se indican varios detalles importantes:

- La barra lateral, contiene información que usted debe leer; se trata de las reglas de uso del foro.
- Los botones para crear nuevos posts. El de arriba es para crear un post a partir de un hipervínculo. El segundo es para crear un post de texto (cualquiera de los dos sirve y es posible hacer un post mixto).
- La casilla de mail de reddit, indicada como “PMs”, ya que PM es el acrónimo de “Personal Messages” (Mensajes Personales). El botón cambia de color a rojo cuando usted tiene nuevos mensajes.

Figure 5: creando un nuevo post en r/imser.

4.3 Crear un nuevo post.

Cuando usted quiera hacer un nuevo post, se encontrará con la interfaz que ve en la figura 5. Dado que es una interfaz muy intuitiva, sólo vamos a explicar aquí el estilo Markdown para dar formato al texto que escribimos en reddit.

Figure 6: leyendo un post

El estilo Markdown es una forma de dar formato básico al texto que resulta muy práctica y amigable (de hecho este mismo documento fue creado usando Markdown). En particular, ya que este es un curso de programación, nos interesa diferenciar con claridad lo que es `código` de lo que es texto. El estilo convencional usa alguna fuente "`monospace`" para esto. Para eso hay dos maneras básicas:

1. Si el código está incluido dentro de una línea de texto cualquiera, usamos los acentos graves para delimitarlos. Por ejemplo: `` `length(x)` `` (en teclados en español suele encontrarse arriba del Shift derecho y a izquierda del Enter, o a la derecha de la Ñ).

2. Si el código debe estar en una o varias líneas aparte, hay que dejar 4 espacios en blanco antes de cada línea. Por ejemplo:

```
Esto sería texto normal, antes de enviar el post...

    Esto sería código y se verá en alguna
    fuente monospace al enviar el post.
```

Cada vez que escriba un texto en reddit tome en cuenta estas y algunas otras reglas de formato (no son muchas). Recuerde que tiene la referencia en el borde mismo del espacio para escribir.

4.4 Leyendo / guardando un post.

Por último, en la figura 6 se muestra la interfaz de lectura de un post. En este caso no hay comentarios aún. Puede ver que hay varias opiciones:

- Votar por el post. Cada usuario puede votar a favor o en contra de un post. Esto permite seleccionar por la cantidad de votos a los más útiles.
- Guardar el post. Como usuario de reddit usted puede guardar posts que le resultan particularmente interesantes. Podrá acceder a ellos rápidamente en el futuro cuando usted lo desee.
- Marcar como solucionado. Cuando la pregunta del OP ("Original Poster", es el acrónimo estándar para referirse a quien inició la conversación) se ha respondido satisfactoriamente, usted puede agregarle la etiqueta de *Solucionado*. Esto es muy útil para estudiantes y más aún para los profesores.

  (Nota: la etiqueta solucionado es una versión modificada de la original de reddit "NSFW"; estando afuera de r/imser usted va a ver esa etiqueta en lugar de "Solucionado".)

Apéndice I: instrucciones generales para los repartidos

La información que aquí complementa la que se encuentra en el texto principal.

Archivos incluidos:

El archivo con los ejercicios de cada práctico debe bajarse y descomprimirse dentro de la carpeta del curso, creando la subcarpeta `rep-X` (`X` es el número de repartido). Usted deberá abrir RStudio y seleccionar dicha carpeta como su directorio de trabajo con `setwd` o **Ctrl+Shift+K**. En esta carpeta se encuentran algunos archivos que usted deberá modificar. En el archivo *letra.pdf* se indican cuáles son los archivos de ejercicio (scripts de R) y cuáles son los archivos que usted no debe modificar. Los ejercicios tienen nombres como `n-xxxxx.R`, en donde `n` es el número de ejercicio.

Mecanismo de corrección:

Lo primero que debe hacer es cargar el archivo `evaluar.R` con la función `source`:

```
source("evaluar.R", encoding = "UTF-8")
```

En caso de que ocurra un error o se vea otro mensaje en la consola, verifique que los archivos se descomprimieron correctamente y que usted está trabajando en la carpeta correspondiente con el comando `getwd()`.

Cada vez que termina un ejercicio, o cuando los hizo todos, ejecute:

```
evaluar()
```

Si usted no hace esto no habrá registro de sus notas en el archivo "datos" contenido en la carpeta del repartido. Recuerde que en todo momento puede verificar su puntaje con la función `verNotas()`.

Al finalizar

Una vez terminados los ejercicios (y usada la función `evaluar` para guardar sus notas), vaya a la página web del curso, a la sección de entregas. Allí usted deberá subir el archivo *datos*. Este será corregido posteriormente por los profesores del curso.

Entregas con retraso: en el curso es posible entregar con retraso los ejercicios de los repartidos, pero con un costo en el puntaje. Por ejemplo, si usted entrega al día siguiente de la fecha límite, tendrá un 5% de amonestación. Cada día extra es otro 5%: si en la página del curso se indica que usted entregó con 4 días y 12 horas de retraso, entonces el porcentaje de penalización total es de $5 \cdot 5 = 25$. Siguiendo el ejemplo anterior, tendría una nota de 75% para el repartido.

La fórmula general es, para D días y H horas de retraso (H > 0):

$$P = (D + 1) \cdot 5$$

(Cambia a $P = D \cdot 5$ si H $= 0$.)

Código de Honor

Si bien animamos a que trabaje en equipos y que haya un intercambio fluido en los foros del curso, es fundamental que las respuestas a los cuestionarios y ejercicios de programación sean fruto del trabajo individual. En particular, no utilize el código creado por sus compañeros, programe sus propias instrucciones. De lo contrario, estará saboteando su propio proceso de aprendizaje. Esto también implica evitar exponer el código propio a sus colegas. Como profesores estamos comprometidos a dar las herramientas y explicaciones adecuadas a fin de que pueda encontrar su propias soluciones para los ejercicios.

En casos de planteos de dudas a través del foro en los que considere que es imposible no exponer su código, hágalo, pero recuerde que una vez respondida la duda usted debe borrarlo. Si no lo hace los profesores podrán borrar su pregunta original.

Apéndice II: nota final del curso

El siguiente código sirve para calcular la nota final a partir de dos vectores: `notas.cuestionarios` y `notas.repartidos`.

```
nc <- mean(notas.cuestionarios)
nr <- min(100, mean(notas.repartidos))  # El máximo posible es 100
nota.final <- 100 * (0.65 * nr + 0.35 * nc)
```

Soluciones de los ejercicios

1) Errores:

1. En este caso hay un error en el nombre de la función: la función `mean` se escribe con minúsculas:

   ```
   mean(5:7, na.rm = TRUE)
   ```

2. Aquí hay dos interpretaciones posibles:

   a. que falta una coma entre 8.564432 y el 3:

   ```
   round(8.564432, 3)
   ```

   b. que sobra el espacio en blanco entre 8.564432 y 3:

   ```
   round(8.5644323)
   ```

   No podemos resolver con mayor precisión el problema ya que no tenemos el contexto en el que se ejecuta el comando.

3. El objeto `bigcity` no se encuentra, debido a que el paquete boot no fue cargado en la sesión. Se puede comprobar que es un objeto perteneciente a este paquete con el comando:

   ```
   ??bigcity
   ```

   Se soluciona el error cargando dicho paquete, así:

   ```
   library(boot)
   head(bigcity)
   ```

---

2) Uso de la función mean:

```
promedio <- mean(x)
```

---

3) Secuencias de números enteros:

```
10:10000
20:10
-8:6
6:-8
```

---

4) Secuencias de números (seq):

```
seq(2, 110, by = 2)
seq(1, 110, by = 2)
seq(9, 0, length = 101)
```

---

5) Concatenación

Ambas son aceptables:

```
mi.otro.vector <- c(45, -76, 3, 4, 5, 6, 0.333)
mi.otro.vector <- c(45, -76, 3:6, 0.333)
```

---

6) Modificación de un vector

Ambas son aceptables:

```
mi.vector[2:3] <- c(100, 104)
mi.vector[c(2, 3)] <- c(100, 104)
```

---

7) Vector invertido

```
mi.vector[6:1]
```

---

8) Matemática

a. Para acceder a la ayuda (ambas sirven):

```
?log
help("log")
```

El argumento se llama “base”

```
log(8, 3)
log(13, 5)
log(1.5, 10) # o log10(1.5)
log(7)       # o log(7, exp(1))
log(6, 2)    # o log2(6)
```

b. La función se llama `range`. El argumento `finite` y puede tomar dos valores: TRUE o FALSE.

```
range(x)                # 1 y 100
range(x)                # -100 y -1
range(x, finite = TRUE) # 3.141593 y 2980.957987
```

c. Polinomio:

```
x <- c(-1, 1.5, 4)
5 * x ^ 3 - 2 * x ^ 2 + 11
```

d. Seno y coseno, con 100 valores de `x` aleatorios:

```
x <- runif(100, -1, 1)
cos(x) ** 2 + sin(x) ** 2
# o
cos(x) ^ 2 + sin(x) ^ 2
```

e. Diagonal de un cuadrado:

```
a <- 1:100
D <- a * sqrt(2)
# o
D <- a * 2 ^ (1 / 2)
```

f. Sumatoria (número e), para cualquiera de los valores de `n`:

```
i <- 0:n
sum(1 / factorial(i))
```

g. Integral:

```
integral(log, 0, 20)
```

---

9) Solución genérica

```
mi.vector[length(mi.vector):1]
```

---

10) 1-varianza.R

```
out <- sum(s) / (length(x) - 1)
```

---

11) 2-zenon.R:

```
n <- 20
e <- 1:n
s <- 1 / (2 ** e)
out <- sum(s2)
```

Ejercicio extra del repartido