Universidad Nacional de Rosario
Facultad de Ciencias Económicas y Estadística
Licenciatura en Estadística | Licenciatura en Ciencia de Datos

Programación 1

Mgs. Lic. Marcos Prunello
Tec. Cesar Mignoni
Lic. Gino Bartolelli
Lic. Iván Millanes
Julián L'Heureux

Año 2025

Tabla de contenidos

Bienvenida

¡Hola!

Bienvenidos a este curso introductorio de programación para estudiantes de Estadística y Ciencia de Datos. A lo largo de estas páginas, nos embarcaremos en un recorrido que tiene como objetivo fundamental desarrollar habilidades de pensamiento algorítmico y resolución de problemas computacionales. Aprender a programar no es solo escribir código: implica descomponer problemas complejos en partes más pequeñas, identificar patrones, diseñar soluciones y comprender cómo una computadora ejecuta instrucciones paso a paso.

En este camino utilizaremos **R**, un lenguaje ampliamente utilizado en Estadística y Ciencia de Datos. Sin embargo, **en este curso no nos enfocaremos en sus poderosas herramientas de modelado, manipulación de datos o visualización**. Aprenderás de eso en otras asignaturas, como en **Laboratorio de Datos 1**. En su lugar, usaremos R como vehículo para aprender los principios esenciales de la programación. Por ejemplo, veremos cómo lograr que la computadora haga una división, cuando en realidad es algo sencillo que R o cualquier otro entorno ya lo sabe hacer. Este enfoque nos permitirá desarrollar habilidades fundamentales: estructurar código de manera ordenada, depurar errores con criterio, organizar proyectos de software y escribir funciones reutilizables. Aprenderemos a pensar como programadores, lo que no solo facilitará el uso posterior de herramientas más avanzadas, sino que también brindará una base sólida para afrontar nuevos lenguajes y paradigmas de programación.

La idea es que desarrollemos nuestro pensamiento lógico y algorítmico, mientras nos familiarizamos con el uso de nuestra computadora y archivos, con el objetivo de que salgas listo para seguir profundizando tus conocimientos, por ejemplo, en la asignatura **Programación 2**, donde usarás Python.

¡Comencemos este viaje!

Estructura del libro

Este libro se organiza en **unidades**, cada una de las cuales se corresponde con una unidad del programa de la asignatura y está compuesta por capítulos que abordan un tema en particular. De vez en cuando se incluyen algunos elementos especiales, que están identificados de la siguiente forma:

RESUMEN
Esto es la presentación de una unidad o un capítulo.

EJEMPLO
Esto es un ejemplo.

PARA RESOLVER

Esto es un ejercicio.

CONCEPTO CLAVE
Esto es una definición.

INFO IMPORANTE
Esto es una idea fundamental para tener en cuenta.

COMENTARIO ADICIONAL

Esto es un comentario adicional, que puede ser considerado como secundario o dejado de lado.

Licencia

Información general sobre la asignatura

Programa

En este enlace podés consultar el programa para esta asignatura.

Aula virtual

Esta asignatura posee un espacio virtual en la plataforma Comunidades 3 de la Universidad Nacional de Rosario, que será utilizado para dar anuncios y noticias, responder consultas en los foros, entregar y realizar devoluciones sobre los trabajos prácticos, informar calificaciones, etc. Es importante que visites este espacio de forma regular. Si no te ha llegado la clave para poder ingresar, contactá a los docentes.

Cursado

Actividades semanales

Cada semana durante la extensión del cuatrimestre encontrarás las indicaciones referidas a los contenidos que debés cubrir para avanzar en tiempo y forma con esta asignatura. En este cronograma de trabajo verás de forma detallada cuáles son los capítulos del libro a estudiar y los ejercicios de las prácticas a resolver, semana a semana. Todos los materiales estarán publicados con anticipación, incluyendo las respuestas a los problemas, de modo que puedas autogestionar tus tiempos. Además, en el cronograma podrás ver todas las fechas importantes que deben ser respetadas, como las de las evaluaciones.

Clases

Como acompañamiento a tu propio avance por la asignatura, cada semana podés asistir a dos clases:

- Una es virtual y se dicta los días lunes de 14:30 a 16:30 para todo el estudiantado. Estas clases tienen el objetivo de presentar los contenidos conceptuales de cada semana, mediante la exposición interactiva del tema a abordar.

- La otra clase es presencial y se dicta en el Laboratorio de Computación de la Escuela de Estadística, para cada comisión por separado.

Si por alguna razón una clase se suspende (por ejemplo, es feriado), ¡no te preocupes! Todo el material correspondiente a la semana en cuestión estará publicado y podrás resolver tus dudas en las consultas presenciales o virtuales, o en los foros del aula virtual. Tenés que tener en cuenta la semana siguiente no repasaremos el contenido de la semana en la que no tuviste clase, sino que avanzaremos con los temas que tocan esa semana, según el cronograma.

Información importante para las clases, como enlaces para conectarte o días y horarios de cada comisón, son publicados en el aula virtual.

Consultas

Podés resolver tus dudas recurriendo a estos medios:

- **Foros de consulta en el aula virtual**.
- **Clases de consulta virtuales**. Los enlaces y horarios se publican en el aula virtual. Si ningún estudiante se conecta, la clase se da por finalizada luego de 10 minutos de espera.
- **Clases de consulta presenciales en el laboratorio de computación**. Estas clases tendrán una duración determinada. Durante ese tiempo, además de resolver dudas con los docentes, podés recurrir al laboratorio para hacer uso de las computadoras y resolver ejercicios u otros trabajos.

Cronograma

Visitá periódicamente este cronograma para autogestionar tu tránsito por los temas y materiales de la asignatura:

Semana	Inicio / Fin	Tópico	Capítulos a leer	Ejercicios a resolver	Importante
1	17/3 al 23/3	Unidad 1	1 a 3	1 a 4	
2	24/3 al 30/3	Unidad 1	4	5 a 9	Feriado: lun 24/3
3	31/3 al 6/4	Unidades 1 y 2	5 a 7	10 a 13 (Unidad 1)	Feriado: mié 2/4
4	7/4 al 13/4	Unidad 2	10	1 a 6	
5	14/4 al 20/4	Unidad 2	11 y 12	7 a 9	Feriados: Jue 17/4 y Vie 18/4
6	21/4 al 27/4	Unidad 3	14	1 a 5	
7	28/4 al 4/5	Unidad 3	15 y 16	6 a 12	Feriados: Jue 1/5 y Vie 2/5

Semana	Inicio / Fin	Tópico	Capítulos a leer	Ejercicios a resolver	Importante
8	5/5 al 11/5	Semana del parcial	Estudio y repaso	Estudio y repaso	**Parcial**: jue 8/5 a las 18:00, todas las comisiones juntas
9	12/5 al 18/5	Unidad 4	19 a 21	Todos los ejercicios	
10	19/5 al 25/5	Unidad 5	24	A definir	
11	26/5 al 1/6	Unidad 5	25	A definir	
12	2/6 al 8/6	Unidad 5	26	A definir	
13	9/6 al 15/6	Recup. / Unidad 6	A definir	A definir	**Entrega TP**: martes 10/6. **Recup**: jue 12/6 a la tardecita, todas las comisiones juntas
14	16/6 al 22/6	Unidad 6	A definir	A definir	**Entrega tarea promoción**: dom 22/6. Feriados: lun 16/6 y vie 20/6
15	23/6 al 29/6	Unidad 6	A definir	A definir	Vie 27/6: notificación de la condición final en la asignatura

Evaluación

La evaluación de la asignatura se realiza a través de un examen parcial individual y un trabajo práctico grupal. Ambas actividades son evaluadas con una calificación entre 0 y 10 y se ponderan para obtener una nota de cursada (60% examen parcial, 40% trabajo práctico). Al finalizar el cursado, cada estudiante obtiene una de las siguientes condiciones:

- **Libre**: obtienen la condición de "libre" aquellos estudiantes con nota de cursada inferior a 6 o alguna nota menor a 4, ya sea en el examen parcial o en el trabajo práctico.
- **Regular**: obtienen la condición de "regular" aquellos estudiantes con nota de cursada mayor o igual a 6, siempre que no tengan ninguna nota menor a 4 en el examen parcial o en el trabajo práctico
- **Promovido**: obtienen la condición de "promovido" aquellos estudiantes que con nota de cursada mayor o igual a 8, que no tengan ninguna nota menor a 6 en el examen parcial o en el trabajo práctico y que aprueben una tarea de realización opcional e individual al final del cuatrimestre. La nota de cursada se computa como nota final de la asignatura.

Más detalles sobre cada componente evaluativo pueden encontrarse en el programa o en el aula virtual.

Código de conducta

En esta asignatura promovemos un ambiente de respeto, colaboración y aprendizaje mutuo. Para garantizar una experiencia positiva para todas las personas involucradas, establecemos las siguientes normas de convivencia:

Respeto y trato cordial

- Todas las personas deben ser tratadas con respeto, independientemente de su experiencia, formación, identidad de género, origen, creencias u opiniones.
- Se espera un trato cordial y profesional entre estudiantes, docentes y personal de la facultad.
- No se tolerarán expresiones de discriminación, violencia verbal, acoso o cualquier forma de descalificación personal.

Participación y comunicación

- Se fomenta el intercambio de ideas y el debate académico en un marco de respeto.
- Se debe permitir que todas las voces sean escuchadas sin interrupciones ni actitudes despectivas.
- El uso de lenguaje claro y respetuoso es fundamental, tanto en interacciones presenciales como en medios digitales relacionados con la asignatura.
- Se valora la cooperación en actividades grupales, con una distribución equitativa de tareas y reconocimiento del aporte de cada integrante.
- Se invita a hacer preguntas y solicitar aclaraciones cuando sea necesario, en un clima de confianza y respeto.
- Los comentarios hacia otros estudiantes y docentes deben ser constructivos y enfocados en la mejora del aprendizaje.

Responsabilidad y compromiso

- La entrega de trabajos deben realizarse con responsabilidad. En caso de dificultades, se recomienda comunicarse con el equipo docente con anticipación.
- Se espera honestidad académica en todas las actividades del curso.
- El uso de dispositivos electrónicos en clase debe estar alineado con los objetivos de la misma, evitando distracciones innecesarias.

Uso responsable de inteligencia artificial para resolver problemas

- Las herramientas de inteligencia artificial (IA) pueden ser útiles en ciertos contextos, pero su uso excesivo o inadecuado puede limitar el aprendizaje real. Es fundamental que los estudiantes realicen el esfuerzo de comprender y desarrollar las soluciones por sí mismos.

- Para aprender a programar y desarrollar pensamiento lógico, es necesario escribir código, enfrentar errores y corregirlos. Depender demasiado de la IA puede generar una falsa sensación de dominio sin una comprensión profunda.

- En las evaluaciones y entregas, **queda estrictamente prohibido el plagio y todo tipo de copia**, incluyendo el uso de respuestas generadas por IA sin una comprensión real del contenido.

- Si se detecta plagio o copia en cualquier instancia de evaluación, o falta de comprensión de las soluciones entregadas, se aplicarán las medidas correspondientes según las normativas de la institución.

El cumplimiento de estas normas contribuye a un entorno en el que todas las personas puedan participar activamente y aprovechar al máximo la experiencia educativa. Ante cualquier inquietud o situación que afecte la convivencia, se recomienda comunicarlo al equipo docente o a las autoridades institucionales.

Unidad 1. Introducción a la programación con R

RESUMEN

Esta unidad introduce los conceptos fundamentales para comenzar a programar en R, proporcionando las bases necesarias para desarrollar código claro, estructurado y funcional. Iniciamos con una exploración de qué significa programar, abordando las etapas del diseño algorítmico y la codificación. Luego, presentamos R y RStudio, sus características principales y su instalación, junto con los primeros comandos y estructuras esenciales del lenguaje.

A lo largo de la unidad, aprenderemos sobre los distintos tipos de objetos y operadores en R, así como la organización eficiente del trabajo mediante archivos, proyectos y buenas prácticas. También abordaremos la gestión de errores, la importancia de seguir una guía de estilo para escribir código legible y el uso de paquetes para ampliar las capacidades del lenguaje. Finalmente, incluimos una reseña histórica de la computación y una introducción a los lenguajes de programación y sus niveles de abstracción.

Capítulo 1

Programación

RESUMEN

En este capítulo definiremos qué es la programación y exploraremos las etapas del diseño algorítmico y la codificación, aspectos fundamentales para desarrollar soluciones computacionales. Luego, introduciremos R como lenguaje de programación y explicaremos por qué es una herramienta tan utilizada en Estadística y Ciencia de Datos.

1.1 Qué es la programación

Las computadoras son una parte esencial de nuestra vida cotidiana. Casi todos los aparatos que usamos tienen algún tipo de computadora capaz de ejecutar ciertas tareas: lavarropas con distintos modos de lavado, consolas de juegos para momentos de entretenimiento, calculadoras súper potentes, computadoras personales que se usan para un montón de propósitos, teléfonos celulares con un sinfín de aplicaciones y miles de cosas más.

Todos estos dispositivos con computadoras de distinto tipo tienen algo en común: alguien "les dice" cómo funcionar, es decir, les indica cuáles son los pasos que deben seguir para cumplir una tarea. De eso se trata la **programación**: es la actividad mediante la cual las *personas* (o algoritmos de inteligencia artificial) le entregan a una *computadora* un conjunto de instrucciones para que, al ejecutarlas, ésta pueda *resolver un problema*. Los conjuntos de instrucciones que reciben las computadoras reciben el nombre de *programas*.

La programación es un proceso creativo: en muchas ocasiones la tarea en cuestión puede cumplirse siguiendo distintos caminos y al **programar** debemos imaginar cuáles son y elegir uno. Algunos de estos caminos pueden ser mejores que otros, pero en cualquier caso la computadora se limitará a seguir las instrucciones ideadas por nosotros.

Estas instrucciones deben ser transmitidas en un idioma que la computadora pueda entender sin ambigüedad. Para eso debemos aprender algún **lenguaje de programación**, que no es más que un lenguaje artificial compuesto por una serie de expresiones que la computadora puede interpretar. Las computadoras interpretan nuestras instrucciones de forma muy literal, por lo tanto a la hora de programar hay que ser muy específicos. Es necesario respetar las reglas del lenguaje de programación y ser claros en las indicaciones provistas.

Ahora bien, ¿por qué debemos estudiar programación en carreras como Licenciatura en Estadística y Licenciatura en Ciencia de Datos? La actividad de los profesionales que trabajamos con datos está atravesada en su totalidad por la necesidad de manejar con soltura herramientas informáticas que nos asisten en las distintas etapas de nuestra labor: recolección y depuración de conjuntos de datos, aplicación de distintas metodologías de análisis, visualización de la información, comunicación efectiva de los resultados, despliegue y puesta en producción de modelos, etc. Por eso, en la asignatura **Programación 1** estudiaremos los conceptos básicos de esta disciplina, fomentando la ejercitación del pensamiento abstracto y lógico necesario para poder entendernos hábilmente con la computadora y lograr que la misma realice las tareas que necesitamos.

1.2 Etapas en la programación

Mencionamos anteriormente que la *programación* consiste en instruir a una computadora para que resuelva un problema y que la comunicación de esas instrucciones debe ser realizada de forma clara. Es por eso que, ante un problema que debe ser resuelto computacionalmente, el primer paso es pensar detalladamente cuál puede ser una forma de resolverlo, es decir, crear un *algoritmo*.

CONCEPTO CLAVE
Un **algoritmo** es una estrategia consistente de un conjunto ordenado de pasos que nos lleva a la solución de un problema o alcance de un objetivo. Luego, hay que traducir el algoritmo elegido al idioma de la computadora.

Entonces, podemos decir que la resolución computacional de un problema consiste de dos etapas básicas:

1.2.1 Diseño algorítmico

Cotidianamente, hacemos uso de algoritmos para llevar adelante casi todas las actividades que realizamos: preparar el desayuno, sacar a pasear la mascota, poner en la tele un servicio de *streaming* para ver una película, etc. Cada una de estas tareas requiere llevar adelante algunas acciones de forma ordenada, aunque no hagamos un listado de las mismas y procedamos casi sin pensar.

Sin embargo, cuando estamos pensando la solución para un problema que va a resolver programando, debemos pensar uno por uno cuáles son todos los pasos a seguir, para asegurarnos de que cuando la computadora los siga, llegue a la solución. Suele ser útil escribir en borrador cuáles son esos pasos, de forma clara y ordenada, e incluso hacer diagramas. Una vez que hemos imaginado como resolver el problema mediante programación, podemos pasar a la siguiente etapa.

1.2.2 Codificación

Una vez que tenemos diseñada la solución al problema, hay que comunicársela a la computadora para que la siga. Para que ella pueda entender nuestro algoritmo, debemos traducirlo en un *lenguaje de programación*, que, como dijimos antes, es un idioma artificial diseñado para expresar cómputos que puedan ser llevados a cabo por equipos electrónicos, es decir es un medio de comunicación entre el humano y la máquina.

Al aprender sobre programación, comenzamos enfrentándonos a problemas simples para los cuales la etapa del diseño algorítmico parece sencilla, mientras que la codificación se torna dificultosa, ya que hay que aprender las reglas del lenguaje de programación. Sin embargo, mientras que con práctica rápidamente podemos ganar facilidad para la escritura de código, el diseño algorítmico se torna cada vez más desafiante al encarar problemas más complejos.

1.3 El lenguaje R y su importancia en Estadística y Ciencia de Datos

R es un lenguaje de programación y un entorno de software diseñado específicamente para el análisis estadístico y la visualización de datos. Fue creado por Ross Ihaka y Robert Gentleman como una implementación libre del lenguaje S, desarrollado en los laboratorios AT&T Bell. Desde su lanzamiento, R ha crecido hasta convertirse en una de las herramientas más utilizadas en Estadística, Ciencia de Datos e investigación en una amplia variedad de disciplinas. Su código es abierto y es mantenido por una comunidad activa de desarrolladores y usuarios, con actualizaciones constantes y una enorme cantidad de paquetes adicionales que amplían sus capacidades.

Una de las principales razones por las que R es tan popular es su flexibilidad. A diferencia de herramientas de software más rígidas, que solo permiten aplicar métodos predefinidos, R ofrece un lenguaje de programación completo que permite desarrollar soluciones adaptadas a problemas específicos. Esto significa que los usuarios pueden definir sus propias funciones, automatizar procesos y realizar simulaciones personalizadas, algo fundamental cuando se trabaja con problemas complejos que requieren enfoques innovadores.

Sin embargo, esta flexibilidad también implica una curva de aprendizaje más pronunciada en comparación con herramientas en las que sólo hay apuntar con el mouse y hacer clic para elegir algunas opciones en un menú. Algunos programadores provenientes de otros lenguajes pueden encontrar a R poco intuitivo o un tanto ambiguo en algunas cuestiones, pero es que ha sido pensado como una herramienta de programación para profesionales no formados en esa disciplina.

La comunidad de desarrolladores y usuarios de R ha desarrollado múltiples soluciones para hacer el lenguaje cada vez más eficiente y accesible. Existen paquetes optimizados para manejar grandes volúmenes de datos con rapidez y entornos o plataformas como RStudio, Shiny y Quarto que han ampliado el alcance de R, permitiendo desde el desarrollo de aplicaciones interactivas hasta la implementación de modelos en producción. Gracias a este continuo desarrollo del ecosistema R, este software es una opción potente y versátil para la Estadística y Ciencia de Datos.

INFO IMPORANTE

Este libro, y la asignatura a la que pertenece, no se enfoca en las poderosas herramientas de análisis de datos que ofrece R, como la modelización estadística, la manipulación de datos o la visualización gráfica. En su lugar, enseñaremos fundamentos generales de programación, y los ejemplificaresmos particularemente con R. Por ejemplo, en lugar de usar funciones que ya fueron programadas por otras personas, reinventaremos la rueda y crearemos nuestras propias funciones; o en lugar de usar paquetes que nos ayudan a realizar procesos iterativos con gran eficiencia, exploraremos estructuras de control tradicionales como bucles. Es decir, frente a variados problemas vamos a dedicarnos a crear soluciones que ya existen y están disponibles en R, pero lo haremos con el fin de utilizar dicho lenguaje para aprender y ejercitar nociones básicas de programación. Esto nos ayudará a desarrollar un pensamiento algorítmico y ganar habilidad para generar soluciones propias y comprender con mayor profundidad lo que sucede “detrás de escena” en cada análisis.

Capítulo 2

Primeros pasos con R y RStudio

RESUMEN

En este capítulo daremos nuestros primeros pasos con R y RStudio, una interfaz que simplica el uso de este lenguaje de programación. Conoceremos sus características y veremos cómo instalar ambos softwares. Aprenderemos a ejecutar comandos en la consola, escribir y guardar código en *scripts*, y utilizaremos nuestras primeras funciones en R. Estos conceptos sentarán las bases para programar de manera organizada y eficiente en los próximos capítulos.

2.1 R y RStudio

Si bien R será nuestro medio de comunicación con la computadora, vamos a usarlo a través de otro programa que brinda algunas herramientas para facilitar nuestro trabajo de programación, es decir, vamos a usar un **entorno integrado de desarrollo** (o *IDE*, por *integrated development environment*). Un IDE es un programa que hace que la codificación sea más sencilla porque permite manejar varios archivos de código, visualizar el *ambiente* de trabajo, utilizar resaltado con colores para distintas partes del código, emplear autocompletado para escribir más rápido, explorar páginas de ayuda, implementar estrategias de depuración e incluso intercalar la ejecución de instrucciones con la visualización de los resultados mientras avanzamos en el análisis o solución del problema. El IDE más popularmente empleado para programar con R es **RStudio**.

2.2 Instalación

Para instalar estos programas, se deben visitar las páginas oficiales de R y de RStudio, descargar los instaladores y ejecutarlos. En este documento encontrarás una guía paso a paso, o también podés mirar este video con las indicaciones.

Si experimentás algún problema con la instalación, hay una alternativa para que no pierdas tiempo hasta que los docentes puedan ayudarte a resolverlo. RStudio puede ser usado online sin que lo tengas que instalar. Sólo necesitás conexión a internet. Si necesitás usar esto porque la instalación falló, seguí las instrucciones de este archivo.

2.3 Paneles de RStudio

Cuando se abre RStudio se pueden visualizar tres paneles principales (Figura 2.1):

- En el panel de la izquierda la pestaña más importante es **Console** (*consola*), que es donde se ejecutan las instrucciones de R en tiempo real. Es la ventana que usamos para comunicarnos con R. Ahí se escriben las instrucciones para que R las *evalúe* (también decimos, que las *ejecute* o *corra*) y también es el lugar donde se visualizan los resultados.
- En el panel de arriba a la derecha la pestaña más importante es **Environment** (*entorno* o *ambiente*), que se encarga de mostrar los elementos que tenemos a disposición para programar. Al inicio de la sesión de trabajo, se encuentra vacío.
- En el panel de abajo a la derecha, las pestañas más importantes son:
  - **Files**: explorador de archivos de la computadora.
  - **Plots**: ventana donde aparecen los gráficos si es que nuestro código produce alguno (no lo usaremos en este curso).
  - **Packages**: listado de los "paquetes" (herramientas adicionales) que tenemos instalados.
  - **Help**: manual de ayuda de R.

Más adelante profundizaremos en el uso de estos componentes de RStudio.

2.4 Uso de la consola de R

La consola de R en RStudio permite ejecutar comandos de manera inmediata. Al presionar `Enter` las instrucciones escritas serán evaluadas, produciendo algún resultado. Por ejemplo, podemos escribir expresiones matemáticas sencillas, como la suma `1 + 2`. Para esto, tenemos que hacer clic en la última línea de la consola, al lado del indicador `>` (llamado *prompt*), para asegurarnos que el cursor esté allí titilando. La presencia del *prompt* `>` en esa última línea nos indica que R está preparado para recibir una nueva instrucción. Escribimos ahí la cuenta `1 + 2` y pulsamos `Enter` (Figura 2.2).

El resultado o **salida** se ve inmediatamente debajo de la instrucción: se trata del número `3`, por supuesto. Antes aparece la anotación `[1]`, que indica que la primera y única línea de la salida muestra el primer y único resultado de la instrucción evaluada. En algunas operaciones, la salida está compuesta por muchos elementos y ocupa varias líneas. En ese caso R muestra un número entre corchetes al comienzo de cada línea de la salida, para darnos una idea de cuántos elementos nos está mostrando. Por ahora podemos ignorarlo.

Probemos con más cálculos matemáticos:

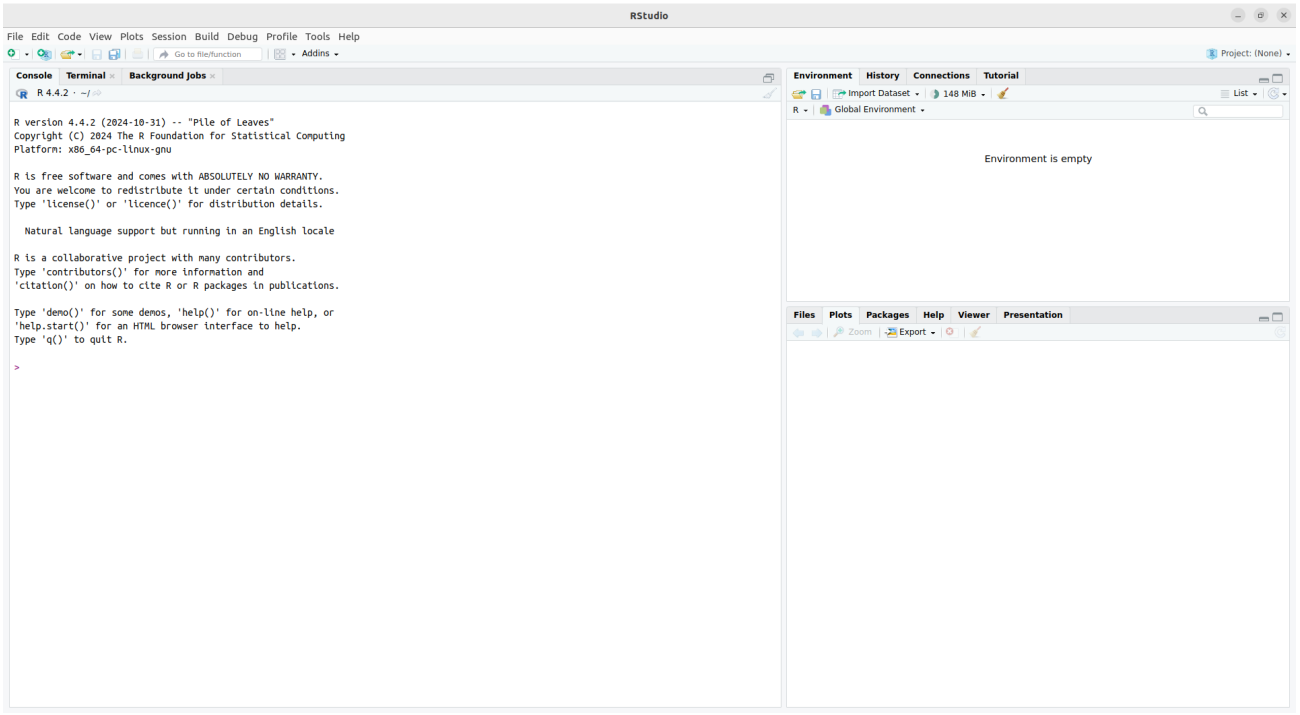


Figura 2.1: RStudio al abrirlo.

```
1 + 2
```

```
[1] 3
```

```
5 * 3
```

```
[1] 15
```

```
100 / 4
```

```
[1] 25
```

```
3^2
```

```
[1] 9
```

```
3 - (2 * 9)
```

```
[1] -15
```

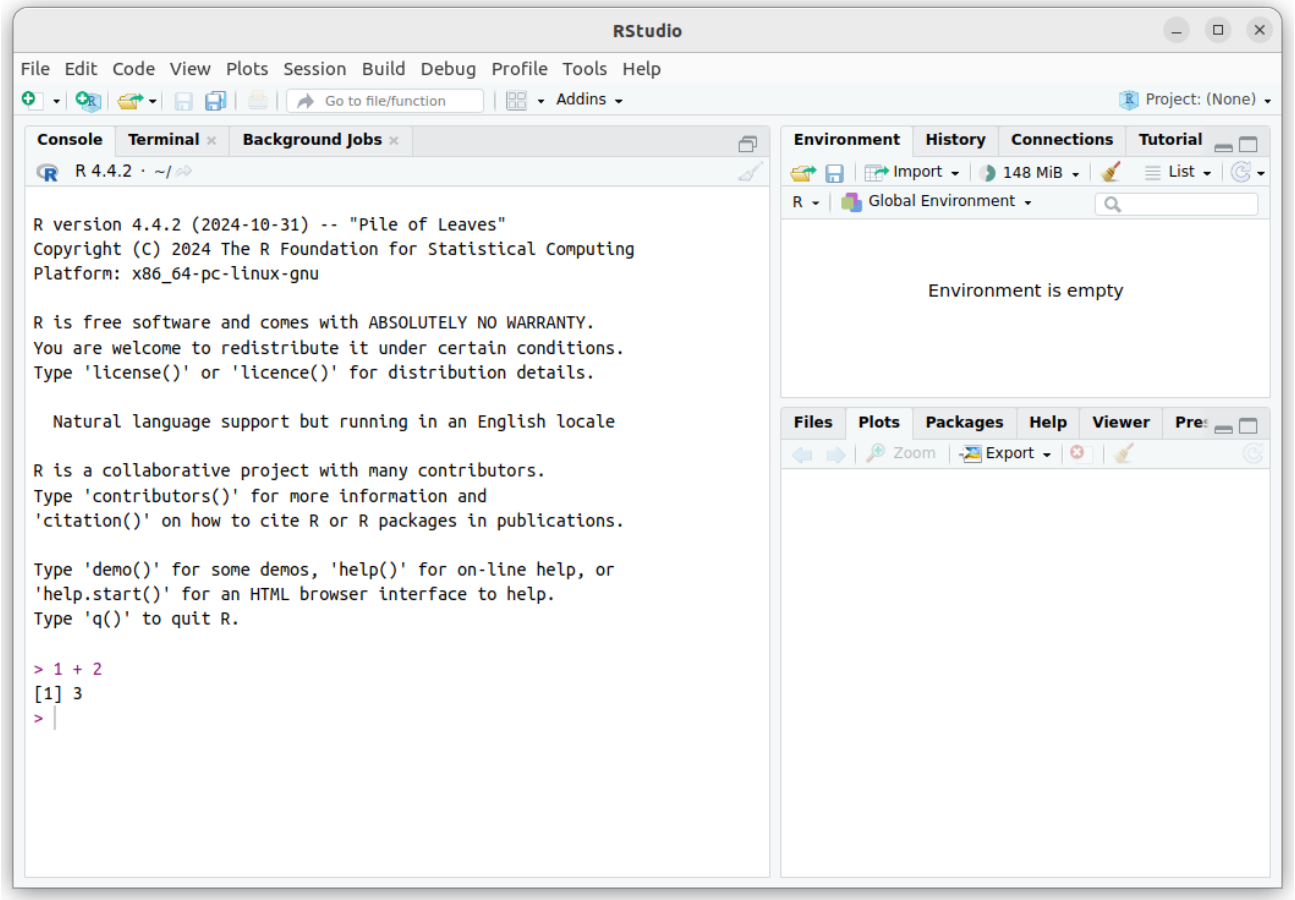


Figura 2.2: Ejemplo de una primera instrucción de programación en: la suma entre 1 y 2.

INFO IMPORANTE

A veces nos pasa que escribimos una instrucción de forma incompleta y presionamos `Enter`. En esta situación, la consola muestra al comienzo de la línea el símbolo `+`, señalando que falta "algo más" para que el comando esté completo y se pueda mostrar el resultado. Tenemos que completar lo que falta y presionar `Enter` otra vez, o presionar `Esc` para cancelar esa instrucción y que la consola vuelva a mostrar el *prompt* `>`, indicando que podemos escribir el código de nuevo desde cero.

En el siguiente ejemplo, en la consola escribí `100 /` y presioné `Enter` dos veces. Como la cuenta quedó incompleta, la consola muestra el `+`:

```
> 100 /
+
+
```

La solución es apretar `Esc` para cancelar, o completar la instrucción:

```
> 100 /
+
+
+ 4
```

```
[1] 25
```

Es importante reconocer que no podemos escribir una nueva instrucción en la consola cuando está el `+` porque algo de lo anterior quedó incompleto. Debemos solucionarlo, ver que aparezca otra vez el `>` y entonces sí volver a escribir un comando.

Por otro lado, si escribimos una instrucción que R no sabe interpretar o que presenta algún tipo de problema, la salida mostrará un mensaje de error. Por ejemplo, el símbolo para hacer divisiones es `/` y no `%`. Si lo usamos, pasa esto:

```
100 % 4
```

```
Error in parse(text = input): <text>:1:5: unexpected input
1: 100 % 4
        ^
```

2.5 Archivos de código o *scripts*

Hasta ahora hemos usado la consola de R para ejecutar comandos de manera inmediata. Sin embargo, cuando trabajamos con tareas de programación más complejas, es importante **guardar nuestro**

código para poder reutilizarlo, modificarlo y compartirlo. Para esto, usamos los **scripts**.

CONCEPTO CLAVE
Un **archivo de código** o **script** es un archivo de texto que contiene una serie de instrucciones de escritas en algún lenguaje de programación. En lugar de escribir y ejecutar los comandos uno por uno en la consola, podemos escribirlos en un script y ejecutarlos cuando sea necesario. Esto nos permite organizar mejor nuestro trabajo y evitar repetir tareas manualmente. A veces usamos el término **programa** como sinónimo de script.

El uso de scripts en lugar de escribir código directamente en la consola tiene varias ventajas:

- **Reproducibilidad**: podemos volver a ejecutar nuestro programa sin necesidad de reescribirlo.
- **Organización**: podemos estructurar el código en secciones claras.
- **Depuración**: es más fácil detectar y corregir errores en un script que en la consola.

2.5.1 Crear un script

Para crear un nuevo script en RStudio podemos seguir algunas de estas opciones:

- Ir a `File > New > R Script`.
- Usar el atajo `Ctrl + Shift + N`.
- Hacer clic en el primer ícono de la barra de herramientas (hoja en blanco con signo +)

El sector izquierdo de RStudio se subdivide en dos paneles: abajo queda la consola y arriba aparece el editor de scripts (Figura 2.5). Podemos crear o abrir más de un script a la vez, cada uno aparece como una pestaña de este panel.

2.5.2 Escribir código y guardar el script

Una vez creado el script, podemos escribir ahí todo nuestro código de R. Para no perder el trabajo debemos guardar este documento en nuestra computadora, con alguna de estas opciones:

- Ir a `File > Save`.
- Usar el atajo `CTRL + S`.
- Usar el ícono de guardar en la barra de herramientas.

La primera vez que guardamos el script recién creado, tendremos que elegir un nombre para el archivo y un lugar en la computadora para su ubicación. En el caso presentado en la Figura 2.4, elegimos el nombre "ejemplos".

Ahora la pestaña del editor de scripts muestra el nombre elegido para el archivo seguido por `.R`, es decir, vemos que dice: `ejemplos.R`. El nombre de un archivo informático se compone de dos partes: la **raíz**

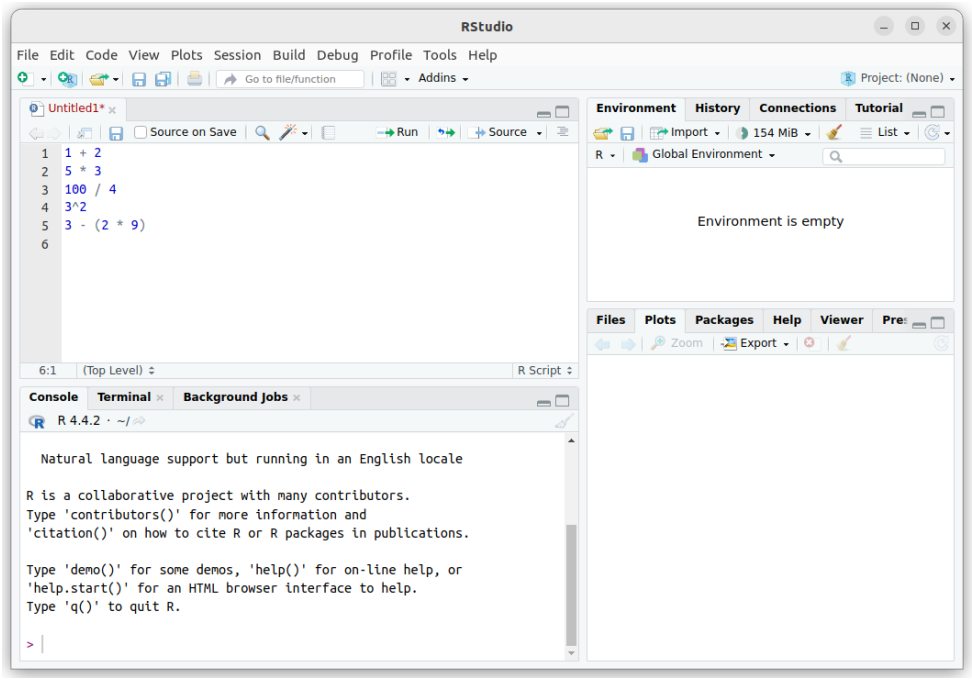


Figura 2.3: Código escrito en un nuevo script.

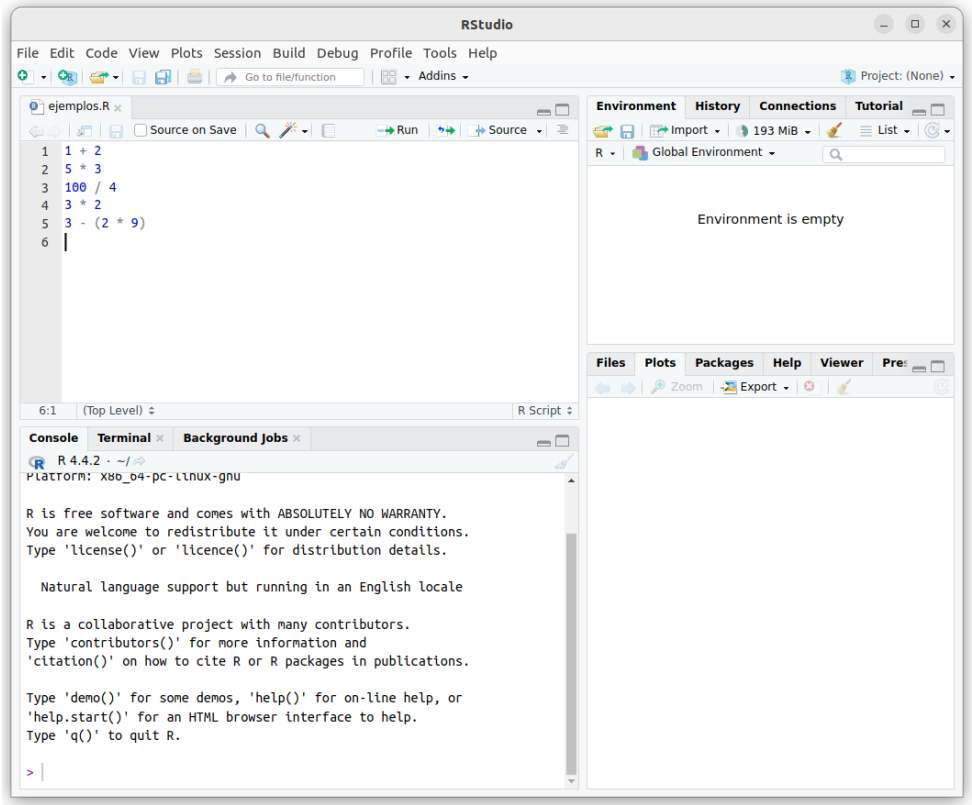


Figura 2.4: El nuevo script ha sido guardado bajo el nombre ‘ejemplos.R.

y la **extensión**. La raíz es el nombre principal que elegimos para identificar el archivo (`ejemplos`), mientras que la extensión es un sufijo separado por un punto que indica el tipo de archivo y con qué programas se puede abrir. Por ejemplo, los archivos de texto suelen tener la extensión `.txt`, las imágenes pueden ser `.jpg` o `.png` y las hojas de cálculo de Excel suelen ser `.xlsx`. En el caso de los scripts de R, la extensión es `.R`, lo que indica que el archivo contiene código en el lenguaje R y puede ser ejecutado dentro de RStudio o cualquier otro entorno compatible. Al guardar un archivo de código, se agrega automáticamente la extensión en el nombre y esto nos permite organizarlos adecuadamente y asegurarnos de que R los reconozca como archivos de código.

INFO IMPORANTE

Si buscamos el archivo recién creado en el Explorador de archivos de Windows, puede que en su nombre no se vea la extensión. Windows, por defecto, oculta las extensiones de los archivos, pero es posible mostrarlas siguiendo estos pasos:

1. Abrir el Explorador de archivos.
2. Acceder a la configuración de vista:
   - En **Windows 10 y 11**, hacé clic en la pestaña **Vista** en la parte superior.
   - En **Windows 11**, si no ves la pestaña, haz clic en **Ver > Mostrar**.
3. Activar la visualización de extensiones: marcá la opción **"Extensiones de nombre de archivo"**.

Habilitar esta opción es útil para evitar confusiones entre tipos de archivos. Dos archivos diferentes pueden tener en su nombre la misma raíz, pero tratarse de distintas cosas porque tienen diferente extensión.

A medida que seguimos editando nuestro script de código agregando nuevas instrucciones de programación, es conveniente guardar frecuentemente los cambios añadidos.

2.5.3 Ejecutar código desde un script

Escribir código en un script no lo ejecuta automáticamente. Para ejecutarlo, podemos:

- Seleccionar una o varias líneas del script y presionar `Ctrl + Enter` (Windows/Linux) o `Cmd + Enter` (Mac).
- Seleccionar una o varias líneas y hacer clic en el botón `Run` en la parte superior del editor.

Si empleamos cualquiera de esas opciones sin tener líneas de código seleccionadas, se ejecutará una sola línea, aquella sobre la cual esté colocado el cursor.

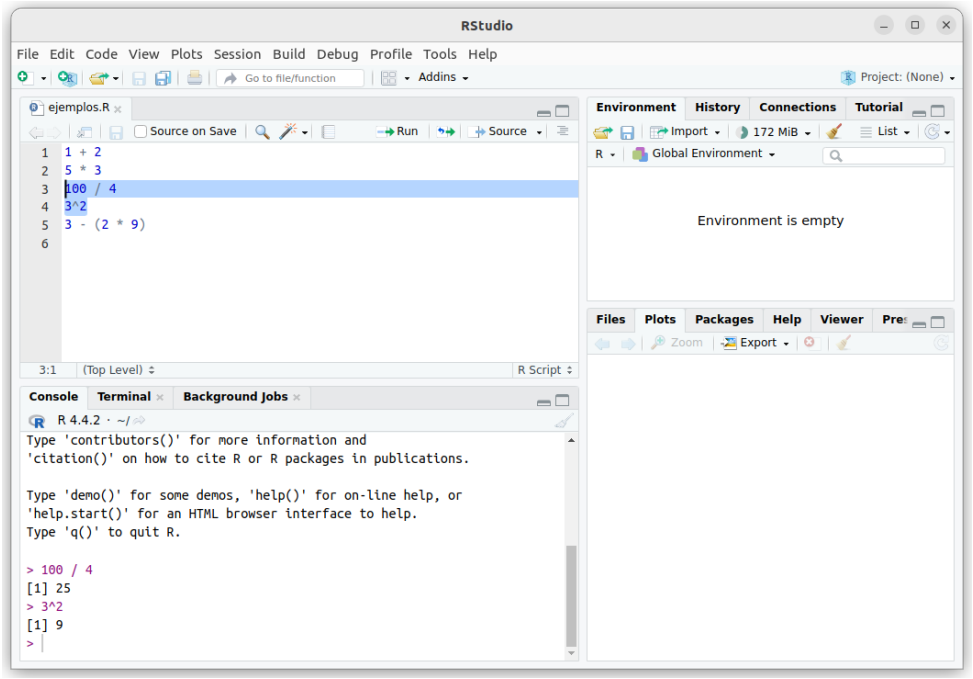


Figura 2.5: Se han ejecutado las líneas seleccionadas del script. En la consola se ven los resultados.

2.5.4 Comentarios en el código

En todo lenguaje de programación existe un carácter especial que, al ser colocado al comienzo de una línea de código, le indica al software que dicha línea no debe ser evaluada. Esto se utiliza para incluir **comentarios**, es decir, líneas escritas en español que ayudan a documentar lo que hace cada parte de nuestro programa. Los comentarios no afectan la ejecución del código y son fundamentales para hacer que nuestro trabajo sea comprensible para nosotros y otras personas. También sirven para marcar distintas partes del script. En R, los comentarios se escriben con el símbolo `#` (Figura 2.6).

2.6 Funciones

En los ejemplos anteriores hemos realizado algunas operaciones básicas, como sumas o multiplicaciones. Otros cálculos matemáticos requieren que usemos **funciones**. Por ejemplo, para calcular una raíz cuadrada debemos usar la función `sqrt` (del inglés *sq*uared *root*):

```
sqrt(49)
```

```
[1] 7
```

Usamos funciones como `sqrt` para pedirle a R que realice algún tipo de operación. Como resultado R nos devuelve cierta respuesta o información (“salida”).

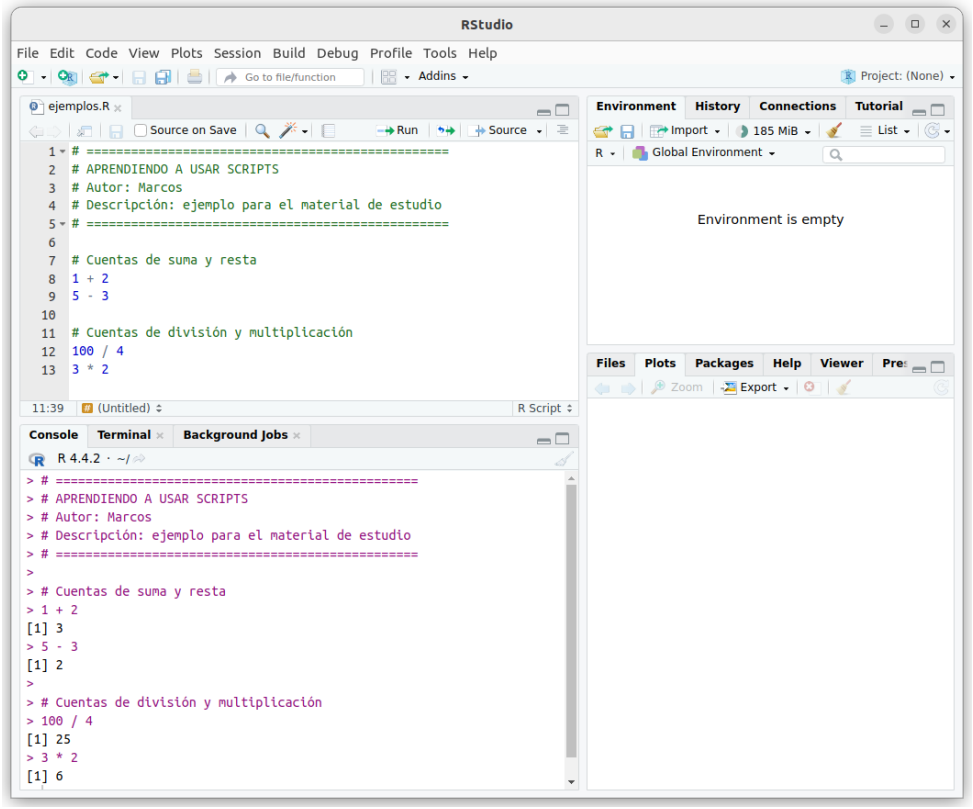


Figura 2.6: Comentarios en el script. Todo el contenido del script fue evaluado, por eso se ve también en la consola. Las líneas comentadas son ignoradas, sólo se muestran resultados para las demás.

Las funciones están representadas por su nombre, seguido por los paréntesis `()`. Dentro de los paréntesis se colocan las porciones de información que queremos compartir con la función, que reciben el nombre de **argumentos** o **parámetros**. Una función puede depender de uno o más argumentos. Si hay más de uno, se separan con comas.

Cuando usamos una función, generalmente se dice que la estamos **llamando**, **invocando** o **corriendo**. Por ejemplo, *corremos* la función `log` con el argumento `100`. Esta función calcula, por defecto, el logaritmo natural:

```
log(100)
```

```
[1] 4.60517
```

En este caso `100` representa un valor numérico que se pasa como argumento a la función para que la misma opere. Algunas funciones predefinidas en R pueden trabajar con más de un argumento, en cuyo caso hay que enumerarlos dentro de los paréntesis, separados con comas. Por ejemplo, si en lugar de calcular el logaritmo natural (cuya base es la constante matemática e), queremos calcular un logaritmo en base 10, podemos hacer lo siguiente:

```
# Logaritmo de 100 en base 10
log(100, 10)
```

```
[1] 2
```

¿Cómo sabemos que la función `log()` se puede usar de esa forma, con uno o dos argumentos, cambiando o no el valor de la base con respecto a la cual toma el logaritmo? Lo aprendemos al leer el manual de ayuda de R.

Toda función de R viene con un instructivo que detalla cómo se usa, qué argumentos incluye y otras aclaraciones. Lo encontramos en la pestaña de Ayuda (*Help*) en el panel de abajo a la derecha en RStudio. Otras formas de abrir la página de ayuda sobre una función es correr en la consola alguna de estas sentencias:

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

Esa página de ayuda tiene bastante información, porque reúne explicaciones sobre muchas funciones relacionadas con logaritmos y exponenciales, pero podemos detenernos en algunas partes más importantes (Figura 2.7).

En la sección *Usage* ("uso") descubrimos que la función `log()` puede usarse con dos argumentos: `x` y `base`. En la sección *Arguments* entendemos que `x` es el número al cual le vamos a sacar el logaritmo y `base` es la base con respecto a la cual se toma el logaritmo. Por eso, al correr `log(100, 10)`, estamos calculando el logaritmo de `x = 100` con `base = 10`.

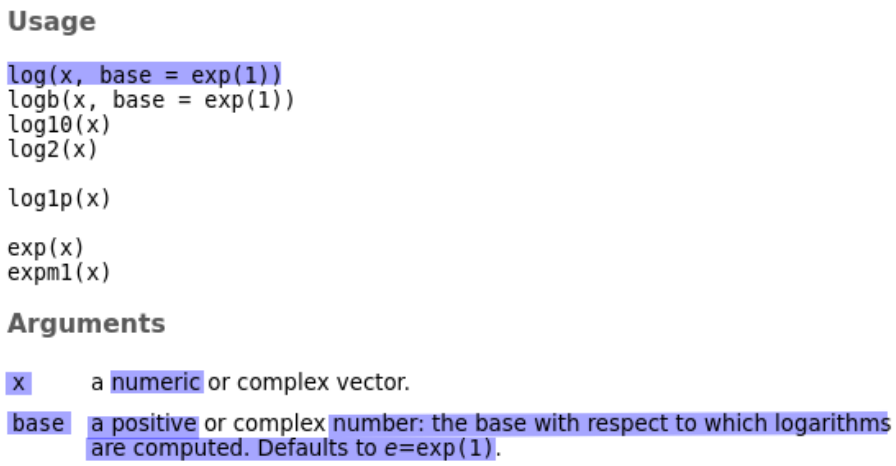


Figura 2.7: Captura de pantalla de la ayuda sobre la función `log()`.

Vemos, además, una diferencia en la forma en que `x` y `base` aparecen en la descripción: `log(x, base = exp(1))`. Cuando un argumento tiene un signo `=` significa que tiene asignado un **valor por defecto** y que no es obligatorio usarlo. Por eso, cuando corremos `log(100)` estamos calculando el logaritmo de `x = 100` con la base elegida por R: `base = exp(1)`, que es la forma que tiene R de nombrar a la constante $e = 2.718282...$ (es el logaritmo natural). Si quiero cambiar la base, debo proveer un valor, por ejemplo, `log(100, 10)`. Por el contrario, el argumento `x` no tiene asignado un valor por default. Eso significa que obligatoriamente tenemos que proveer un valor para el mismo.

R también permite usar una función escribiendo los nombres de los argumentos, lo cual muchas veces es muy esclarecedor, en especial cuando las funciones llevan muchos argumentos:

```
log(x = 100, base = 10)
```

```
[1] 2
```

Si escribimos los nombres de los parámetros explícitamente como en el caso anterior, podemos cambiar su orden, sin alterar el resultado:

```
log(base = 10, x = 100)
```

```
[1] 2
```

Si no escribimos los nombres, el orden importa. R hace corresponder los valores provistos con los argumentos presentados en la ayuda sobre la función, uno por uno, en orden:

```
# Toma el logaritmo de x = 100 con base = 10
log(100, 10)
```

```
[1] 2
```

```
# Toma el logaritmo de x = 10 con base = 100
log(10, 100)
```

```
[1] 0.5
```

EJEMPLO

Presentamos los siguientes ejemplos a modo de resumen:

```
# Tres usos equivalentes de la función para obtener el
# logaritmo de x = 100 con base = 10:
log(100, 10)
```

```
[1] 2
```

```
log(x = 100, base = 10)
```

```
[1] 2
```

```
log(base = 10, x = 100)
```

```
[1] 2
```

```
# Si no indico la base, se toma el valor por defecto (logaritmo
# natural). Ambas expresiones son equivalentes:
log(100)
```

```
[1] 4.60517
```

```
log(x = 100)
```

```
[1] 4.60517
```

```
# Si no indico el argumento obligatorio x, obtengo un error:
log(base = 10)
```

```
Error: argument "x" is missing, with no default
```

Capítulo 3

Objetos y ambiente

RESUMEN

En este capítulo exploramos los fundamentos de R como un lenguaje basado en **objetos**. Aprenderemos qué es un objeto, los diferentes tipos de datos que R maneja y cómo se organizan dentro de **vectores atómicos**. También veremos cómo crear nuevos objetos en R y cómo estos componenen al **ambiente global**. Estos conceptos son esenciales para trabajar de manera eficiente y comprender el comportamiento del lenguaje en futuras aplicaciones.

3.1 Objetos

Anteriormente usamos R para realizar cálculos matemáticos como:

```
5 * 6
```

```
[1] 30
```

Los operandos en ese cálculo y su resultado no están guardados en ningún lugar de nuestra computadora. Podríamos decir que lo que vemos ahí son las huellas de algunos números que existieron brevemente en la memoria de nuestra computadora pero ya desaparecieron. Si queremos usar ese resultado para realizar otro cálculo, tendremos que pedirle a R que calcule `5 * 6` de nuevo.

Claramente, al resolver problemas complejos no podemos trabajar con resultados o valores efímeros. Tenemos que guardarlos en algún lugar para poder reutilizarlos. En R podemos hacer:

```
x <- 5 * 6
```

En este caso que estamos analizando, `x` es un ejemplo de un **objeto** creado por nosotros.

CONCEPTO CLAVE
Los objetos son estructuras capaces de almacenar las distintas piezas de información, (o *datos*), que podemos manipular para resolver una tarea de programación. Dependiendo de las características de un objeto en particular, podemos hacer con él diferentes operaciones. Nuestro programa, a lo largo de su ejecución, va creando o modificando los objetos existentes.

Un nuevo objeto se crea con el **operador de asignación** (`<-`, "operador flecha"). Como lo vamos a usar muchísimas veces, es conveniente recordar su atajo para escribirlo rápidamente con el teclado: `Alt + -` (teclas `Alt` y guión medio). De esta forma, informalmente la línea `x <- 5 * 6` puede ser leída así: "creamos un objeto llamado `x` que contiene al resultado de la operación `5 * 6` (30)".

Una vez que almacenamos un valor en `x`, podemos utilizarlo en nuevos cálculos:

```
x * 100
```

```
[1] 3000
```

```
80 - x
```

```
[1] 50
```

Hay distintos tipos de objetos, algunos con estructuras muy simples (como aquellos que sólo almacenan un único valor, al igual que `x` en el ejemplo) y otros mucho más complejos (como aquellos que sirven para representar conjuntos de datos completos o resultados de algún análisis). Cada lenguaje de programación propone su propio catálogo de tipos de objetos y cada programador puede crear otros tipos nuevos. Durante las primeras unidades sólo emplearemos el tipo de objeto más sencillo que R ofrece.

3.2 Vectores atómicos de R

El **tipo de objeto más simple y básico en R** se llama **vector atómico** (*atomic vector*). De hecho, en el ejemplo, `x` es un vector atómico que hospeda un único valor numérico. En Matemática, la palabra "vector" hace referencia a un conjunto de números ordenados. Los vectores atómicos de R también pueden contener más de un valor, pero por ahora sólo consideraremos situaciones donde cada vector atómico contiene un sólo valor (como `x`). A los objetos que se usan para almacenar un solo valor a veces también se les dice **variables** (porque suele ocurrir que ese valor va cambiando a lo largo del programa).

Un vector atómico puede guardar otras cosas además de números. R define seis tipos básicos de vectores atómicos dependiendo de cómo son los datos que guardan: *doubles*, *integers*, *characters*, *logicals*, *complex* y *raw*[1]. Los últimos dos sirven para guardar números complejos y *bytes* en crudo, respectivamente y muy rara vez son usados para tareas de análisis de datos, por lo cual no los volveremos a nombrar. Vamos a ver a los otros, uno por uno.

3.2.1 *Doubles* (dobles)

Un vector atómico de tipo *double* almacena números reales (positivos o negativos, grandes o chicos, con decimales o sin decimales). Casi siempre que usamos números para analizar datos en R, empleamos este tipo de vector. Con estos objetos podemos realizar operaciones aritméticas comunes, como en los ejemplos que ya vimos antes.

La función `typeof()` se usa para preguntar a R de qué tipo es un objeto:

```
typeof(x)
```

```
[1] "double"
```

3.2.2 *Integers* (enteros)

Un vector atómico de tipo *integer* también almacena números, pero sólo números enteros. Muchos lenguajes de programación tratan a los números enteros de forma diferente al resto de los números porque utilizan estrategias especiales para almacenarlos en la memoria de la compu (ocupan menos espacio y su representación tiene mayor precisión).

Para distinguir a los *integers* de los *doubles*, R los muestra con una `L` al costado. De hecho, podemos crear un vector atómico de tipo *integer* agregando la `L`

```
y <- 1L
typeof(y)
```

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

En aplicaciones de análisis de datos, es raro que los números con los que se trabaja sean sólo enteros, por lo que podemos prestarle poca atención a esto por ahora. Es bueno saber que existen, porque en algunas salidas a veces aparece esa `L` y ahora sabemos de qué se trata.

En algunos contextos, R directamente les dice *numeric* tanto a los *doubles* como a los *integers*.

[1] Los nombramos en inglés puesto que esos son sus nombres formales.

3.2.3 *Characters* (caracteres)

Un vector atómico de tipo *character* almacena texto, es decir, una cadena de caracteres (una palabra o palabras). No es posible hacer operaciones matemáticas con este tipo de vector. Para crear un vector de tipo *character* debemos encerrar entre comillas el texto que será almacenado en él:

```
z <- "Hola, ¿cómo estás?"
z
```

```
[1] "Hola, ¿cómo estás?"
```

```
typeof(z)
```

```
[1] "character"
```

3.2.4 *Logicals* (lógicos)

Un vector atómico de tipo *logical* puede almacenar el valor lógico `TRUE` (verdadero) o el valor lógico `FALSE` (falso). Si bien es poco común que este tipo de valores aparezca de forma natural en conjuntos de datos a analizar, son sumamente importantes. Permiten hacer comparaciones y entender su resultado, así como también evaluar condiciones para decidir o no implementar algunas acciones en el proceso que estamos llevando adelante.

Podemos crear un vector *logical* así:

```
v <- TRUE
v
```

```
[1] TRUE
```

```
typeof(v)
```

```
[1] "logical"
```

Notar que los valores lógicos no se escriben entre comillas, puesto que no son cadenas de texto. Tal vez crear vectores lógicos no es lo más común; es más frecuente que los valores lógicos surjan como resultado de algunas operaciones. Por ejemplo:

```
x > y
```

```
[1] TRUE
```

```
x < y
```

```
[1] FALSE
```

```
is.logical(x)
```

```
[1] FALSE
```

```
is.logical(v)
```

```
[1] TRUE
```

```
is.numeric(x)
```

```
[1] TRUE
```

RESUMEN

¿Para qué necesitamos distintos tipos de vectores atómicos? Para poder establecer qué operaciones se pueden realizar con unos y otros, y que todo tenga sentido. Por ejemplo, podemos hacer sumas aritméticas con vectores numéricos pero no con vectores carácter:

```
# Se puede:
x * 100
```

```
[1] 3000
```

```
x + y
```

```
[1] 31
```

```
# No se puede:
x + z
```

```
Error in x + z: non-numeric argument to binary operator
```

PARA RESOLVER

Seleccionar la respuesta correcta.
¿Cuál de los siguientes es un valor numérico?

- (A) "dos"
- (B) "2"
- (C) 2

¿Cuál de los siguientes es un valor lógico?

- (A) FALSE
- (B) "FALSE"

3.3 Nombres de los objetos

De manera general, al nombre de un objeto se le dice **identificador**, ya que se trata de secuencia de caracteres que sirve para identificarlo a lo largo de un programa. Nombrar los objetos hace posible referirse a los mismos. La elección de los identificadores es una tarea del programador, pero cada lenguaje tiene sus propias reglas. Por ejemplo, en R los nombres de los objetos:

- deben empezar con una letra o un punto (no pueden empezar con un número);
- sólo pueden contener letras, números, guiones bajos y puntos; y
- no se pueden usar las siguientes palabras como nombres, ya que son están reservadas por el lenguaje: `break`, `else`, `FALSE`, `for`, `function`, `if`, `Inf`, `NA`, `NaN`, `next`, `repeat`, `return`, `TRUE`, `while`.

Es aconsejable elegir un nombre que sea representativo de la información que va a guardar el objeto, ya que esto facilita la lectura y la comprensión tanto del algoritmo como del programa. Por ejemplo, si se necesita un objeto para guardar el valor numérico del precio de algún producto, el identificador `p` sería una mala elección, mientras que `precio` sería mejor. Si se necesitan varios identificadores para distinguir los precios de diversos productos, podríamos usar algo como `precio_manzana`, `precio_banana`, etc.

Por otro lado, no es posible usar como identificador a `precio manazana`, puesto que un nombre no puede tener espacios. Otra opción podría ser `preciomanzana` o `precioManzana`, pero en este curso seguimos la convención de usar guiones bajos para facilitar la lectura de nombres compuestos por más de una palabra (esto se conoce como *snake case*).

Es importar que R distingue entre mayúsculas y minúsculas, por lo que `precio` y `Precio` pueden referirse a distintos objetos, con diferentes valores almacenados:

```
precio <- 15
Precio <- 2
precio + Precio
```

```
[1] 17
```

No es aconsejable tener objetos cuyos nombres sólo difieran en mayúsculas o minúsculas como en el ejemplo anterior, puesto que sirve para confusión. En general, preferimos evitar usar mayúsculas.

3.4 Actualizar la información guardada en un objeto

El valor guardado en un objeto puede cambiar en cualquier momento del programa. Además, podemos usar otros objetos para calcular el valor que será guardado. Por ejemplo, imaginemos que un programa contabiliza el stock disponible de un artículo en un comercio. Inicialmente había 43 artículos, pero en el día se vendieron 29 y se compraron otros 12 al proveedor para reponer. Al finalizar la jornada, para saber cuántos hay en stock hay que tomar la cantidad disponible original, restar la cantidad que se vendió y sumar la cantidad que se compró. El código podría lucir así:

```
stock <- 43
ventas <- 29
compras <- 12
stock <- stock - ventas + compras
stock
```

```
[1] 26
```

```
cat("Hay un stock de", stock, "artículos disponibles.")
```

```
Hay un stock de 26 artículos disponibles.
```

INFO IMPORANTE

En el ejemplo anterior usamos por primera vez la función `cat()` para emitir un mensaje, concatenando cadenas de texto encerradas entre comillas y el valor de una variable a la cual hacemos referencia por su nombre (`stock`).

El valor `43` que originalmente estaba guardado en `stock` se perdió para siempre en el preciso momento cuando se ejecutó la línea `stock <- stock - ventas + compras`, que "sobrescribió" su valor. Es importante sobrescribir el valor de un objeto sólo si estamos seguros de que es lo correcto.

También se debe tener en cuenta que podemos reemplazar el objeto representado con un nombre por otro de un tipo diferente, y a R no le va a molestar:

```
# x es un vector atómico de tipo numeric:
x <- 100
x
```

```
[1] 100
```

```
# ahora x pasa a ser un vector atómico de tipo caracter:
x <- "hola"
x
```

```
[1] "hola"
```

Otros lenguajes no admiten este comportamiento. Por el contrario, requieren se "declare" el nombre y el tipo de cada objeto antes de ser usados y, si bien se puede actualizar su valor, éste siempre debe ser del mismo tipo. R es un **lenguaje dinámico** que no tiene este recaudo.

> **COMENTARIO ADICIONAL**
>
> Expresiones como "crear un objeto llamado `x` que contiene el valor `100`" o "sobreescribir, actualizar o reemplazar su valor" nos permiten imaginar lo que sucede y encarar tareas generales de programación en R, pero en realidad son formas simplificadas y poco precisas de describir procesos más complejos relacionados al funcionamiento de R. Estudiantes con experiencia en programación pueden opcionalmente referirse a este material para leer más.

3.5 Ambiente

CONCEPTO CLAVE

En R, un environment (entorno o ambiente) es un espacio donde se almacenan los objetos creados durante una sesión de trabajo. Se trata una estructura clave en la gestión de variables y funciones dentro de un programa.

Durante la evaluación de un programa coexisten muchos *environments*. Vamos a hablar un poco más de esto cuando aprendamos a crear nuevas funciones, pero en general podemos programar en R sin preocuparnos por este tema, que es bastante complejo y avanzado.

Lo fundamental es saber que el conjunto de objetos que vamos creando en nuestro programa forman parte de un ambiente llamado *Global Environment* y es el que vemos en la pestaña *Environment* del panel superior derecho de RStudio (Figura 14.1).

También podemos ver en la consola un listado de todos los nombres de los objetos que existen en el ambiente con la función `ls()`:

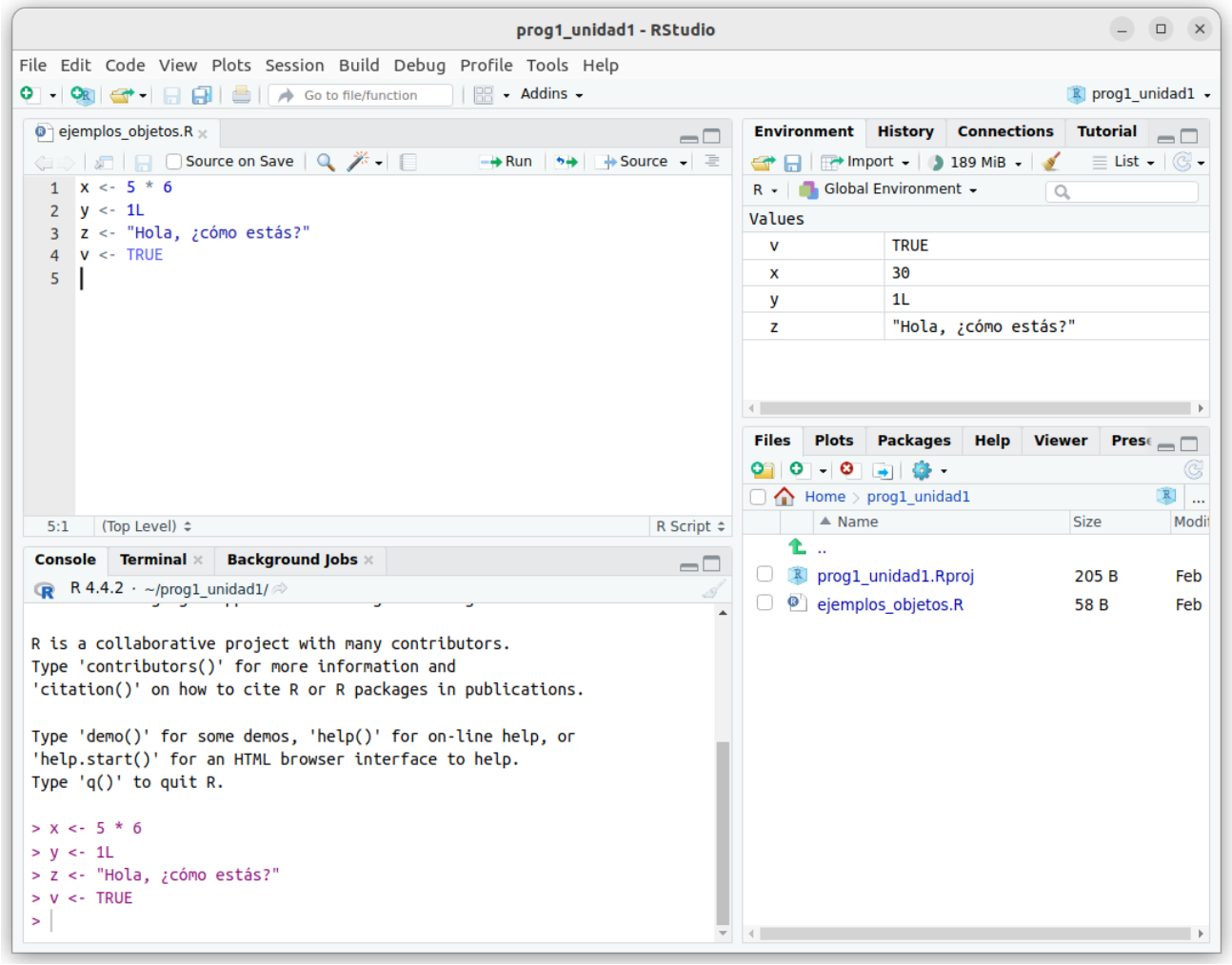


Figura 3.1: Captura de pantalla del *Global Environment*.

```
ls()
```

```
[1] "v"             "x"             "y"             "z"
```

Los objetos que aparecen listados en esta salida o en el panel *Environment* son los que podemos usar para programar, porque están disponibles en nuestro ambiente. Si por error queremos usar un objeto que aún no fue definido en el ambiente global, obtenemos un error así:

```
w * 10
```

```
Error: object 'w' not found
```

Si necesitamos borrar un objeto del ambiente global podemos usar `rm()` indicando como argumento el nombre del objeto a eliminar:

```
rm(x)
```

Si necesitamos borrar todos los objetos del ambiente podemos usar el ícono de la escoba en la pestaña *Environment* o ejecutar:

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

A pesar de que el ambiente nos muestre todos los objetos creados durante el trabajo o análisis, es fundamental que el código que los generó esté siempre escrito y guardado en un script. Con un script siempre es posible recrear el entorno de trabajo, pero si sólo tenemos los objetos en el ambiente, no podemos adivinar con qué código fueron creados. Para asegurar que nuestros scripts sean la referencia principal y es el respaldo de todo lo que se hace en un análisis de datos, se recomienda configurar RStudio para que no guarde automáticamente el ambiente cuando se cierra. Esto se logra yendo al menú `Tools > Global Options` y estableciendo las opciones de configuración tal como se observa en la Figura 3.2. Aunque al principio puede resultar incómodo, ya que cada vez que reinicies RStudio deberás volver a ejecutar el código para generar tus objetos, esta práctica evita problemas a largo plazo. Dejar solo los resultados en el entorno sin registrar el código que los generó puede dificultar la reproducibilidad del análisis en el futuro.

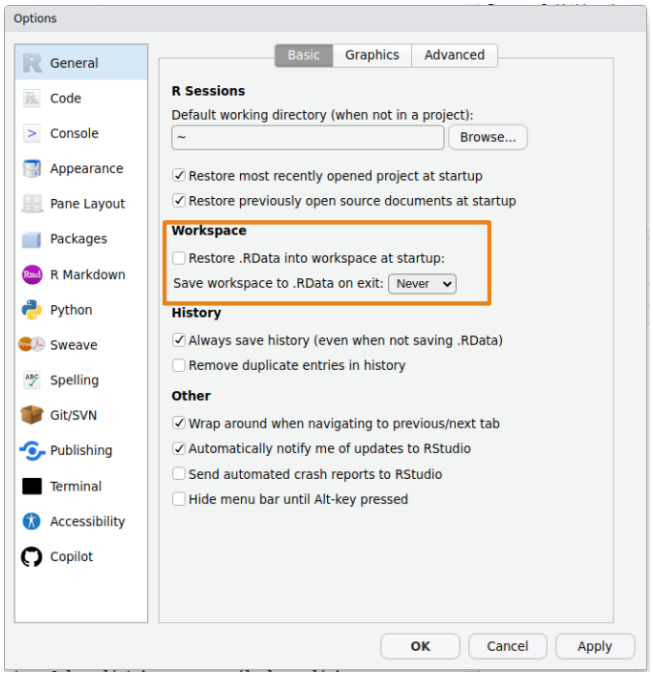


Figura 3.2: Estas opciones garantizan que el ambiente de trabajo esté limpio cada vez que iniciamos RStudio.

Capítulo 4

Operadores

RESUMEN

El desarrollo de un programa involucra la necesidad de efectuar operaciones de distinto tipo entre los valores guardados en los objetos: suma, resta, concatenación de caracteres, comparaciones, etc. Los elementos que describen el tipo de operación a realizar entre dos objetos se denominan **operadores**.

En este capítulo exploraremos los operadores en R, fundamentales para realizar cálculos, comparaciones y evaluaciones lógicas en nuestros programas. Comenzaremos con los operadores aritméticos, que permiten realizar operaciones matemáticas básicas. Luego, abordaremos los operadores relacionales, esenciales para comparar valores, y los operadores lógicos, que nos ayudan a evaluar condiciones. También estudiaremos el orden de precedencia en R, clave para interpretar correctamente las expresiones, y la evaluación en cortocircuito, una optimización importante en decisiones condicionales.

4.1 Operadores aritméticos

Los **operadores aritméticos** permiten realizar operaciones matemáticas con vectores atómicos que almacenen valores numéricos, como *double* o *integer* (Tabla 4.1).

Tabla 4.1: Operadores aritméticos.

Operación	Operador	Ejemplo de uso	Resultado con `x <- 7` e `y <- 3`
Suma	`+`	`x + y`	`10`
Resta	`-`	`x - y`	`4`

Operación	Operador	Ejemplo de uso	Resultado con `x <- 7` e `y <- 3`
Multiplicación	`*`	`x * y`	21
División	`/`	`x / y`	2.33
Potenciación	`^`	`x ^ y`	343
División entera	`%/%`	`x %/% y`	2
División módular (resto de la división)	`%%`	`x %% y`	1

Los operadores aritméticos actúan con un orden de prioridad establecido, también conocido como **orden de evaluación** u **orden de precedencia**, tal como estamos acostumbrados en matemática. Las expresiones entre paréntesis se evalúan primero. Si hay paréntesis anidados se evalúan desde adentro hacia afuera. Dentro de una misma expresión, los operadores se evalúan en este orden:

1. Potenciación (`^`)
2. División entera y módulo (`%/%`, `%%`)
3. Multiplicación y división (`*`, `/`)
4. Suma y resta (`+`, `-`)

Si la expresión presenta operadores con igual nivel de prioridad, se evalúan de izquierda a derecha.

EJEMPLO

Tabla 4.2: Ejemplos de operaciones aritméticas según el orden de precedencia de R.

Operación	Resultado
`4 + 2 * 4`	12
`23 * 2 / 5`	9.2
`3 + 5 * (10 - (2 + 4))`	23
`2.1 * 1.5 + 12.3`	15.45
`2.1 * (1.5 + 12.3)`	28.98
`1 %% 4`	1
`8 * (7 - 6 + 5) %% (1 + 8 / 2) - 1`	7

INFO IMPORANTE

Los operadores aritméticos también se pueden aplicar con valores lógicos. En este caso, `TRUE` es considerado como `1` y `FALSE`, como `0`[1]:

```
TRUE + TRUE
```

```
[1] 2
```

```
TRUE + FALSE
```

```
[1] 1
```

4.2 Operadores relacionales o de comparación

Los **operadores relacionales** sirven para comparar dos valores de cualquier tipo y dan como resultado un valor lógico, `TRUE` o `FALSE`.

Tabla 4.3: Operadores relacionales

Comparación	Operador	Ejemplo de uso	Resultado con `x <- 7` e `y <- 3`
Mayor que	`>`	`x > y`	`TRUE`
Menor que	`<`	`x < y`	`FALSE`
Mayor o igual que	`>=`	`x >= y`	`TRUE`
Menor o igual que	`<=`	`x <= y`	`FALSE`
Igual a	`==`	`x == y`	`FALSE`
Distinto a	`!=`	`x != y`	`TRUE`

[1] Esta conversión de un tipo de valor a otro se llama *coerción*.

EJEMPLO

Ejemplos del uso de operadores relacionales:

```
a <- 3
b <- 4
d <- 2
e <- 10
f <- 15

(a * b) == (d + e)
```

```
[1] TRUE
```

```
(a * b) != (f - b)
```

```
[1] TRUE
```

Es interesante notar que primero se evalúan las operaciones a cada lado de los operadores relacionales y luego se hace la comparación. Es decir, **los operadores aritméticos preceden a los relacionales en el orden de prioridad**. Por eso, en los ejemplos anteriores en realidad no son necesarios los paréntesis y podríamos omitirlos:

```
a * b == d + e
```

```
[1] TRUE
```

```
a * b != f - b
```

```
[1] TRUE
```

PARA RESOLVER

Para pensar… ¿en base a qué criterio se determina si un valor de tipo *character* es mayor qué otro? Mirá este ejemplo:

```
texto1 <- "Hola"
texto2 <- "Chau"
texto3 <- "Adiós"

texto1 > texto2
```

```
[1] TRUE
```

```
texto3 > texto2
```

```
[1] FALSE
```

PARA RESOLVER

¿Qué valor lógico devuelve esta operación?

```
texto1 == "hola"
```

- (A) TRUE
- (B) FALSE

4.3 Operadores lógicos

Mientras que los operadores relacionales comparan cualquier tipo de valores, los **operadores lógicos** sólo toman operandos de tipo *logical* y producen también un resultado lógico.

Tabla 4.4: Operadores lógicos

Operación	Operador	Ejemplo de uso	Resultado con `x <- TRUE` e `y <- FALSE`
Conjunción	`&&`	`x && y`	`FALSE`
Disyunción	`||`	`x || y`	`TRUE`
Negación	`!`	`!x`	`FALSE`

Veamos uno por uno:

- La operación de conjunción devuelve un valor `TRUE` sólo si son verdaderas **ambas** expresiones que vincula. Ejemplo: `(3 > 2) && (3 > 5)` resulta en `TRUE && FALSE` y esto es `FALSE`.
- La operación de disyunción devuelve un valor `TRUE` si **al menos una** de las dos expresiones que vincula es verdadera. Ejemplo: `(3 > 2) || (3 > 5)` resulta en `TRUE || FALSE` y esto es `TRUE`.
- La operación de negación niega un valor lógico, es decir, devuelve el opuesto. Ejemplo: `!(3 > 2)` resulta en `!TRUE` y esto es `FALSE`.

La **tabla de verdad** o **tabla de valores de verdad** se utiliza para mostrar todos los resultados posibles de estas operaciones lógicas:

Tabla 4.5: Tabla de la verdad

x	y	!x	x && y	x || y
TRUE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE

x	y	!x	x && y	x || y
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE

Con estos operadores es posible construir evaluaciones lógicas algo más elaboradas como los siguientes ejemplos:

1. **Determinar si el valor numérica guardado en la variable x está entre 5 y 7**. Tal vez tu intuición te sugiere que la expresión lógica a evaluar en este caso debe ser `5 < valor < 7`, pero esto genera un error en R. Para saber si `valor` está entre 5 y 7, se tiene que evaluar por separado que `valor` sea mayor que 5 y también menor que 7, y ambas condiciones deben ser verdaderas.

```
valor <- 6.4
(valor > 5) && (valor < 7)
```

```
[1] TRUE
```

```
valor <- 2.1
(valor > 5) && (valor < 7)
```

```
[1] FALSE
```

2. **Establecer si el valor de tipo carácter almacenado en la variable `nacionalidad` sea igual a una de dos opciones.**

```
nacionalidad <- "Argentino"
(nacionalidad == "Uruguayo") || (nacionalidad == "Chileno")
```

```
[1] FALSE
```

3. **Verificar que el valor guardado en `nacionalidad` no coincida con "Argentino".**

```
nacionalidad <- "Uruguayo"
!(nacionalidad == "Argentino")
```

```
[1] TRUE
```

4. **Chequear que el valor numérico guardado en x no sea igual a 2 ni a 3.**

   Opción correcta 1: `(x != 2) && (x != 3)`

```
# Da verdadero porque x no es ni 2 ni 3
x <- 10
(x != 2) && (x != 3)
```

```
[1] TRUE
```

```
# Da falso porque x es igual a 3
x <- 3
(x != 2) && (x != 3)
```

```
[1] FALSE
```

Opción correcta 2: `!(x == 2 || x == 3)`

```
# Da verdadero porque x no es ni 2 ni 3
x <- 10
!(x == 2 || x == 3)
```

```
[1] TRUE
```

```
# Da falso porque x es igual a 3
x <- 3
!(x == 2 || x == 3)
```

```
[1] FALSE
```

Opción incorrecta: `(x != 2) || (x != 3)`

```
# Como la primera parte es verdadera (porque x es igual a 3), la
# conjunción es verdadera, cuando quisiéramos que en este caso el
# resultado sea FALSO
x <- 3
(x != 2) || (x != 3)
```

```
[1] TRUE
```

INFO IMPORANTE

Es importante notar que todos los paréntesis usados en el código de R de los ejemplos 1, 2 y 4 son innecesarios, puesto que **los operadores relacionales preceden a los lógicos en el orden de prioridad**. Sin embargo, a veces preferimos usar paréntesis para que la lectura sea más sencilla. Retomando el ejemplo 1, notemos que ambas expresiones son equivalentes:

```
valor <- 2.1
(valor > 5) && (valor < 7)
```

```
[1] FALSE
```

```
valor > 5 && valor < 7
```

```
[1] FALSE
```

PARA RESOLVER

¿Cuál es el resultado de las siguientes operaciones?

```
x <- 2
y <- -2
```

- `x > 0 && y < 0`: TRUE / FALSE
- `x > 0 || y < 0`: TRUE / FALSE
- `!(x > 0 && y < 0)`: TRUE / FALSE

COMENTARIO ADICIONAL

Tanto para la conjunción como para la disyunción, R provee dos operadores diferentes, los ya mencionados `&&` y `||` y otros que no repiten el símbolo, `&` y `|`. La diferencia entre las dos versiones se hace notar cuando operamos con vectores atómicos que almacenen más de un valor, por lo cual por ahora podemos ignorarla. Usaremos la versión de símbolos dobles.

4.4 Orden de precedencia completo en R

Resumiendo la información anterior, a continuación se presenta el orden de precedencia completo de los operadores en R que utilizaremos[2]:

Tabla 4.6: Orden de precedencia de los operadores en R.

Orden	Operaciones	Operadores
1	Potenciación	`^`
2	Signo de un número (ej: -3)	`+`, `-`
3	División entera y resto	`%/%`, `%%`
4	Multiplicación y división	`*`. `/`
5	Suma y resta	`+`. `-`
6	Operadores de comparación	`<`, `>`, `<=`, `>=`, `==`, `!=`
7	Negación	`!`
8	Conjunción	`&&`, `&`
9	Disyunción	`||`, `|`
10	Asignación	`<-`

Dentro de una misma expresión, operadores con igual prioridad se evalúan de izquierda a derecha.

[2]Hay algunos operadores más que no vamos a usar, pero que pueden ser consultados en `?Syntax`.

4.5 Evaluación en cortocircuito

Para evaluar la operación de conjunción `x && y`, en R se comienza por evaluar la expresión del primer operando `x` y si su resultado es `FALSE` ya no se evalúa la expresión `y` del segundo operando. Esto es porque si `x` es `FALSE`, el resultado de `x && y` ya no depende de `y`, será siempre `FALSE`. Por este motivo se dice que el operador `&&` se evalúa en *cortocircuito*. La evaluación en cortocircuito evita realizar operaciones innecesarias[3].

Por ejemplo:

```
f <- 1
g <- 2

# La primera parte da TRUE, se continúa con la segunda, pero da error porque no
# existe un objeto llamado h
(g > f) && (f > h)
```

```
Error: object 'h' not found
```

```
# La primera parte da FALSE, entonces toda la operación será FALSE, no se
# continúa con la segunda parte, con lo cual no se intenta usar el objeto
# inexistente h y no hay error
(g < f) && (f > h)
```

```
[1] FALSE
```

La operación de disyunción también se evalúa en cortocircuito, es decir, si se encuentra que uno de los operandos es `TRUE`, no hace falta evaluar los restantes, puesto que el resultado general será `TRUE`:

```
# Es TRUE porque la primera parte es TRUE, sin evaluar la segunda, que daría
# error
(g > f) || (f > h)
```

```
[1] TRUE
```

```
# Como la primera parte es FALSE, debe evaluar la segunda, no encuentra a h y da
# error
(f > g) || (f > h)
```

```
Error: object 'h' not found
```

[3] El otro operador de conjunción, `&`, no evalúa en cortocircuito, además de poseer otras diferencias.

Capítulo 5

Organización de archivos

RESUMEN

Mantener una buena organización de archivos es fundamental para trabajar de manera eficiente. Una estructura clara y ordenada facilita el acceso a scripts, datos y resultados, evitando confusión y errores al ejecutar código. En este capítulo, exploraremos conceptos clave como las carpetas, archivos y rutas informáticas, la importancia del directorio de trabajo y cómo utilizar los **RStudio Projects** para estructurar mejor nuestros proyectos.

5.1 Carpetas, archivos y rutas informáticas

En las tareas de programación y de análisis de datos se trabaja con muchos archivos de distinto tipo al mismo tiempo (scripts, conjuntos de datos, archivos con resultados, gráficos, etc.). Resulta fundamental mantener un orden para que todo funcione bien y prestar atención dónde guardamos nuestros archivos y elegir esa ubicación de forma cuidadosa.

En una computadora, los archivos se organizan de manera jerárquica dentro de carpetas y subcarpetas, lo que facilita su acceso y gestión. La organización de estos archivos sigue un esquema de árbol, donde las carpetas actúan como contenedores que agrupan archivos relacionados. Este sistema permite a los usuarios almacenar y clasificar la información de forma ordenada y accesible.

La Figura 5.1 muestra como ejemplo un trabajo de análisis de datos de una encuesta a estudiantes. Dentro de la carpeta `Documentos`, se ha creado un directorio llamado `encuesta_estudiantes` para guardar allí absolutamente todos los archivos relacionados con este caso. Incluso se pueden usar subcarpetas para distribuirlos de forma bien clara, dentro de esa carpeta principal. Dentro de esa carpeta, se crearon otras subcarpetas para agrupar los archivos de forma ordenada. Este ejemplo esconde un principio muy importante: es bueno tener la costumbre de crear una carpeta específica

para almacenar todos los archivos vinculados al trabajo que estamos realizando y no dejar archivos tirados por cualquier lugar en la computadora.

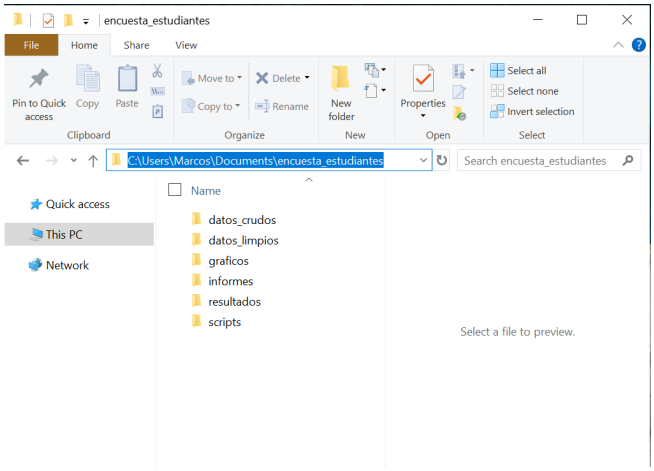


Figura 5.1: Organización de carpetas para analizar los datos de una encuesta.

Cada archivo o carpeta en una computadora tiene una **ruta informática** o **path** que indica su ubicación dentro del sistema de almacenamiento. Esta ruta es como una dirección que permite encontrar un archivo o carpeta específica. La ruta se lee desde una ubicación principal en el disco de la computadora y sigue el camino de las carpetas y subcarpetas hasta llegar al archivo deseado.

Por ejemplo, en relación a la Figura 5.1, la ruta informática que identifica la ubicación de la carpeta `datos_crudos` es `C:\Users\Marcos\Documents\encuesta_estudiantes\datos_crudos`. Si dentro de ella hay una planilla de Excel con los datos de la encuesta, llamada `datos_encuesta.xlsx`, su ruta informática será `C:\Users\Marcos\Documents\encuesta_estudiantes\datos_crudos\datos_encuesta.xlsx`. En sistemas operativos como Windows, las rutas suelen comenzar con una letra de unidad, como "C:", seguida de las carpetas y subcarpetas.

Cuando guardamos un archivo en algún lugar de la computadora estamos definiendo cuál es su ruta informática para que distintos programas de la computadora puedan encontrarlo. Tener esta noción es fundamental a la hora de programar. Por ejemplo, puede ser que nuestro script de R necesite importar los datos del archivo `datos_encuesta.xlsx`. Para esto tal vez necesitemos escribir un comando especial que incluya su ruta informática para que R pueda encontrar el archivo. Si no la escribimos bien, R producirá un error diciendo que el archivo no existe.

INFO IMPORANTE

Para saber con exactitud cuál es la ruta informática de un archivo en Windows, podemos seguir alguna de estas opciones:

1. **Usando el Explorador de Archivos**: ubicar el archivo, hacer clic derecho y seleccionar **"Copiar como ruta"**. Luego podemos pegar la ruta en cualquier lugar (`Ctrl + V`).
2. **Desde la Barra de Direcciones**: abrir la carpeta donde está el archivo, hacer clic en la barra de direcciones, copiar la ruta (`Ctrl + C`) y al pegarla donde se necesite, agregar manualmente el nombre del archivo.
3. **Desde las Propiedades del Archivo**: hacer clic derecho sobre el archivo y seleccionar **"Propiedades"**. En la pestaña **"General"**, copiar el contenido del campo **Ubicación** y, al pegarlo, agregar el nombre del archivo.

Es importante recordar lo siguiente: si bien Windows usa barras diagonales `/` para mostrar rutas informáticas, R sólo las reconoce si las escribimos con barras invertidas `\`.

Tener un buen sistema para ordenar archivos también es una gran recomendación al cursar una carrera universitaria, ya que necesitarás manejar múltiples archivos de distintas asignaturas a la vez. Podrías crear un esquema de trabajo como el de la Figura 5.2.

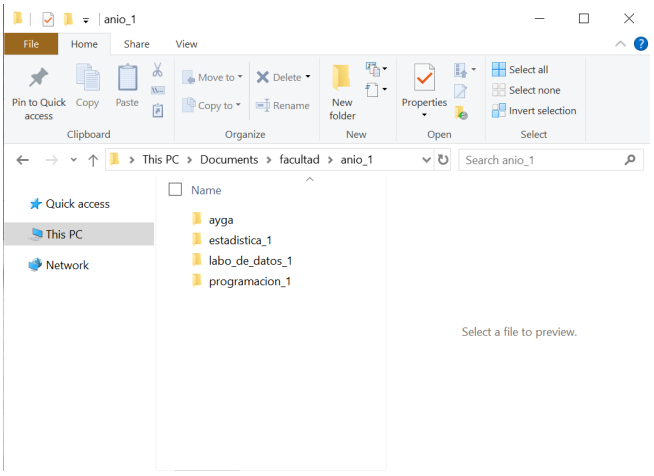


Figura 5.2: Organización de carpetas para las asignaturas de la facultad.

Una buena idea adicional es que alojes la carpeta `facultad` y todo su contenido dentro de algún sistema de sincronización de archivos como **Google Drive** o **OneDrive**, para que tengas un respaldo en la nube y puedas acceder al contenido desde el celular u otras computadoras, manteniendo siempre todo sincronizado.

Para nuestra asignatura, te sugerimos que dentro de la carpeta `facultad/anio_1/programacion_1`, definas una carpeta distinta para cada unidad o trabajo práctico (por ejemplo, `unidad_1`, `unidad_2`, etc.). Esto ayudará a mantener el orden. No hace falta que crees a mano cada una de estas carpetas. Las vamos a crear con RStudio, lo cual resultará en beneficios adicionales.

5.2 Directorio de trabajo

Como hemos visto en la sección anterior, nuestra computadora organiza todos sus archivos bajo un sistema jerárquico de carpetas y subcarpetas. Entre todas ellas, en cada sesión de trabajo R posa su mirada en una de forma particular, la cual recibe el nombre **directorio de trabajo** (o *working directory*, *wd*). Aquí es donde R busca los archivos que le pedís que cargue y donde colocará los archivos que le pedís que guarde. El directorio de trabajo por default suele ser la carpeta *Documentos* o alguna equivalente según el sistema operativo y es la que se muestra en el panel `Files` cuando iniciamos RStudio. Otras formas de saber cuál es la carpeta de la computadora que actúa como *working directory* en una sesión de trabajo son:

- Leer la ruta informática escrita en la parte superior del panel de la consola, al lado del logo y la versión de R (Figura 5.3).

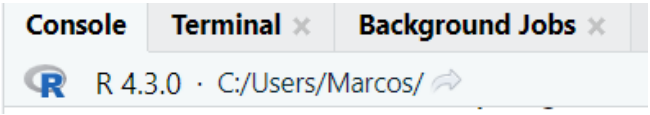


Figura 5.3: Indicación del *working directory* arriba en la consola

- Ejecutar en la consola la instrucción `getwd()`. Por ejemplo:

```
getwd()
```

```
"C:/Users/Marcos"
```

Ese resultado implica que en este momento R puede ver y acceder de manera directa a todos los archivos que hay en esa carpeta, sin necesidad de escribir la ruta informática completa para hacer referencia a cada uno de ellos. Si creamos un nuevo script y apretamos el ícono de guardar, R nos ofrece guardarlo en el *working directory*. Si queremos importar un conjunto de datos, lo buscará ahí, a menos que le indiquemos otra cosa. Si queremos guardar cualquier resultado de nuestro análisis, también lo guardará en esa carpeta.

Por esa razón, es muy útil que, al trabajar con R, el *working directory* no sea la carpeta que aparece por defecto, sino aquella carpeta en la que tengamos guardados todos los archivos referidos al problema que estamos resolviendo. Para poder cambiar y elegir como *working directory* a cualquier carpeta de nuestra computadora que nos interese podemos usar una instrucción que se llama `setwd()`. Sin embargo, a continuación aprenderemos algo mejor.

5.3 Organización del trabajo con RStudio Projects

A partir de las dos secciones anteriores, llegamos a las siguientes conclusiones:

- Cuando encaramos un trabajo de programación o de análisis de datos, tenemos que destinar una carpeta específica de nuestra computadora para guardar ahí todos los archivos relacionados.

- Si estamos usando R, tenemos que setear como *working directory* a dicha carpeta, para que podamos acceder con facilidad a los archivos y guardar allí los archivos nuevos que generemos, sin tener que depender de rutas informáticas largas.

RStudio nos permite trabajar de esa forma a través los **RStudio Projects**.

CONCEPTO CLAVE
Al crear un RStudio Project (o sencillamente, proyecto), se genera una nueva carpeta en la computadora con el objetivo de colocar allí todos los archivos relacionados con un trabajo específico, incluyendo scripts, datos, gráficos y otros documentos.

Para crear un nuevo proyecto en RStudio, seguimos estos pasos (Figura 5.4):

1. Ir al menú **File** (Archivo) y seleccionar **New Project...** (Nuevo Proyecto).

2. Elegir **New Directory** (Nueva carpeta, Figura 5.4).
3. Ingresar un nombre para el proyecto y elegir la ubicación donde se guardará en nuestra computadora.

4. Hacer clic en **Create Project** (Crear Proyecto).

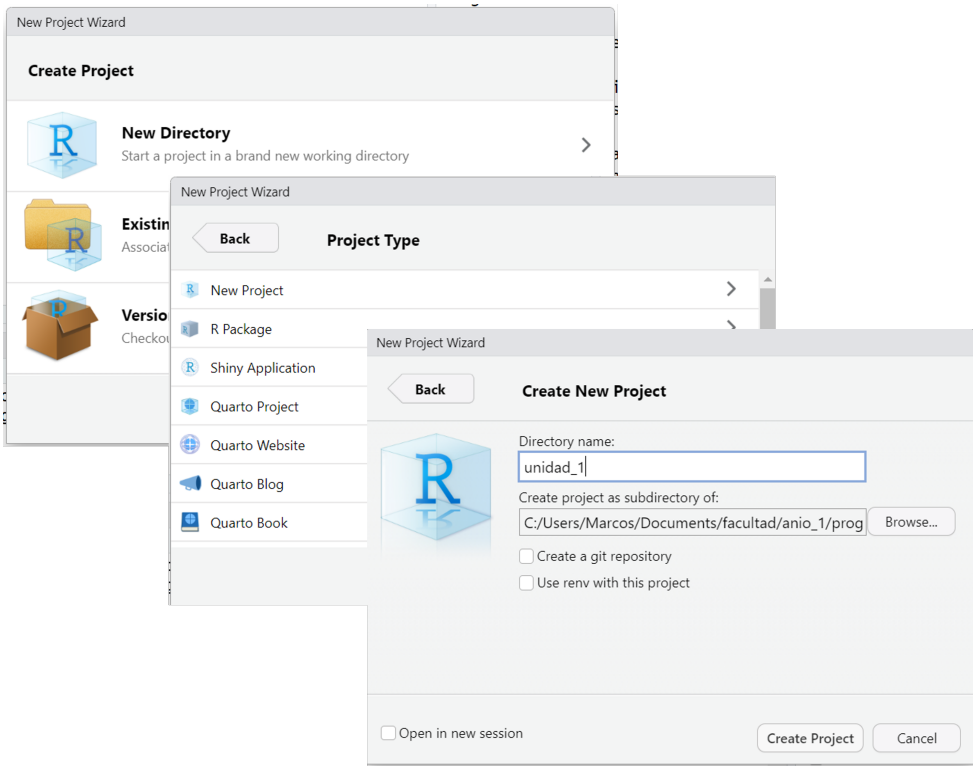


Figura 5.4: Creación de un nuevo proyecto de RStudio.

Como resultado, se crea una carpeta con el nombre elegido y se reinicia RStudio, con una nueva sesión de trabajo, en la cual dicha carpeta es configurada como *working directory*. Podemos leer y guardar los archivos relacionados con nuestro trabajo de forma directa en esa carpeta, sin tener que usar rutas informáticas largas.

INFO IMPORANTE

Usar proyectos de RStudio tiene varias ventajas:

- Cada proyecto tiene su propia carpeta, lo que evita mezclar archivos de diferentes trabajos (o asignaturas).
- Se automatiza la configuración del directorio de trabajo.
- Se pueden gestionar múltiples trabajos simultáneamente. Podemos cambiar de un proyecto a otro sin perder el contexto de cada análisis (scripts abiertos, objetos en el ambiente, etc.). Abrimos cada proyecto en una instancia de RStudio diferente, pudiendo tener varias abiertas a la vez.
- Todo el código y los archivos quedan organizados en un solo lugar, lo que facilita compartir o retomar un proyecto en el futuro.

Cuando dejamos de trabajar en el proyecto, ya sea porque terminamos o debemos continuar más tarde, cerramos RStudio y listo. Para continuar trabajando en otro momento, tenemos que volver a abrir el proyecto, de alguna de estas formas:

- A diferencia de cualquier otra carpeta, un *RStudio Project* incluye un archivo de extensión `.Rproj`. Si lo abrimos, se abre el proyecto una nueva sesión de trabajo en RStudio, que ya tiene seteado como *working directory* a dicha carpeta.
- En caso de que ya tengamos RStudio abierto:
  - Podemos ir a `File > Open project` y buscar en la computadora la carpeta del proyecto, para seleccionar el archivo `.Rproj`.
  - O bien, podemos seleccionar un proyecto de la lista de proyectos abiertos recientemente (`File > Recent projectos` o en la esquina superior derecha de RStudio).

Al abrir un *RStudio Project*, los scripts que estaban abiertos en el editor la última vez que trabajamos en este proyecto, vuelven a aparecer tal como los dejamos, sin importar que estemos trabajando de manera intermitente en distintos proyectos.

Al trabajar con un proyecto, si necesitamos referirnos en el código a la ubicación de un archivo sólo debemos usar rutas informáticas relativas, no absolutas. Una **ruta absoluta** es la ruta completa, como `C:\Usuarios\Marcos\Documentos\encuesta_estudiantes\datos_crudos\datos_encuesta.xlsx`. En cambio, una **ruta relativa** es relativa al *working directory*, es decir, al directorio de inicio del proyecto. Suponiendo que nuestro proyecto se corresponde con la carpeta `encuesta_estudiantes`, la ruta relativa que tenemos que usar para ubicar a la planilla de Excel sólo está compuesta por `datos_crudos\datos_encuesta.xlsx`.

Las rutas relativas son importantes: funcionan siempre bien, a pesar de que el código lo use otra persona en otra computadora. La primera parte de la ruta absoluta, que contiene incluso hasta el nombre de usuario, cambia de computadora a computadora, pero la parte que se usa en la ruta relativa es siempre la misma. Todas las computadoras que tengan una copia del proyecto podrán usar el mismo código sin problemas para encontrar a todos los archivos involucrados.

COMENTARIO ADICIONAL

Tal vez hayas notados que los nombres de carpeta de los ejemplos lucen algo raros, no tienen espacios, mayúsculas o tildes. Es recomendable usar nombres de carpetas y archivos que no contengan espacios, tildes, la letra "ñ" ni caracteres especiales, y preferiblemente en minúsculas. Esto se debe a que algunos sistemas operativos y programas pueden interpretar estos caracteres de manera diferente, lo que puede generar errores al acceder a los archivos o al ejecutar código. Además, cuando trabajamos con rutas de archivos en R, los espacios pueden causar problemas si no se manejan correctamente. Una práctica común es utilizar guiones bajos (`_`) o guiones medios (`-`) en lugar de espacios, por ejemplo, `encuesta_estudiantes` en lugar de `Encuesta de Estudiantes`. Esto es una sugerencia que ayuda a evitar errores y asegura que los archivos sean accesibles sin complicaciones en cualquier sistema.

PARA RESOLVER

Vamos a establecer el siguiente modo de trabajo para este curso. En tu computadora, tal como mencionamos antes, creá una carpeta para guardar todo lo relacionado a tu carrera (puede estar en `Documentos` o dentro de `Google Drive` u otro sistema de sincronización y respaldo). Luego, creá una subcarpeta para los elementos relacionados al primer año de cursado. A continuación, creá una carpeta para cada materia que estás cursando, incluída **Programación 1**. El resultado tiene que ser similar al que se ve en la Figura 5.2.

Cada vez que comencemos una nueva etapa en **Programación 1**, vamos a crear un nuevo proyecto de RStudio para gestionar todos los archivos referidos a ella. Particularmente, ahora es momento de crear un proyecto llamado `unidad_1`, dentro de la carpeta `programacion_1`. Guardarás ahí todos los scripts que desarrollaremos a lo largo de la unidad, con ejemplos y soluciones de ejercicios. Cada vez que quieras continuar trabajando en los materiales de esta unidad, tendrás que abrir el proyecto a partir del archivo `unidad_1.Rproj`. Tomá de ejemplo la Figura 5.4.

Capítulo 6

Errores de programación, guías de estilo y paquetes de R

RESUMEN

En este capítulo hablaremos sobre la posibilidad de cometer errores al programar, desde errores de sintaxis que impiden la ejecución del código hasta errores lógicos que pueden pasar desapercibidos y afectar los resultados. También destacaremos la importancia de seguir una guía de estilo para escribir código claro y fácil de depurar. Además, conoceremos el sistema de paquetes en R, una de sus características más poderosas, que permite ampliar sus funcionalidades mediante herramientas creadas por la comunidad. Aprenderemos cómo instalar y cargar paquetes para aprovechar al máximo este ecosistema.

6.1 Errores de programación

Apenas iniciemos nuestro camino en el mundo de la programación nos daremos cuenta que tendremos siempre ciertos compañeros de viaje: los *errores*. Muchas veces nos pasará que queremos ejecutar nuestro código y el mismo no anda o no produce el resultado esperado. No importa cuán cuidadosos seamos, ni cuánta experiencia tengamos, los errores están siempre presentes. Con el tiempo y práctica, vamos a poder identificarlos y corregirlos con mayor facilidad, pero probablemente nunca dejemos de cometerlos.

A los errores en programación se los suele llamar *bugs* (insecto o bicho en inglés) y el proceso de la corrección de los mismos se conoce como *debugging* (depuración)[1]. Se dice que esta terminología proviene de 1947, cuando una computadora en la Universidad de Harvard (la *Mark II*) dejó de funcionar y finalmente se descubrió que la causa del problema era la presencia de una polilla en

[1] Algunos usan el término bug para referirse exclusivamente a errores lógicos.

un relé electromagnético de la máquina. Sin embargo, otros historiadores sostienen que el término ya se usaba desde antes.

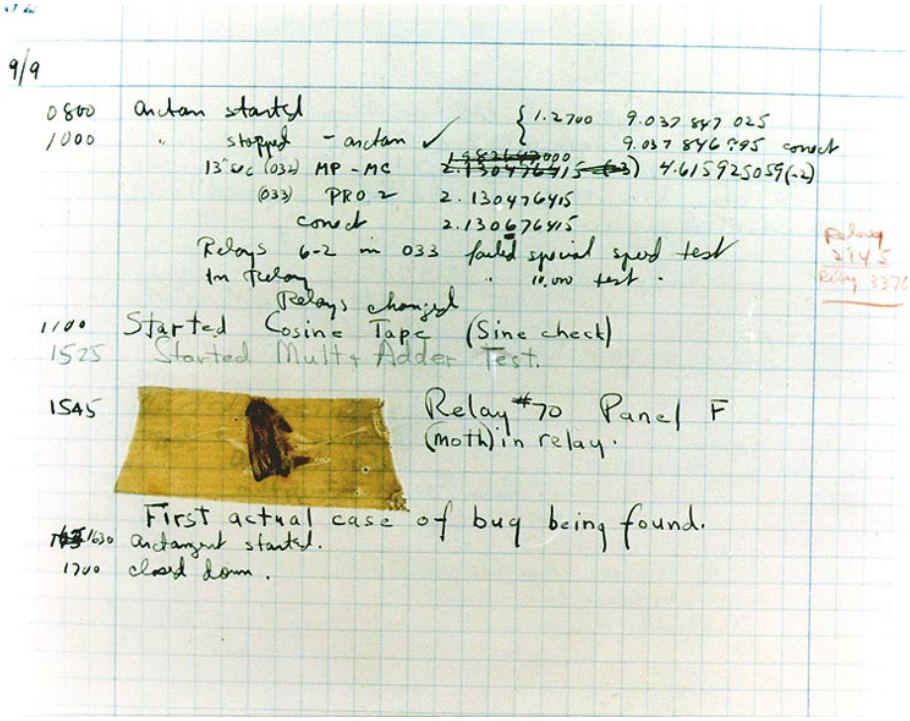


Figura 6.1: La polilla (bug) encontrada por la científica de la computación Grace Hooper en la Mark II fue pegada con cinta en un reporte sobre el malfuncionamiento de la máquina.

A continuación se presenta una clasificación de los errores que se pueden cometer en programación:

- *Errores de sintaxis.* Tal como el lenguaje humano, los lenguajes de programación tienen su propio vocabulario y su propia sintaxis, que es el conjunto de reglas gramaticales que establecen cómo se pueden combinar las distintas partes. Estas reglas sintácticas determinan que ciertas instrucciones están correctamente construidas, mientras que otras no. Cuando ejecutamos un programa, se chequea si el mismo es sintácticamente correcto. Si hemos violado alguna regla, por ejemplo, nos faltó una coma o nos sobra un paréntesis, mostrará un mensaje de error y debemos editar nuestro programa para corregirlo. En estos casos, hay que interpretar el mensaje de error, revisar el código y corregir el error.
- *Errores lógicos.* Se presentan cuando el programa no tiene errores de sintaxis pero arroja resultados incorrectos o ningún resultado. El software no muestra mensajes de error, debido a que, por supuesto, no sabe cuál es el resultado deseado, sino que sólo se limita a hacer lo que hemos programado. En estos casos hay que revisar el programa para encontrar algún error en su lógica. Este tipo de errores suelen ser los más problemáticos. Algunas ideas para enfrentarlos incluyen volver a pensar paso por paso lo que se debería hacer para solucionar el problema y compararlo con lo que se ha programado, agregar pasos para mostrar resultados intermedios o emplear herramientas especializadas de *debugging* (llamadas *debuggers*) para explorar el código paso a paso hasta identificar el error.

- *Errores en la ejecución* (*runtime errors*). Se presentan cuando el programa está bien escrito, sin errores lógicos ni sintácticos, pero igualmente se comporta de alguna forma incorrecta. Se dan a pesar de que el programa ande bien en el entorno de desarrollo del programador, pero no cuando algún usuario lo utiliza en algún contexto particular. Puede ser que se intente abrir un archivo que no existe, que el proceso supere la memoria disponible, que tomen lugar operaciones aritméticas no definidas como la división por cero, etc.

Los errores en la programación son tan comunes, que un científico de la computación muy reconocido, Edsger Dijkstra, dijo una vez: "si la depuración es el proceso de eliminar errores, entonces la programación es el proceso de generarlos". Ante la presencia de uno, no hay más que respirar profundo y con paciencia revisar hasta encontrarlo y solucionarlo.

6.2 Guías de estilo

Es sumamente importante mantener la prolijidad en la escritura del código para facilitar su lectura, especialmente cuando estamos trabajando con problemas largos. Siempre hay que tener en cuenta de que cuando escribimos un programa, tenemos dos públicos potenciales: integrantes de nuestro equipo de trabajo que tienen leer el código y hacer sus propios aportes y nosotros mismos en el futuro, cuando retomemos código hecho en el pasado y necesitemos interpretar qué es lo que hicimos hacer.

Es por eso que se establecen conjuntos de reglas para controlar y unificar la forma de escribir programas, que se conocen como **guía de estilo**. Estas reglas cubren aspectos como, por ejemplo, la forma de escribir comentarios en el código, la utilización de espacios o renglones en blanco, el formato de los nombres para los elementos que creamos nosotros mismos (como las funciones) y para los archivos que generamos, etc. Una *guía de estilo* no indica la única forma de escribir código, ni siquiera la forma correcta de hacerlo, sino que establece una convención para que todos trabajen de la misma forma, basándose en costumbres que sí se ha visto que pueden tener más ventajas que otras.

Para programar en R existe una guía de estilo muy popular llamada The tidyverse style guide, creada por los desarrolladores de RStudio (Posit). En este curso vamos a adherir a sus recomendaciones. Si bien es una lectura muy interesante, particularmente si tenés intenciones de profundizar tus conocimientos sobre programación en R, no es necesario que lean dicha guía completa. Por ahora es suficiente con que imiten con atención el estilo que usamos en los ejemplos provistos en esta guía y sigan algunas recomendaciones generales como las siguientes:

- Como dijimos antes, escribimos los nombres de nuevos objetos con *snake case* y en minúscula:
  - Ok: `mi_objeto`.
  - Evitar: `MIOBJETO`, `miObjeto`.
- En R los espacios en blanco son ignorados, colocarlos o no no produce errores en el código. Sin embargo, se recomienda hacer uso de los mismos para facilitar la lectura. Se sugiere colocar espacios alrededor del operador de asignación `<-` y de los operadores matemáticos (excepto la potencia). No colocamos espacio después de abrir o antes de cerrar un paréntesis.
  - Ok: `z <- (a + b)^2 / d`
  - Evitar: `z<-( a + b ) ^ 2/d`

- Usamos un espacio después de poner una coma y entre el signo igual que se usa para definir argumentos en una función:
  - Ok: `log(100, base = 10)`
  - Evitar: `log(100 ,base=10)`
- Los nombres de archivo deben describir su contenido y evitar espacios, símbolos y caracteres especiales, y preferentemente estar escritos en minúsculas.
  - Ok: `analisis_exploratorio.R`
  - Evitar: `Análisis exploratorio.R`, `codigo.r`

Recordemos siempre que seguir un buen estilo para programar es como hacer uso de una correcta puntuación cuando escribimos, podemos entendernos sin ella, peroesmuchomasdificilleersinolarespetamosno?[2]

6.3 Paquetes de R

6.3.1 Diseño del sistema R

R está diseñado como un sistema modular, dividido en dos partes principales:

- **R Base**: Se instala automáticamente cuando descargamos R desde CRAN (Comprehensive R Archive Network). Incluye las funciones fundamentales del lenguaje, como operadores matemáticos, herramientas para manipular datos y funciones estadísticas básicas.
- **Paquetes adicionales**: Son extensiones opcionales que amplían las funcionalidades de R. Cada paquete es un conjunto de funciones y datos diseñados para tareas específicas, como visualización de datos, modelado estadístico o manipulación avanzada de estructuras de datos.

6.3.2 Instalación de paquetes

Para utilizar un paquete en R, primero debemos instalarlo. La mayoría de los paquetes están disponibles en CRAN y, teniendo conexión a internet, se pueden descargar directamente con el siguiente comando:

```
install.packages("nombre-del-paquete")
```

Esto descargará e instalará el paquete en la computadora, permitiéndolo usarlo en futuras sesiones.

[2] Frase tomada de acá.

6.3.3 Carga de paquetes

Después de instalar un paquete, es necesario **cargarlo en la sesión de R** para poder utilizar sus funciones. Es como abrirlo para sacar las funciones que están guardadas dentro. Para hacerlo, usamos la función `library()`:

```
library("nombre-del-paquete")
```

Si intentás usar una función de un paquete sin haberlo cargado previamente, R mostrará un error indicando que el objeto no fue encontrado.

Es importante recordar que **la instalación de un paquete solo se hace una vez**, pero **debemos cargarlo en cada nueva sesión de R** en la que queramos utilizarlo. Cuando cerramos RStudio, los paquetes se descargan de la memoria, por lo que es necesario volver a llamarlos con `library()` la próxima vez que los necesitemos.

Este sistema modular permite que R sea altamente flexible, permitiendo a los usuarios instalar solo las herramientas que realmente necesitan para sus análisis.

6.3.4 Creación de paquetes en R

Los paquetes en R no solo son desarrollados por expertos en estadística y computación, sino también por cualquier persona que quiera compartir herramientas útiles con la comunidad. Investigadores, analistas de datos y programadores contribuyen a la expansión del ecosistema de R creando paquetes que facilitan tareas específicas.

Si bien algunos paquetes pueden ser muy sofisticados, la creación de uno no es algo exclusivo de grandes desarrolladores. De hecho, al finalizar esta materia, contarás con los conocimientos necesarios para construir tu propio paquete desde cero. Esto te permitirá organizar y compartir funciones de manera eficiente, facilitando su reutilización en distintos proyectos.

6.3.5 Paquetes utilizados en esta materia

Como estaremos aprendiendo nociones de programación, no necesitaremos usar paquetes adicionales en el contexto de esta asignatura, pero sí usarás muchos en otras materias.

Capítulo 7

Lectura opcional

RESUMEN

Para comprender mejor el entorno en el que programamos, es útil conocer algunos aspectos históricos y conceptuales de la computación. En este capítulo opcional, exploraremos una breve reseña sobre el desarrollo de las computadoras, desde sus orígenes hasta la actualidad. También definiremos los conceptos de **hardware** y **software**, elementos fundamentales que permiten el funcionamiento de cualquier sistema informático. Aunque no es imprescindible para aprender R, esta información proporciona un contexto valioso sobre la evolución de la tecnología y su impacto en la programación.

7.1 Breve reseña histórica sobre la programación

La historia de la programación está vinculada directamente con la de la computación. Esta palabra proviene del latín *computatio*, que deriva del verbo *computare*, cuyo significado es "enumerar cantidades". Computación, en este sentido, designa la acción y efecto de computar, realizar una cuenta, un cálculo matemático. De allí que antiguamente computación fuese un término usado para referirse a los cálculos realizados por una persona con un instrumento expresamente utilizado para tal fin (como el ábaco, por ejemplo) o sin él. En este sentido, la computación ha estado presente desde tiempos ancestrales, sin embargo debemos remontarnos al siglo XVII para encontrar los primeros dispositivos diseñados para automatizar cómputos matemáticos.

En 1617 el matemático escocés John Napier (el mismo que definió los logaritmos) inventó un sistema conocido como *los huesos de Napier* o *huesos neperianos* que facilitaba la tarea de multiplicar, dividir y tomar raíces cuadradas, usando unas barras de hueso o marfil que tenían dígitos grabados. Esta fue la base para otras ideas más avanzadas, entre ellas la que dio origen a la primera calculadora mecánica, inventada por el alemán Wilhelm Schickard en 1623, capaz de realizar cómputos aritméticos sencillos

funcionando a base de ruedas y engranajes. Se componía de dos mecanismos diferenciados, un ábaco de Napier de forma cilíndrica en la parte superior y un mecanismo en la inferior para realizar sumas parciales de los resultados obtenidos con el aparato de la parte superior. Fue llamado *reloj calculador*. A partir de aquí se fueron desarrollando otros modelos, todos ellos teniendo en común el hecho de ser puramente mecánicos, sin motores ni otras fuentes de energía. El operador ingresaba números ubicando ruedas de metal en posiciones particulares y al girarlas otras partes de la máquina se movían y mostraban el resultado. Algunos ejemplos son las calculadoras del inglés William Oughtred en 1624, de Blaise Pascal en 1645 (llamada *pascalina*), la de Samuel Morland en 1666 y las de Leibniz, en 1673 y 1694.

Figura 7.1: De izquierda a derecha: los huesos de Napier (Museo Arqueológico Nacional de España), el reloj calculador de Schickard (Museo de la Ciencia de la Universidad Pública de Navarra) y una pascalina del año 1952

El siglo XVIII trajo consigo algunos otros diseños, pero un gran salto se dio a comienzos del siglo XIX de mano de un tejedor y comerciante francés, Joseph Jacquard. En 1801 creó un telar que tenía un sistema de tarjetas perforadas para controlar las puntadas del tejido, de forma que fuera posible *programar* una gran diversidad de tramas y figuras. Sin saberlo, Jacquard sentó una idea fundamental para la creación de las computadoras.

Figura 7.2: Un telar de Jacquard y sus tarjetas perforadas en el Museo de la ciencia y la industria en Mánchester.

En 1822 el matemático británico Charles Babbage publicó un diseño para la construcción de una

máquina diferencial, que podía calcular valores de funciones polinómicas mediante el método de las diferencias. Este complejo sistema de ruedas y engranajes era el primero que podía trabajar automáticamente utilizando resultados de operaciones previas. Si bien el diseño era viable, por motivos técnicos y económicos no lo pudo concretar (sólo construyó un modelo de menor escala). Sin embargo, Babbage no se dio por vencido y en 1837 presentó el diseño de una *máquina analítica*, un aparato capaz de ejecutar cualquier tipo de cálculo matemático y que, por lo tanto, se podría utilizar con cualquier propósito. Tal como el telar de Jacquard, la operación de esta máquina sería controlada por un patrón de perforaciones hechas sobre una tarjetas que la misma podría leer. Al cambiar el patrón de las perforaciones, se podría cambiar el comportamiento de la máquina para que resuelva diferentes tipos de cálculos. Para la salida de resultados, la máquina sería capaz de perforar tarjetas. Además, funcionaría con un motor a vapor y su tamaño hubiese sido de 30 metros de largo por 10 de ancho. Si bien Babbage tampoco llegó a concretar en vida este diseño que dejó plasmado en más de 300 dibujos y 2200 páginas por motivos políticos, se lo considera como la primera conceptualización de lo que hoy conocemos como computadora, por lo cual Babbage es conocido como *el padre de la computación*.

En 1843 Lady Ada Lovelace, una matemática y escritora británica, publicó una serie de notas sobre la máquina analítica de Babbage, en las que resaltaba sus potenciales aplicaciones prácticas, incluyendo la descripción detallada de tarjetas perforadas para que sea capaz de calcular los números de Bernoulli. Al haber señalado los pasos para que la máquina pueda cumplir con estas y otras tareas, Ada es considerada actualmente como la primera programadora del mundo, a pesar de que en la época no fue tomada en serio por la comunidad científica, principalmente por su condición de mujer.

Figura 7.3: Charles Babbage, Ada Lovelace y el algoritmo que publicó Ada para calcular los números de Bernoulli con la máquina analítica de Charles.

La utilidad de las tarjetas perforadas quedó confirmada en 1890, cuando Herman Hollerith las utilizó para automatizar la tabulación de datos en el censo de Estados Unidos. Las perforaciones en determinados lugares representaban información como el sexo o la edad de las personas, logrando que se pudieran lograr clasificaciones y conteos de forma muy veloz. Así, se tardaron sólo 3 años en procesar la información del censo, cinco años menos que en el anterior de 1880. Con el fin de comercializar esta tecnología, Hollerith fundó una compañía que terminaría siendo la famosa International Business Machine (IBM), empresa líder en informática hasta el día de hoy.

Sin embargo, la visión de Babbage de una computadora programable no se hizo realidad hasta los años 1940, cuando el advenimiento de la electrónica hizo posible superar a los dispositivos mecánicos existentes. John Atanasoff y Clifford Berry (Iowa State College, Estados Unidos) terminaron en 1942 en Iowa State College (Estados Unidos) una computadora electrónica capaz de resolver sistemas de ecuaciones lineales simultáneas, llamada *ABC* (por "Atanasoff Berry Computer"). La misma contaba con 300 tubos de vacío, unas bombillas de vidrio con ciertos componentes que podían recibir y modificar una señal eléctrica mediante el control del movimiento de los electrones produciendo una respuesta, que habían sido presentados por primera vez en 1906 por el estadounidense Lee De Forest. La *ABC* dio comienzo a la conocida como la *primera generación de computadoras* basadas en el empleo de tubos de vacío.

La primera computadora electrónica de propósito general fue la *ENIAC*, *Electronic Numerical Integrator and Computer*, completada por Presper Eckert y John Mauchly en la Universidad de Pensilvania. Podía realizar cinco mil operaciones aritmética por segundo y tenía más de 18000 tubos de vacío, ocupando una sala de 9x15 metros en un sótano de la universidad donde se montó un sistema de aire acondicionado especial.

Ni la *ABC* ni la *ENIAC* eran reprogramables: la *ABC* servía el propósito específico de resolver sistemas de ecuaciones y la *ENIAC* era controlada conectando ciertos cables en un panel, lo que hacía muy compleja su programación. El siguiente gran avance se produjo en 1945, cuando el matemático húngaro-estadounidense John von Neumann (Universidad de Princeton) propuso que los programas, es decir, las instrucciones para que la máquina opere, y también los datos necesarios, podrían ser representados y guardados en una memoria electrónica interna. Así nació el concepto de *programa almacenado* (o *stored-program*), en contraposición con el uso de tableros de conexiones y mecanismos similares de los modelos vigentes. Los creadores de la ENIAC, bajo la consultoría de von Neumann, implementaron esto en el diseño de su sucesora, la *EDVAC*, terminada en 1949. También ya había experimentado con esta idea el alemán Konrad Zuse, quien entre 1937 y 1941 desarrolló la *Z3*, por lo cual es considerada por algunos como la primera máquina completamente automática y programable. En lugar de usar tubos de vacíos, empleaba un conjunto de 2600 relés, unos dispositivos electromagnéticos inventados en 1835 y empleados, por ejemplo, en telegrafía. El modelo original de la Z3 fue destruido en Berlín por un bombardeo durante la segunda guerra mundial.

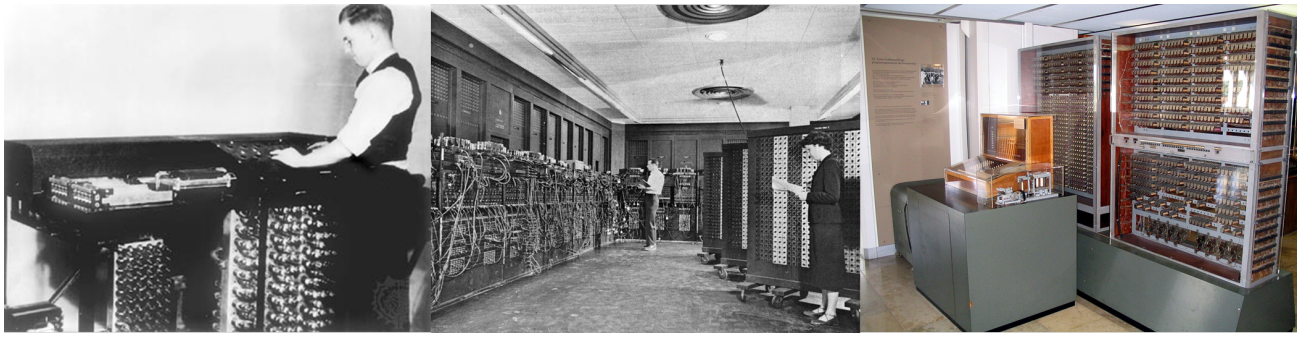


Figura 7.4: De izquierda a derecha: las computadoras ABC, ENIAC y Z3

Este nuevo paradigma cambió la historia de la computación, como también lo hizo la invención del *transistor* en 1947 en los Laboratorios Bell. Un *transistor* es un dispositivo electrónico semiconductor

que entrega una señal de salida en respuesta a una señal de entrada, mucho más pequeño que los tubos de vacío y que consumen menos energía eléctrica. Así, una computadora podía tener cientos de miles de transistores, no obstante ocupando mucho espacio.

Desde entonces, la computación ha evolucionado muy rápidamente, con la introducción de nuevos sistemas y conceptos, que llegan a los complejos y poderosos diseños electrónicos que caracterizan la vida actual. En un intento de caracterizar y resumir esta impactante evolución, algunos historiadores dividen al desarrollo de las computadoras modernas en "generaciones".

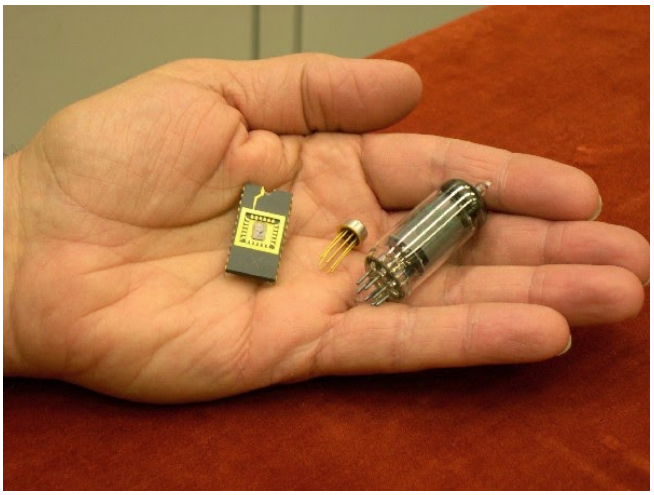


Figura 7.5: De derecha a izquierda: un tubo de vacío, un transistor y un chip.

7.2 Niveles de abstracción de los lenguajes de programación

Si bien hay distintos lenguajes de programación, una computadora en definitiva es un aparato que sólo sabe *hablar* en *binario*, es decir, sólo interpreta señales eléctricas con dos estados posibles, los cuales son representados por los dígitos binarios 0 y 1. Toda instrucción que recibe la computadora se construye mediante una adecuada y larga combinación de ceros y unos[1]. Este sistema de código con ceros y unos que la computadora interpreta como instrucciones o conjuntos de datos se llama *lenguaje de máquina* (o código de máquina).

Programar en lenguaje de máquina es muy complejo y lento, es fácil cometer errores pero es difícil arreglarlos. Por eso a principios de la década de 1950 se inventaron los *lenguajes ensambladores*, que usan palabras para representar simbólicamente las operaciones que debe realizar la computadora. Cada una de estas palabras reemplaza un código de máquina binario, siendo un poco más fácil programar. Imaginemos que deseamos crear un programa que permita sumar dos números elegidos por una persona. La computadora puede hacer esto si se lo comunicamos mediante un mensaje compuesto por una larga cadena de ceros y unos (lenguaje de máquina) que a simple vista no podríamos entender. Sin embargo, escrito en lenguaje ensamblador, el programa se vería así (por ejemplo):

[1] Como podés leer en Sección 7.1, las primeras computadoras no se manejaban con lenguajes de programación, sino que para introducir información e instrucciones en las primeras computadoras se usaban tarjetas perforadas, en las cuales los orificios representaban un "0" y las posiciones que no los tenían se entendían como un "1", de modo que la máquina podía operar empleando el sistema binario.

```
mov ah,01h  ;Se guarda 01h en el registro ah
int 21h     ;Se llama a la interrupción 21h
sub al,30h  ;Se resta 30h para obtener el número ingresado
mov var1,al ;Se guarda el número ingresado en var1
mov ah,01h  ;Se guarda 01h en el registro ah
int 21h     ;Se llama a la interrupción 21h
sub al,30h  ;Se resta 30h para obtener el número ingresado
add al,var1 ;Se suma var1 y al y el resultado se guarda en el registro al
mov dl,al   ;Se guarda el contenido del registro al en el registro dl
add dl,30h  ;Se suma 30h al registro dl para obtener la suma en ASCII
mov ah,02h  ;Se guarda 02h en el registro ah
int 21h     ;Se llama a la interrupción 21h
```

Figura 7.6: Programa en lenguaje ensamblador para leer dos números, sumarlos y mostrar el resultado. Al final de cada línea hay una descripción de la operación realizada.

El programa que se encarga de traducir esto al código de máquina se llama *ensamblador*. A pesar de que no haya ceros y unos como en el lenguaje de máquina, probablemente el código anterior tampoco sea fácil de entender. Aparecen instrucciones que tal vez podemos interpretar, como *add* por sumar o *sub* por substraer, pero está lleno de cálculos hexadecimales, referencias a posiciones en la memoria de la computadora y movimientos de valores que no lo hacen muy amigable. Por eso, a pesar de que la existencia de los lenguajes ensambladores simplificó mucho la comunicación con la computadora, se hizo necesario desarrollar lenguajes que sean aún más sencillos de usar.

Por ejemplo, con el lenguaje que vamos a aprender, R, el problema de la imagen anterior, que consiste en pedirle a una persona que ingrese dos números para luego sumarlos se resumen en las siguientes líneas de código:

```
n1 <- scan()
n2 <- scan()
print(n1 + n2)
```

En las dos primeras líneas con la instrucción `scan()` (que quiere decir “escanear”, “leer”) se le pide a la persona que indique dos números y en la tercera línea se muestra el resultado de la suma, con la instrucción `print()` (“imprimir”, “mostrar”). Mucho más corto y entendible.

Esta simplificación es posible porque nos permitimos *ignorar* ciertos aspectos del proceso que realiza la computadora. Todas esas acciones que se ven ejemplificadas en la imagen con el código ensamblador se llevan a cabo de todas formas, pero no lo vemos. Nosotros sólo tenemos que aprender esas últimas tres líneas de código, de forma que nos podemos concentrar en el problema a resolver (ingresar dos números, sumarlos y mostrar el resultado) y no en las complejas operaciones internas que tiene que hacer el microprocesador.

En programación, la idea de simplificar un proceso complejo ignorando algunas de sus partes para comprender mejor lo que hay que realizar y así resolver un problema se conoce como **abstracción**[2]. Esto quiere decir que los lenguajes de programación pueden tener distintos niveles de abstracción:

- *Lenguajes de bajo nivel de abstracción*: permiten controlar directamente el *hardware* de la computadora, son específicos para cada tipo de máquina, y son más rígidos y complicados de entender para nosotros. El lenguaje ensamblador entra en esta categoría.
- *Lenguajes de alto nivel de abstracción*: diseñados para que sea fácil para los humanos expresar los algoritmos sin necesidad de entender en detalle cómo hace exactamente el hardware para ejecutarlos. El lenguaje que utilizaremos en este taller, R, es de alto nivel. Son independientes del tipo de máquina.
- *Lenguajes de nivel medio de abstracción*: son lenguajes con características mixtas entre ambos grupos anteriores.

Figura 7.7: Distintos lenguajes de programación y sus logos.

Si bien podemos programar usando un lenguaje de alto nivel para que nos resulte más sencillo, *alguien* o *algo* debe traducirlo a lenguaje de máquina para que la computadora, que sólo entiende de ceros y unos, pueda realizar las tareas. Esto también es necesario incluso si programáramos en lenguaje ensamblador. Para estos procesos de traducción se crearon los *compiladores* e *intérpretes*.

Un *compilador* es un programa que toma el código escrito en un lenguaje de alto nivel y lo traduce a código de máquina, guardándolo en un archivo que la computadora ejecutará posteriormente (archivo ejecutable). Para ilustrar el rol del compilador, imaginemos que alguien que sólo habla español le quiere mandar una carta escrita en español a alguien que vive en Alemania y sólo habla alemán. Cuando esta persona la reciba, no la va a entender. Se necesita de un intermediario que tome la carta en español, la traduzca y la escriba en alemán y luego se la mande al destinatario, quien ahora sí la podrá entender. Ese es el rol de un *compilador* en la computadora. Ahora bien, el resultado de la traducción, que es la carta escrita en alemán, sólo sirve para gente que hable alemán. Si se quiere enviar el mismo mensaje a personas que hablen otros idiomas, necesitaremos hacer la traducción que corresponda. De

[2]La abstracción no es una idea exclusiva de la programación. Se encuentra, también, por ejemplo, en el *arte abstracto*.

la misma forma, el código generado por un compilador es específico para cada máquina, depende de su arquitectura.

Además de los compiladores, para realizar este pasaje también existen los *intérpretes*. Un intérprete es un programa que traduce el código escrito en lenguaje de alto nivel a código de máquina, pero lo va haciendo a medida que se necesita, es decir, su resultado reside en la memoria temporal de la computadora y no se genera ningún archivo ejecutable. Siguiendo con el ejemplo anterior, es similar a viajar a Alemania con un intérprete que nos vaya traduciendo en vivo y en directo cada vez que le queramos decir algo a alguien de ese país. En su implementación por defecto, el lenguaje R es interpretado, no compilado.

Concluyendo, gracias al concepto de la *abstracción* podemos escribir programas en un lenguaje que nos resulte fácil entender, y gracias al trabajo de los *compiladores* e *intérpretes* la computadora podrá llevar adelante las tareas necesarias.

7.3 Software y hardware

Como podemos ver, en la historia de la computación hubo dos aspectos que fueron evolucionando: las máquinas y los programas que las dirigen. Hacemos referencia a estos elementos como *hardware* y *software* respectivamente, y es la conjunción de ambos la que le da vida a la computación y hace posible la programación.

Figura 7.8: Representación de la diferencia entre hardware y software.

El *hardware* es el conjunto de piezas físicas y tangibles de la computadora. Existen diversas formas de clasificar a los elementos que componene al hardware, según distintos criterios:

Critero	Clasificación
Según su utilidad	Dispositivos de Dispositivos de Dispositivos de Dispositivos de
Según su ubicación	Dispositivos int Dispositivos ext
Según su importancia	Hardware princ

Critero	Clasifi
	Hardw

Tabla 7.2: Clasificación del hardware.

Critero	Clasificación	Descripción	Ejemplos
Según su utilidad	Dispositivos de procesamiento	Son los que reciben las instrucciones mediante señales eléctricas y usan cálculos y lógica para interpretarlas y emitir otras señales eléctricas como resultado.	microprocesador, tarjeta gráfica, tarjeta de sonido, etc.
	Dispositivos de almacenamiento	Son capaces de guardar información para que esté disponible para el sistema.	disco duro, pen drive, DVD, etc.
	Dispositivos de entrada	Captan instrucciones por parte de los usuarios y las transforman en señales eléctricas interpretables por la máquina.	teclado, mouse, touch pad, etc.
	Dispositivos de salida	Transforman los resultados de los dispositivos de procesamiento para presentarlos de una forma fácilmente interpretable para el usuario.	monitor, impresora, etc
Según su ubicación	Dispositivos internos	Generalmente se incluye dentro de la carcasa de la computadora.	microprocesador, disco rígido, ventiladores, módem, tarjeta gráfica, fuente de alimentación, puertos, etc.
	Dispositivos externos o periféricos	No se incluye dentro de la carcasa de la computadora y está al alcance del usuario	monitor, teclado, mouse, joystick, micrófono, impresora, escáner, pen drive, lectores de código de barras, etc.
Según su importancia	Hardware principal	Dispositvos esenciales para el funcionamiento de la computadora	microprocesador, disco rígido, memoria RAM, fuente de alimentación, monitor, etc.
	Hardware complementario	Aquellos elementos no indispensables (claramente, dependiendo del contexto, alguna pieza del hardware que en alguna situación podría considerarse complementaria, en otras resulta principal).	

Por otro lado tenemos al *software*, que es el conjunto de todos los programas (es decir, todas las instrucciones que recibe la computadora) que permiten que el hardware funcione y que se pueda concretar la ejecución de las tareas. No tiene una existencia física, sino que es intangible. El software se puede clasificar de la siguiente forma:

Clasificación	Descripción
Software de sistema o software base	Son los programas informáticos que están escritos en lenguaje de b
Software de aplicación o utilitario	Son los programas o aplicaciones que usamos habitualmente para r
Software de programación o de desarrollo	Son los programas y entornos que nos permiten desarrollar nuestra

Tabla 7.4: Clasificación del software.

Clasificación	Descripción	Ejemplos
Software de sistema o software base	Son los programas informáticos que están escritos en lenguaje de bajo nivel como el de máquina o ensamblador y cuyas instrucciones controlan de forma directa el hardware	BIOS o UEFIs (sistemas que se encargan de operaciones básicas como el arranque del sistema, la configuración del hardware, etc), sistemas operativos (Linux, Windows, iOS, Android), controladores o *drivers*, etc.
Software de aplicación o utilitario	Son los programas o aplicaciones que usamos habitualmente para realizar alguna tarea específica.	procesadores de texto como Word, reproductor de música, Whatsapp, Guaraní, navegadores web, juegos, etc.
Software de programación o de desarrollo	Son los programas y entornos que nos permiten desarrollar nuestras propias herramientas de software o nuevos programas. Aquí se incluyen los lenguajes de programación	C++, Java, Python, R, etc.

Capítulo 8

Práctica de la Unidad 1

8.1 Ejercicio 1

Usar la consola de R en RStudio para realizar las siguientes operaciones:

- Sumar 25 y 17.
- Multiplicar 6 por 8.
- Calcular la raíz cuadrada de 144.

Luego, crear un nuevo script en RStudio y escribir en él las operaciones anteriores. En una línea anterior, agregar el comentario "Ejercicio 1". Guardar el script con el nombre `resolucion_practica_1.R` en cualquier lugar de tu computadora. Continuar completando el script la solución de los restantes ejercicios de esta práctica, usando comentarios para identificarlos.

8.2 Ejercicio 2

a. Encontrar la página de ayuda en R para la función `round()`.

b. Descubrir qué argumentos acepta la función y qué hace cada uno de ellos.

c. ¿Cuál de esos argumentos es de uso obligatorio y cuál, opcional?
d. Usar la función para redondear el valor `3.14159` con 0, 1 o 2 decimales.
e. Escribir tres formas distintas de usar la función para redondear el valor `3.14159` con dos decimales/

8.3 Ejercicio 3

En R, crear los siguientes objetos y observar sus valores:

```
x <- 10
y <- "Hola"
z <- 5
```

a. ¿Qué tipo de objeto es cada uno?
b. Modificar el objeto `z` para almacene el resultado de multiplicar a `x` por 6.
c. ¿Qué sucede si intentás sumar `x` e `y`? Explicar por qué ocurre esto.

8.4 Ejercicio 4

Descargar el archivo practica1_ambiente.RData. Su extensión `.RData` indica que contiene objetos de R. Cuando este archivo se lee con R, se cargan en el *environment* los objetos que tiene almacenados. Vamos a leerlo para identificar qué tipo de objetos aparecen en nuestro ambiente, con alguna de estas opciones:

- Hacer doble clic en el archivo `practica1_ambiente.RData`.
- En la pestaña *Environment* de RStudio, hacer clic en el ícono de abrir y seleccionar el archivo `practica1_ambiente.RData`.
- Ejecutar la siguiente instrucción:

```
load("ruta/completa/hasta/el/archivo/practica1_ambiente.RData")
```

Una vez que el archivo haya sido cargado en R, responder: ¿cuántos objetos fueron incorporadas al ambiente al haber cargado el archivo? ¿Cuáles son sus nombres o identificadores? ¿Qué tipo de dato contiene cada uno? ¿Qué valor contiene cada uno?

8.5 Ejercicio 5

Teniendo en cuenta los objetos cargados en el ambiente en el ejercicio anterior:

a. Evaluar si al menos una de las variables `var2` o `var5` contiene un número negativo, escribiendo la correspondiente operación en R.
b. Evaluar si ambas variables `var2` y `var5` contienen un número negativo, escribiendo la correspondiente operación en R.
c. Evaluar si al dividir `var2` por la suma entre `var5` y 100, el resto es menor que 10, escribiendo la correspondiente operación en R.
d. Evalar si los valores guardados en `var3` y `var6` son iguales o no. Inspeccione dichos valores y comente lo observado.

8.6 Ejercicio 6

Sin utilizar R, calcular el valor resultante de las siguientes operaciones, para cada uno de los casos presentados en las columnas. Luego, verificar en R.

Operación	`edad <- 21, altura <- 1.90`	`edad <- 17 , altura <- 1.90`	`edad <- 21, altura <- 1.50`
`(edad > 18) && (altura < 1.70)`			
`(edad > 18) || (altura < 1.70)`			
`!(edad > 18)`			

8.7 Ejercicio 7

Aplicando las reglas de prioridad en los operadores aritméticos, anticipar el resultado de la siguiente expresión. Luego, verificar en R.

```
1 + 2 + (3 + 4) * ((5 * 6 %% 7 * 8) - 9) - 10
```

8.8 Ejercicio 8

¿Para qué valores de `x` la siguiente expresión resulta verdadera?: `(x != 4) || (x != 17)`.

8.9 Ejercicio 9

Se desea determinar si un determinado año es bisiesto. Si bien pensamos que los años bisiestos ocurren cada 4 años, los procesos astronómicos que dan origen a este concepto son algo más complejos. Dado que en realidad la Tierra tarda 365.25 días en completar su órbita anual alrededor del sol, agregar un día extra una vez cada 4 años ayuda a mantener el calendario en sincronización con el sol, pero aún queda un pequeño desfasaje. Por lo tanto, la regla completa dice que los años bisiestos ocurren cada 4 años, excepto los terminados en 00, los cuales son bisiestos sólo si son divisibles por 400. Es decir, los años como 1600, 1700, 1800 son bisiestos si son divisibles por 400. Por ejemplo, el año 1900 no fue bisiesto a pesar de ser divisible por 4, pero el año 2000 si lo fue por ser divisible por 400. Entonces, para que un año dado sea bisiesto, se debe cumplir una de las siguientes condiciones:

- El año es divisible por 4 pero no divisible por 100, o
- El año es divisible por 400.

Siendo `año` el nombre del objeto cuyo valor es el año que se desea evaluar, expresar la operación lógica que devuelve el valor `TRUE` sólo si `año` es un año bisiesto.

8.10 Ejercicio 10

Para cada bloque de código, determinar el valor final de cada variable antes de ejecutarlo en R. Luego, comprobar la respuesta ejecutando el código.

a. ¿Cuáles son los valores finales de `a` y `b`?

```
a <- 10
b <- a * 2
a <- a + 5
b <- b - a
```

b. ¿Cuáles son los valores finales de `m` y `n`?

```
m <- 5
n <- 2 * m
m <- m + 3
n <- n + m
m <- n - 4
```

c. ¿Cuál es el valor final de `y`?

```
x <- 6
y <- 2
x <- x / y + x * y
y <- x^2 %% 10
y <- y * 2
y
```

d. ¿Cuál es el valor final de `resultado`?

```
a <- 5
b <- 2
c <- 3

resultado <- a^b - (c * b) + (a %% c)
```

e. ¿Cuáles son los valores finales de `x`, `y` y `z`?

```
x <- 8
y <- 3
z <- 2

x <- x %% y + z^y
y <- (x + y) %/% z
z <- z + x - y
```

8.11 Ejercicio 11

Escribir una expresión en R para calcular el área total y el volumen de un prisma rectangular, considerando que las longitudes se guardan en las variables numéricas `a`, `b` y `h`, como se muestra en la imagen.

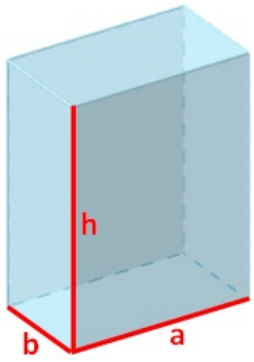


Figura 8.1: Prisma rectangular.

8.12 Ejercicio 12

Responder las siguientes preguntas:

a. ¿Cuál es la ruta informática o *path* del script `resolucion_practica_1.R` que creaste en el ejercicio 1?
b. ¿Cuál es el directorio de trabajo o *working directory* en tu sesión de trabajo actual?

8.13 Ejercicio 13

a. Crear en tu computadora una carpeta para guardar todo lo relacionado a tus estudios (puede estar en `Documentos` o dentro de `Google Drive` u otro sistema de sincronización y respaldo).
b. Crear una subcarpeta para los elementos relacionados al primer año de cursado.
c. En ella, crear una carpeta para cada materia que estás cursando, incluyendo una para **Programación 1**, con el nombre `programacion_1`.
d. Crear un proyecto de RStudio llamado `unidad_1`, dentro de la carpeta `programacion_1`.
e. Guardar en la carpeta del proyecto todos los archivos que hayas utilizado. Por ejemplo, el script `resolucion_practica_1.R` y el archivo descargado `practica1_ambiente.RData`.
f. A partir de ahora, cada vez que comiences una nueva unidad en esta asignatura, crear un nuevo proyecto como hicimos para la Unidad 1.

Capítulo 9

Actividad de autoevaluación 1

INFO IMPORANTE
Esta autoevaluación DEBE ser completada sin usar R para poder razonar las preguntas.

9.1 Pregunta 1

Dada la operación:

```
(x > y) || !(x / 2 > 10)
```

indique su resultado (`TRUE` o `FALSE`) para cada una de las siguientes asignaciones de valor para las variables `x` e `y`:

```
a) x <- 15
   y <- 30
```

Seleccione una:

- (A) TRUE
- (B) FALSE

Explicación

Como `x = 15` y `y = 30`, entonces `(15 > 30)` es `FALSE`.

`15/2 = 7.5` y `(7.5 > 10)` también es `FALSE`.

Sin embargo, `!(FALSE) = TRUE`.

Finalmente, `FALSE || TRUE = TRUE` ya que el operador `||` resulta `TRUE` si al menos una de las dos condiciones es `TRUE`.

```
b) x <- 22
   y <- 30
```

Seleccione una:

- (A) TRUE
- (B) FALSE

Explicación

Como `x = 22` y `y = 30`, entonces `(22 > 30)` es `FALSE`

`22/2 = 11`, y `(11 > 10)` es `TRUE`

Sin embargo, `!(TRUE) = FALSE`

Finalmente, `FALSE || FALSE = FALSE`

```
c) x <- 32
   y <- 30
```

Seleccione una:

- (A) TRUE
- (B) FALSE

Explicación

Como `x = 32` y `y = 30` entonces `(32 > 30)` es `TRUE`

`32/2 = 16` y `(16 > 10)` es `TRUE`

Sin embargo, `!(TRUE) = FALSE`

Finalmente, `TRUE || FALSE = TRUE`

9.2 Pregunta 2

¿Cuál es el valor almacenado en `var1` luego de ejecutar las siguientes acciones?

```
var1 <- 100
var2 <- var1 %% 30
var3 <- var1 + var2
var3 <- var3 - 3 * var2
var1 <- var3 + 20
```

La respuesta es (ingrese un numero): _____

Explicación

1) `var1` recibe el valor `100`

2) `var2` almacena el resto de dividir `var1` entre 30:

- Como `30 × 3 = 90` y `100 - 90 = 10`, entonces:
- `var2 = 10`

3) `var3` guarda la suma de `var1` y `var2`:

- `100 + 10 = 110`

4) Modificación de `var3`:

- Se actualiza `var3` restando 3 veces `var2`:
- `110 - (3 × 10) = 80`

5) Actualización final de `var1`:

- `var1` toma el valor de `var3 + 20`
- `80 + 20 = 100`

Tras todas las operaciones, `var1` vuelve a su valor inicial: `100`.

9.3 Pregunta 3

Los operadores aritméticos tienen mayor orden de precedencia que los operadores relacionales (es decir, se evalúan antes). ¿Es esta afirmación verdadera o falsa?

- (A) TRUE
- (B) FALSE

Explicación

Jerarquía de operadores:

- Aritméticos (`^`, `%%`, `/`, `*`, `+`, `-`): Mayor precedencia.
- Relacionales (`>`, `<`, `==`, `!=`, `>=`, `<=`): Menor precedencia.

Ejemplo:

`3 + 5 * 2 > 1`

Paso 1 (aritméticos): `5 * 2 = 10` → `3 + 10 = 13` Paso 2 (relacional): `13 > 10` → `TRUE`

Regla clave: `Paréntesis () > Aritméticos > Relacionales > Lógicos`

9.4 Pregunta 4

Una inmobiliaria necesita guardar el valor de una operación de venta de un inmueble efectuada durante el corriente año en una variable para realizar operaciones posteriores. Entre los siguientes posibles nombre de variable indique cual sería el más adecuado:

- (A) `ventainmuebleanio2025`
- (B) `VentaOperación`
- (C) `Valor-Venta-Inmueble-2025`
- (D) `venta inmueble 2025`
- (E) `venta_inmueble_2025`
- (F) `2025_venta_inmueble`
- (G) `operación_Año2025`
- (H) `valorVentaInmueble2025`

Explicación

`VentaOperación`: No es la mejor opción porque incluye un acento ("ó") en el nombre de la variable, lo que puede generar problemas de codificación en algunos entornos. Además, se recomienda usar guiones bajos para separar palabras en minúsculas y mejorar la legibilidad.

`2025_venta_inmueble`: En general no es buena práctica que los nombres de variables comiencen con un número en ningún lenguaje de programación. R y la mayoría de lenguajes no lo permiten, y aunque algunos sí lo hacen, dificulta la lectura y puede causar errores.

`Valor-Venta-Inmueble-2025`: No se recomienda usar guiones medios en nombres de variables porque este carácter está reservado para la operación de resta. En muchos lenguajes, incluyendo R, provocaría errores de sintaxis. Siempre es mejor usar guiones bajos.

`venta inmueble 2025`: Los espacios entre palabras no son válidos en nombres de variables. El sistema interpretaría "venta", "inmueble" y "2025" como tres variables distintas. Para unir palabras, se usan guiones bajos.

`valorVentaInmueble2025`: Este tipo de escritura se conoce como *camelCase*, y aunque muchos lo utilizan, en R la convención más clara y extendida es el uso de *snake_case* (con guiones bajos). Funciona, pero no sigue el estilo típico de R.

`operación_Año2025`: Este nombre contiene un acento ("ó") y la letra "ñ", lo que puede causar conflictos en algunos entornos. Para evitar errores, es mejor evitar caracteres especiales como estos. Ademas utiliza mayúsculas, lo que puede complicar innecesariamente la escritura del nombre de la variable.

`ventainmuebleanio2025`: Aunque técnicamente válido, este nombre es largo y poco intuitivo porque carece de separadores entre palabras. Usar guiones bajos mejora significativamente la legibilidad.

9.5 Pregunta 5

Indicar si la siguiente afirmación es verdadera o falsa:

"Al crear un nuevo Rproject en RStudio se inicia una nueva sesión, se setea el directorio de trabajo y se cargan todas las librerias que necesito para trabajar automáticamente"

- (A) TRUE
- (B) FALSE

Explicación

Un nuevo **RProject** en RStudio inicia una nueva sesión de R (limpia el entorno de trabajo) y setea el directorio de trabajo al directorio raíz del proyecto, pero no carga librerías instaladas. Las librerías deben cargarse manualmente en cada sesión o mediante scripts usando el comando `library()`.

Unidad 2. Estructuras de control

RESUMEN

Un programa está compuesto por una sucesión ordenada de instrucciones que se ejecutan una detrás de otra, de forma secuencial. Sin embargo, con frecuencia es necesario recurrir a comandos especiales que alteran o controlan el orden en el que se ejecutan las instrucciones. Llamamos **estructuras de control del flujo del código** al conjunto de reglas que permiten controlar el orden o la cantidad de veces con la que se ejecutan las instrucciones de un algoritmo o programa. Las mismas pueden clasificarse en **condicionales** e **iterativas** y son el tema de estudio de esta unidad.

Capítulo 10

Estructuras de control condicionales

RESUMEN

En algunas partes de un programa puede ser útil detenerse para elegir entre una o más opciones disponibles para continuar con la ejecución del código. Para elegir, planteamos una pregunta o **evaluación lógica**, cuya respuesta deberá ser `VERDADERO` (`TRUE`) o `FALSO` (`FALSE`). Según el resultado, el algoritmo seguirá ciertas acciones e ignorará otras. Estas preguntas y respuestas representan procesos de toma de decisión que conducen a diferentes caminos dentro de un programa, permitiéndonos que la solución para el problema en cuestión sea flexible y se adapte a distintas situaciones. Este tipo de estructuras de control de las instrucciones de programación reciben el nombre de **condicionales** (o *estructuras de selección*) y pueden ser **simples**, **dobles** o **múltiples**.

10.1 Estructuras condicionales simples

Postulan una evaluación lógica y, si su resultado es `TRUE`, se procede a ejecutar las acciones delimitadas entre las llaves que definen el cuerpo de esta estructura. se expresan en R con la siguiente sintaxis:

```
if (condición) {
    hacer esto
}
```

La palabra `if` indica el comando de evaluación lógica, `condición` indica la evaluación a realizar y entre llaves se detallan las instrucciones que se realizan sólo si se cumple la condición, es decir, si la evaluación resulta `TRUE`. Dentro de las llaves puede haber una o muchas líneas de código. Si la condición no se verifica (es `FALSE`), no se ejecuta ninguna acción y el programa sigue su estructura secuencial con el código que prosigue a la última llave.

EJEMPLO

El siguiente programa registra la edad de una persona y, en el caso de que sea mayor de edad, avisa que puede votar en las elecciones provinciales de Santa Fe:

```
# Persona A
edad <- 27
if (edad >= 18) {
    cat("Edad =", edad, "años: puede votar")
}
```

```
Edad = 27 años: puede votar
```

```
# Persona B
edad <- 15
if (edad >= 18) {
    cat("Edad =", edad, "años: puede votar")
}

# no se ejecuta ninguna acción
```

Notar que si bien el uso de *sangrías* o *indentación* en el código dentro de las llaves es opcional, decidimos emplearlo para facilitar la lectura y la identificación del inicio y el fin del bloque condicional. Mantener la prolijidad en nuestros programas es esencial.

10.2 Estructuras condicionales dobles

La estructura simple sólo provee un curso de acción en el caso de que la evaluación sea `TRUE`. La estructura doble permite especificar, además, qué se debe hacer en el caso de que le resultado sea `FALSE`. La sintaxis es:

```
if (condición) {
    hacer esto
} else {
    hacer otra cosa
}
```

Dentro del primer bloque de llaves se escriben las instrucciones que se ejecutan si se cumple la condición, mientras que en el segundo, luego de la expresión `else` (“si no”), se incluyen las que se evalúan si no se verifica la misma.

EJEMPLO

Retomamos el caso anterior para emitir un mensaje cualquiera sea la situación:

```
# Persona A
edad <- 27
if (edad >= 18) {
    cat("Edad =", edad, "años: puede votar")
} else {
    cat("Edad =", edad, "años: no puede votar")
}
```

```
Edad = 27 años: puede votar
```

```
# Persona B
edad <- 15
if (edad >= 18) {
    cat("Edad =", edad, "años: puede votar")
} else {
    cat("Edad =", edad, "años: no puede votar")
}
```

```
Edad = 15 años: no puede votar
```

10.3 Estructuras condicionales múltiples o anidadas

Permiten combinar varias estructuras condicionales para establecer controles más complejos sobre el flujo de la ejecución de las instrucciones, representando una toma de decisión múltiple. Podemos ejemplificar la sintaxis de la siguiente forma:

```
if (condición 1) {
    hacer esto
} else if (condición 2) {
    hacer otra cosa
} else {
    hacer otra cosa distinta
}
```

En la estructura anterior, hay una primera evaluación lógica. Si su resultado es `TRUE`, se ejecuta el código encerrado en el primer juego de llaves y el resto es ignorado. En cambio, si su resultado es `FALSE`, se procede a evaluar la segunda evaluación lógica, que da lugar a la ejecución del segundo bloque de código si el resultado es `TRUE` o del tercer bloque, si su resultado es `FALSE`.

El último bloque de acciones, indicado con `hacer otra cosa distinta`, se evaluará sólo si ninguna de las condiciones lógicas anteriores fue `TRUE`.

EJEMPLO

Completamos el caso de la edad y la votación, considerando la no obligatoriedad para las personas mayores:

```
# Persona A
edad <- 27
if (edad < 18) {
    cat("Edad =", edad, "años: no puede votar")
} else if (edad >= 70) {
    cat("Edad =", edad, "años: puede votar opcionalmente")
} else {
    cat("Edad =", edad, "años: debe votar obligatoriamente")
}
```

```
Edad = 27 años: debe votar obligatoriamente
```

```
# Persona B
edad <- 15
if (edad < 18) {
    cat("Edad =", edad, "años: no puede votar")
} else if (edad >= 70) {
    cat("Edad =", edad, "años: puede votar opcionalmente")
} else {
    cat("Edad =", edad, "años: debe votar obligatoriamente")
}
```

```
Edad = 15 años: no puede votar
```

```
# Persona C
edad <- 81
if (edad < 18) {
    cat("Edad =", edad, "años: no puede votar")
} else if (edad >= 70) {
    cat("Edad =", edad, "años: puede votar opcionalmente")
} else {
    cat("Edad =", edad, "años: debe votar obligatoriamente")
}
```

```
Edad = 81 años: puede votar opcionalmente
```

PARA RESOLVER

¿Cuál será el valor final de `resultado` después de ejecutar el siguiente código en R?

```
x <- 20
if (x < 10) {
  resultado <- x * 20
} else if (x < 20) {
  resultado <- x / 2
} else {
  resultado <- x + 100
}
resultado
```

`resultado` es igual a: _____.

Capítulo 11

Estructuras de control iterativas

RESUMEN

Las estructuras de control iterativas son útiles cuando la solución de un problema requiere que se ejecute repetidamente un determinado conjunto de acciones. El número de veces que se debe repetir dicha secuencia de acciones puede ser fijo o puede variar dependiendo de algún dato o condición a evaluar en el programa. Dependiendo de esto, podemos hacer uso de diferentes estructuras de control iterativas, como las de tipo **for** o las de tipo **while**, que presentamos en este capítulo.

11.1 Estructuras de control iterativas con un número fijo de iteraciones: for

Una estructura **for** se aplican cuando se conoce de antemano el número exacto de veces que se debe repetir una secuencia de instrucciones dentro de un programa. También se conoce como *bucle o loop for*. La sintaxis es:

```
for (variable in conjunto) {
  hacer esto
}
```

En lo anterior, `conjunto` representa un conjunto de elementos, usualmente números o cadenas de texto. El bloque de instrucciones encerrados entre las llaves (que puede contener una o muchas líneas) se ejecutará tantas veces como elementos haya en `conjunto`. Por ejemplo:

```
for (variable in c("Hola", "cómo", "estás")) {
  print("Esto es una repetición.")
}
```

```
[1] "Esto es una repetición."
[1] "Esto es una repetición."
[1] "Esto es una repetición."
```

En el ejemplo, con la expresión `c("Hola", "cómo", "estás")` se define un conjunto de tres cadenas de texto. Dado que hay tres elementos en ese conjunto, la acción indicada entre las llaves se realiza exactamente tres veces. Dicha acción consiste en imprimir en la consola el mensaje "Esto es una repetición" y se realiza gracias al uso de la función `print()`.

¿Y para qué está `variable` en esa estructura? Se trata de un objeto que recibe el nombre de **variable o índice de iteración**. En cada repetición y respetando el orden, el objeto `variable` va a recibir el valor de uno de los elementos del conjunto. En este caso, durante la primera iteración, `variable` almacena el valor `"Hola"`. En la segunda repetición, el valor `"cómo"` y en la tercera, `"estás"`. De hecho, si queremos podemos usar este objeto, cuyo valor va cambiando iteración tras iteración, en las acciones que se implementan dentro de las llaves:

```
for (variable in c("Hola", "cómo", "estás")) {
  print("-------------------------")
  print("Esto es una repetición:")
  print(variable)
}
```

```
[1] "-------------------------"
[1] "Esto es una repetición:"
[1] "Hola"
[1] "-------------------------"
[1] "Esto es una repetición:"
[1] "cómo"
[1] "-------------------------"
[1] "Esto es una repetición:"
[1] "estás"
```

La variable de iteración, que en el ejemplo se llamó `variable`, en realidad puede llevar cualquier nombre que queramos. Es común usar sencillamente el nombre `i`. También es usual que el conjunto de elementos contenga números:

```
for (i in c(1, 2, 3, 4, 5)) {
  print("-------------------------")
  print("En esta iteración i vale:")
  print(i)
}
```

```
[1] "-------------------------"
[1] "En esta iteración i vale:"
[1] 1
[1] "-------------------------"
[1] "En esta iteración i vale:"
[1] 2
[1] "-------------------------"
[1] "En esta iteración i vale:"
[1] 3
[1] "-------------------------"
[1] "En esta iteración i vale:"
[1] 4
[1] "-------------------------"
[1] "En esta iteración i vale:"
[1] 5
```

En los elementos del conjunto son números enteros ordenados, se puede usar un atajo con la forma `inicio:fin`:

```
for (i in 1:5) {
  print("-------------------------")
  print("En esta iteración i vale:")
  print(i)
}
```

```
[1] "-------------------------"
[1] "En esta iteración i vale:"
[1] 1
[1] "-------------------------"
[1] "En esta iteración i vale:"
[1] 2
[1] "-------------------------"
[1] "En esta iteración i vale:"
[1] 3
[1] "-------------------------"
[1] "En esta iteración i vale:"
[1] 4
[1] "-------------------------"
[1] "En esta iteración i vale:"
[1] 5
```

EJEMPLO

El siguiente bloque de código calcula y muestra la **tabla de multiplicar del ocho**:

```
for (i in 0:10) {
  resultado <- 8 * i
  cat("8 x", i, "=", resultado, "\n")
}
```

```
8 x 0 = 0
8 x 1 = 8
8 x 2 = 16
8 x 3 = 24
8 x 4 = 32
8 x 5 = 40
8 x 6 = 48
8 x 7 = 56
8 x 8 = 64
8 x 9 = 72
8 x 10 = 80
```

INFO IMPORANTE

Para emitir mensajes, podemos usar `print()` o `cat()`:

- `print()` tiene la ventaja de que al terminar de emitir el mensaje, agrega un salto de línea: lo próximo que se escriba, aparecerá en un nuevo renglón de la consola. No obstante, no nos permite de forma sencilla concatenar varias piezas de información para armar frases complejas.
- `cat()` nos permite unir cadenas de texto y valores guardados en variables para armar cualquier frase que queramos, separando entre comas cada una de las partes. Sin embargo, no incluye automáticamente un salto de línea: lo próximo que se escriba queda pegado a lo anterior en el mismo renglón. Para evitar esto, incluimos el carácter especial `\n` que representa un salto de línea. Si no lo agreamos, el resultado se ve así:

```
for (i in 0:10) {
  resultado <- 8 * i
  cat("8 x", i, "=", resultado)
}
```

```
8 x 0 = 08 x 1 = 88 x 2 = 168 x 3 = 248 x 4 = 328 x 5 = 408 x 6 = 488 x 7 = 568 x 8 = 648 x 9 = 72
```

EJEMPLO

Imaginemos que queremos escribir un programa que permita calcular la quinta potencia de cualquier número, por ejemplo, 2^5. Para esto, se debe tomar el número 2 y multiplicarlo por sí mismo 5 veces. Por lo tanto, una posible solución es:

```
x <- 2
resultado <- 1
resultado <- resultado * x
resultado <- resultado * x
resultado <- resultado * x
resultado <- resultado * x
resultado <- resultado * x
cat(x, "a la quinta es igual a", resultado)
```

```
2 a la quinta es igual a 32
```

Ya que sabemos que la multiplicación se debe repetir 5 veces, podemos resumir lo anterior con la siguiente estructura:

```
x <- 2
resultado <- 1
for (i in 1:5) {
    resultado <- resultado * x
}
cat(x, "a la quinta es igual a", resultado)
```

```
2 a la quinta es igual a 32
```

PARA RESOLVER

¿Cuál será el valor final de `salida` después de ejecutar el siguiente código en R?

```
salida <- 30
for (i in 1:4) {
  salida <- salida - i
}
```

`salida` es igual a: ___.

11.2 Estructuras de control iterativas con un número indeterminado de iteraciones: while

En otras circunstancias se necesita repetir un bloque de acciones sin conocer con exactitud cuántas veces, sino que esto depende de algún otro aspecto del programa. Las iteraciones deben continuar *mientras que* se verifique alguna condición, dando lugar a la estructura **while**, también conocida como *bucle o loop controlado por una condición.*

En una estructura **while** (mientras), el conjunto de instrucciones se repite mientras que se siga evaluando como `TRUE` una condición declarada al inicio del bloque. Cuando la condición ya no se cumple, el proceso deja de ejecutarse. La sintaxis es:

```
while (condición) {
    hacer esto
}
```

El flujo de ejecución es el siguiente:

1. Se evalúa la `condición`.
2. Si la `condición` es `TRUE`, se ejecuta el bloque de código dentro del `while`.
3. Una vez ejecutado el bloque de código, se vuelve a evaluar la `condición`.
4. Si la `condición` sigue siendo `TRUE`, se repite el proceso.
5. Si la `condición` es `FALSE`, el bucle termina y el programa continúa con las instrucciones que siguen después del `while`.

EJEMPLO

La variable `y` tiene un valor inicial igual a 5. Dado que este valor es mayor a 0, se ejecuta por primera vez el bloque entre llaves: le restamos 1 e imprimimos el resultado, el valor 4. Este valor sigue siendo mayor que 0, por lo tanto el proceso se repite. En una de las iteraciones, `y` queda con el valor 1. Dado que es mayor que 0, la condición es `TRUE`, a `y` le restamos uno, queda igual a 0 y lo imprimimos. La condición se evalúa una vez más, pero resulta en `FALSE`, por lo tanto no se detiene la repetición.

```
y <- 5
while (y > 0) {
  y <- y - 1
  print(y)
}
```

```
[1] 4
[1] 3
[1] 2
[1] 1
[1] 0
```

El valor 0 se imprime porque una vez dentro del bloque, se ejecutan todas sus acciones. El procesamiento no se interrumpe cuando `y` toma el valor 0, sino cuando se vuelve a evaluar la condición lógica y esta es `FALSE`, después de la última vuelta. En otras palabras, **la evaluación de la condición sólo se lleva a cabo al inicio de cada iteración**; si la condición se vuelve `FALSE` en algún punto durante la ejecución del bloque, el programa no lo nota hasta que termine de ejecutarlo y la condición sea evaluada antes de comenzar la próxima iteración.

PARA RESOLVER

En el siguiente bloque intercambiamos de lugar las líneas dentro de las llaves. ¿En qué difiere el resultado?

```
y <- 5
while (y > 0) {
  print(y)
  y <- y - 1
}
```

```
[1] 5
[1] 4
[1] 3
[1] 2
[1] 1
```

Como ya observamos, la evaluación de la condición se lleva a cabo antes de cada iteración, incluso antes de ejecutar el código dentro del bloque por primera vez. Si la condición es `FALSE` inicialmente, entonces las acciones en el cuerpo de la estructura no se ejecutan nunca:

```
y <- -5
while (y > 0) {
  print(y)
  y <- y - 1
}
```

EJEMPLO

Este programa divide un número por 2 mientras que el resultado sea mayor o igual a 1 (es decir, hasta encontrar un valor menor que 1):

```
x <- 100
while (x >= 1) {
    x <- x / 2
    print(x)
}
```

```
[1] 50
[1] 25
[1] 12.5
[1] 6.25
[1] 3.125
[1] 1.5625
[1] 0.78125
```

PARA RESOLVER

¿Cuál será la salida del siguiente código en R?

```
x <- 1
suma <- 0

while (suma + x <= 15) {
  suma <- suma + x
  cat(suma, " ")
  x <- x + 1
}
```

- (A) 1 2 3 4 5

- (B) 1 3 6 10 15
- (C) 1 3 6 10 15 21

COMENTARIO ADICIONAL

En R existe otra estructura iterativa, muy relacionada con el *while*. Se trata de la estructura *repeat*, que repite indefinidamente el bloque de instrucciones entre llaves. Para detener las iteraciones, se incluye dentro del bloque una evaluación lógica con un *if* y la instrucción `break`. Si la condición es `TRUE`, se ejecuta el `break`, deteniendo el proceso iterativo.

```
x <- 100
repeat {
  x <- x / 2
  print(x)
  if (x < 1) break
}
```

```
[1] 50
[1] 25
[1] 12.5
[1] 6.25
[1] 3.125
[1] 1.5625
[1] 0.78125
```

En general, no utilizaremos la estructura *repeat*, pero es otra herramienta disponible.
Así como contamos con la instrucción `break` para detener un proceso iterativo (ya sea un *for*, *while* o *repeat*), también contamos con `next` para saltar una iteración. Notemos su efecto:

```
x <- 100
while (x >= 1) {
    x <- x / 2
    print(x)
}
```

```
[1] 50
[1] 25
[1] 12.5
[1] 6.25
[1] 3.125
[1] 1.5625
[1] 0.78125
```

```r
x <- 100
while (x >= 1) {
    x <- x / 2
    if (x == 12.5) next
    print(x)
}
```

```
[1] 50
[1] 25
[1] 6.25
[1] 3.125
[1] 1.5625
[1] 0.78125
```

11.3 Ejemplos

Con las sentencias de tipo **while** se debe tener mucha precaución, puesto que si la evaluación lógica no está bien especificada o nunca deja de ser evaluada como `TRUE`, se incurre en un *loop* infinito: el programa nunca deja de repetir el bloque (al menos hasta que la máquina se tilde o se produzca un error por desbordamiento de memoria, por ejemplo).

La siguiente situación ilustra esto:

```r
var <- 9
while (var < 10) {
    var <- var - 1
    cat("var =", var, "No puedo parar!!!\n")
}
```

```
var = 8 No puedo parar!!!
var = 7 No puedo parar!!!
var = 6 No puedo parar!!!
var = 5 No puedo parar!!!
var = 4 No puedo parar!!!
var = 3 No puedo parar!!!
var = 2 No puedo parar!!!
var = 1 No puedo parar!!!
var = 0 No puedo parar!!!
var = -1 No puedo parar!!!
var = -2 No puedo parar!!!
.
.
.
```

Si ejecutás ese código, vas a tener que forzar "a mano" el detenimiento del procesamiento, con el botón rojo de "Stop" arriba a la derecha de la consola, o con el atajo `CTRL + C`. También podemos implementar un "conteo" de las iteraciones y agregar un `break`: si el bloque se repitió, por ejemplo, 15 veces, que se detenga:

```
conteo <- 0
var <- 9
while (var < 10) {
    var <- var - 1
    cat("var =", var, "No puedo parar!!!\n")
    conteo <- conteo + 1
    if (conteo == 15) break
}
cat("Se hicieron", conteo, "iteraciones.")
```

```
var = 8 No puedo parar!!!
var = 7 No puedo parar!!!
var = 6 No puedo parar!!!
var = 5 No puedo parar!!!
var = 4 No puedo parar!!!
var = 3 No puedo parar!!!
var = 2 No puedo parar!!!
var = 1 No puedo parar!!!
var = 0 No puedo parar!!!
var = -1 No puedo parar!!!
var = -2 No puedo parar!!!
var = -3 No puedo parar!!!
var = -4 No puedo parar!!!
var = -5 No puedo parar!!!
var = -6 No puedo parar!!!
Se hicieron 15 iteraciones.
```

Las distintas estructuras iterativas se pueden combinar entre sí, dando lugar flujos de ejecución del código altamente flexibles. En el siguiente caso se tienen dos estructuras `for` anidadas:

```
for (i in 1:3) {
    for (j in 1:2) {
        suma <- i + j
        cat("| i vale", i, "| j vale", j, "| La suma es igual a", suma, "|\n")
    }
}
```

```
| i vale 1 | j vale 1 | La suma es igual a 2 |
| i vale 1 | j vale 2 | La suma es igual a 3 |
```

```
| i vale 2 | j vale 1 | La suma es igual a 3 |
| i vale 2 | j vale 2 | La suma es igual a 4 |
| i vale 3 | j vale 1 | La suma es igual a 4 |
| i vale 3 | j vale 2 | La suma es igual a 5 |
```

En primer lugar, `i` toma el valor 1, y entonces `j` varía de 1 a 2, generando las combinaciones `i = 1, j = 1` e `i = 1, j = 2`. Luego de que el *loop* de `j` finalice habiendo recorrido todo su campo de variación, comienza la segunda iteración del *loop* de `i`, actualizándose su valor a 2 y comenzando otra vez el *loop* de `j`, que varía de 1 a 2. Así, se generan las combinaciones `i = 2, j = 1` e `i = 2, j = 2`. Finalmente, se actualiza `i` y pasa a valer 3, generando las combinaciones `i = 3, j = 1` e `i = 3, j = 2`. Para cada combinación, se muestra el valor de la suma entre `i` y `j`.

Recordemos que las variables de iteración pueden recibir cualquier nombre:

```
for (guau in 1:3) {
    for (miau in 1:2) {
        suma <- guau + miau
        cat("| guau vale", guau, "| miau vale", miau, "| La suma es igual a", suma, "|\n")
    }
}
```

```
| guau vale 1 | miau vale 1 | La suma es igual a 2 |
| guau vale 1 | miau vale 2 | La suma es igual a 3 |
| guau vale 2 | miau vale 1 | La suma es igual a 3 |
| guau vale 2 | miau vale 2 | La suma es igual a 4 |
| guau vale 3 | miau vale 1 | La suma es igual a 4 |
| guau vale 3 | miau vale 2 | La suma es igual a 5 |
```

Con estas herramientas, podemos crear programas que nos permitan verificar algunos principios matemáticos. Por ejemplo, sabemos que la suma de los primeros n números naturales es igual a $n(n+1)/2$. Entonces, la suma de los 50 primeros números naturales $(1 + 2 + 3 + ... + 50)$ debe ser igual a $50 \times 51/2 = 1275$. ¿Será verdad?

```
n <- 50
suma <- 0
for (i in 1:n) {
    suma <- suma + i
}
suma
```

```
[1] 1275
```

Ahora que nos quedamos tranquilos de que ese postulado matemático se cumple, si necesitamos esa suma podemos hacer sencillamente:

```
n * (n + 1) / 2
```

```
[1] 1275
```

Es momento de dejar volar nuestra imaginación y plantearnos cualquier problema de este estilo que se nos ocurra, ya estamos en condiciones de resolverlo. Por ejemplo, sumemos números naturales hasta que la suma pase el valor 100:

```
suma <- 0
nro_natural <- 1
while (suma < 100) {
    suma <-  suma + nro_natural
    nro_natural <- nro_natural + 1
}
cat("Se deben sumar los primeros", nro_natural - 1, "números naturales para superar el valor
```

```
Se deben sumar los primeros 14 números naturales para superar el valor 100.
```

```
cat("La suma de los primeros", nro_natural - 1, "números naturales es igual a", suma, ".")
```

```
La suma de los primeros 14 números naturales es igual a 105 .
```

> **PARA RESOLVER**
>
> Analizar con atención el caso anterior y explicar por qué se utiliza `nro_natural - 1` para indicar cuántos números naturales formaron parte de la suma.

¿Alguien sabe cuántos múltiplos de 8 menores a 150 hay? Contemos[1]:

```
# cada nro es múltiplo de sí mismo, así que el primero es el mismo 8
multiplo <- 8
# contamos que ya tenemos identificado al primer múltiplo
conteo <- 1
# encontramos los siguientes múltiplos sumando de a 8
while (multiplo < 150) {
    print(multiplo)
    multiplo <- multiplo + 8
    conteo <- conteo + 1
}
```

[1] Para gente curiosa: este conteo se puede hacer sencillamente con `150 %/% 8 + 1`. ¿Por qué?

```
[1] 8
[1] 16
[1] 24
[1] 32
[1] 40
[1] 48
[1] 56
[1] 64
[1] 72
[1] 80
[1] 88
[1] 96
[1] 104
[1] 112
[1] 120
[1] 128
[1] 136
[1] 144
```

```
cat("Hay", conteo, "múltiplos de 8 menores que 150.")
```

```
Hay 19 múltiplos de 8 menores que 150.
```

Capítulo 12

Práctica de la Unidad 2

12.1 Ejercicio 1

Escribir un programa en R para determinar si un número entero dado es par o impar.

12.2 Ejercicio 2

Escribir un programa en R para leer tres números y determinar cuál es el mayor.

12.3 Ejercicio 3

Una fábrica organiza a sus operarios en tres turnos de trabajo rotativos: mañana, tarde y noche. Los turnos mañana y tarde se pagan $4000 la hora, mientras que en el turno noche se paga un adicional de $2000. Además, los domingos se paga un adicional de $1000 la hora. Escribir un programa en R que permita calcular cuánto se le debe pagar a un operario por un día de trabajo, dados el turno, el día de la semana y la cantidad de horas trabajadas. Los turnos son identificados mediante los caracteres `"M"`, `"T"` y `"N"`; los días de semana mediante las primeras tres letras (`"DOM"`, `"LUN"`, `"MAR"`, `"MIE"`, `"JUE"`, `"VIE"`, `"SAB"`).

12.4 Ejercicio 4

Escribir un programa en R para convertir un valor de temperatura expresado en grados Celsius a su equivalente en grados Farenheit y viceversa. El problema debe leer la magnitud a convertir en la variable numérica `temp` y el tipo de conversión en la variable caracter `modo`, que puede tomar los valores `"C a F"` o `"F a C"`.

12.5 Ejercicio 5

Escribir un programa en R que permita calcular:

a. la suma de los `n` primeros números naturales.
b. la suma de los cuadrados de los `n` primeros números naturales.
c. el producto de los primeros `n` números naturales impares.
d. la suma de los cubos de los `n` primeros números naturales pares.

12.6 Ejercicio 6

Escribir un programa en R que permita calcular el factorial de un número natural `n`, `n!`. Realizarlo de dos formas, empleando estructuras iterativas diferentes.

12.7 Ejercicio 7

En el siglo XIII el matemático italiano Leonardo Fibonacci ideó una secuencia matemática que lleva su nombre, intentando explicar el crecimiento geométrico de una población de conejos. Los primeros dos términos de la secuencia son 0 y 1, y cada uno de los subsecuentes términos es la suma de los dos anteriores. De esta forma, el inicio de la secuencia de Fibonacci es: 0 - 1 - 1 - 2 - 3 - 5 - 8 - … . Escribir un programa en R para mostrar todos los términos de la secuencia de Fibonacci menores que 10000.

12.8 Ejercicio 8

Se tiene una caja fuerte que, para poder abrirla, se desea crear una lista con todas las combinaciones posibles. Se sabe que:

- La combinación tiene 3 cifras.
- La combinación es múltiplo de 11.
- La combinación no es mayor que 800.
- La combinación no es múltiplo de 8.
- La combinación no comienza con 0.

Escribir un programa en R que permita imprimir una lista con todas las combinaciones posibles bajo las condiciones anteriores.

12.9 Ejercicio 9

En un pequeño pueblo en expansión, la población es igual a 1000 al inicio del año. La población aumenta regularmente un 2% anual. Además, 50 nuevos habitantes se mudan al pueblo cada año.

a. Al finalizar el año, ¿cuál es el tamaño de la población? Expresar una fómula general usando la simbología `po` para el valor inicial de la población (1000), `tasa` para el porcentaje de aumento anual (2) e `inmigrantes` para la cantidad de nuevos habitantes que llegan por año (50).

b. ¿Cuántos años tienen que transcurrir para que el pueblo alcance una población mayor o igual a 1200 habitantes? Realice la cuenta para cada año, tomando la parte entera del resultado en cada oportunidad, en el caso de que el resultado tenga decimales.

c. Escribir un programa en R que permita responder la pregunta del punto anterior para este caso y para cualquier otro, en el que cambien los parámetros del problema (la población inicial, población objetivo, tasa de crecimiento anual y cantidad de habitantes que inmigran al pueblo). El programa debe terminar emitiendo una descripción de la situación del problema y cuántos años deben transcurrir para llegar a la población objetivo.

d. Repetir el cálculo para la situación en la que la población inicialmente es de 10000 habitantes, hay un crecimiento anual del 3%, se añaden 100 inmigrantes por año y se desea saber cuántos años llevará superar los 50000 habitantes.

Capítulo 13

Actividad de autoevaluación 2

INFO IMPORANTE
Esta autoevaluación DEBE ser completada sin usar R para poder razonar las preguntas.

13.1 Pregunta 1

Se desea determinar si una persona puede conducir o no, en función de las siguientes reglas:

- Si tiene 18 años o más y tiene carnet, puede conducir.
- Si tiene 70 años o más, o no tiene carnet, necesita una evaluación adicional.

Se presentan distintas opciones para hacer la evaluación requerida:

```
# Opción A
if (edad >= 18 && tiene_carnet) {
  print("Puede conducir")
} else if (edad >= 70 || !tiene_carnet) {
  print("Requiere evaluación adicional")
}

# Opción B
if (edad >= 18 && tiene_carnet) {
  print("Puede conducir")
}
if (edad >= 70 || !tiene_carnet) {
  print("Requiere evaluación adicional")
}
```

```
# Opción C
if (edad >= 70 || !tiene_carnet) {
  print("Requiere evaluación adicional")
} else if (edad >= 18 && tiene_carnet) {
  print("Puede conducir")
}

# Opción D
if (edad >= 18 && tiene_carnet) {
  print("Puede conducir")
} else {
  print("Requiere evaluación adicional")
}
```

¿Cuál es la opción correcta?:

- (A) A
- (B) B
- (C) C
- (D) D

Explicación

Del enunciado hay que tener en cuenta que algunas personas pueden cumplir las dos reglas a la vez. Por ejemplo: si una persona tiene 80 años y carnet, puede conducir pero además necesita una evaluación adicional (por la edad). Esto significa que las condiciones NO son mutuamente excluyentes (pueden cumplirse sin excluirse una a la otra). Para resolver este tipo de condiciones se utilizan `if` independientes así, si una persona cumple las dos condiciones, recibe ambos mensajes.

Problemas con las otras opciones:

- Opciones `A` y `D`: usan `else / else if`, es decir, solo muestran uno de los dos mensajes (aunque ambas reglas apliquen). Ejemplo: 80 años con carnet solo diría "Puede conducir" (y se olvida de la evaluación).
- Opción `C`: pone la evaluación primera, si alguien tiene +70 años, siempre mostrará solo "Necesita evaluación" (aunque igual pueda conducir).

Conclusión clave:

Cuando dos reglas pueden cumplirse juntas, usamos `if` independientes. Si usamos `else / else if`, el programa se detiene en la primera condición `TRUE`.

13.2 Pregunta 2

¿Cuál de las siguientes expresiones **NO** muestra la tabla del 8?

```
# Opción A
for (i in 0:10) {
  cat("8 por", i, "es", 8 * i, "\n")
}

# Opción B
numero <- 0
for (i in 0:10) {
  cat("8 por", i, "es", numero, "/n")
  numero <- numero + 8
}

# Opción C
numero <- 1
for (i in 0:10) {
  cat("8 por", i, "es", numero, "\n")
  numero <- numero * 8
}
```

La opción que no muestra la tabla del 8 es la:

- (A) A
- (B) B
- (C) C

Explicación

- La opción `A` está bien porque multiplica directamente 8 por cada valor de `i`.
- La opción `B` también es correcta, porque empieza desde cero y va sumando 8 en cada paso, lo cual genera la tabla correctamente.
- En cambio, la opción `C` no muestra la tabla del 8 por dos razones:
  - En primer lugar, el primer `cat("8 por", i, "es", numero, "\n")` da como resultado `8 por 0 es 1` puesto que la variable `numero` se inicializa en 1.
  - Además, en cada vuelta se multiplica el resultado anterior por 8, en lugar de sumarle 8, lo cual genera una secuencia exponencial.

13.3 Pregunta 3

En una fábrica, los tornillos se empaquetan de a 10 y luego los paquetes se agrupan de a 5 en cajas. En la etapa de control de calidad se eligen al azar 3 cajas y se pesan todos los tornillos. Se desea escribir un programa que informe el peso de cada tornillo evaluado, indicando a qué caja y paquete pertenece. Además, se desea informar el porcentaje de tornillos evaluados que pesan más de 1 gramo, puesto que se los considera defectuosos.

Completar el siguiente fragmento de código, escribiendo las partes faltantes en los campos en blanco disponibles debajo. Para las variables de iteración se puede utilizar nombres descriptivos como `caja`, `paquete` y `tornillo` o los genéricos `i`, `j` o `k`:

```
cajas <- 3
paquetes <- 5
tornillos <- 10
defectuosos <- 'A ___'

for ('B ___' in 1:'C ___') {
  for ('D ___' in 1:'E ___') {
    for ('F ___' in 1:'G ___') {
      # Se pesa el tornillo (la siguiente línea genera un número al azar)
      peso <- round(rnorm(1, 0.99, 0.01), 4)
      cat(
        "El tornillo", 'H ___', "del paquete", 'I ___', "de la caja", 'J ___',
        "pesa", 'K ___', "gramos.\n"
      )
      if (peso > 'L ___') {
        defectuosos <- 'M ___' + 1
      }
    }
  }
}
total_tornillos <- cajas * paquetes * 'N ___'
porcentaje <- round(defectuosos / 'Ñ ___' * 100)
cat("El porcentaje de tornillos defectuosos es", 'O ___', "%")
```

- A: _
- B: _____
- C: _____
- D: _____
- E: _____
- F: _____
- G: _____
- H: _____
- I: _____
- J: _____

- K: _____
- L: _
- M: _____
- N: _____
- Ñ: _____
- O: _____

Explicación

En este ejercicio, queremos recorrer todas las `cajas`, luego cada `paquete` dentro de cada caja, y finalmente cada `tornillo` dentro de cada paquete. Como ya están definidas las cantidades totales de `cajas`, `paquetes` y `tornillos`, esas variables se deben usar como los límites de los bucles. Para verificar si un tornillo está en buen estado, usamos una condición simple: si su peso es mayor a 1, contamos un tornillo defecutoso más, en una variable que fue iniciada en 0:

```
cajas <- 3
paquetes <- 5
tornillos <- 10
defectuosos <- 0

for (caja in 1:cajas) {
  for (paquete in 1:paquetes) {
    for (tornillo in 1:tornillos) {
      # Se pesa el tornillo (la siguiente línea genera un número al azar)
      peso <- round(rnorm(1, 0.99, 0.01), 4)
      cat(
        "El tornillo", tornillo, "del paquete", paquete, "de la caja", caja,
        "pesa", peso, "gramos.\n"
      )
      if (peso > 1) {
        defectuosos <- defectuosos + 1
      }
    }
  }
}
total_tornillos <- cajas * paquetes * tornillos
porcentaje <- round(defectuosos / total_tornillos * 100)
cat("El porcentaje de tornillos defectuosos es", porcentaje, "%")
```

13.4 Pregunta 4

A partir del siguiente fragmento de código, responder las siguientes preguntas:

```
var <- 0
n <- 1

while (n <= 5) {
    var <- var + n * n
    n <- n + 1
}
```

- ¿Qué valor almacena la variable `n` al finalizar?

  Respuesta: _

- ¿Qué valor almacena la variable `var` al finalizar?

  Respuesta: ___

- ¿Cuántas veces se ejecutó el bloque de código encerrado en la estructura iterativa?

  Respuesta: _ veces.

- Si en lugar de asignar al inicio `n <- 1`, se hubiese asignado `n <- 6`, ¿cuál sería el valor de `var` al finalizar?

  Respuesta: _

Explicación

Este código usa un bucle `while` que se repite mientras la condición `n <= 5` se cumpla. Arranca con `n` en 1 e incrementa de a uno en cada vuelta. En cada iteración, suma el cuadrado de `n` a `var`. Entonces, lo que estamos haciendo es calcular la suma de los cuadrados de los números del 1 al 5. Al finalizar, `n` vale 6 porque ya no cumple la condición para seguir. `var` vale 55 porque es $1^2 + 2^2 + 3^2 + 4^2 + 5^2$. El bloque se ejecuta 5 veces, una por cada valor de `n` de `1` a `5`. Y si hubiéramos empezado con `n <- 6`, como no se cumple la condición desde el principio, el bucle no se ejecuta y `var` queda en 0.

13.5 Pregunta 5

Analizar el código y determinar si las siguientes afirmaciones son verdaderas o falsas:

```
stock <- 7
tipo_producto <- "alimento"

while (stock > 0) {
  if (stock >= 5 && tipo_producto == "alimento") {
    cat("¡OFERTA EN ALIMENTOS!", "\n")
  } else if (stock <= 3 && tipo_producto == "alimento") {
    cat("¡QUEDAN POCAS UNIDADES!", "\n")
  }
  stock <- stock - 1
```

```
}

cat("Producto no disponible", "\n")
```

- El mensaje `"¡QUEDAN POCAS UNIDADES!"` se imprimirá 3 veces.
- (A) Verdadero
- (B) Falso
- Si `tipo_producto <- "limpieza"` el código no se ejecutará.
- (A) Verdadero
- (B) Falso
- El bucle `while` se ejecuta 6 veces.
- (A) Verdadero
- (B) Falso
- El valor final de `stock` al terminar el bucle es 1.
- (A) Verdadero
- (B) Falso
- Si `stock <- 0` se imprime `"Producto no disponible"`.
- (A) Verdadero
- (B) Falso
- Si las condiciones que evalúan el `if` y `else if` se invirtieran en posición la salida del programa cambiaría.
- (A) Verdadero
- (B) Falso

Explicación

Este ejercicio muestra cómo el orden de las condiciones y el valor de las variables iniciales afectan la ejecución:

- El mensaje `"¡QUEDAN POCAS UNIDADES!"` se imprime 3 veces porque eso pasa cuando `stock` baja de 3 a 1, cumpliendo la condición del `else if`.

- El bucle se ejecuta 7 veces en total, porque empieza en 7 y termina cuando `stock` baja a 0.
- Si `tipo_producto` tuviese el valor `"limpieza"`, el bucle sí se ejecuta, pero no se imprime ninguno de los mensajes dentro del `if` o `else if`, ya que esas condiciones no se cumplen.
- El valor final de `stock` es 0, no 1, porque en cada vuelta se resta 1 hasta llegar a cero.
- Si arrancáramos con `stock <- 0`, el bucle no se ejecuta pero igual se imprime `"Producto no disponible"`, porque esa línea está fuera del `while`.
- Por último, invertir el orden de `if` y `else if` no cambia el resultado en este caso porque las condiciones nunca se solapan; o se cumple una o la otra, pero nunca ambas al mismo tiempo (son excluyentes).

Unidad 3. Descomposición algorítmica

RESUMEN

En este capítulo exploraremos un concepto fundamental en la programación: la creación de **nuevas funciones**. A medida que los programas crecen en tamaño y complejidad, escribir el mismo código repetidas veces no solo es ineficiente, sino que también aumenta el riesgo de cometer errores. Para solucionar esto, podemos encapsular bloques de código en funciones reutilizables, mejorando la claridad y la organización de nuestros programas.

Capítulo 14

Creación de nuevas funciones en R

RESUMEN

En este capítulo analizaremos por qué el uso de funciones es una herramienta poderosa en la resolución de problemas y cómo nos permite estructurar mejor nuestro código mediante la **descomposición algorítmica**. Luego, aprenderemos cómo definir funciones en R, especificando parámetros con valores por defecto y controlando su comportamiento.

14.1 La importancia de la descomposición algorítmica

Un principio clave en la resolución de problemas es la **descomposición algorítmica**, es decir, dividir un problema complejo en partes más pequeñas y manejables. En programación, esto se traduce en la creación de **subalgoritmos**: fragmentos de código que resuelven una parte específica del problema. Este enfoque, también conocido como **descomposición modular**, facilita la comprensión del código y permite reutilizar soluciones ya escritas.

En R, los subalgoritmos se implementan a través de **funciones**. Una función encapsula una serie de instrucciones y puede ser invocada desde distintos puntos del código cada vez que se necesite, sin necesidad de reescribir esas instrucciones encapsuladas una y otra vez. Esto aporta varios beneficios fundamentales:

- **Mejor legibilidad del código**: si el programa es muy largo porque las mismas instrucciones aparecen muchas veces, encerrarlas dentro de una función que es invocada en una sola línea cada vez que se necesita hace que el código se vuelva más corto. Además, al dar a una función un nombre descriptivo, el propósito de la línea de código que la invoca se vuelve más claro. Todo esto resulta en código más comprensible para las personas.

- **Facilidad de mantenimiento**: si hay que modificar una funcionalidad, basta con actualizar la función sólo en el lugar donde está definida, en lugar de hacerlo en múltiples fragmentos de código copiados y pegados.
- **Reducción de errores**: copiar y pegar código manualmente puede dar lugar a errores accidentales, como olvidar cambiar un nombre de variable en alguna de las copias.

- **Mayor eficiencia y reutilización**: una vez definida, una función puede utilizarse muchas veces, en el mismo o en otros proyectos, ahorrando tiempo y esfuerzo.

Las funciones son fundamentales para organizar y estructurar programas, ya que permiten dividir un problema en partes más pequeñas y reutilizar código en diferentes puntos del programa sin necesidad de repetirlo.

14.2 Definición de una función

CONCEPTO CLAVE
En R, una función es un bloque de código reutilizable que realiza una tarea específica. Las funciones toman argumentos de entrada, ejecutan una serie de instrucciones y pueden devolver un resultado.

Para ejemplificar, podemos decir que la noción de *función* en programación se asemeja a la idea matemática de *función de una o más variables*. Pensemos en la función $f(x, y) = x^2 + 3y$ (ejemplo 1). Si queremos saber cuál es el valor numérico de la función f cuando x toma el valor 4 e y toma el valor 5, reemplazamos en la expresión anterior las variables por los valores mencionados y obtenemos: $f(4, 5) = 4^2 + 3 \times 5 = 31$.

Podemos definir dicha función en R de la siguiente manera:

```
f <- function(x, y) {
    resultado <- x^2 + 3 * y
    return(resultado)
}
```

La estructura general de la definición de una función consta de tres componentes:

```
nombre <- function(argumentos) {
  cuerpo
}
```

1. **Nombre**: elegido por nosotros, respetando las reglas para la elección de nombres para objetos y buscando que provea una buena descripción del propósito de la función. Usamos el operador de asignación (`<-`) para asociar ese nombre a la definición de la función, señalada con la palabra clave `function`.

2. **Argumentos** o **parámetros**: listado de piezas de información que la función necesita para operar y que pueden variar cada vez que es invocada. Se listan entre paréntesis y separados por comas, a la derecha de la palabra clave `function`.
3. **Cuerpo**: conjunto de instrucciones de programación que la función ejecuta cada vez que es invocada, encerrado por un par de llaves. Generalmente finaliza con la función `return()`, que indica el fin de la ejecución y provee el objeto que la función devuelve.

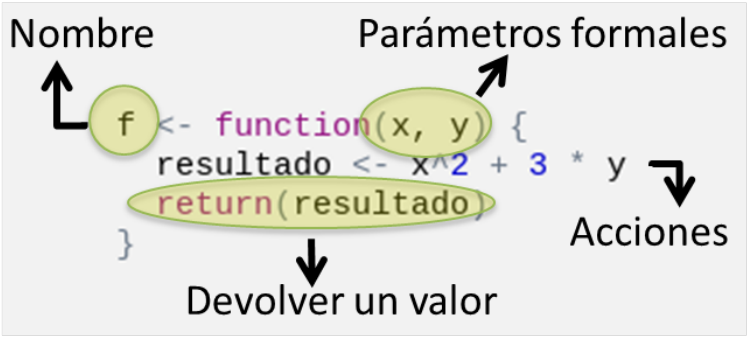


Figura 14.1: Estructura de una función en R.

Una vez que la definición de la función es ejecutada, pasa a formar parte de los objetos que conforman al ambiente global, como se puede apreciar al verla listada en el panel *Environment* de RStudio. A partir de este momento, podemos utilizarla, como parte de otro programa. Para invocarla, escribimos el nombre de la función y entre paréntesis los valores que nos interesan para el cálculo:

```
# Ejemplos de uso de la función f
f(4, 5)
```

```
[1] 31
```

```
f(6, -5)
```

```
[1] 21
```

```
f(0, 0)
```

```
[1] 0
```

Los **parámetros** o **argumentos** constituyen el *input* o información de entrada con la cual se realizarán las operaciones. Considerando el ejemplo, decimos que x e y son los **parámetros formales** o **ficticios**, ya que son símbolos que permiten expresar de manera general las acciones que la función ejecuta. Describen lo que uno diría en palabras: "hay que tomar a x, elevarlo al cuadrado y sumarle la y multiplicada por 3".

Los valores en los cuales se quiere evaluar la función se llaman **parámetros actuales** o **reales**. Por ejemplo, si nos interesa calcular $f(4, 5)$, los valores 4 y 5 son los parámetros actuales y se establece una correspondencia entre el parámetro formal x y el actual 4, así como entre la y y el 5. El resultado que se obtiene, como observamos antes, es 31 y este es el valor que la función *devuelve*.

Recordando lo discutido en Sección 2.6, podemos apreciar que los siguientes usos de la función `f()` son equivalentes:

```
f(4, 5)
```

```
[1] 31
```

```
f(x = 4, y = 5)
```

```
[1] 31
```

```
f(y = 5, x = 4)
```

```
[1] 31
```

Sin embargo, no son equivalentes los siguientes:

```
# Siguiendo el orden de definición, x recibe el valor 4 e y recibe el 5:
f(4, 5)
```

```
[1] 31
```

```
# Siguiendo el orden de definición, x recibe el valor 5 e y recibe el 4:
f(5, 4)
```

```
[1] 37
```

A continuación, podemos ver casos que generan error por hacer un uso incorrecto de la función:

```
# Error por omitir un argumento de uso obligatorio (x recibe 4, falta y)
f(4)
```

```
Error in f(4): argument "y" is missing, with no default
```

```
# Error por proveer más argumentos de los declarados en la definición
f(4, 5, 6)
```

```
Error in f(4, 5, 6): unused argument (6)
```

14.3 Función `return()`

La instrucción `return()` provoca la inmediata finalización de la ejecución de la función e indica cuál es el objeto que la misma devuelve como resultado. Aunque generalmente lo encontramos al final de la definición de la función, es posible incluir más de un `return()` para que el resultado dependa de alguna condición, aunque sólo uno llegue a ejecutarse.

EJEMPLO

Uso de dos instrucciones `return()` en la definición de una función que devuelve cuál es el mayor entre dos números:

```
maximo <- function(num1, num2) {
    if (num1 > num2) {
        return(num1)
    } else {
        return(num2)
    }
}

maximo(0, 10)
```

```
[1] 10
```

```
maximo(0, -10)
```

```
[1] 0
```

La función `return()` puede omitirse, ya que si no está presente se devuelve el resultado de la última expresión analizada. Por eso, las siguientes funciones son equivalentes:

```
# return explícito
sumar1 <- function(x, y) {
    resultado <- x + y
    return(resultado)
```

```
}

# return implícito
sumar2 <- function(x, y) {
    x + y
}

sumar1(4, 5)
```

```
[1] 9
```

```
sumar2(4, 5)
```

```
[1] 9
```

No obstante, es aconsejable usar `return()` para evitar ambigüedades y ganar en claridad. Además, en funciones más complejas, su uso puede ser indispensable para indicar el término de la evaluación de la función.

14.4 Argumentos con valores asignados por defecto

Hemos visto que algunos argumentos de las funciones predefinidas de R tienen valores asignados por defecto, como es el caso de la función `log()`, que a menos que indiquemos otra cosa opera con la base natural. Cuando definimos nuestras propias funciones, también es posible asignarle un valor por defecto a uno o más de sus argumentos.

Recordemos la definición de la función `f`:

```
f <- function(x, y) {
    resultado <- x^2 + 3 * y
    return(resultado)
}
```

Esta función también podría ser definida así:

```
nueva_f <- function(x, y = 100) {
    resultado <- x^2 + 3 * y
    return(resultado)
}
```

Esto significa que si no proveemos un valor para el argumento `y`, se le asignará por default el valor 100. Luego:

```
nueva_f(4)
```

```
[1] 316
```

En el caso anterior se hace corresponder el 4 al primer argumento de la función, `x`, y como no proveemos ningún otro parámetro actual, `y` recibe por defecto el valor 100 y se calcula: `x^2 + 3 * y = 16 + 300 = 316`.

Por supuesto, podemos proveer cualquier otro valor para `y`, de modo que no se use el valor asignado por default:

```
nueva_f(4, 5)
```

```
[1] 31
```

Como `x` no tiene valor asignado por default en la función `nueva_f()`, siempre debemos pasarle uno. En caso contrario, recibiremos un error:

```
nueva_f(y = 5)
```

```
Error in nueva_f(y = 5): argument "x" is missing, with no default
```

```
nueva_f()
```

```
Error in nueva_f(): argument "x" is missing, with no default
```

14.5 ¿Dónde escribimos el código que define nuestras funciones?

El código que define una función tiene que ser ejecutado **antes** que el código que pretende usarla, de modo que la función aparezca entre los objetos disponibles del *Global Environment*. No podemos usar una función cuya definición no haya sido ejecutada, puesto que produciremos un error que indica que tal objeto no existe. Esto nos obliga a pensar dónde escribimos el código para crear nuevas funciones y en qué momento es ejecutado.

En proyectos de pequeña extensión, donde todo el problema se resuelve en un único y acotado *script* y sólo se definen unas pocas nuevas funciones, podemos incluirlas al comienzo del archivo, para que sean evaluadas antes del código que las invoca.

EJEMPLO

A continuación se presenta el contenido de un breve script de código que comienza con la definición de dos funciones, usadas posteriormente.

```
# ------------------------------------------------------------------
# DEFINICIÓN DE FUNCIONES
# ------------------------------------------------------------------

f <- function(x, y) {
    resultado <- x^2 + 3 * y
    return(resultado)
}

maximo <- function(num1, num2) {
    if (num1 > num2) {
        return(num1)
    } else {
        return(num2)
    }
}

# ------------------------------------------------------------------
# PROGRAMA
# ------------------------------------------------------------------

rtdo1 <- f(2, 5)
rtdo2 <- f(3, 10)
rtdo3 <- maximo(rtdo1, rtdo2) + 20
cat("El resultado es", rtdo3)
```

PARA RESOLVER

¿Cuál es el mensaje que emite la última línea del script del ejemplo?

_____.

Cuanto más grande o complejo es el problema a resolver, más funciones deben ser programadas. Por eso, con el objetivo de tener un mayor orden en nuestro código, podemos escribir nuestras funciones en uno o más archivos separados, creados específicamente para esto ("scripts de funciones"). Si lo hacemos, al comienzo del script en el que estamos resolviendo un problema que involucra el uso de las funciones creadas, debemos incluir una instrucción para que el script de funciones sea evaluado, de modo que las funciones ahí definidas sean ejecutadas y pasen a formar parte del *Global Environment*. Esta instrucción es la función `source()`, que toma como único argumento el nombre del script de funciones.

Para ilustrar esto, vamos a recordar que en el ejercicio 6 de la Práctica 2 escribimos un programa para el cálculo de factoriales. Dado que los mismos son muy útiles en variadas aplicaciones, podemos

definir una nueva función se encargue de obtenerlos. Escribimos la definición en un script llamado `funciones.R`, cuyo contenido es:

```
fact <- function(n) {
    resultado <- 1
    if (n > 0) {
        for (i in 1:n) {
            resultado <- resultado * i
        }
    }
    return(resultado)
}
```

Luego, en cualquier script donde se presente un problema que requiera el cálculo de factoriales, vamos a pedirle a R que ejecute el código guardado en `funciones.R` con la sentencia `source()`, como paso inicial. Por ejemplo, en el script `mis_factoriales.R` se usa la función `fact()` para calcular el factorial de los primeros diez números naturales. Su contenido es:

```
# ---------------------------------------------------------------
# PROGRAMA: Mostrar los factoriales de los 10 primeros naturales
# ---------------------------------------------------------------

source("funciones.R")

for (j in 1:10) {
    cat("El factorial de", j, "es igual a", fact(j), "\n")
}
```

```
El factorial de 1 es igual a 1
El factorial de 2 es igual a 2
El factorial de 3 es igual a 6
El factorial de 4 es igual a 24
El factorial de 5 es igual a 120
El factorial de 6 es igual a 720
El factorial de 7 es igual a 5040
El factorial de 8 es igual a 40320
El factorial de 9 es igual a 362880
El factorial de 10 es igual a 3628800
```

Gracias a `source()` todas las funciones definidas en el archivo `funciones.R` aparecerán en el entorno y no hay necesidad ni siquiera de abrirlo. Esto funcionará siempre que este archivo esté guardado en el directorio de trabajo. En caso contrario, se debe indicar en `source()` el *path* completo hacia el script de funciones (por ejemplo, `C:/Documentos/Facultad/funciones.R`), pero esto no es recomendable. Es preferible que todos los archivos estén correctamente organizados en la carpeta principal de

nuestro proyecto y, en caso de necesitarlo, usemos rutas relativas con respecto a la misma, tal como mencionamos en Sección 5.3.

Más adelante veremos una forma mejor de guardar y distribuir las funciones que inventamos: podemos crear un nuevo paquete de R que las almacene.

Capítulo 15

Alcance de las variables

RESUMEN

En este capítulo abordaremos el concepto de **ambiente de una función**, explicando cómo se gestionan las variables dentro de una función y cómo se relacionan con las variables que residen en el *Global Environment*.

COMENTARIO ADICIONAL

Advertencia sobre el contenido de este capítulo

El material presentado en este capítulo es una **simplificación con fines didácticos**. Se han omitido ciertos detalles técnicos y se han realizado aproximaciones para facilitar la comprensión de los conceptos fundamentales. En realidad, muchos de los temas aquí tratados son más **complejos y matizados**, especialmente en lo que respecta a aspectos avanzados del lenguaje R, la gestión de ambientes y la evaluación de expresiones. Quienes deseen profundizar en estos temas con mayor rigor, pueden explorar fuentes adicionales, como las descriptas en la bibliografía.

15.1 Pasaje de parámetros

Las funciones y los programas desde los que se invocan comunican información entre sí a través de los parámetros. Esta comunicación recibe el nombre de **pasaje de argumentos** y se puede realizar de distintas formas, siendo las más comunes *por valor* o *por referencia*. Algunos lenguajes de programación trabajan con uno u otro sistema, mientras que otros lenguajes permiten el uso de ambos.

En R, el pasaje de argumentos es **por valor**. Esto quiere decir que los argumentos representan valores que se transmiten **desde** el programa que invoca la función **hacia** la misma. Los objetos del *Global Environment* provistos como argumentos en la llamada a la función no serán modificados por su ejecución. Este sistema funciona de la siguiente forma:

1. Se evalúan los argumentos actuales usados en la invocación a la función.
2. Los valores obtenidos se *copian* en los argumentos formales dentro de la función.
3. Los argumentos formales se usan como variables dentro de la función. Aunque los mismos sean modificados (por ejemplo, se les asignen nuevos valores), no se modifican los argumentos actuales en el *Global Environment*, sólo sus copias dentro de la función.

EJEMPLO

Analicemos el siguiente bloque de código para apreciar cómo es el pasaje de la información a través de los argumentos:

```
# ------------------------------------------------------------
# DEFINICIÓN DE FUNCIONES
# ------------------------------------------------------------

fun <- function(x, y) {
    x <- x + 1
    y <- y * 2
    return(x + y)
}

# ------------------------------------------------------------
# PROGRAMA
# ------------------------------------------------------------

a <- 3
b <- 5
d <- fun(a, b)
cat(a, b, d)
```

```
3 5 14
```

```
cat(x, y)
```

```
Error in cat(x, y): object 'x' not found
```

Si el pasaje de argumentos se hace por valor, los cambios producidos en el cuerpo de la función sobre los parámetros formales no son transmitidos a los parámetros actuales en el *Global Environment*. Esto significa que los formales son una "copia" de los actuales. Los pasos que se siguen:

1. En el *Global Environment*, se asignan los valores: `a = 3`, `b = 5`.
2. Al invocar la función, se establece la correspondencia entre `a` y `x` (que recibe el valor 3) y entre `b` e `y` que recibe el valor 5.
3. Dentro del cuerpo de la función, se ejecuta su primera línea, resultando en la siguiente asignación de valor: `x = 3 + 1 = 4`.
4. Segunda línea de la función: `y = 5 * 2 = 10`.
5. La función devuelve el valor `x + y = 4 + 10 = 14`.
6. El valor 14 es asignado a la variable `d` del *Global Environment*.
7. Se escribe: `3 5 14`.
8. Se produce un error porque en el *Global Environment* no han sido definidos objetos con el nombre `x` o `y`.

15.2 Ambiente global

El ejemplo anterior nos permite darnos cuenta que los objetos que definimos pueden "vivir" en distintos lugares… es decir, existen distintos *entornos*, *ambientes* o *environments*.

En Sección 3.5 presentamos al *Global Environment*, que es el espacio de trabajo principal donde se almacenan los objetos creados durante una sesión de R. Es el nivel más alto en la jerarquía de entornos y es donde se guardan variables, funciones y otros elementos definidos por el usuario. A los objetos definidos en este ambiente se les suele decir **variables globales**.

15.3 Ambiente local de una función

El **ambiente o entorno local** para una función en particular es el espacio donde existen las variables creadas en su cuerpo. Este ambiente es **temporal** y solo es accesible mientras la función está en ejecución. El entorno local se crea cuando se invoca a la función y en él se almacenan los valores de los argumentos de la función y las variables definidas dentro de la misma. Una vez que la función termina su ejecución, **el ambiente local desaparece**, y todas las variables dentro de él se eliminan automáticamente (a menos que sean devueltas como resultado mediante `return()`).

A los objetos definidos en este ambiente se les suele decir **variables locales**. El uso de *variables locales* tiene muchas ventajas. Permiten independizar el trabajo que realiza una función de las instrucciones de programación escritas en el script que la invoca, ya que las variables definidas localmente en una función no son reconocidas fuera de ella. La comunicación entre el ambiente global y el ambiente local de la función debe establecerse a través de la lista de parámetros y del objeto devuelto por la función. Esta característica hace posible dividir grandes trabajos de programación en piezas más pequeñas y que, por ejemplo, diferentes programadores puedan trabajar independientemente en un mismo proyecto, al encargarse del desarrollo de algunas de las funciones.

15.4 Variables locales *vs* variables globales

Las variables locales sólo pueden ser usadas por las instrucciones que están dentro de esa función, mientras que el *Global Environment* desconoce su existencia. Las variables locales a una función no tienen nada que ver con las variables que puedan ser declaradas con el mismo nombre en otros ambientes, ya sea el global o ambientes locales de otras funciones.

EJEMPLO

Decimos que `z` es una variable global porque ha sido definida por el programa en el *Global Environment*. Por otro lado, las variables `a` y `b` son locales a la función `f1` y no se pueden usar desde el *Global Environment*, porque dejan de existir una vez que termina la ejecución de `f1`. El error se genera porque el programa quiere usar a la variable `a`, que no existe en el *Global Environment*.

```
# ----------------------------------------------------------------
# DEFINICIÓN DE FUNCIONES
# ----------------------------------------------------------------

f1 <- function(x) {
    a <- x - 10
    b <- x + 10
    return(a + b)
}

# ----------------------------------------------------------------
# PROGRAMA
# ----------------------------------------------------------------

z <- f1(50)
z
```

```
[1] 100
```

```
z + a
```

```
Error: object 'a' not found
```

Una variable local no puede ser usada en el *Global Environment*, pero una variable global sí puede ser usada en el ambiente local de una función.

EJEMPLO

En el siguiente caso, la función `f2` puede hacer uso de la variable global `y` sin habérsela compartido a través de los argumentos. Cuando en el cuerpo de una función se quiere hacer uso de una variable (`y`), R primero la busca en el ambiente local. Si existe allí, opera con el valor que tiene almacenado. Si no existe, en lugar de producir un error, la busca en un ambiente superior, desde el cual la función fue invocada, en este caso, el *Global Environment*. Si existe allí, opera con el valor que tiene almacenado. Si no existe, R produce un error.

```
# ------------------------------------------------------------------
# DEFINICIÓN DE FUNCIONES
# ------------------------------------------------------------------

f2 <- function(x) {
    a <- x * y
    return(a)
}

# ------------------------------------------------------------------
# PROGRAMA
# ------------------------------------------------------------------

y <- 20
f2(2)
```

```
[1] 40
```

```
y <- 18
f2(2)
```

```
[1] 36
```

INFO IMPORANTE

La práctica anterior no es recomendable: si bien evaluemos `f2(2)` dos veces, el resultado no fue el mismo, porque depende de cuánto vale `y` en el ambiente global en el momento que `f2` es invocada. Además de ser confuso, esto es una violación al principio de **transparencia referencial**: una función idealmente sólo debe utilizar objetos mencionados en la lista de argumentos o definidos localmente, sin emplear variables globales. En particular, si hablamos de una función donde el pasaje de parámetros es por valor, esta práctica garantiza que la misma siempre devuelva el mismo resultado cada vez que sea invocada con los mismos valores en los argumentos de entrada, sin producir ningún efecto secundario en el *Global Environment*. El uso de variables globales dentro de los ambientes locales de las funciones permite escribir programas que carecen de transparencia referencial.

Se puede usar el mismo nombre para variables locales y globales, pero dentro del ambiente local de una función toma precedencia la variable local. Esto se conoce como **name masking**, porque la variable definida dentro de la función "enmascara" nombres definidos fuera de ella.

EJEMPLO

En el siguiente caso hay una variable global `a` en el *Global Environment* que recibe el valor `70` y otra variable `a` que es local a la función `f3`. Cuando `f3` calcula `a + b`, lo hace con el valor de su variable local (`x - 10`) y no con el valor de la variable global (`70`):

```
# ------------------------------------------------------------
# DEFINICIÓN DE FUNCIONES
# ------------------------------------------------------------

f3 <- function(x) {
    a <- x - 10
    b <- x + 10
    cat("Acá, dentro de la f3, el valor de a es", a)
    return(a + b)
}

# ------------------------------------------------------------
# PROGRAMA
# ------------------------------------------------------------

a <- 70
z <- f3(50)
z
cat("Acá, en el programa principal, el valor de a es", a)
a + z
```

```
Acá, dentro de la f3, el valor de a es 40[1] 100
Acá, en el programa principal, el valor de a es 70[1] 170
```

Se debe prestar atención que con la función `cat()` en R se muestra en pantalla un mensaje en el momento en el que se ejecuta esa acción. Si el mensaje incluye mostrar valores guardados en objetos, se mostrarán los valores que los mismos tienen dentro del ambiente que está activo en ese momento. Por otro lado, lo devuelto por `return()` es el resultado de la ejecución de la función: el valor que la función entrega puede ser asignado a otro objeto en el *Global Environment*, como ocurre en la línea de `z <- f3(50)`.

PARA RESOLVER

Dado el siguiente código, determinar, sin ejecutarlo en R, el resultado devuelto por cada llamada a la función `calcular()` presentada debajo.

```r
calcular <- function(a, b = 50) {
  x <- a + b
  return(x)
}

x <- 10
b <- 100
```

- `calcular(5, 3)`:
- (A) 8
- (B) 14
- (C) 52
- (D) 60
- (E) 110
- (F) 150
- (G) 200
- (H) Error
- `calcular(2)`:
- (A) 8
- (B) 14
- (C) 52
- (D) 60
- (E) 110

- (F) 150
- (G) 200
- (H) Error
- `calcular(4, x)`:
- (A) 8
- (B) 14
- (C) 52
- (D) 60
- (E) 110
- (F) 150
- (G) 200
- (H) Error
- `calcular(b, x)`:
- (A) 8
- (B) 14
- (C) 52
- (D) 60
- (E) 110
- (F) 150

- (G) 200
- (H) Error
- `calcular(a = x)`:
- (A) 8
- (B) 14
- (C) 52
- (D) 60
- (E) 110
- (F) 150
- (G) 200
- (H) Error
- `calcular(a = b)`:
- (A) 8
- (B) 14
- (C) 52
- (D) 60
- (E) 110
- (F) 150
- (G) 200

- (H) Error
- `calcular(a = b, b = b)`:
- (A) 8
- (B) 14
- (C) 52
- (D) 60
- (E) 110
- (F) 150
- (G) 200
- (H) Error
- `calcular(b = x)`:
- (A) 8
- (B) 14
- (C) 52
- (D) 60
- (E) 110
- (F) 150
- (G) 200
- (H) Error

Capítulo 16

Más allá de la definición de funciones

RESUMEN

En este capítulo profundizamos en aspectos esenciales para escribir funciones claras y comprensibles en R, que no fallen… o que avisen cuando lo hacen. Aprenderemos a utilizar el objeto especial `NULL` para controlar el comportamiento de nuestras funciones, a gestionar errores y mensajes mediante el uso de `stop()`, `warning()` y `message()` para comunicar problemas o información útil al usuario, y a documentar las funciones correctamente usando el sistema **Roxygen**. Estas herramientas nos permitirán crear funciones más claras, seguras y fáciles de reutilizar.

16.1 El objeto `NULL`

Generalmente los lenguajes de programación poseen un elemento conocido como NULO, para representar un objeto vacío, sin información. En R, `NULL` representa la **ausencia total de un objeto** o valor. Es un objeto en sí mismo y no pertenece a ningún tipo de objeto básico (como numérico, lógico o carácter). Se usa para indicar que una variable o un elemento de una estructura de datos no existe. Suele ser usado como el objeto devuelto por funciones cuando no hay un resultado válido para retornar.

EJEMPLO

Definimos una función para calcular el perímetro de un cuadrado en base a la longitud de uno de sus lados. Este cálculo sólo tiene sentido si el argumento `lado` es un valor positivo. Si no lo es, la función devuelve `NULL`.

```
perimetro_cuadrado <- function(lado) {
  if (lado > 0) {
    return(lado * 4)
  } else {
    return(NULL)
  }
}

perimetro_cuadrado(10)
```

```
[1] 40
```

```
perimetro_cuadrado(-2)
```

```
NULL
```

```
# podemos guardar el resultado en una nueva variable
x <- perimetro_cuadrado(-2)
typeof(x)
```

```
[1] "NULL"
```

```
is.numeric(x)
```

```
[1] FALSE
```

```
is.null(x)
```

```
[1] TRUE
```

Hay funciones que devuelven el objeto `NULL` de forma invisible. Esto quiere decir que, si bien lo devuelven, no se imprime en la consola. Este es el caso de la función `cat()` que usamos para construir mensajes:

```
nombre <- "Andrea"

# Esribe un mensaje, aparentemente no devuelve nada...
cat("Hola,", nombre)
```

```
Hola, Andrea
```

```
# Asignamos su resultado a una variable:
resultado <- cat("Hola,", nombre)
```

```
Hola, Andrea
```

```
# Imprimimos en la consola y nos encontramos que cat() devuelve NULL,
# pero de forma invisible
resultado
```

```
NULL
```

Entonces si definimos una función con el objetivo de generar un mensaje, podemos prescindir del uso de `return()` y la función devolverá de forma invisible el objeto `NULL`, aunque no lo notemos ni nos interese usarlo:

```
saludar <- function(nombre) {
  cat("¡Hola, ", nombre, "! ¿En qué puedo ayudarte hoy?", sep = "")
}

saludar("Andrea")
```

```
¡Hola, Andrea! ¿En qué puedo ayudarte hoy?
```

```
saludar("Gonzalo")
```

```
¡Hola, Gonzalo! ¿En qué puedo ayudarte hoy?
```

```
saludar("Lucía")
```

```
¡Hola, Lucía! ¿En qué puedo ayudarte hoy?
```

COMENTARIO ADICIONAL

La mayoría de las funciones devuelven valores de forma **visible**: si se ejecutan en un entorno interactivo como la consola de R, el resultado se muestra automáticamente en pantalla. Este es el comportamiento por defecto de las funciones que escribimos. Recordemos la función `f`:

```
f <- function(x, y) {
    resultado <- x^2 + 3 * y
    return(resultado)
}

f(4, 5)
```

```
[1] 31
```

Podemos "invisibilizar" el resultado devuelto por una función, así:

```
f_invisible <- function(x, y) {
    resultado <- x^2 + 3 * y
    return(invisible(resultado))
}

f_invisible(4, 5)
```

La función devuelve un resultado, pero no se ve en la consola. Para usarlo o verlo, debemos guardarlo en una nueva variable:

```
resultado <- f_invisible(4, 5)
resultado
```

```
[1] 31
```

Terminamos esta sección mencionando que en R, existen otros valores especiales que representan diferentes tipos de información ausente, indefinida o nula. Aunque parezcan similares, tienen diferencias fundamentales en cuanto a su significado, uso y comportamiento en operaciones y no deben confundirse con el objeto `NULL`:

- `NA` son las siglas de *Not Available* y es un tipo especial valor lógico que generalmente representa datos faltantes o desconocidos. No es un objeto en sí mismo. Propaga su presencia en operaciones matemáticas y lógicas, ya que cualquier operación con `NA` generalmente devuelve `NA`.

  ```
  y <- 100
  z <- NA
  y + z
  ```

  ```
  [1] NA
  ```

- `NaN` son las siglas de *Not a Number* y es un valor numérico que generalmente surge como resultado de operaciones aritméticas imposibles de calcular, como indeterminaciones, raíces negativas, etc.

```
0 / 0
```

```
[1] NaN
```

```
log(-1)
```

```
Warning in log(-1): NaNs produced
```

```
[1] NaN
```

```
sqrt(-1)
```

```
Warning in sqrt(-1): NaNs produced
```

```
[1] NaN
```

16.2 Manejo de errores y mensajes

Ya hemos visto en varias ocasiones que cuando no usamos las funciones de R como deberíamos, obtenemos un mensaje de error. Las funciones que creamos nosotros también pueden contar con esta característica. Si la función no puede completar su tarea, debe lanzar un error utilizando `stop()`, que interrumpe inmediatamente su ejecución, o bien emitir un mensaje o advertencia.

INFO IMPORANTE

Los **mecanismos de manejo de errores, advertencias y mensajes** nos permiten:

- Detectar y comunicar problemas de manera clara al usuario.
- Evitar que el programa continúe ejecutándose con resultados incorrectos.
- Manejar el error sin interrumpir el flujo de ejecución general (no lo veremos en este material).

R proporciona varias herramientas para estos fines, siendo las más comunes `stop()`, `warning()` y `message()`.

16.2.1 `stop()`: para errores críticos

La función `stop()` se usa para **detener inmediatamente la ejecución** de una función cuando ocurre una situación que impide que pueda continuar correctamente. El texto proporcionado como argumento se muestra al usuario como un error.

EJEMPLO

Controlamos que el argumento `nombre` sea de tipo carácter para poder emitir un saludo de manera adecuada:

```
saludar <- function(nombre) {
  if (!is.character(nombre)) {
    stop("Debe proveer una cadena de texto con el nombre de la persona.")
  }
  cat("¡Hola, ", nombre, "! ¿En qué puedo ayudarte hoy?", sep = "")
}

saludar("Eli")
```

```
¡Hola, Eli! ¿En qué puedo ayudarte hoy?
```

```
saludar(100)
```

```
Error in saludar(100): Debe proveer una cadena de texto con el nombre de la persona.
```

16.2.2 `warning()`: para advertencias no fatales

La función `warning()` se utiliza cuando hay algo que podría estar mal, pero no impide continuar con la ejecución. La función sigue adelante, pero el usuario recibe una advertencia.

EJEMPLO

Verificamos que el argumento `b` que seré el divisor en la cuenta no sea igual a cero.

```
division <- function(a, b) {
  if (b == 0) {
    warning("El divisor es 0. El resultado es una indefinición.")
  }
  return(a / b)
}

division(10, 2)
```

```
[1] 5
```

```
division(10, 0)
```

```
Warning in division(10, 0): El divisor es 0. El resultado es una indefinición.
```

```
[1] Inf
```

16.2.3 `message()`: para informar sin interrumpir

Cuando se quiere **comunicar algo al usuario sin que se considere un error o advertencia**, se puede usar `message()`. Es útil para brindar información adicional, como el progreso de una operación.

EJEMPLO

```
cuadrado <- function(x) {
  if (!is.numeric(x)) {
    stop("x debe ser un valor numérico.")
  }
  message("Calculando el cuadrado del número...")
  resultado <- x^2
  message("Cálculo finalizado.")
  return(resultado)
}

cuadrado(4)
```

```
Calculando el cuadrado del número...
```

```
Cálculo finalizado.
```

```
[1] 16
```

INFO IMPORANTE

Las siguientes son algunas buenas prácticas al manejar errores:

- Informar claramente qué salió mal y, si es posible, cómo corregirlo.
- Validar los argumentos de entrada antes de realizar operaciones.
- Usar `stop()` para errores que impiden continuar y `warning()` para situaciones potencialmente problemáticas pero no fatales.
- Recordar que una buena función no solo produce un resultado correcto, sino que también falla de manera informativa cuando algo no está bien.

16.3 Documentación de las funciones

En el contexto de la programación, documentar significa escribir indicaciones para que otras personas puedan entender lo que queremos hacer en nuestro código o para que sepan cómo usar nuestras funciones. Como vimos en Sección 2.6, todas las funciones predefinidas de R están documentadas para que podamos buscar orientación sobre su uso en el panel de ayuda si lo necesitamos. Cuando estamos creando nuestras propias funciones, es importante que también incluyamos comentarios para guiar a otras personas (y a nosotros mismos en el futuro, si nos olvidamos) para qué y cómo se usa lo que estamos desarrollando.

Estas aclaraciones pueden incluirse antes de la definición de la función mediante líneas comentadas con `#` o podemos hacerlo siguiendo estándares ya establecidos por la comunidad de desarrolladores. Si lo hacemos, gozaremos de la ventaja de que las páginas de ayuda sobre nuestras funciones se puedan generar automáticamente cuando las incluimos en la creación de nuevo paquete de R, como veremos en la última unidad de la asignatura.

RStudio ofrece ayuda para escribir la documentación de una función bajo el sistema **Roxygen**, que provee pautas para escribir comentarios con un formato especial, incluyendo toda la información requerida para describir qué hace una función justo antes de su definición. Podemos usar este sistema para desarrollar la costumbre de escribir la documentación al mismo tiempo que creamos la función, sin que se vuelva una carga pesada para más adelante.

Para ejemplificar, retomemos la función que escribimos para calcular factoriales. Ya que aprendimos a originar errores, le agregamos la verificación para el argumento `n`:

```
fact <- function(n) {
  if (n < 0 || n != floor(n)) {
    stop("n debe ser entero no negativo.")
  }
    resultado <- 1
    if (n > 0) {
        for (i in 1:n) {
            resultado <- resultado * i
```

```
        }
    }
    return(resultado)
}
```

Al trabajar en el editor de scripts y con el cursor posicionado dentro del cuerpo de la función, vamos al menú `Code` y elegimos la opción `Insert Roxygen Skeleton`. Por encima de la función se incluirá un "esqueleto" o "plantilla" para que podamos comenzar a escribir la documentación:

```
#' Title
#'
#' @param n
#'
#' @return
#' @export
#'
#' @examples
fact <- function(n) {
  if (n < 0 || n != floor(n)) {
    stop("n debe ser entero no negativo.")
  }
  resultado <- 1
  if (n > 0) {
    for (i in 1:n) {
      resultado <- resultado * i
    }
  }
  return(resultado)
}
```

Presentamos algunas pautas generales para entender la estructura de los comentarios *Roxygen*:

- Un bloque **Roxygen** es una secuencia de líneas que comienzan con `#'` (opcionalmente precedido por un espacio en blanco).
- La primera línea es el título de la función, que no tiene que coincidir con su nombre. En este caso, podemos poner: "Cálculo de factoriales".
- Luego se especifican los distintos campos de la documentación, haciendo uso de *etiquetas* (*tags*) que comienzan con `@`, aparecen al principio de una línea y su contenido se extiende hasta el inicio de la siguiente etiqueta o el final del bloque. Sirven para señalar qué tipo de información vamos a escribir (por ejemplo, qué hace cada argumento, qué devuelve la función, etc.). Algunas de las etiquetas más importantes a incluir son:
  - `@description`: es lo que aparece primero en la documentación y debe describir brevemente qué hace la función.

- `@details`: esta sección proporciona cualquier otro detalle importante sobre el funcionamiento de la función.
- `@param`: se detalla para qué sirve cada parámetro de la función. Debe proporcionar un resumen conciso del tipo de parámetro (por ejemplo, es un `character` o un `numeric`). Es una oración, por lo que debe comenzar con mayúscula y terminar con punto. Puede abarcar varias líneas (o incluso párrafos) si es necesario. Todos los parámetros deben estar documentados, cada uno con su propia etiqueta. RStudio automáticamente incluye tanta etiquetas como parámetros formales hayamos definido.
- `@return`: explica qué objeto devuelve la función.
- `@examples`: incluye ejemplos del uso de la función.
- En el esqueleto se incluye también la etiqueta `@export`, que sólo es relevante en el contexto del desarrollo de nuevos paquetes, por lo cual por ahora la eliminamos.

Teniendo en cuenta lo anterior, completamos la documentación para nuestra función:

```
#' Cálculo de factoriales
#'
#' @description
#' Calcula el factorial de números enteros no negativos.
#'
#' @details
#' Produce un error si se quiere calcular el factorial de un número negativo.
#'
#' @param n Número entero no negativo para el cual se calcula el factorial.
#'
#' @return El factorial de n.
#'
#' @examples
#' fact(5)
#' fact(0)
#'
fact <- function(n) {
  if (n < 0 || n != floor(n)) {
    stop("n debe ser entero no negativo.")
  }
  resultado <- 1
  if (n > 0) {
    for (i in 1:n) {
      resultado <- resultado * i
    }
  }
  return(resultado)
}
```

Capítulo 17

Práctica de la Unidad 3

INFO IMPORANTE

Instrucciones generales para resolver los problemas de esta práctica:

1. Abrir RStudio y crear un nuevo proyecto llamado `unidad3`, para guardar allí todos los archivos que usaremos. Asegurarse de que RStudio esté trabajando con este proyecto abierto.
2. Al comenzar a resolver cada ejercicio:

a. Eliminar todos los objetos del *Global Environment*, para evitar confusiones con objetos que hayan sido creados para resolver otro problema.
b. Crear y guardar en la carpeta del proyecto un nuevo script con el nombre `ejercicio_*.R` para almacenar de manera organizada la solución de cada problema (por ejemplo, `ejercicio_01.R`, `ejercicio_02.R`, etc.)
c. A menos que se indique lo contrario, utilizar cada uno de estos scripts para escribir el código que crea la función pedida en el ejercicio y también el código con ejemplos para usarla.

17.1 Ejercicio 1

a. Definir una nueva función `f1(x1, x2, x3)` que calcule y devuelva la siguiente expresión matemática:

$$\frac{x_1}{x_2} + x_3^2 + x_2 * x_3$$

Ejemplo de su uso:

```
> f1(5, 2, 3)
[1] 17.5
```

b. Modificar el código de `f1` para crear una función `f2(x1, x2, x3)` que realiza el mismo cálculo, pero asumiendo que los argumentos `x2` y `x3` son opcionales. Si el usuario de la función no provee un valor para ellos, deben tomar el valor `1`. Chequear que el resultado coincide con los siguientes ejemplos y analizar por qué se originan:

```
> f2(5, 2, 3)
[1] 17.5

> f2(5)
[1] 7

> f2(5, 2)
[1] 5.5

> f2(5, x3 = 3)
[1] 17

> f2(x2 = 2, x3 = 3)
Error in f2(x2 = 2, x3 = 3) : argument "x1" is missing, with no default
```

c. Modificar el código de `f2` para crear una función `f3(x1, x2, x3)` que realiza el mismo cálculo, con los mismos valores por defecto para `x2` y `x3`, pero que devuelve `-100` si alguno de los argumentos es un valor negativo. Ejemplos de su uso:

```
> f3(5, 2, 3)
[1] 17.5

> f3(-5, 2, 3)
[1] -100

> f3(-5)
[1] -100

> f3(5, x3 = -3)
[1] -100
```

17.2 Ejercicio 2

Dados dos números enteros `a` y `b` que pueden ser negativos o positivos, crear una función llamada `suma_secuencia(a, b)` para calcular la suma de todos los números enteros entre `a` y `b`, incluyéndolos. Si estos números son iguales, la función debe devolver el valor que comparten. Ejemplos de su uso:

```
> suma_secuencia(1, 3)
[1] 6
> suma_secuencia(30, 40)
[1] 385
> suma_secuencia(5, 2)
[1] 14
> suma_secuencia(-2, 3)
[1] 3
> suma_secuencia(-7, -5)
[1] -18
> suma_secuencia(-3, -3)
[1] -3
> suma_secuencia(3, 3)
[1] 3
> suma_secuencia(-3, -5)
[1] -12
```

17.3 Ejercicio 3

Escribir un programa en R para la creación de la función `triangulos(a, b, c)` que a partir de la longitud de los tres lados de un triángulo `a`, `b` y `c` (valores positivos) lo clasifica con los siguientes resultados posibles:

- No forman un triángulo (un lado mayor que la suma de los otros dos).
- Triángulo equilátero (tres lados iguales).
- Triángulo isósceles (dos lados iguales).
- Triángulo escaleno (tres lados distintos).

Como resultado, la función devuelve uno de estos valores de tipo carácter, según corresponda: “no es triángulo”, “equilátero”, “isósceles” o “escaleno”.

Ejemplos de uso:

```
triangulos(2, 3, 4)
[1] "escaleno"
triangulos(2, 3, 10)
[1] "no es triángulo"
```

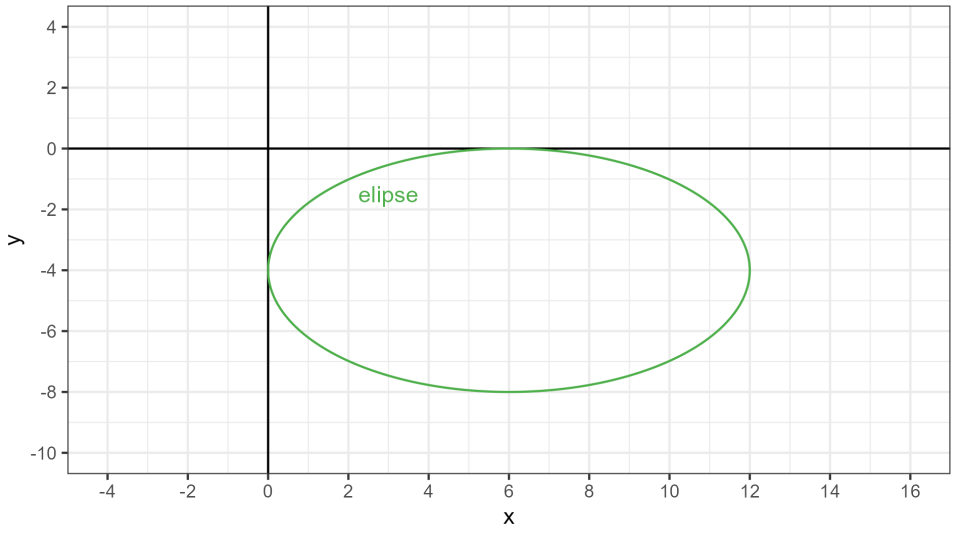
17.4 Ejercicio 4

Escribir un programa en R para la creación de la función `elipse(x, y)` que permite determinar si un punto de coordenadas (x, y) está dentro o no de la elipse definida por la ecuación:

$$\frac{(x-6)^2}{36} + \frac{(y+4)^2}{16} = 1$$

Si el punto está contenido en la elipse, la función devuelve el valor lógico `TRUE` y en caso contrario, `FALSE`. En caso de que sea invocada sin valores para los argumentos `x` e `y`, la función realiza la misma evaluación pero para el origen, es decir, para el punto $(0, 0)$.

Observación: si un punto se encuentra exactamente sobre la curva definida por la elipse, la fórmula anterior evaluada en las coordenadas (x, y) del punto es exactamente igual a 1. Si el punto está dentro de la elipse, da menor que 1. Si está fuera, da mayor que 1. A continuación se presenta la representación gráfica de la elipse en cuestión:



Ejemplos del uso de la función:

```
elipse(3, 7)
[1] FALSE
elipse(6, -4)
[1] TRUE
elipse()
[1] FALSE
```

17.5 Ejercicio 5

Imaginemos que con los números impares podemos crear una pirámide como la que se muestra a continuación:

```
             1
          3     5
       7     9    11
   13    15    17    19
21    23    25    27    29
```

Una pirámide puede tener cualquier cantidad de líneas. Definir una función llamada `suma_piramide(n)` que calcule la suma de los números impares en alguna la fila número `n`. Por ejemplo:

```
> suma_piramide(1)
[1] 1
> suma_piramide(2)
[1] 8
> suma_piramide(3)
[1] 27

# Evaluamos la suma de cada una de las primeras 10 filas
for (n in 1:10) {
   suma <- suma_piramide(n)
   cat("Los impares de la fila", n, "suman", suma, "\n")
}

Los impares de la fila 1 suman 1
Los impares de la fila 2 suman 8
Los impares de la fila 3 suman 27
Los impares de la fila 4 suman 64
Los impares de la fila 5 suman 125
Los impares de la fila 6 suman 216
Los impares de la fila 7 suman 343
Los impares de la fila 8 suman 512
Los impares de la fila 9 suman 729
Los impares de la fila 10 suman 1000
```

17.6 Ejercicio 6

a. Sin utilizar la computadora, indique cuál es el valor devuelto por `g(a, b)` luego de que este programa sea evaluado:

```
f <- function(a = 10) {
a <- (a - 10) * (a + 10)
  return(a)
}

g <- function(x, y) {
```

```
  b <- x - y * 2
  c <- b * f(b)
  d <- f() - c
  return(d)
}

a <- 6
b <- 1
g(a, b)
```

b. Sin utilizar la computadora, indique cuál es el valor de `z` que se muestra el algoritmo y explique por qué se indica que la última línea produce un error:

```
f1 <- function(a, b) {
  x <- a + b
  y <- x + 2
  return(y)
}

f2 <- function(x) {
  return(x^2)
}

# PROGRAMA: Ejemplo de ámbito de las variables
x <- 3
y <- 5
a <- f1(x, y)
z <- x + f2(a)
print(z)
print(a + b) # esta línea produce un error
```

c. Sin usar la computadora, indique cuál es el resultado de evaluar `a + b + c + d` en la última línea del siguiente código:

```
f = function(x, y = 5, z = x + y) {
  u = z - x - y
  return(u)
}

a = f(10)
b = f(10, 10)
c = f(10, 10, 10)
d = f(10, z = 10)

a + b + c + d
```

17.7 Ejercicio 7

Escribir un programa en R para la creación de la función `resolvente(a, b, c)` que muestra las soluciones de la ecuación de segundo grado $ax^2 + bx + c = 0$, empleando la fórmula resolvente:

$$x_{1,2} = \frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$$

Observaciones:

- El programa debe emitir mensajes aclaratorios si hay una solución real doble o dos soluciones complejas (en este caso, no las calcula).
- La función *escribe* en pantalla las soluciones de la ecuación y no es necesario que devuelva ningún objeto en particular.
- Para calcular una raíz cuadrada, podemos usar la función `sqrt()`.
- Si `a` es igual a cero, usar la función `stop()` para devolver un error informativo.

Ejemplos de uso:

```
> resolvente(1, -1, -2)
Hay dos soluciones reales -1 y 2

> resolvente(1, 2, 1)
Hay una solución real doble: -1

> resolvente(1, 1, 1)
Las soluciones son complejas.

> resolvente(0, 1, 1)
Error in resolvente(0, 1, 1) : (a) debe ser distinto de cero
```

17.8 Ejercicio 8

Escribir un programa en R para la creación de la función `es_primo(n)` que devuelve el valor lógico `TRUE` si el natural `n` es un número primo o `FALSE` en caso contrario. Recordar la siguiente definición:

> Un número primo es un número natural mayor que 1, que tiene únicamente dos divisores positivos distintos: él mismo y el 1.

Si el argumento de entrada no es un natural mayor que 1, la función debe imprimir un `warning` y devolver `FALSE` como en estos ejemplos:

```
> es_primo(47)
[1] TRUE

> es_primo(253)
[1] FALSE

> es_primo(2)
[1] TRUE

> es_primo(7.18)
[1] FALSE
Warning message:
In es_primo(7.18) : (n) no es entero

> es_primo(0)
[1] FALSE
Warning message:
In es_primo(0) : (n) no es mayor que 1
```

17.9 Ejercicio 9

Escribir un programa en R para la creación de la función `cociente(dividendo, divisor)` que permite obtener cociente entero y resto en la división de dos números naturales (llamados `dividendo` y `divisor`) empleando únicamente operaciones aritméticas de suma y resta. La función escribe un mensaje en pantalla con los valores del dividendo, divisor, cociente y resto, mientras que devuelve el valor del cociente.

Ejemplos de su uso:

```
cociente(1253, 4)

Dividendo: 1253   # mensajes escritos
Divisor: 4
Cociente: 313
Resto: 1
[1] 313           # valor devuelto

cociente(3, 4)

Dividendo: 3      # mensajes escritos
Divisor: 4
Cociente: 0
Resto: 3
[1] 0             # valor devuelto
```

17.10 Ejercicio 10

Escribir un programa en R para la creación de la función `max_com_div(a, b)` que permite calcular el máximo común divisor (*mcd*) de los números naturales `a` y `b`, empleando el algoritmo de Euclides, que propone:

- Dividir al mayor por el menor.
- Si el resto es cero, el divisor es el máximo común divisor.
- Si el resto no es cero, dividir el divisor por el resto.
- Evaluar el nuevo resto de la misma forma y repetir hasta hallar un resto igual a cero. Cuando esto ocurre, el último divisor es el *mcd*.

Ejemplos de uso:

```
max_com_div(100, 24)
4

max_com_div(25, 100)
25

max_com_div(24, 24)
24
```

17.11 Ejercicio 11

a. Descargar el archivo `funciones_unidad3.R` en el que se encuentra ya definida la función `fact()` tal como se presentó en Sección 16.3. Guardar este archivo en la carpeta del proyecto de esta unidad.

b. Agregar en ese script código para crear una nueva función `combinatorio(m, n)` que calcula el número combinatorio `m` tomado de a `n` (también llamado *coeficiente binomial*), siendo estos números naturales tales que `m >= n`. La función `combinatorio()` debe invocar a la función `fact()` ya provista, teniendo en cuenta que un número combinatorio se define como:

$$C(m,n) = \binom{m}{n} = \frac{m!}{(m-n)!n!}$$

c. Crear otro script llamado `ejercicio_11.R`, en el cual escribiremos ejemplos de uso de la función `combinatorio(m, n)`. En primera instancia, incluir en este script la sentencia `source(funciones_unidad3.R)` para que el contenido del script de funciones sea ejecutado y las mismas sean creadas en el ambiente global. Luego, utilizar la función `combinatorio(m, n)` para ejemplificar las siguientes propiedades de los números combinatorios:

1. $\binom{m}{0} = 1$
2. $\binom{m}{m} = 1$

3. $\binom{m}{1} = m$
4. $\binom{m}{n} = \binom{m}{m-n}$
5. $\binom{m}{n} = \binom{m-1}{n-1} + \binom{m-1}{n}$

d. Es importante recordar que los números combinatorios sólo están definidos para `m >= n`. Probar qué ocurre pasando un valor de `n` mayor que `m`.

e. El número combinatorio `m` tomado de a `n` **con reposición** se define como:

$$\bar{C}(m,n) = \binom{m+n-1}{n} = \frac{(m+n-1)!}{(m-1)!n!}$$

En base a lo realizado anteriormente, crear la función `combinatorio2(m, n, r)` para generalizar el cálculo de números combinatorios, siendo `r` un argumento adicional que toma el valor lógico `TRUE` si el cálculo es con reposición, y `FALSE` en caso contrario.

Consideraciones:

- Usar la función `combinatorio(m, n)` para implementar esta nueva función.
- Por defecto, la función debe hacer el cálculo del número combinatorio sin reposición.
- Para `m = 5` y `n = 2`, pruebe si el número de combinaciones posibles es mayor con o sin resposición.

17.12 Ejercicio 12

Escribir la documentación de cada una de las funciones creadas en esta unidad, siguiendo el formato **Roxygen**.

Capítulo 18

Actividad de autoevaluación 3

INFO IMPORANTE
Esta autoevaluación DEBE ser completada sin usar R para poder razonar las preguntas.

18.1 Pregunta 1

¿Qué problema/s tiene la siguiente definición de una función en R?

```
f0 <- function(a b) {
  x <- a + b
  x * 100
}
```

- (A) Falta una sentencia de tipo return()
- (B) Falta una coma
- (C) Las dos anteriores
- (D) Ningún problema

18.2 Pregunta 2

Considerar la siguiente función:

```r
f1 <- function(x, y) {
  if(y == "operar") {
    x <- x * 2
    rtdo <- x + 100
  } else {
    rtdo <- x
  }
  return(rtdo)
}
```

¿Qué valor se obtiene si se evalúa `f1(2, "operar")`?

- (A) 2
- (B) 4
- (C) 102
- (D) 104

¿Qué valor se obtiene si se evalúa `f1(2, "no operar")`?

- (A) 2
- (B) 4
- (C) 102
- (D) 104

¿Qué valor se obtiene si se evalúa `f1("operar", 2)`?

- (A) 2
- (B) 4
- (C) 102
- (D) 104
- (E) Tira error
- (F) Devuelve una cadena de texto

¿Qué valor se obtiene si se evalúa `f1(2)`?

- (A) 2
- (B) 4
- (C) 102
- (D) 104
- (E) Tira error
- (F) Devuelve una cadena de texto

¿Qué valor se obtiene si se evalúa `f1("2", "operar")`?

- (A) 2
- (B) 4
- (C) 102
- (D) 104
- (E) Tira error
- (F) Devuelve una cadena de texto

¿Qué valor se obtiene si se evalúa `f1("2", "jaja")`?

- (A) 2
- (B) 4
- (C) 102
- (D) 104
- (E) Tira error
- (F) Devuelve una cadena de texto

18.3 Pregunta 3

Considerando la función `f1` anterior junto con la función `f2` y el programa que se muestran a continuación, indicar si cada una de las siguientes afirmaciones es VERDADERA o FALSA o elegir la respuesta correcta:

```r
# Definición de funciones
f1 <- function(x, y) {
  if(y == "operar") {
    x <- x * 2
    rtdo <- x + 100
  } else {
    rtdo <- x
  }
  return(rtdo)
}

f2 <- function(z) {
  return(z + b)
}

# Programa
x <- 10
palabra <- "operar"
a <- f1(x, palabra)
cat("Mensaje 1: el valor encontrado es ", a, "y la variable x vale ", x, "\n")
b <- 12
c <- f2(15)
cat("Mensaje 2: el valor encontrado es ", c, "\n")
d <- f1(5, "no operar") + f1(5, "operar")
cat("Mensaje 3: el valor encontrado es", d, "\n")
```

La variable `d` es local a `f1`:

- (A) Verdadero
- (B) Falso

La variable `c` es global:

- (A) Verdadero
- (B) Falso

La función `f2` respeta el principio de transparencia referencial:

- (A) Verdadero
- (B) Falso

La función `f2` puede devolver distintos resultados aunque le pasemos siempre el mismo valor para el parámetro formal `z`:

- (A) Verdadero
- (B) Falso

La variable `rtdo` existe en el ambiente global del programa:

- (A) Verdadero
- (B) Falso

La variable `rtdo` es local a la función `f1`:

- (A) Verdadero
- (B) Falso

El texto mostrado por el primer mensaje será:

- (A) Mensaje 1: el valor encontrado es 10 y la variable x vale 10
- (B) Mensaje 1: el valor encontrado es 10 y la variable x vale 120
- (C) Mensaje 1: el valor encontrado es 120 y la variable x vale 10
- (D) Mensaje 1: el valor encontrado es 120 y la variable x vale 20

El texto mostrado por el segundo mensaje será:

- (A) Mensaje 2: el valor encontrado es 12
- (B) Mensaje 2: el valor encontrado es 15
- (C) Mensaje 2: el valor encontrado es 27
- (D) No se muestra un mensaje porque se produce error

El texto mostrado por el tercer mensaje será:

- (A) Mensaje 3: el valor encontrado es 5
- (B) Mensaje 3: el valor encontrado es 110
- (C) Mensaje 3: el valor encontrado es 115
- (D) No se muestra un mensaje porque se produce error

18.4 Pregunta 4

Seleccionar la opción correcta:

`warning()`

- (A) Muestra un mensaje que detiene inmediatamente la ejecución de la función.
- (B) Muestra un mensaje que no detiene la ejecución del programa pero advierte al usuario de un posible error.
- (C) Muestra un mensaje en la consola sin que sea considerado un error o advertencia.

`message()`

- (A) Muestra un mensaje que detiene inmediatamente la ejecución de la función.
- (B) Muestra un mensaje que no detiene la ejecución del programa pero advierte al usuario de un posible error.
- (C) Muestra un mensaje en la consola sin que sea considerado un error o advertencia.

`stop()`

- (A) Muestra un mensaje que detiene inmediatamente la ejecución de la función.
- (B) Muestra un mensaje que no detiene la ejecución del programa pero advierte al usuario de un posible error.
- (C) Muestra un mensaje en la consola sin que sea considerado un error o advertencia.

`NULL`, `NA` y `NaN` son tres objetos que tienen comportamientos semejantes en las operaciones.

- (A) Verdadero
- (B) Falso

Unidad 4. Uso de la terminal

RESUMEN

En esta unidad nos adentraremos en el uso de la terminal, una herramienta poderosa que permite interactuar directamente con el sistema operativo a través de comandos de texto. Aprenderemos a utilizarla en el entorno Windows para realizar tareas básicas como crear carpetas o ejecutar programas. También aprenderemos a correr *scripts* de R desde la terminal y a desarrollar programas que interactúen con el usuario, recibiendo información y reaccionando en consecuencia.

Capítulo 19

La terminal

RESUMEN

En este capítulo aprenderemos a usar la **terminal**, una herramienta clave para interactuar con el sistema operativo de manera eficiente. A través de comandos, podemos realizar tareas rápidamente sin depender de la interfaz gráfica. Aprenderemos a abrir una terminal en Windows y a usar los comandos básicos para navegar por directorios, gestionar archivos y realizar tareas comunes de manera más rápida.

19.1 Introducción

Cuando encendemos nuestra computadora, normalmente interactuamos con el sistema operativo a través de una **interfaz gráfica** (GUI), utilizando ventanas, menús, íconos, el mouse, el teclado o incluso pantallas táctiles. Sin embargo, existe otra forma de comunicarse con la computadora: escribiendo comandos en una ventana especial llamada **terminal**. La terminal interpreta estos comandos y los traduce en instrucciones que la computadora puede ejecutar. Aunque hoy en día la mayoría de los usuarios utilizan la interfaz gráfica, años atrás este método era la única forma disponible para operar un sistema.

¿Por qué aprender a usar la terminal si contamos con una interfaz gráfica que parece más sencilla? La razón principal es la **eficiencia**: muchas tareas tediosas y repetitivas se pueden realizar de manera mucho más rápida mediante comandos. Por ejemplo, copiar decenas de archivos que cumplen ciertos criterios de nombre puede hacerse con un solo comando en lugar de seleccionar manualmente uno por uno. Además, muchas herramientas de diagnóstico y administración de sistemas, especialmente en el ámbito de redes y servidores, solo están disponibles a través de la terminal. El uso de la terminal también permite la **automatización** de tareas.

Si bien al principio puede parecer intimidante, no es un mundo completamente desconocido: ya estamos acostumbrados a escribir comandos en la **consola de R**, interpretando respuestas y corrigiendo errores. Aprender a usar la terminal en Windows será un paso natural que te permitirá tener un **mayor control** sobre nuestro entorno de trabajo y nos abrirá nuevas posibilidades para **automatizar y optimizar** tareas cotidianas.

19.2 Conceptos relacionados

En el mundo de la programación y la informática, hay muchos términos relacionados con la interacción textual con la computadora que a veces se usan como sinónimos sin demasiada preocupación, como si fuesen distintas formas de llamar a lo mismo: una ventanita donde puedo escribir comandos y hacer que sucedan cosas en la computadora. Si bien está aceptado usar cualquiera de estas palabras, hay pequeñas diferencias entre los conceptos, que acá tratamos de resumir:

- **Línea de comandos (Command Line Interface, CLI)**. La línea de comandos es una forma de interactuar con el sistema operativo escribiendo comandos, en lugar de usar ventanas y botones (interfaz gráfica o *Graphical User Interface*, GUI). La línea de comandos no es un programa, sino un modo de interacción. Para usarla, necesitamos programas específicos como las terminales y las shells.
- **Shell**. Una shell es un programa que interpreta los comandos que escribimos y los traduce en instrucciones que el sistema operativo puede ejecutar. Cada shell tiene su propio lenguaje de comandos y sus propias capacidades. Algunas de las más comunes son:

Shell	Sistema	Características principales
CMD.exe	Windows	Shell clásica de Windows. Limitada, pero simple.
PowerShell	Windows	Más moderna y poderosa. Permite automatizar tareas.
Bash	Linux, macOS, WSL	Muy usada en sistemas Unix y entornos Linux en Windows. Potente y versátil.
sh, zsh, fish	Unix-like	Otras shells populares, con distintos niveles de funcionalidad y facilidad de uso.

- **Terminal (o emulador de terminal)**. Una terminal (o emulador de terminal) es un programa que permite enviar comandos a una shell y mostrar sus respuestas. Es como una "ventana" desde la cual nos comunicamos con la computadora usando solo texto. Se llama emulador porque hoy en día normalmente se abre desde una interfaz gráfica, simulando las terminales físicas antiguas. La terminal no ejecuta directamente los comandos, sino que los pasa a la shell que esté configurada. Ejemplos de terminales:
  - En Windows: CMD, PowerShell, Windows Terminal
  - En Linux/macOS: GNOME Terminal, Konsole, Terminal.app
- **Consola**. La palabra **consola** tiene varios usos:

  - Históricamente, se refería a un dispositivo físico (un teclado y una pantalla conectados directamente a una computadora o servidor).
  - Actualmente, en el lenguaje cotidiano, muchas veces se usa como sinónimo de terminal.

19.3 Comandos básicos para el uso de la terminal en Windows

En este curso utilizaremos **CMD.exe** para trabajar con la terminal en Windows. CMD es más sencillo y directo que la otra opción disponible (PowerShell) y nos permitirá concentrarnos en aprender los comandos básicos sin distracciones.

Se puede abrir la terminal **CMD.exe** así:

1. Hacer clic en el botón de **Inicio** (el ícono de Windows en la esquina inferior izquierda).
2. Escribir `cmd` en el cuadro de búsqueda.
3. Aparecerá una aplicación llamada **Símbolo del sistema** o **Command Prompt**.
4. Hacer clic en ella para abrirla.

Al igual que R, una terminal siempre tiene un directorio de trabajo, cuya ruta se muestra al comienzo de cada línea, antes del *prompt* `>`, que señala que podemos escribir un comando. También se puede abrir la terminal desde el Explorador de archivos. En este caso, el directorio de trabajo será la carpeta desde la que se abrió la terminal.

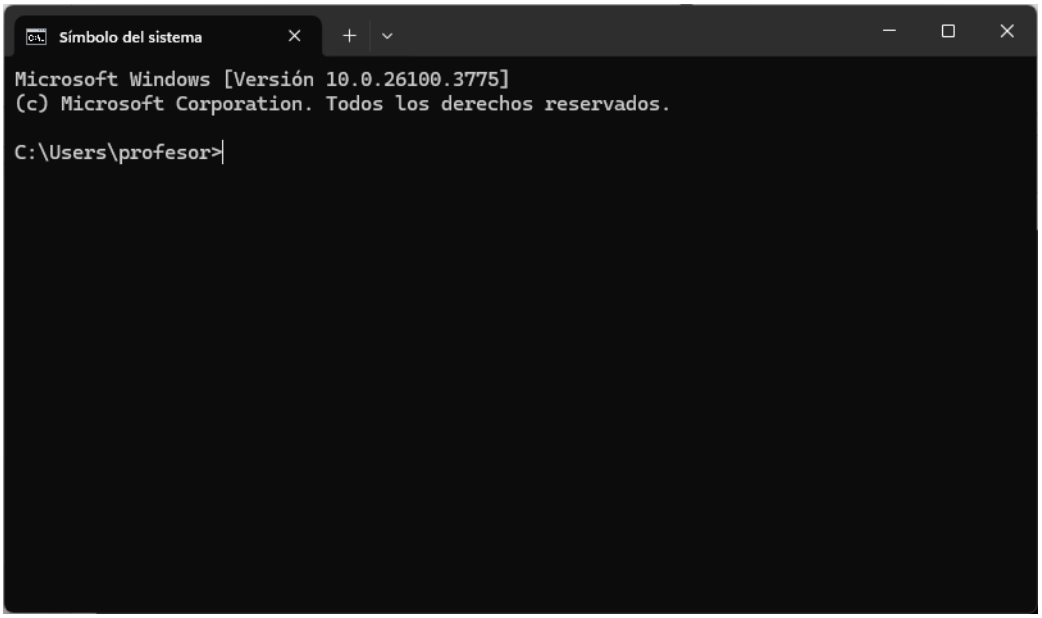


Figura 19.1: Línea de comandos de Windows. El directorio de trabajo es `C:\Users\profesor`.

A continuación, se presentan algunos comandos esenciales que pueden usarse en CMD:

Comando	Descripción
`dir`	Lista los archivos y carpetas en el directorio actual.
`cd`	Muestra el directorio de trabajo actual
`cd nombre_carpeta`	Cambia al directorio especificado.
`cd ..`	Sube un nivel en el árbol de directorios.
`mkdir nombre`	Crea una nueva carpeta.
`del nombre`	Elimina un archivo.
`rmdir nombre`	Elimina una carpeta (debe estar vacía).
`copy archivo1.txt archivo2.txt`	Copia `archivo1.txt` y crea una copia llamada `archivo2.txt`.
`move archivo1.txt carpeta\`	Mueve `archivo1.txt` a una carpeta.
`rename nombre_viejo.txt nombre_nuevo.txt`	Cambia el nombre de un archivo.
`cls`	Limpia la pantalla.
`echo texto`	Imprime “texto” en la terminal.
`echo primer texto > archivo.txt`	Escribe “primer texto” en el archivo `archivo.txt`
`echo segundo texto >> archivo.txt`	Agrega “segundo texto” en el archivo `archivo.txt`, como nueva línea, sin que se borre lo anterior.
`type archivo.txt`	Muestra el contenido de un archivo de texto.
`help`	Ver la ayuda de los comandos.
`exit`	Cerrar la terminal.

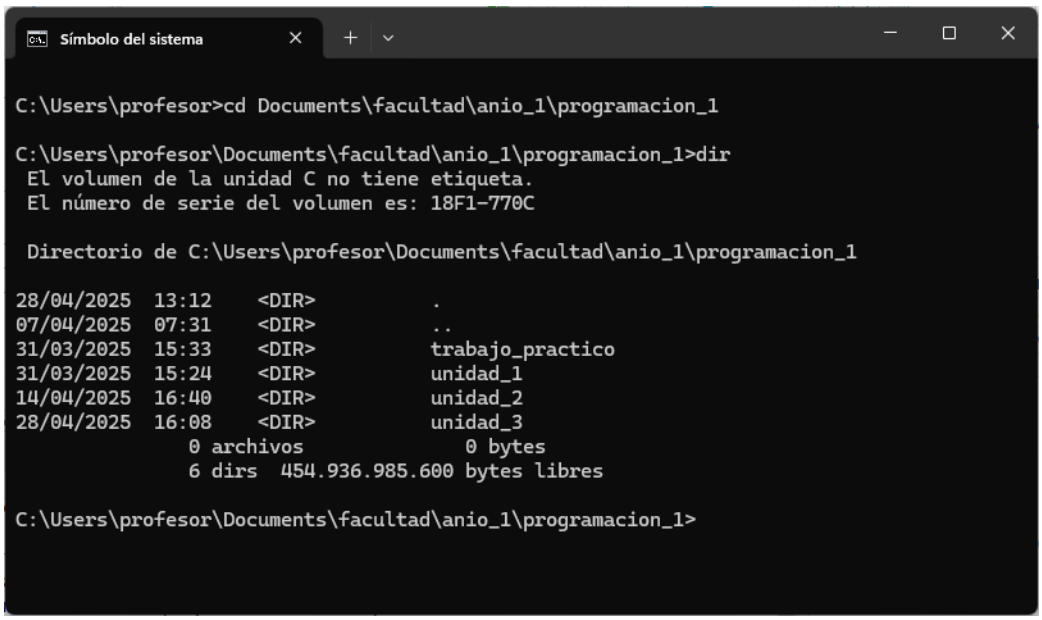


Figura 19.2: Ejemplo del uso de los comandos `cd` y `dir` para cambiar de carpeta y mostrar el contenido.

PARA RESOLVER

Realicemos la siguiente lista de actividades:

1. Ir al directorio de nuestra asignatura (escribí un *path* adecuado para tu computadora:

```
cd Documents\facultad\anio_1\programacion_1
```

2. Listar las carpetas y archivos allí guardados:

```
dir
```

3. Crear una nueva carpeta para la Unidad 4 (no será un RStudio Project porque no es creado desde RStudio, pero no importa):

```
mkdir unidad_4
```

4. Crear un archivo de texto simple, llamado `archivo1.txt` y con el contenido "Este es mi primer archivo">

```
echo Este es mi primer archivo > archivo1.txt
```

5. Copiar el archivo:

```
copy archivo1.txt copia_archivo1.txt
```

   Ahora tenemos dos archivos: `archivo1.txt` y `copia_archivo1.txt`.

6. Renombrar el archivo copiado:

```
rename copia_archivo1.txt archivo2.txt
```

7. Crear una subcarpeta llamada `textos`:

```
mkdir textos
```

8. Mover el segundo archivo a la subcarpeta:

```
move archivo2.txt textos
```

9. Verificar que los archivos se movieron:

```
cd textos
dir
```

10. Mostrar en la terminal el contenido de `archivo2.txt`:

```
type archivo2.txt
```

Capítulo 20

Ejecución de *scripts* de R desde la terminal

RESUMEN

En este capítulo aprenderemos a **ejecutar scripts de R directamente desde la terminal**, una habilidad esencial para automatizar tareas y trabajar de manera más eficiente. Si bien RStudio y otros entornos gráficos son herramientas poderosas, ejecutar scripts desde la terminal ofrece ventajas importantes, como mayor rapidez, control sobre el entorno y la capacidad de automatizar procesos sin depender de interfaces gráficas. Hacer esto es necesario cuando tenemos que programar alguna tarea de gran escala que se ejecutará de manera remota en algún servidor o cuando necesitamos encapsular nuestro programa para que otros lo puedan correr sin siquiera saber nada de R.

20.1 Requisitos

Para poder ejecutar scripts de R desde la terminal de Windows, necesitamos:

1. Editar la variable de entorno `PATH`

Cuando queramos correr un programa de R usaremos el comando `Rscript`. Por default, la terminal lo desconoce, porque se trata de un programa que se agregó en nuestra computadora el día que instalamos R. Por eso necesitamos indicarle al sistema operativo que `Rscript` es un comando que se instaló con R y que lo puede encontrar en la carpeta de los archivos de instalación del programa R. Esto hay que hacerlo una sola vez por computadora editando las **variables de entorno de Windows**, que son cadenas de texto que contienen información acerca del sistema. Se logra siguiendo estos pasos:

1. Fijarse en qué carpeta de la computadora está instalado R. Seguramente lo encuentres si, abriendo el explorador de archivo, vas siguiendo este camino: `Este equipo > Windows (C:) > Archivos de programa > R > R-version > bin`. En esta carpeta tiene que haber dos archivos, llamados `R.exe` y `Rscript.exe`. Si es así, hacé clic con el botón derecho del mouse sobre cualquiera de ellos, luego en "Propiedades" y copiá el *path* que aparece en "Ubicación" (deberías copiar algo como `C:\Program Files\R\R-4.4.3\bin`). También podés copiar el *path* que se ve arriba en la barra de navegación.

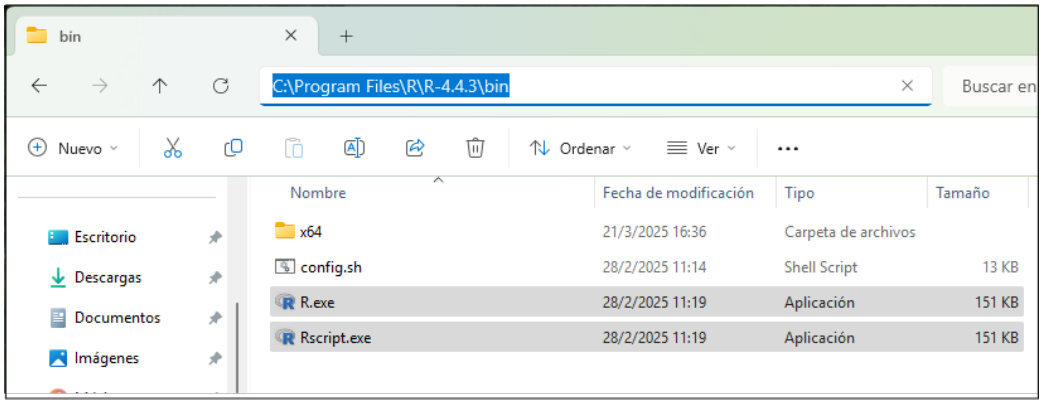


Figura 20.1: Archivos de instalación de R en Windows.

2. Hacer clic en "Inicio" (logo de Windows de la barra de tareas) o presionar la tecla Windows del teclado, escribir "Entorno" y hacer clic en la opción "Editar las variables de entorno del sistema".

3. Hacer clic en el botón "Variables de entorno" (abajo).

4. En el cuadro "Variables del sistema" (abajo), hacer clic en la variable "Path" y luego en "Editar".

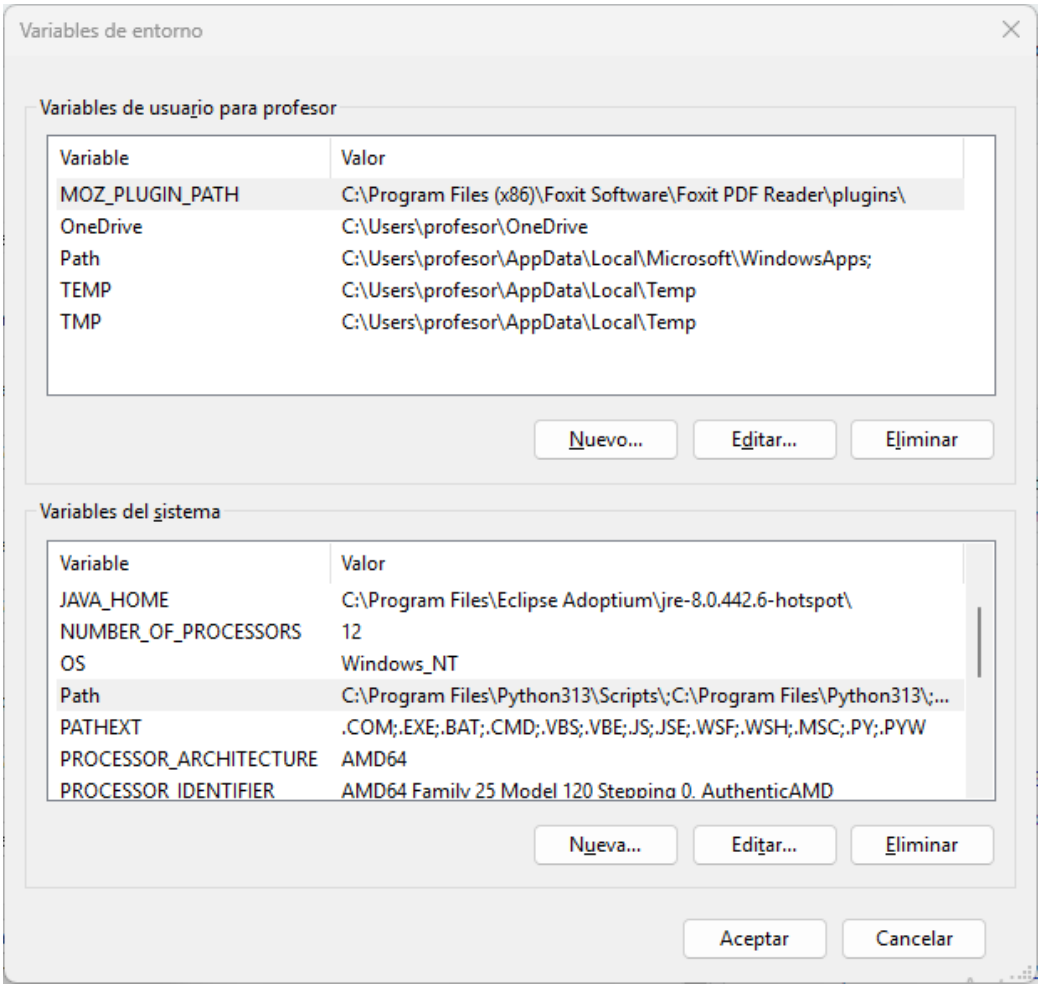


Figura 20.2: Variables de entorno de Windows.

5. Hacer clic en "Nuevo", pegar la dirección copiada antes (por ejemplo, `C:\Program Files\R\R-4.4.3\bin`) y dar `Enter`.

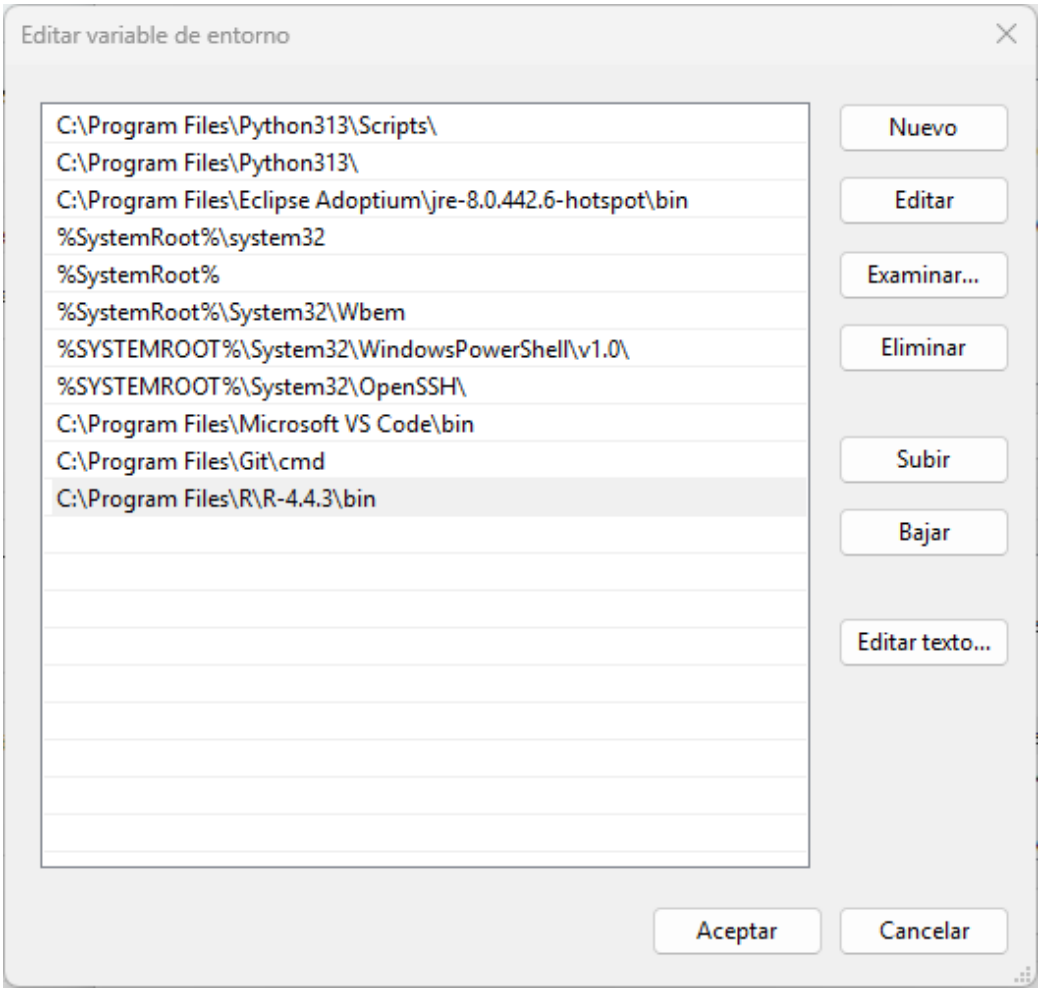


Figura 20.3: Editar variable de entorno.

6. Luego, hacer clic en "Aceptar" tres veces para confirmar y cerrar todo.

2. Abrir la terminal y cambiar el directorio de trabajo

A la hora de ejecutar un script desde la terminal, tenemos que cambiar el directorio de trabajo a la carpeta en la cual está guardado, con el comando `cd`. Luego, ejecutamos el comando `Rscript nombre_archivo.R`. Si no queremos cambiar el directorio de trabajo, debemos escribir el *path* completo hacia el script, pero esto es menos cómodo (por ejemplo: `Rscript C:\Users\Documents\nombre_archivo.R`).

Los scripts utilizados en los ejemplos de esta unidad pueden ser descargados desde este archivo comprimido y ubicados de forma ordenada en algún lugar de la computadora, idealmente en la carpeta destinada para esta unidad, como se muestra en la siguiente imagen.

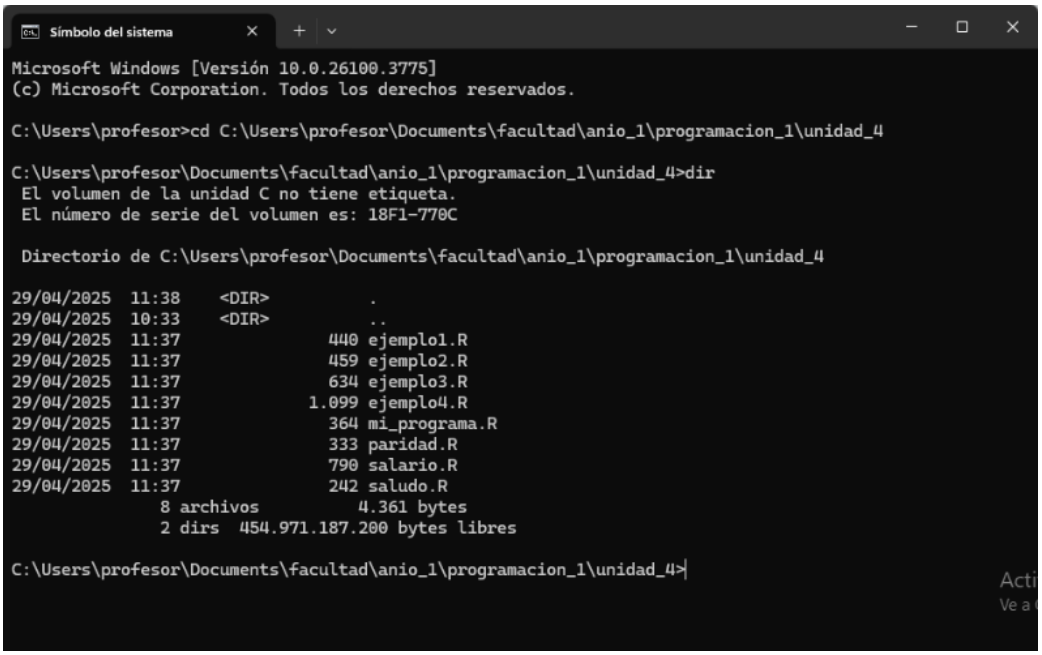


Figura 20.4: Cambio del directorio de trabajo a la carpeta que contiene los scripts de R.

20.2 Ejecución de programas sin interacción del usuario

Empezaremos con el ejemplo propuesto por el script `mi_programa.R` que tiene este contenido:

```
a <- "¡Hola, Mundo!"
b <- 3
d <- 5
cat("=========================================\n")
cat("                 RESULTADOS               \n")
cat("=========================================\n\n")
cat("El valor de b es ", b, ", mientras que d vale ", d, ".\n\n", sep = "")
cat("La suma entre ellos es igual a ", b + d, ".\n\n", sep = "")
cat("Este es un saludo:", a)
```

Todo lo que en el programa está encerrado en una llamada a la función `cat()` es lo que se mostrará como mensajes en la terminal cuando el script sea ejecutado. En una terminal abierta y habiendo cambiado el directorio hacia la carpeta donde está guardado el script, utilizamos el comando `Rscript mi_programa.R` y veremos el siguiente resultado:

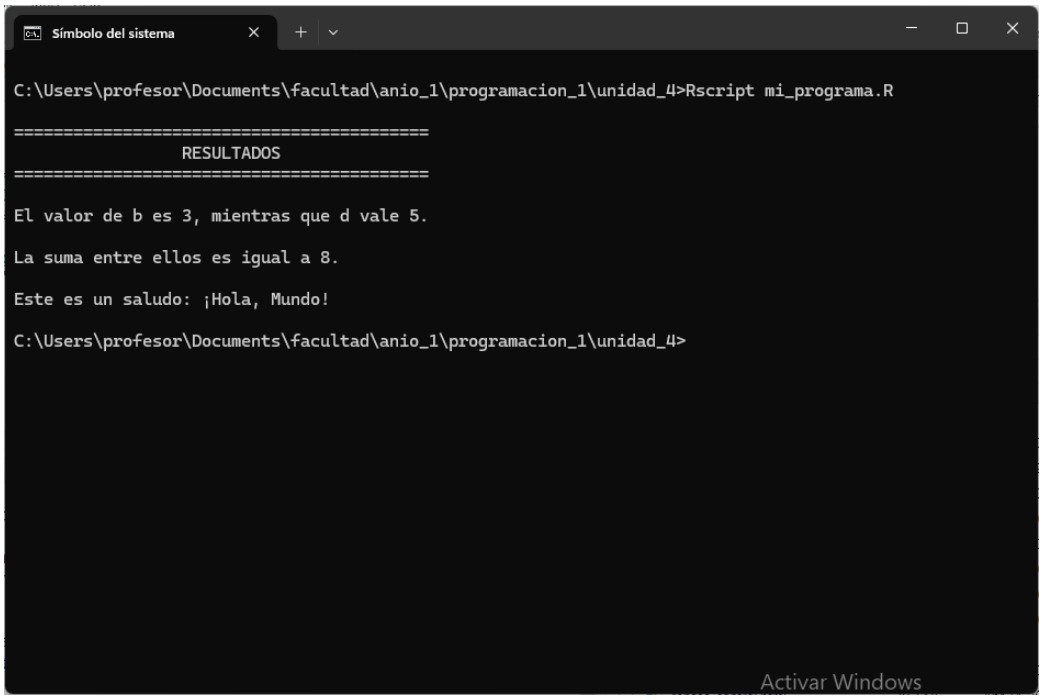


Figura 20.5: Correr el script `mi_programa.R` desde la terminal.

20.3 Ejecución de programas interactivos

Correr scripts desde la terminal nos permite crear programas interactivos, que soliciten información al usuario y actúen en función de esa entrada. Este enfoque es particularmente útil cuando necesitamos que el usuario proporcione datos específicos o haga elecciones durante la ejecución del programa, sin depender de interfaces gráficas complejas. Al usar funciones de R como `scan()`, podemos diseñar scripts que personalicen su comportamiento según las respuestas del usuario.

Por ejemplo, el script `saludo.R` preguntará al usuario "¿Cómo te llamás?" y luego responderá con un saludo. Su contenido es:

```
cat("****************************************\n")
cat("¿Cómo te llamás?\n> ")
nombre <- scan(file = "stdin", what = character(), n = 1, quiet = TRUE)
cat("¡Hola, ", nombre, "!\n", sep = "")
cat("****************************************\n")
```

La función `scan()` es la que permite *escanear* o *leer* valores que los usuarios ingresen por la terminal. Sus argumentos incluyen:

- `file`: especifica desde donde se lee la información. Debemos setearlo como `file = "stdin"` porque vamos a leer información desde la terminal.

- `what`: determina el tipo de dato a leer, generalmente un valor numérico o una cadena de texto. Si queremos leer un número, no es necesario usarlo, ya que por default se tiene `what = double()` y hace eso. En cambio, si queremos leer texto, como en este ejemplo, debemos indicar `what = character()`.
- `n`: cantidad de valores a leer (uno solo en el ejemplo).
- `quiet`: valor lógico que determina si la función opera escribiendo detalles adicionales en la consola (`FALSE`, valor por defecto, no lo desemos) o de forma silenciosa, sin escribir nada adicional (`TRUE`, es lo que queremos).

Cuando ejecutamos este script, vemos:

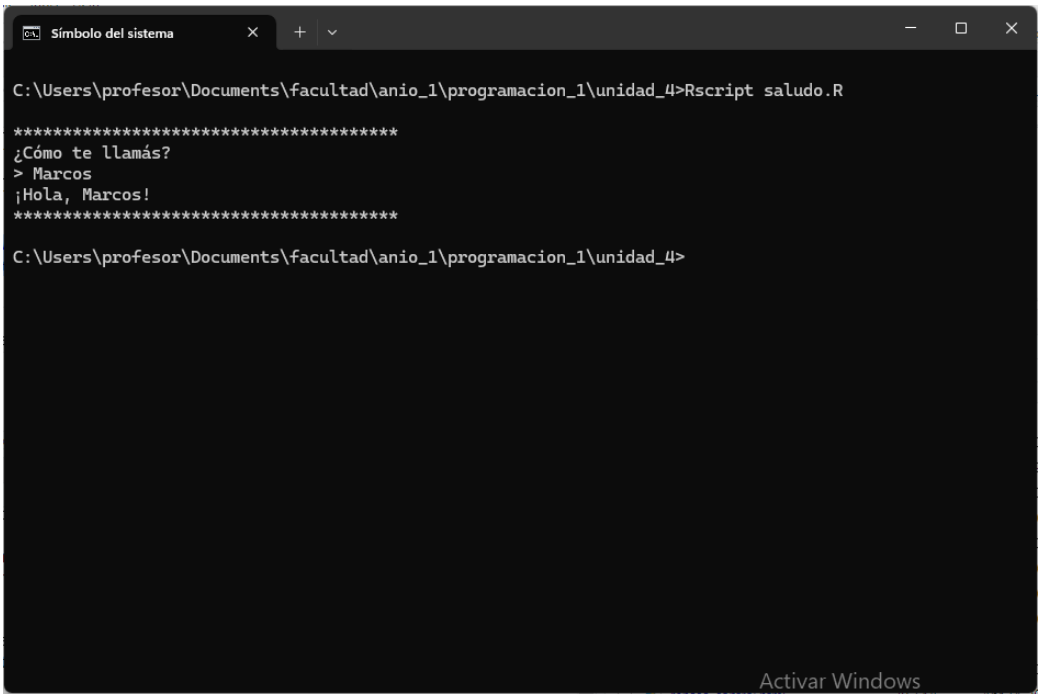


Figura 20.6: Correr el script `saludo.R` desde la terminal.

Pasamos a otro ejemplo. El siguiente script le pide a la persona que lo esté usando que indique cualquier número y luego le comunica si es par o impar:

```
cat("=========================================\n")
cat("           PARIDAD DE UN NÚMERO          \n")
cat("=========================================\n\n")
cat("Ingrese un número entero y presione enter:\n")
n <- scan(file = "stdin", n = 1, quiet = TRUE)
if (n %% 2 == 0) {
  cat(n, "es par\n")
} else {
  cat(n, "es impar\n")
}
```

Esto es lo que ocurre en la terminal:

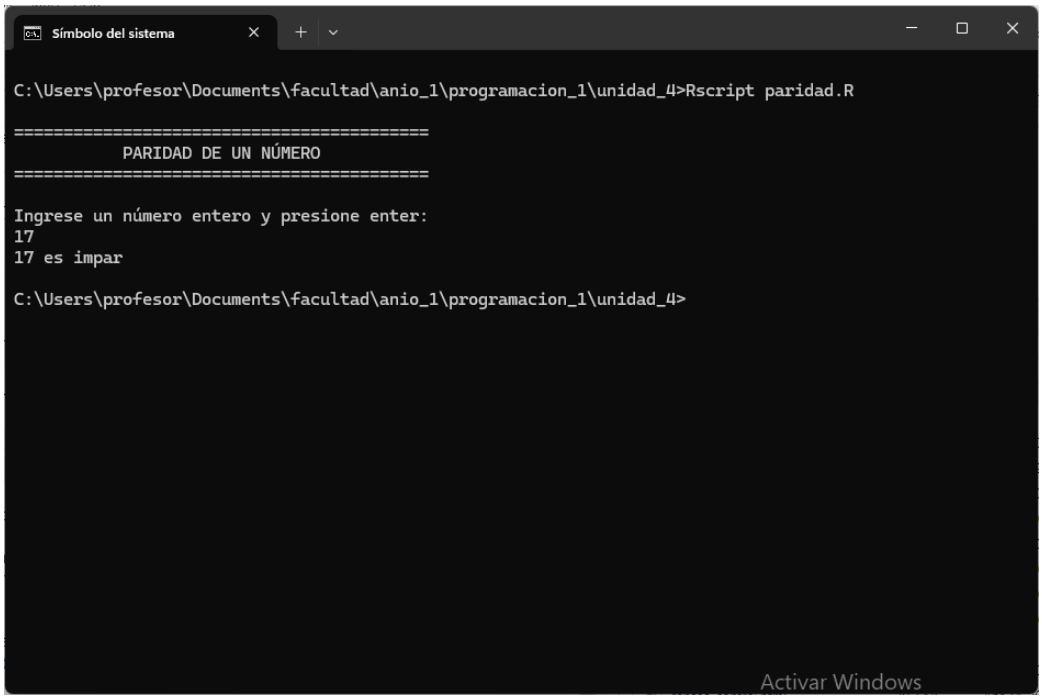


Figura 20.7: Correr el script `paridad.R` desde la terminal.

Ahora recordemos el ejercicio de la práctica 2 en el que escribimos un programa para calcular salarios, en función del día de la semana, el turno y las horas trabajadas. Podemos adaptarlo para esta información de la que depende el cálculo final se tome desde la terminal:

```r
# PROGRAMA: "Determinar salario"

# Valores fijados
valor_hora <- 4000
valor_adicional_noche <- 2000
valor_adicional_domingo <- 1000

# Pedir valores para calcular el salario
cat("Ingrese la cantidad de horas trabajadas:\n")
horas <- scan("stdin", n = 1, quiet = TRUE)
cat("\nIngrese el día de la semana (DOM LUN MAR MIE JUE VIE SAB):\n")
dia <- scan("stdin", what = "", n = 1, quiet = TRUE)
cat("\nIngrese el turno (M T N):\n")
turno <- scan("stdin", what = "", n = 1, quiet = TRUE)

# Cálculo para el pago de ese día a ese empleado
salario <- horas * valor_hora
```

```r
if (turno == "N") {
  salario <- salario + horas * valor_adicional_noche
}
if (turno == "DOM") {
  salario <- salario + horas * valor_adicional_domingo
}
cat("\nEl salario que se debe abonar es $", salario, "\n", sep = "")
```

Ejecutamos el script `salario.R`:

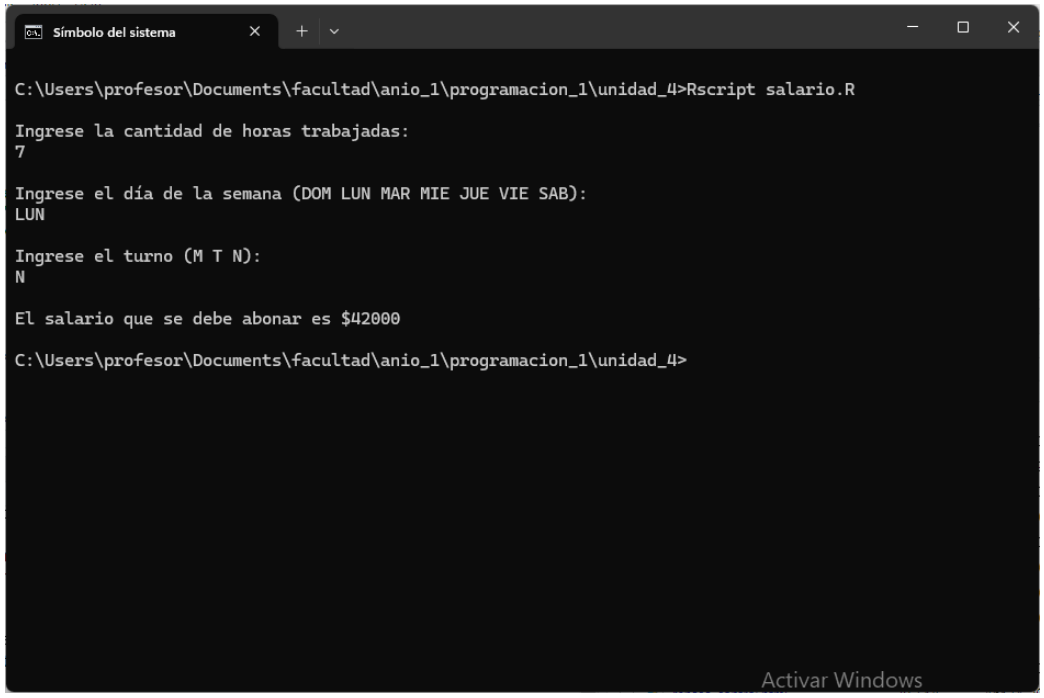


Figura 20.8: Correr el script `salario.R` desde la terminal.

Capítulo 21

Uso de argumentos en la línea de comandos al ejecutar código de R

RESUMEN

En este capítulo **de lectura opcional** exploraremos ejemplos más avanzados de interacción entre R y la terminal, centrándonos en el uso de argumentos que se pasan al ejecutar scripts con `Rscript`. Aprenderemos cómo capturar y procesar estos argumentos con la función `commandArgs()`, cómo validar su cantidad y contenido, y cómo convertirlos a otros tipos de datos cuando sea necesario. También veremos ejemplos prácticos y mencionaremos herramientas adicionales que permiten gestionar opciones de manera más sofisticada.

En ejemplos anteriores hemos visto cómo capturar distintas piezas de información de forma interactiva mediante la función `scan()` mientras estamos ejecutando un programa de R desde la terminal. En otras ocasiones, en lugar de pausar la ejecución del programa a la espera de que el usuario ingrese algún valor, es conveniente especificar algunas opciones directamente en la línea de comando, al lado instrucción `Rscript` que ejecuta el código.

Por ejemplo, imaginemos que tenemos un programa llamado `resumen.R` que se encarga de hacer un análisis descriptivo de un conjunto de datos, guardados en algún archivo cuyo nombre debe especificar el usuario para que el programa lo lea. Supongamos también que el nombre del archivo de datos es `mayo.txt`. El usuario puede mandar a correr el programa que hace el análisis sobre este archivo de datos indicando su nombre como un argumento adicional de esta forma:

```
Rscript resumen.R mayo.txt
```

Ahora pensemos que este mismo tipo de análisis se repite todos los meses con datos nuevos. En lugar de modificar nuestro script `resumen.R` para que lea un archivo con otro nombre, ejecutamos lo anterior con el nombre del archivo que corresponda y listo:

```
Rscript resumen.R junio.txt
Rscript resumen.R julio.txt
Rscript resumen.R agosto.txt
```

Para que esto funcione, el programa que está guardado en `resumen.R` debe ser capaz de capturar el nombre del archivo que tiene leer y que el usuario se lo está pasando como un argumento adicional en la instrucción `Rscript`.

La función que se encarga de capturar los argumentos adicionales que enviamos desde la terminal es `commandArgs()`. Toma todos los elementos que escribamos al final de la línea de `Rscript` y los reúne en un **vector atómico de tipo carácter**.

INFO IMPORANTE

En la Unidad 1 mencionamos que los vectores atómicos pueden contener más de un valor, pero hasta acá no los usamos de esa forma. Lo veremos recién en la próxima unidad. Los ejemplos que siguen hacen uso de vectores con varios elementos, que se corresponden con todos los argumentos que un usuario envía desde la terminal. Por esta razón, **es recomendable que sigas leyendo si ya sabés usar vectores con más de un elemento, o que vuelvas a ver estos ejemplos más adelante, después de que hayas estudiado la unidad 5**.

Los scripts utilizados en los ejemplos pueden ser descargados desde este archivo comprimido.

En primer lugar vamos a analizar al script `ejemplo1.R`, con el siguiente contenido:

```
# Capturar los argumentos pasados desde la terminal en un vector
args <- commandArgs(trailingOnly = TRUE)

# Contar cuántos argumentos nos pasaron
cat("Nos pasaron", length(args), "argumentos.\n\n")

# Mostrar los argumentos que nos pasaron
cat("Los argumentos que nos pasaron son:\n")
cat(args, "\n")

# Aunque los argumentos sean números, son tomados como carácter
cat("\nLos argumentos se toman como valores de tipo:\n")
typeof(args)
```

Vamos a ejecutarlo desde la terminal con los argumentos “hola”, “chau” y “4”. Obtenemos:

```
Rscript ejemplo1.R hola chau 4
```

```
Nos pasaron 3 argumentos.

Los argumentos que nos pasaron son:
hola chau 4

Los argumentos se toman como valores de tipo:
[1] "character"
```

Si lo ejecutamos sin argumentos:

```
Rscript ejemplo1.R
```

```
Nos pasaron 0 argumentos.

Los argumentos que nos pasaron son:


Los argumentos se toman como valores de tipo:
[1] "character"
```

Ahora vamos a suponer que el programa `ejemplo2.R` tiene como objetivo contar un chiste o decir un refrán, según lo que se le pida en el único argumento que se le pasa al correrlo desde la terminal. Si el argumento es igual a "chiste", se cuenta el chiste; si es igual a "refran" se cuenta el refrán; y en otro caso no se hace nada. El contenido del archivo es:

```
# Capturar los argumentos pasados desde la terminal en un vector
args <- commandArgs(trailingOnly = TRUE)

if (args[1] == "chiste") {
  cat("- Juan, cómo has cambiado.\n- Yo no soy Juan.\n- Más a mi favor.\n\n")
} else if (args[1] == "refran") {
  cat("No por mucho madrugar amanece más temprano.\n\n")
} else {
  # Genero un error para que el programa se detenga, avisando lo que pasa
  stop("El argumento provisto debe ser igual a chiste o refran.\n")
}
```

Ejecutamos este archivo pasando distintos valores para su argumento:

```
Rscript ejemplo2.R refran
```

```
No por mucho madrugar amanece más temprano.
```

```
Rscript ejemplo2.R chiste
```

```
- Juan, cómo has cambiado.
- Yo no soy Juan.
- Más a mi favor.
```

```
Rscript ejemplo2.R hola
```

```
Error: El argumento provisto debe ser igual a chiste o refran.
Execution halted
```

Podemos controlar la cantidad de argumentos admitidos generando errores en el código para aquellas situaciones donde el usuario envíe menos o más que la cantidad deseada. Por ejemplo, en el caso anterior, es obligatorio enviar uno y sólo un argumento. Modificamos el script para que lo tenga en cuenta, dando lugar al programa `ejemplo3.R`:

```
# Capturar los argumentos pasados desde la terminal en un vector
args <- commandArgs(trailingOnly = TRUE)

# Controlar la cantidad de argumentos
if (length(args) == 0 || length(args) > 1) {
  stop("Debe proveer exactamente un argumento, que debe ser igual a chiste o refran.\n")
}

if (args[1] == "chiste") {
  cat("- Juan, cómo has cambiado.\n- Yo no soy Juan.\n- Más a mi favor.\n\n")
} else if (args[1] == "refran") {
  cat("No por mucho madrugar amanece más temprano.\n\n")
} else {
  # Genero un error para que el programa se detenga, avisando lo que pasa
  stop("El argumento provisto debe ser igual a chiste o refran.\n")
}
```

Veamos lo que pasa si cumplimos o no con la cantidad exacta de argumentos que hay que pasarle al código de R:

```
Rscript ejemplo3.R
```

```
Error: Debe proveer exactamente un argumento, que debe ser igual a chiste o refran.
Execution halted
```

```
Rscript ejemplo3.R chiste refran
```

```
Error: Debe proveer exactamente un argumento, que debe ser igual a chiste o refran.
Execution halted
```

```
Rscript ejemplo3.R chiste
```

```
- Juan, cómo has cambiado.
- Yo no soy Juan.
- Más a mi favor.
```

Imaginemos por último que es obligatorio pasar un primer argumento ("chiste" o "refran") y que opcionalmente se puede pasar un segundo argumento, que se va a tratar de un número para indicar cuántas veces queremos que el chiste o el refrán se repita. Como todos los argumentos se pasan como datos de tipo carácter, para poder usar el número tendremos que convertirlo a dato de tipo numérico. Añadimos esta característica en el script `ejemplo4.R`:

```
# Capturar los argumentos pasados desde la terminal en un vector
args <- commandArgs(trailingOnly = TRUE)

# Si no proveyó argumentos, generar un error y que se detenga el programa
if (length(args) == 0) {
  stop("Debe proveer al menos un argumento (chiste o refran).")
}

# Si proveyó más de 2 argumentos, generar un error y que se detenga el programa
if (length(args) > 2) {
  stop("No debe proveer más de 2 argumentos. El primero es obligatorio (chiste o refran) y e
}

# Si no hay segundo argumento, args[2] es NA
if (is.na(args[2])) {
  n <- 1
} else {
  n <- as.numeric(args[2])
}

# Repetir n veces
for (i in 1:n) {
  if (args[1] == "chiste") {
    cat("- Juan, cómo has cambiado.\n- Yo no soy Juan.\n- Más a mi favor.\n\n")
  } else if (args[1] == "refran") {
    cat("No por mucho madrugar amanece más temprano.\n\n")
```

```
  } else {
    # Genero un error para que el programa se detenga, avisando lo que pasa
    stop("El argumento provisto debe ser igual a chiste o refran.\n")
  }
}
```

Veamos ahora cómo funciona:

```
Rscript ejemplo4.R refran 5
```

```
No por mucho madrugar amanece más temprano.

No por mucho madrugar amanece más temprano.

No por mucho madrugar amanece más temprano.

No por mucho madrugar amanece más temprano.

No por mucho madrugar amanece más temprano.
```

```
Rscript ejemplo4.R chiste 3
```

```
- Juan, cómo has cambiado.
- Yo no soy Juan.
- Más a mi favor.

- Juan, cómo has cambiado.
- Yo no soy Juan.
- Más a mi favor.

- Juan, cómo has cambiado.
- Yo no soy Juan.
- Más a mi favor.
```

```
Rscript ejemplo4.R refran
```

```
No por mucho madrugar amanece más temprano.
```

```
Rscript ejemplo4.R
```

```
Error: Debe proveer al menos un argumento (chiste o refran).
Execution halted
```

Además de la función `commandArgs()` existen paquetes de R para poder trabajar con argumentos y opciones de formas mucho más elaboradas, como los paquetes `argparse` y `optparse`, entre otros.

Capítulo 22

Práctica de la Unidad 4

22.1 Ejercicio 1

El equipo de administración de usuarios de un sistema de ventas debe organizar los datos de los usuarios registrados y temporales. Para hacerlo, se necesita realizar las siguientes acciones desde la terminal:

a. Crear una carpeta principal llamada `organizacion_usuarios`.

b. Dentro de ella, crear tres subcarpetas: `registrados`, `temporales_abril` e `historial`.

c. Agregar información ficticia de al menos tres usuarios registrados y tres usuarios temporales. Cada usuario debe tener un archivo propio, cuyo nombre debe ser un código alfanumérico de 8 caracteres que finalice con `_T` para los usuarios temporales y `_R` para los usuarios registrados (por ejemplo: `ABC1234X_R.txt`). Se puede inventar cualquier código. El contenido de los archivos debe ser:

   - Usuarios registrados: dirección IP, nombre de usuario y correo electrónico.
   - Usuarios temporales: dirección IP y nombre de invitado (ejemplo: "invitado01")

   Asegurarse de que cada archivo esté guardado en la carpeta correspondiente.

d. Uno de los usuarios temporales se ha registrado oficialmente en el sistema. Se deben seguir los siguientes pasos para actualizar la información:

   - Copiar el archivo de ese usuario desde `temporales_abril` a la carpeta `registrados`.
   - Editar el archivo del usuario para agregar el nombre de usuario y el correo electrónico correspondiente.
   - Renombrar el archivo para que su nombre termine en `_R` en lugar de `_T`.
   - Mostrar el contenido del archivo actualizado por pantalla para asegurarse que la información fue correctamente registrada.
   - Finalmente, eliminar el archivo original de la carpeta `temporales_abril`.

e. Además, se debe considerar que los archivos temporales se borran mensualmente, pero se guarda un respaldo en la carpeta `historial`. Para lograr esto:

   - Copiar a la carpeta historial `historial` cada uno de los archivos de la carpeta `temporales_abril`.
   - Borrar la carpeta `temporales_abril` y crear el directorio para el próximo mes, `temporales_mayo`.

22.2 Ejercicio 2

Crear un script en R llamado `menu.R` que muestre un menú de opciones en la terminal, lea la opción elegida por el usuario y ejecute una acción diferente según la opción seleccionada. El menú debe contener al menos tres opciones: "1. Saludar", "2. Mostrar fecha" y "3. Salir". El programa debe:

- Mostrar el menú al usuario.
- Leer la opción ingresada.
- Responder con una acción distinta para cada opción:
  - Si elige "1", pedirle al usuario que ingrese su nombre, mostrar un saludo personalizado (por ejemplo: "¡Hola, Pepito!") y mostrar otra vez el menú principal.
  - Si elige "2", mostrar la hora actual (para obtenerla se puede usar el comando de R `format(Sys.time(), "%H:%M:%S")`) y mostrar otra vez el menú principal.
  - Si elige "3", pedirle al usuario que ingrese su nombre, mostrar un saludo personalizado, por ejemplo: "¡Chau, Pepito!" y finalmente finalizar su ejecución.

Si el usuario ingresa una opción no válida, se debe mostrar un mensaje como "Opción inválida" y solicitar nuevamente que ingrese su elección.

```
Rscript menu.R

=== MENÚ PRINCIPAL ===

1. Saludar
2. Mostrar hora actual
3. Salir

Elegí una opción: 2

La hora actual es: 11:44:30

=== MENÚ PRINCIPAL ===

1. Saludar
2. Mostrar hora actual
3. Salir
```

```
Elegí una opción: 3

Ingresá tu nombre para despedirte: Eugenia
¡Chau, Eugenia!
```

22.3 Ejercicio 3

Se desea desarrollar un programa que calcule la calificación promedio otorgada a una película por un grupo de jueces de cine. En primer lugar, el programa debe preguntar cuántos jueces integran el grupo, que debe tener un mínimo de 3 y un máximo de 6. Si el número ingresado no cumple con esta condición, el programa debe mostrar un mensaje adecuado y no continuar con el resto de las instrucciones.

Si el número de jueces está dentro del intervalo aceptado, el programa debe:

- Solicitar al usuario que ingrese el nombre de la película.
- Solicitar las calificaciones otorgadas por cada uno de los jueces. Las calificaciones deben ser ingresadas una por una y se debe chequear que cada una esté dentro del rango válido de 0 a 10. En caso contrario, debe mostrar un mensaje de error y solicitar nuevamente esa calificación.
- Calcular el promedio de los puntajes ingresados.
- Mostrar el nombre de la película y su puntaje promedio.

Ejemplo del uso del script

```
Rscript evaluar_pelicula.R

================================================

      SISTEMA DE EVALUACIÓN DE PELÍCULAS

================================================

Ingrese la cantidad de jueces en el grupo:
3

Ingrese el nombre de la película:
Shrek 1

Ingrese la calificación del juez 1:
8.7

Ingrese la calificación del juez 2:
9

Ingrese la calificación del juez 3:
```

```
9.2

La clasificación promedio para la película <Shrek> es 8.97 puntos.
```

22.4 Ejercicio 4

En el campus virtual de una universidad, los usuarios pueden acceder a diferentes salas, cada una correspondiente a una materia específica (por ejemplo, Programación 1, Laboratorio de Datos 1, Estadística 1). Cada sala está protegida por una contraseña única. Para acceder, el usuario debe ingresar la contraseña correcta. En caso de error, debe intentarlo nuevamente hasta lograrlo o hasta que se alcance un límite máximo de intentos, tras lo cual la cuenta será bloqueada.

Para el caso de la sala de Programación 1, la contraseña es "amoprogramar". Vamos a suponer que un usuario quiere ingresar a esta sala. Escribir un programa que:

- Muestre un mensaje solicitando el ingreso de la contraseña.
- Lea el valor ingresado por el usuario.
- Inicie un proceso de verificación:
  - Si la contraseña ingresada es incorrecta, debe mostrar:
    `"Contraseña incorrecta. Ingrésela nuevamente."`

    y permitir un nuevo intento.
  - Si la contraseña es correcta, debe mostrar:
    `"¡Contraseña correcta! Puede continuar con sus estudios."`

A partir de este escenario, se deben proponer **dos versiones** del programa, considerando los siguientes casos:

a. El usuario puede intentar indefinidamente hasta ingresar la contraseña correcta.

b. El usuario tiene un máximo de cinco intentos. Por cada intento fallido, el mensaje emitido debe indicar la cantidad de intentos restantes. Si no ingresa la contraseña correcta en ese número de intentos, se debe mostrar el mensaje:
   `"Demasiados intentos fallidos. Su cuenta ha sido bloqueada."`

22.5 Ejercicio 5

En el ejercicio 7 de la practica 3 se tuvo que crear una función llamada `resolvente(a, b, c)`, que muestra las soluciones de la ecuación de segundo grado $ax^2 + bx + c = 0$, empleando la fórmula resolvente:

$$x_{1,2} = \frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$$

Modificar el código e incluirlo en un script que pueda ser ejecutado desde la terminal de manera que se cumplan los siguientes requisitos:

- Pedir al usuario que ingrese los coeficientes `a`, `b` y `c` uno por uno.
- Si el coeficiente `a` es cero, mostrar un mensaje de error y finalice el programa.
- Mostrar la ecuación ingresada en formato legible (ejemplo: `x² - 5x + 6 = 0`).
- Calcular el discriminante (`D = b^2 - 4 * a * c`) y mostrar su valor.
- Clasificar las soluciones de acuerdo al resultado de `D`:
  - Si `D > 0`: mostrar las dos soluciones reales distintas con un mensaje descriptivo como: "Existen dos soluciones reales distintas: x = 2, x = 3".
  - Si `D = 0`: mostrar la única solución real (doble) con un mensaje descriptivo como "Existe una única solución real: x = 2 (raiz doble)".
  - Si `D < 0`: indicar que las soluciones son complejas (sin calcularlas).
- Mejorar la experiencia del usuario incluyendo:
  - Mensaje de bienvenida y despedida.
  - Números que se muestren redondeados para facilitar su lectura.

El uso de la función debe ser semejante al siguiente ejemplo:

```
Rscript resolvente.R

========================================
  Calculadora de ecuaciones cuadráticas
========================================

Ingrese el valor del coeficiente 'a':
1
Ingrese el valor del coeficiente 'b':
-1
Ingrese el valor del coeficiente 'c':
2

Ecuación ingresada:  1x²-1x+2=0
```

```
Discriminante (D): -7
Las soluciones son complejas (no reales).

Gracias por usar la calculadora de ecuaciones cuadráticas.
```

22.6 Ejercicio 6 (opcional)

Reescribir el programa anterior de forma que, en lugar de solicitar al usuario que ingrese los valores de `a`, `b` y `c` uno por uno desde la terminal, estos se ingresen directamente como argumentos del sistema al momento de ejecutar el script. Para ello, utilizar `commandArgs(trailingOnly = TRUE)` para capturar los valores y adaptar el código en consecuencia. Asegurarse de incluir validaciones adecuadas para verificar que los argumentos sean numéricos y que `a` no sea igual a cero.

Ejemplo de ejecución desde la terminal:

```
Rscript resolvente_args.R 1 -1 -2
```

```
========================================
  Calculadora de ecuaciones cuadráticas
========================================

Ecuación ingresada:  1x²-1x-2=0
Discriminante (D): 9
Existen dos soluciones reales distintas:
x  = -1 , x  = 2

Gracias por usar la calculadora de ecuaciones cuadráticas.
```

```
Rscript resolvente_args.R 1 2 1
```

```
========================================
  Calculadora de ecuaciones cuadráticas
========================================

Ecuación ingresada:  1x²+2x+1=0
Discriminante (D): 0
Existe una única solución real (raíz doble):
x = -1

Gracias por usar la calculadora de ecuaciones cuadráticas.
```

```
Rscript resolvente_args.R 1 1 1
```

```
========================================
  Calculadora de ecuaciones cuadráticas
========================================

Ecuación ingresada:  1x²+1x+1=0
Discriminante (D): -3
Las soluciones son complejas (no reales).

Gracias por usar la calculadora de ecuaciones cuadráticas.
```

```
Rscript resolvente_args.R 0 1 1
```

```
========================================
  Calculadora de ecuaciones cuadráticas
========================================
Error: Error: el coeficiente 'a' debe ser distinto de cero. Fin del programa.
Execution halted
```

```
Rscript resolvente_args.R 2 hola 1
```

```
========================================
  Calculadora de ecuaciones cuadráticas
========================================
Warning message:
NAs introduced by coercion
Error: Los argumentos ingresados no son valores numéricos.
Execution halted
```

Capítulo 23

Actividad de autoevaluación 4

EN DESARROLLO
El contenido estará disponible a la brevedad.

Soluciones de la Práctica

RESUMEN

En esta sección se presentan las respuestas a todos los ejercicios de la práctica. Al hacer uso de este material, se debe tener en cuenta:

- Muchos ejercicios no tienen una respuesta única y sólo se muestra una solución posible. Aún más, muchas veces elegimos mostrar una solución por ser sencilla o estar alineada con los temas desarrollados, cuando en realidad hay otras formas que tal vez sean mejores para resolver el mismo problema.
- Es importante intentar resolver los ejercicios por cuenta propia y no acercarse a la resolución como primera medida. Leer una respuesta puede darnos la falsa sensación de que entendemos, cuando en realidad enfrentar la resolución de un problema forma independiente puede ser mucho más desafiante. Este material debe servir como un apoyo después de haber trabajado en la resolución de los problemas.

Capítulo 24

Soluciones de la Práctica de la Unidad 1

24.1 Ejercicio 1

Para crear un nuevo script y guardarlo, podés seguir los ejemplos vistos en #sec-scripts. El contenido del script tiene que ser:

```
# Ejercicio 1
25 + 17
6 * 8
sqrt(144)
```

24.2 Ejercicio 2

a. Para acceder a la documentación de la función `round()`, se puede ejecutar en la consola de R:

```
?round
```

o

```
help(round)
```

b. Según la ayuda de R, la función `round()` tiene los siguientes argumentos:

- `x`: El número o vector de números que se desea redondear.

- `digits`: El número de cifras decimales al que se desea redondear `x`. Puede ser positivo (para redondear decimales) o negativo (para redondear a múltiplos de 10).
- `...`: representa el uso opcional de otros argumentos, es algo que podemos ignorar por ahora.

c.

- **Obligatorio:** `x`, ya que es el número a redondear.
- **Opcional:** `digits`, que por defecto es `0`, lo que significa que la función redondeará al número entero más cercano si no se especifica un valor distinto.

d.

```
round(3.14159, digits = 0)  # Redondeo a 0 decimales → Resultado: 3
```

```
[1] 3
```

```
round(3.14159, digits = 1)  # Redondeo a 1 decimal → Resultado: 3.1
```

```
[1] 3.1
```

```
round(3.14159, digits = 2)  # Redondeo a 2 decimales → Resultado: 3.14
```

```
[1] 3.14
```

e.

```
round(3.14159, 2)
```

```
[1] 3.14
```

```
round(x = 3.14159, digits = 2)
```

```
[1] 3.14
```

```
round(digits = 2, x = 3.14159)
```

```
[1] 3.14
```

24.3 Ejercicio 3

```
x <- 10
y <- "Hola"
z <- 5
```

a. Podemos verificar el tipo de cada objeto con la función `typeof()`:

```
typeof(x)
```

```
[1] "double"
```

```
typeof(y)
```

```
[1] "character"
```

```
typeof(z)
```

```
[1] "double"
```

b. Ejecutamos la siguiente instrucción:

```
z <- x * 6
```

Ahora, `z` almacenará el valor:

```
z
```

```
[1] 60
```

c. Obtenemos un error porque `x` es un número (`double`), mientras que `y` es un texto (`character`). En R, no es posible realizar operaciones matemáticas entre objetos de tipo diferente.

```
x + y
```

```
Error in x + y: non-numeric argument to binary operator
```

24.4 Ejercicio 4

```
load("practica1_ambiente.RData")
```

Se incorporaron 6 objetos al ambiente:

Identificador	Tipo de vector	Valor
var1	*logical*	TRUE
var2	*double*	200.12
var3	*character*	"hola"
var4	*character*	"chau"
var5	*integer*	-49L
var6	*character*	"Hola"

```
typeof(var1)
```

```
[1] "logical"
```

```
typeof(var2)
```

```
[1] "double"
```

```
typeof(var3)
```

```
[1] "character"
```

```
typeof(var4)
```

```
[1] "character"
```

```
typeof(var5)
```

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

```
typeof(var6)
```

```
[1] "character"
```

24.5 Ejercicio 5

a.

```
var2 < 0 || var5 < 0
```

```
[1] TRUE
```

b.

```
var2 < 0 && var5 < 0
```

```
[1] FALSE
```

c.

```
var2 %% (var5 + 100) < 10
```

```
[1] FALSE
```

d. No son iguales porque uno tiene una letra mayúscula, “H” es un caracter distinto de “h” y por lo tanto son cadenas de texto diferentes.

```
var3 == var6
```

```
[1] FALSE
```

24.6 Ejercicio 6

Operación	edad <- 21, altura <- 1.90	edad <- 17 , altura <- 1.90	edad <- 21, altura <- 1.50
(edad > 18) && (altura < 1.70)	FALSE	FALSE	TRUE
(edad > 18) || (altura < 1.70)	TRUE	FALSE	TRUE
!(edad > 18)	FALSE	TRUE	FALSE

Verificación en R:

```r
# Primera columna
edad <- 21
altura <- 1.90

(edad > 18) && (altura < 1.70)
```

```
[1] FALSE
```

```r
(edad > 18) || (altura < 1.70)
```

```
[1] TRUE
```

```r
!(edad > 18)
```

```
[1] FALSE
```

```r
# Segunda columna
edad <- 17
altura <- 1.90

(edad > 18) && (altura < 1.70)
```

```
[1] FALSE
```

```r
(edad > 18) || (altura < 1.70)
```

```
[1] FALSE
```

```r
!(edad > 18)
```

```
[1] TRUE
```

```r
# Tercera columna
edad <- 21
altura <- 1.50

(edad > 18) && (altura < 1.70)
```

```
[1] TRUE
```

```
(edad > 18) || (altura < 1.70)
```

```
[1] TRUE
```

```
!(edad > 18)
```

```
[1] FALSE
```

24.7 Ejercicio 7

Razonamiento paso a paso:

```
1 + 2 + (3 + 4) * ((5 * 6 %% 7 * 8) - 9) - 10
1 + 2 + (3 + 4) * ((5 * 6 * 8) - 9) - 10
1 + 2 + (3 + 4) * (240 - 9) - 10
1 + 2 + 7 * 231 - 10
1 + 2 + 1617 - 10
1610
```

Verificación en R:

```
1 + 2 + (3 + 4) * ((5 * 6 %% 7 * 8) - 9) - 10
```

```
[1] 1610
```

24.8 Ejercicio 8

Siempre es verdadera porque sea cual fuere `x`, siempre va a ser distinta a alguno de los dos, incluso si es 4, es distinta a 17 y viceversa.

Recordemos que:

- `!=` significa "distinto de".
- `||` es el operador "o" lógico (OR), que devuelve `TRUE` si al menos una de las condiciones es `TRUE`.

Luego, de la única forma para que la expresión sea `FALSE`, es que ambas condiciones sean `FALSE` al mismo tiempo. Veamos si esto es posible:

- La primera condición `(x != 4)` es `FALSE` solo cuando `x = 4`.
- La segunda condición `(x != 17)` es `FALSE` solo cuando `x = 17`.

Por lo visto, ambas condiciones no pueden ser `FALSE` simultáneamente, ya que un número no puede ser `4` y `17` al mismo tiempo. Dado que siempre hay al menos una condición que es `TRUE`, la expresión es **siempre verdadera**, sin importar el valor de `x`.

24.9 Ejercicio 9

El primer ítem excluye a los años terminados en 00, los cuales son evaluados en la segunda regla. El segundo ítem incluye automáticamente a los divisibles por 4 porque 400 es divisible por 4. Entonces, la operación lógica que determina si un año es bisiesto es: `((año %% 4 == 0) && (año %% 100 != 0)) || (año %% 400 == 0)`.

En R:

```
año <- 2024
((año %% 4 == 0) && (año %% 100 != 0)) || (año %% 400 == 0)
```

```
[1] TRUE
```

```
año <- 2025
((año %% 4 == 0) && (año %% 100 != 0)) || (año %% 400 == 0)
```

```
[1] FALSE
```

24.10 Ejercicio 10

a. ¿Cuáles son los valores finales de `a` y `b`?

```
a <- 10
b <- a * 2
a <- a + 5
b <- b - a
a
```

```
[1] 15
```

```
b
```

```
[1] 5
```

b. ¿Cuáles son los valores finales de `m` y `n`?

```
m <- 5
n <- 2 * m
m <- m + 3
n <- n + m
m <- n - 4
m
```

```
[1] 14
```

```
n
```

```
[1] 18
```

c. ¿Cuál es el valor final de `y`?

```
x <- 6
y <- 2
x <- x / y + x * y
y <- x^2 %% 10
y <- y * 2
y
```

```
[1] 10
```

d. ¿Cuál es el valor final de `resultado`?

```
a <- 5
b <- 2
c <- 3

resultado <- a^b - (c * b) + (a %% c)
resultado
```

```
[1] 21
```

e. ¿Cuáles son los valores finales de `x`, `y` y `z`?

```
x <- 8
y <- 3
z <- 2

x <- x %% y + z^y
y <- (x + y) %/% z
z <- z + x - y
x
```

```
[1] 10
```

```
y
```

```
[1] 6
```

```
z
```

```
[1] 6
```

24.11 Ejercicio 11

Se pueden cambiar los valores de `a`, `b` y `h`.

```
a <- 5
b <- 4
h <- 3
volumen <- a * b * h
area <- 2 * (a * b + a * h + b * h)
cat("El área es igual a", area, "y el volumen es igual a", volumen)
```

```
El área es igual a 94 y el volumen es igual a 60
```

24.12 Ejercicio 12

a. Para identificar la ruta informática de un archivo, podés seguir los pasos mostrados en Sección 5.1.
b. Se puede saber cuál es el *working directory* con:

```
getwd()
```

24.13 Ejercicio 13

El resultado de crear las carpetas y subcarpetas tiene que ser similar al que se ve en la Figura 5.2. El proyecto se crea desde RStudio siguiendo los pasos de la Figura 5.4.

Capítulo 25

Soluciones de la Práctica de la Unidad 2

25.1 Ejercicio 1

```
# PROGRAMA: "Paridad de un número"
x <- 3
if (x %% 2 == 0) {
    cat(x, "es par")
} else {
    cat(x, "es impar")
}
```

```
3 es impar
```

En el código anterior, realizamos una asignación explícita de un valor en la variable (`x <- 3`). Esto nos permite hacer un uso más interactivo del código, que puede ser útil para probar si anda o cuando vamos corriendo por partes para detectar posibles errores.

Otra opción se muestra en el siguiente ejemplo. En lugar de asignarle un valor a `x` como parte del script, indicamos que cuando el código sea ejecutado, el usuario deberá ingresar un valor con el teclado. Para esto empleamos la función `scan()`. Su argumento `n = 1` significa que el usuario sólo deberá ingresar un número, el cual será asignado a la variable `x`. Teniendo un archivo de código con el programa como se muestra a continuación, podemos ejecutarlo de forma completa mediante el botón `Source` de RStudio. Al llegar a la evaluación de `scan()`, deberemos ingresar el valor que deseemos en la consola y dar Enter para que la ejecución continúe.

```
# PROGRAMA: "Paridad de un número"
cat("Ingrese un número entero: ")
x <- scan(n = 1)
if (x %% 2 == 0) {
    cat(x, "es par")
} else {
    cat(x, "es impar")
}
```

25.2 Ejercicio 2

```
# PROGRAMA: "Mayor de tres números"
x <- 5
y <- 5
z <- 7
if (x >= y && x >= z) {
    cat("El mayor es", x)
} else if (y >= z) {
    cat("El mayor es", y)
} else {
    cat("El mayor es", z)
}
```

```
El mayor es 7
```

Observación: se llega a la segunda evaluación lógica cuando `y` es mayor a `x`, o cuando `z` es mayor que `x`, o cuando ambos son mayores a `x`, por lo tanto sólo es necesario saber cuál de ellos es el mayor.

25.3 Ejercicio 3

```
# PROGRAMA: "Determinar salario"

# Valores fijados
valor_hora <- 4000
valor_adicional_noche <- 2000
valor_adicional_domingo <- 1000

# Valores para un cálculo particular
horas <- 8
dia <- "MAR"
```

```
turno <- "T"

# Cálculo para el pago de ese día a ese empleado
salario <- horas * valor_hora
if (turno == "N") {
    salario <- salario + horas * valor_adicional_noche
}
if (turno == "DOM") {
    salario <- salario + horas * valor_adicional_domingo
}
cat("El salario que se debe abonar es", salario)
```

```
El salario que se debe abonar es 32000
```

25.4 Ejercicio 4

```
# PROGRAMA: "Convertir temperatura"
temp <- 20
modo <- "C a F"
if (modo == "C a F") {
    temp_nuevo <- temp * 9 / 5 + 32
    cat(temp, "ºC equivale a", temp_nuevo, "ºF")
} else {
    temp_nuevo <- (temp - 32) * 5 / 9
    cat(temp, "ºF equivale a", temp_nuevo, "ºC")
}
```

```
20 ºC equivale a 68 ºF
```

25.5 Ejercicio 5

a.

```
# PROGRAMA: "Suma de los n primeros números naturales"
n <- 10
suma <- 0
for (i in 1:n) {
    suma <- suma + i
}
cat("La suma de los primeros", n, "números naturales es", suma)
```

```
La suma de los primeros 10 números naturales es 55
```

b.

```r
# PROGRAMA: "Suma de los cuadrados de los n primeros números naturales" -----
n <- 10
suma <- 0
for (i in 1:n) {
    suma <- suma + i^2
}
cat("La suma de los primeros", n, "números naturales al cuadrado es", suma)
```

```
La suma de los primeros 10 números naturales al cuadrado es 385
```

c.

Sabemos el número de iteraciones, `n`, por eso usamos `for`. Por ejemplo, para `n = 4`, los primeros impares son:

```
1 = 2 * 1 - 1
3 = 2 * 2 - 1
5 = 2 * 3 - 1
7 = 2 * 4 - 1
```

De forma general, los primeros 4 impares son `2 * i - 1`, con `i = 1, 2, 3, 4`.

```r
# PROGRAMA: "Producto de los n primeros números naturales impares" ----------
n <- 10
producto <- 1
for (i in 1:n) {
    producto <- producto * (2 * i - 1)
}
cat("El producto de los primeros", n, "números naturales impares es", producto)
```

```
El producto de los primeros 10 números naturales impares es 654729075
```

Otra forma:

```r
# PROGRAMA: "Producto de los n primeros números naturales impares" ----------
n <- 10
producto <- 1
for (i in seq(1, 2 * n - 1, 2)) {
    producto <- producto * i
}
cat("El producto de los primeros", n, "números naturales impares es", producto)
```

```
El producto de los primeros 10 números naturales impares es 654729075
```

La función `seq` crea un vector con una secuencia numérica. El formato con el que está utilizada aquí es `seq(valor_inicial, valor_final, intervalo)`, por lo que creará un vector con los números impares (dado que el intervalo es 2, y comienza con un número impar) empezando desde `1` hasta `2 * n - 1`.

d.

```r
# PROGRAMA: "Suma de los cubos de los n primeros números naturales pares"
n <- 5
suma <- 0
for (i in 1:n) {
    suma <- suma + (2 * i)^3
}
cat("La suma de los cubos de los primeros", n, "números naturales pares es", suma)
```

```
La suma de los cubos de los primeros 5 números naturales pares es 1800
```

25.6 Ejercicio 6

Utilizando estructura de control “PARA”

```r
# PROGRAMA: "Calcular el factorial de n"
n <- 0
factorial <- 1
for (i in 1:n) {
    factorial <- factorial * i
}
cat("El factorial de", n, "es", factorial)
```

```
El factorial de 0 es 0
```

Utilizando estructura de control “MIENTRAS QUE”

```r
# PROGRAMA: "Calcular el factorial de n"
n <- 0
factorial <- 1
i <- 0
while(i < n) {
    i <- i + 1
  factorial <- factorial * i
}
cat("El factorial de", n, "es", factorial)
```

```
El factorial de 0 es 1
```

IMPORTANTE. Notar que la primera opción no arroja de forma correcta el factorial de 0, que por definición es igual a 1. Se podría agregar alguna estructura condicional para esta situación. La segunda opción, en cambio, funciona también para `n <- 0`.

25.7 Ejercicio 7

```
# PROGRAMA: "Secuencia de Fibonacci"
termino1 <- 0
termino2 <- 1
while (termino1 < 10000) {
  print(termino1)
  termino3 <- termino1 + termino2
  termino1 <- termino2
  termino2 <- termino3
}
```

```
[1] 0
[1] 1
[1] 1
[1] 2
[1] 3
[1] 5
[1] 8
[1] 13
[1] 21
[1] 34
[1] 55
[1] 89
[1] 144
[1] 233
[1] 377
[1] 610
[1] 987
[1] 1597
[1] 2584
[1] 4181
[1] 6765
```

25.8 Ejercicio 8

Observar que:

- el valor inicial es el 100 ya que la combinación no comienza por 0.
- el `while` se detiene cuando la combinación es mayor a 800.
- se verifican las condiciones de multiplicidad, y si no se cumplen, se pasa directamente a la siguiente iteración.
- en caso de que se cumplan las condiciones, se imprime la combinación, y se pasa a la siguiente iteración.

```
# PROGRAMA: "Lista de posibles combinaciones"
combinacion <- 100

while (combinacion <= 800) {
  if (combinacion %% 11 == 0 && combinacion %% 8 != 0) {
    print(combinacion)
  }
  combinacion <- combinacion + 1
}
```

```
[1] 110
[1] 121
[1] 132
[1] 143
[1] 154
[1] 165
[1] 187
[1] 198
[1] 209
[1] 220
[1] 231
[1] 242
[1] 253
[1] 275
[1] 286
[1] 297
[1] 308
[1] 319
[1] 330
[1] 341
[1] 363
[1] 374
[1] 385
[1] 396
[1] 407
[1] 418
[1] 429
[1] 451
```

```
[1] 462
[1] 473
[1] 484
[1] 495
[1] 506
[1] 517
[1] 539
[1] 550
[1] 561
[1] 572
[1] 583
[1] 594
[1] 605
[1] 627
[1] 638
[1] 649
[1] 660
[1] 671
[1] 682
[1] 693
[1] 715
[1] 726
[1] 737
[1] 748
[1] 759
[1] 770
[1] 781
```

25.9 Ejercicio 9

a. Al final del primer año, la población será igual a:

$$p0 + p0 \times \frac{tasa}{100} + inmigrantes = 1000 + 1000 * 0.02 + 50 = 1070$$

b. Al final del 2º año, la población será:

$$p0 + p0 \times \frac{tasa}{100} + inmigrantes = 1070 + 1070 * 0.02 + 50 = 1141.4 \cong 1141$$

Recordar que el número de habitantes es un entero, por lo cual tomamos la parte entera.

Al final del tercer año:

$$p0 + p0 \times \frac{tasa}{100} + inmigrantes = 1141 + 1141 * 0.02 + 50 = 1213.82 \cong 1213$$

Se necesitarán 3 años.

c. Haciendo la solución para el problema de forma más general:

```
# Parámetros a definir
p0 <- 1000
p_objetivo <- 1200
tasa <- 2 # La expresamos en porcentaje
inmigrantes <- 50

# Programa
año <- 0
poblacion <- p0
while (poblacion < p_objetivo) {
  # Aplicamos la fórmula y nos quedamos con la parte entera del resultado
  poblacion <- floor(poblacion + poblacion * tasa / 100 + inmigrantes)
  # Contamos un año más
  año <- año + 1
}
cat(
  " Con una población inicial de", p0,
  "habitantes, una tasa de crecimiento anual del\n",
  tasa, "% y", inmigrantes, "nuevos habitantes anuales, se necesitarán",
  año, "años para alcanzar\n una población de", p_objetivo,
  "habitantes. Luego del año", año, "el pueblo contará con una\n población de",
  poblacion, "habitantes."
)
```

```
 Con una población inicial de 1000 habitantes, una tasa de crecimiento anual del
 2 % y 50 nuevos habitantes anuales, se necesitarán 3 años para alcanzar
 una población de 1200 habitantes. Luego del año 3 el pueblo contará con una
 población de 1213 habitantes.
```

d. Cambiamos los valores de las variables y realizamos el cálculo nuevamente:

```
# Parámetros a definir
p0 <- 10000
p_objetivo <- 50000
tasa <- 3 # La expresamos en porcentaje
inmigrantes <- 100

# Programa
año <- 0
poblacion <- p0
while (poblacion < p_objetivo) {
  # Aplicamos la fórmula y nos quedamos con la parte entera del resultado
```

```
  poblacion <- floor(poblacion + poblacion * tasa / 100 + inmigrantes)
  # Contamos un año más
  año <- año + 1
}
cat(
  " Con una población inicial de", p0,
  "habitantes, una tasa de crecimiento anual del\n",
  tasa, "% y", inmigrantes, "nuevos habitantes anuales, se necesitarán",
  año, "años para alcanzar\n una población de", p_objetivo,
  "habitantes. Luego del año", año, "el pueblo contará con una\n población de",
  poblacion, "habitantes."
)
```

```
 Con una población inicial de 10000 habitantes, una tasa de crecimiento anual del
 3 % y 100 nuevos habitantes anuales, se necesitarán 47 años para alcanzar
 una población de 50000 habitantes. Luego del año 47 el pueblo contará con una
 población de 50116 habitantes.
```

Capítulo 26

Soluciones de la Práctica de la Unidad 3

INFO IMPORANTE

Instrucciones generales para resolver los problemas de esta práctica:

1. Abrir RStudio y crear un nuevo proyecto llamado `unidad3`, para guardar allí todos los archivos que usaremos. Asegurarse de que RStudio esté trabajando con este proyecto abierto.
2. Al comenzar a resolver cada ejercicio:

a. Eliminar todos los objetos del *Global Environment*, para evitar confusiones con objetos que hayan sido creados para resolver otro problema.
b. Crear y guardar en la carpeta del proyecto un nuevo script con el nombre `ejercicio_*.R` para almacenar de manera organizada la solución de cada problema (por ejemplo, `ejercicio_01.R`, `ejercicio_02.R`, etc.)
c. A menos que se indique lo contrario, utilizar cada uno de estos scripts para escribir el código que crea la función pedida en el ejercicio y también el código con ejemplos para usarla.

26.1 Ejercicio 1

a. Escribimos la función y la usamos:

```
# Función f1
f1 <- function(x1, x2, x3) {
  resultado <- x1 / x2 + x3^2 + x2 * x3
```

```
  return(resultado)
}

# Ejemplo de su uso
f1(5, 2, 3)
```

```
[1] 17.5
```

b. Escribimos la función y la usamos:

```
# Función f2
f2 <- function(x1, x2 = 1, x3 = 1) {
  resultado <- x1 / x2 + x3^2 + x2 * x3
  return(resultado)
}

# Ejemplos de su uso
f2(5, 2, 3)
```

```
[1] 17.5
```

```
f2(5)
```

```
[1] 7
```

```
f2(5, 2)
```

```
[1] 5.5
```

```
f2(5, x3 = 3)
```

```
[1] 17
```

```
f2(x2 = 2, x3 = 3)
```

```
Error in f2(x2 = 2, x3 = 3): argument "x1" is missing, with no default
```

c. Escribimos la función y la usamos:

```
# Función f3
f3 <- function(x1, x2 = 1, x3 = 1) {
  if (x1 < 0 || x2 < 0 || x3 < 0) {
    return(-100)
  } else {
    resultado <- x1 / x2 + x3^2 + x2 * x3
    return(resultado)
  }
}
```

```
# Ejemplos de su uso
f3(5, 2, 3)
```

```
[1] 17.5
```

```
f3(-5, 2, 3)
```

```
[1] -100
```

```
f3(-5)
```

```
[1] -100
```

```
f3(5, x3 = -3)
```

```
[1] -100
```

26.2 Ejercicio 2

Sin importar el signo de `a` y `b` ni cuál es mayor o menor, la secuencia de todos los enteros se puede generar fácilmente con la expresión `a:b` que usamos como parte de la estructura `for`:

```
1:4
```

```
[1] 1 2 3 4
```

```
4:1
```

```
[1] 4 3 2 1
```

```
3:3
```

```
[1] 3
```

```
-2:4
```

```
[1] -2 -1  0  1  2  3  4
```

```
-2:-8
```

```
[1] -2 -3 -4 -5 -6 -7 -8
```

Por lo tanto, si iteramos con un `for` a través de la secuencia `a:b`, la función puede definirse así:

```
#' Suma de una secuencia de números enteros
#'
#' @description
#' Calcula la suma de un rango de números enteros, incluyendo los extremos.
#'
#' @param a,b números enteros
#'
#' @return suma de la secuencia
#'
#' @examples
#' suma_secuencia(1, 3)
#' suma_secuencia(-2, 3)
#' suma_secuencia(-3, -5)
#' suma_secuencia(3, 3)
#'
suma_secuencia <- function(a, b) {
  suma <- 0
  for (i in a:b) {
    suma <- suma + i
  }
  return(suma)
}
```

Ejemplos de su uso:

```
suma_secuencia(1, 3)
```

```
[1] 6
```

```
suma_secuencia(30, 40)
```

```
[1] 385
```

```
suma_secuencia(5, 2)
```

```
[1] 14
```

```
suma_secuencia(-2, 3)
```

```
[1] 3
```

```
suma_secuencia(-7, -5)
```

```
[1] -18
```

```
suma_secuencia(-3, -5)
```

```
[1] -12
```

```
suma_secuencia(-3, -3)
```

```
[1] -3
```

```
suma_secuencia(3, 3)
```

```
[1] 3
```

26.3 Ejercicio 3

```
#' Clasificación de un triángulo
#'
#' @description
#' Clasifica a un triángulo según las longitudes de sus lados, en escaleno, isósceles
#' o equilátero.
#'
#' @details
#' Se evalúa la desigualdad triangular. Si las medidas de los lados no corresponden
#' a un triángulo, la función devuelve "no es un triángulo".
#'
#' @param a,b,c números reales positivos
#'
#' @return valor carácter que indica el tipo de triángulo.
#'
#' @examples
#' triangulos(2, 3, 4)
#' triangulos(2, 3, 10)
#'
triangulos <- function(a, b, c) {
    if (a > b + c || b > a + c || c > a + b) {
        return("no es triángulo")
    } else if (a == b & a == c) {
```

```
        return("equilátero")
    } else if (a == b || a == c || b == c) {
        return("isósceles")
    } else {
        return("escaleno")
    }
}
```

Alternativamente, podemos prescindir de las estructuras anidadas, gracias a la existencia de las sentencias `return` en cada camino posible. Si alguna condición es `TRUE`, enseguida se devuelve el valor que corresponda y se detiene la ejecución de la función. El resto no se evalúa, lo cual hace innecesario usar `else if` o estructuras anidadas:

```
triangulos <- function(a, b, c) {
    if (a > b + c || b > a + c || c > a + b) {
      return("no es triángulo")
    }
    if (a == b & a == c) {
      return("equilátero")
    }
    if (a == b || a == c || b == c) {
      return("isósceles")
    }
    return("escaleno")
}
```

Ejemplos de uso:

```
triangulos(2, 3, 4)
```

```
[1] "escaleno"
```

```
triangulos(2, 3, 10)
```

```
[1] "no es triángulo"
```

26.4 Ejercicio 4

```r
#' Punto dentro de la elipse
#'
#' @description
#' Determina si un punto está contenido dentro de la elipse definida por la fórmula
#' (x - 6)^2 / 36 + (y + 4)^2 / 16 = 1
#'
#' @details
#' Por defecto se evalúa al origen de coordenadas.
#'
#' @param x,y coordenadas, números reales. Por defecto valen 0.
#'
#' @return valor lógico indicando si el punto está dentro de la elipse descripta.
#'
#' @examples
#' elipse(3, 7)
#' elipse(6, -4)
#' elipse()
#'
elipse <- function(x = 0, y = 0) {
  valor <- (x - 6)^2 / 36 + (y + 4)^2 / 16
    return(valor <= 1)
}
```

Ejemplos de uso:

```r
elipse(3, 7)
```

```
[1] FALSE
```

```r
elipse(6, -4)
```

```
[1] TRUE
```

```r
elipse()
```

```
[1] FALSE
```

26.5 Ejercicio 5

Si bien hay muchas formas de resolver este ejercicio, tal vez usando estructuras iterativas y condicionales, conviene pensarlo desde un punto de vista matemático. La suma de los números impares de la fila n resulta igual a n^3. Para darse cuenta, conviene pensar en los siguientes puntos:

- La suma de los primeros n números naturales es: $n(n+1)/2$. Por ejemplo, si $n = 3$, $1+2+3 = 6 = 3*4/2$.
- La suma de los primeros n números naturales impares es: n^2. Por ejemplo, si $n = 3$, $1+3+5 = 9 = 3^2$.
- En la fila i, hay exactamente i números impares. Por ejemplo, en la fila $i = 3$, hay 3 números: 7, 9 y 11.
- Desde la fila 1 hasta la fila n inclusive, hay $n(n+1)/2$ números impares. En la fila 1 hay 1, en la fila 2 hay 2, etc. Entonces hasta la fila n hay $1+2+3+...+n = n(n+1)/2$ (suma de los primeros n naturales).
- La suma de los números impares desde la fila 1 hasta la fila n inclusive entonces es igual a la suma de los primeros $n(n+1)/2$ números impares: $\left(\frac{n(n+1)}{2}\right)^2$.
- Del mismo modo, la suma de los números impares desde la fila 1 hasta la fila $n-1$ inclusive es igual a la suma de los primeros $(n-1)n/2$ números impares: $\left(\frac{(n-1)n}{2}\right)^2$.
- Como sólo queremos la suma de los impares que están en la fila n, podemos calcular la suma desde la fila 1 hasta la n inclusive, y restarle la suma desde la fila 1 hasta la fila $n-1$ inclusive. El número que buscamos entonces es:

$$\left(\frac{n(n+1)}{2}\right)^2 - \left(\frac{(n-1)n}{2}\right)^2 = n^3$$

El código de la función puede ser sencilamente:

```
#' Suma de fila de la pirámide
#'
#' @description
#' La pirámide se arma con números impares, empezando con el 1 en la cima, el 3 y 5
#' en la segunda fila, y así sucesivamente. Devuelve la suma de los números ubicados
#' en la fila ingresada.
#'
#' @details
#' Se puede demostrar que la suma de los números en la n-ésima fila es igual a
#' n al cubo.
#'
#' @param n Número natural
#'
#' @return suma de los números ubicados en la n-ésima fila de la pirámide.
#'
#' @examples
#' suma_piramide(1)
#' suma_piramide(2)
#' suma_piramide(3)
#'
suma_piramide <- function(n) {
  return(n^3)
}
```

Ejemplos:

```
suma_piramide(1)
```

```
[1] 1
```

```
suma_piramide(2)
```

```
[1] 8
```

```
suma_piramide(3)
```

```
[1] 27
```

```
# Evaluamos la suma de cada una de las primeras 10 filas
for (n in 1:10) {
  suma <- suma_piramide(n)
  cat("Los impares de la fila", n, "suman", suma, "\n")
}
```

```
Los impares de la fila 1 suman 1
Los impares de la fila 2 suman 8
Los impares de la fila 3 suman 27
Los impares de la fila 4 suman 64
Los impares de la fila 5 suman 125
Los impares de la fila 6 suman 216
Los impares de la fila 7 suman 343
Los impares de la fila 8 suman 512
Los impares de la fila 9 suman 729
Los impares de la fila 10 suman 1000
```

26.6 Ejercicio 6

a. Paso a paso:

1. En el ambiente global primero se definen las funciones `f` y `g` y las variables globales `a` y `b`, con los valores 6 y 1, respectivamente.
2. Se invoca `g(a, b)`, resultando en que en el ambiente local de `g`, `x` recibe el valor 6 e `y` el valor 1.
3. En el ambiente local de `g` se crea la variable local `b`, con valor `6 - 2 * 1 = 4`.
4. Desde el ambiente local de `g` se invoca `f(b)`, donde `b` vale `4`.

5. En el ambiente local de `f`, `a` recibe el valor 4, el cual es actualizado por `(4-10)*(4+10)=-84` para finalmente devolver `-84`.
6. De regreso en el ambiente local de `g`, la variable `c` recibe el valor `b*f(b)=4*f(4)=4*(-84)=-336` y se devuelve `-336`.
7. Para definir la variable `d` se vuelve a invocar `f` esta vez sin argumentos explícitos, por lo que en el ambiente local de `f`, `a` recibe el valor 10 y `f() = 0`. El valor final de `d` es `f() - c = 0 - (-336) = 336`.
8. En el ambiente global entonces el resultado de `g(a, b)` es `336`.

b. En este ejemplo, el identificador `a` representa dos variables distintas: una de ellas definida en el *Global Environment* y la otra en el ámbito local de la función `f1`. Lo mismo ocurre con `x`: representa a una variable en el ambiente global, a otra en el ambiente de la función `f1` y a una tercera en el ambiente de la función `f2`. Al ejecutar el algoritmo se obtendrían los siguientes resultados:

1. En el ambiente global, se definen las funciones `f1` y `f2` y las variables `x` e `y` con valores: `x = 3` e `y = 5`.
2. Desde el ambiente global, se invoca a la función `f1`, donde el parámetro formal `a` recibe el valor del parámetro real `x` (`a = 3`), mientras que el parámetro formal `b` recibe el valor del parámetro real `y` (`b = 5`). Dentro de la función `f1`:
   a. Se crea una nueva variable `x` que recibe el valor `x = a + b = 8`. Esta `x` no tiene nada que ver con la del ambiente global.
   b. Se crea una nueva variable `y` que recibe el valor `y = x + 2 = 8 + 2 = 10`. Esta `y` no tiene nada que ver con la del ambiente global.
   c. La función devuelve el valor `y = 10`.
3. En el ambiente global, se le asigna a la variable `a` el valor recién devuelto: `a = 10`. Esta `a` no tiene nada que ver con la variable local de la función `f1`.
4. Desde el ambiente global, se llama a la función `f2`, donde el parámetro formal `x` recibe el valor del parámetro real `a` (`x = 10`). Esta `x` no tiene nada que ver con las anteriores. Dentro de la función `f2`:
   a. Se calcula el cuadrado de `x`: $10^2 = 100$
   b. Se devuelve el valor `100`.
5. En el ambiente global, se suma `x + f2(a) = 3 + 100 = 103` y se asigna este valor a `z`.
6. El algoritmo escribe el valor de `z`, 103.
7. El algoritmo intenta escribir `a + b = 10 + ?`, pero produce error, puesto que `b` no está definida en el ambiente global, no tiene asignado ningún valor. La única variable `b` que existe está en el ámbito de la función `f1`, no en el *Global Environment*.

c. Primero, en el ambiente global se define la función `f` con tres argumentos: `x` (sin valor por defecto), `y` y `z` (opcionales). La función devuelve una combinación lineal de estas tres cantidades. En segundo lugar, se invoca sucesivamente la función `f` para definir cuatro variables globales en el siguiente orden:

1. `a = f(10) = f(10, 5, 10 + 5) = (10+5) - 10 - 5 = 0`
2. `b = f(10, 10) = f(10, 10, 10 + 10) = (10+10) - 10 - 10 = 0`
3. `c = f(10, 10, 10) = 10 - 10 - 10 = -10`

4. `d = f(10, z = 10) = f(10, 5, 10) = 10 - 10 - 5 = -5`

Finalmente, se imprime el resultado de sumar las cuatro cantidades: `0 + 0 - 10 - 5 = -15`

26.7 Ejercicio 7

Hemos definido a las funciones como subalgoritmos que devuelven un objeto. En este caso no nos interesa devolver nada, sino sólo escribir mensajes en la consola, por eso podemos omitir el uso de `return()`:

```
#' Resolviendo ecuaciones de segundo grado
#'
#' @description
#' Encuentra las soluciones de una ecuación de segundo grado a partir del uso de
#' la formula resolvente. Se deben ingresar los coeficientes del polinomio de
#' segundo grado asociado.
#'
#' @details
#' La función imprime un mensaje con el detalle de las soluciones, incluyendo si
#' esta ecuación tiene dos soluciones distintas, una solución doble o ninguna
#' solución dentro de los números reales. La función devuelve un mensaje de error
#' si el coeficiente cuadrático es 0.
#'
#' @param a,b,c Números reales
#'
#' @return NULL, junto con un mensaje con el detalle de las soluciones.
#'
#' @examples
#' resolvente(1, -1, -2)
#' resolvente(1, 2, 1)
#' resolvente(1, 1, 1)
#' resolvente(0, 1, 1)
#'
resolvente <- function(a, b, c) {
  if (a == 0) {
    stop("(a) debe ser distinto de cero")
  }
  discriminante <- b^2 - 4 * a * c
  if (discriminante > 0) {
    x1 <- (-b - sqrt(discriminante)) / (2 * a)
    x2 <- (-b + sqrt(discriminante)) / (2 * a)
    cat("Hay dos soluciones reales", x1, "y", x2, "\n")
  } else if (discriminante == 0) {
    x1 <- -b / (2 * a)
    cat("Hay una solución real doble:", x1, "\n")
```

```
  } else {
    cat("Las soluciones son complejas.\n")
  }
}
```

Ejemplo de uso:

```
resolvente(1, -1, -2)
```

```
Hay dos soluciones reales -1 y 2
```

```
resolvente(1, 2, 1)
```

```
Hay una solución real doble: -1
```

```
resolvente(1, 1, 1)
```

```
Las soluciones son complejas.
```

```
resolvente(0, 1, 1)
```

```
Error in resolvente(0, 1, 1): (a) debe ser distinto de cero
```

Comentarios adicionales. Como hemos mencionado, cuando no se incluye un `return()`, igualmente la función devuelve algo, que es el resultado de la última expresión ejecutada. En todas las situaciones, en esta función la última expresión ejecutada es un `cat()`. Además de escribir un texto, esta función devuelve un valor `NULL` *invisible* (no se imprime en la consola, pero está). Podemos corrobar este comportamiento si usamos la función de esta forma:

```
# Imprime texto en la consola y devuelve un NULL, que se guarda enresultado
resultado <- resolvente(1, -1, -2)
```

```
Hay dos soluciones reales -1 y 2
```

```
# Imprimimos resultado y encontramos que tiene guardado un NULL
resultado
```

```
NULL
```

Sería interesante que esta función pueda devolver las soluciones. Hasta acá sabemos que las funciones pueden devolver un único objeto. Esto puede ser un inconveniente para este problema, ya que nos interesa devolver dos valores (dos soluciones reales), un valor (una solución real doble) o ningún valor (ninguna solución real). Más adelante veremos cómo hacer para poder devolver distinta cantidad de objetos, agrupándolos en otra estructura de datos.

26.8 Ejercicio 8

La solución propone chequear primeramente que el número de entrada sea entero y positivo. Si detecta que una de estos requisitos no se cumple, emite un *warning* y devuelve `FALSE`.

En caso de que se cumplan estos requisitos, el programa directamente devuelve `TRUE` si el número analizado es 2 o 3, ya que sabemos que estos primeros naturales son primos. Para el resto de los números, divide a `n` por todos los naturales desde el `2` hasta `n - 1`. Si al hacer esta división, encuentra un resto igual a cero, significa que `n` es compuesto. Si ninguna división produce resto cero, entonces `n` es primo.

Por ejemplo, para `n = 7`, se hace:

- `7 %% 2 = 1`, sigue.
- `7 %% 3 = 1`, sigue.
- `7 %% 4 = 3`, sigue.
- `7 %% 5 = 2`, sigue.
- `7 %% 6 = 1`, termina la iteración.
- Dado que no se encontraron divisores para 7, es un número primo, se devuelve `TRUE`

Para `n = 9`, se hace:

- `9 %% 2` = 1, sigue.
- `9 %% 3` = 0, devuelve FALSO.
- Dado que se encontró que 9 es múltiplo de 3, no es un número primo y se devolvió `FALSE`.

```
#' Evaluación de números primos
#'
#' @description
#' Determina si un número entero es primo o no.
#'
#' @details
#' La función devuelve un mensaje de advertencia si no se ingresa un número natural
#' mayor que 1. El resultado en este caso será FALSO.
#'
#' @param n Número natural
#'
#' @return Un valor lógico, TRUE si el número es primo, FALSE si no lo es.
#'
#' @examples
#' es_primo(47)
#' es_primo(253)
#' es_primo(2)
#'
es_primo <- function(n) {
  if (n %% 1 != 0) {
    warning("(n) no es entero")
```

```
    return(FALSE)
  }

  if (n <= 1) {
    warning("(n) no es mayor a 1")
    return(FALSE)
  }

    if (n > 3) {
        for (i in 2:(n - 1)) {
            if (n %% i == 0) {
                return(FALSE)
            }
        }
    }

  # solo se llega acá si no se devolvió FALSE antes, es decir, si n es primo
    return(TRUE)
}
```

Ejemplos de uso:

```
es_primo(47)
```

```
[1] TRUE
```

```
es_primo(253)
```

```
[1] FALSE
```

```
es_primo(2)
```

```
[1] TRUE
```

```
es_primo(7.18)
```

```
Warning in es_primo(7.18): (n) no es entero
```

```
[1] FALSE
```

```
es_primo(0)
```

```
Warning in es_primo(0): (n) no es mayor a 1
```

```
[1] FALSE
```

Observación: se puede plantear un algoritmo más eficiente, que realice menos iteraciones. No es necesario iterar hasta `n - 1`, si no hasta el entero inmediato menor a $\sqrt{n}$. Si no se encontró que `n` sea múltiplo de ningún valor menor a $\sqrt{n}$, tampoco lo será con los que siguen. Por otro lado, sería suficiente hacer las divisiones con respecto a los números primos menores $\sqrt{n}$. Se deja propuesto elaborar esta solución alternativa.

26.9 Ejercicio 9

Para poder resolver una división usando solamente sumas y restas, tenemos que pensar que, por ejemplo, hacer 14 dividido 3 nos da cociente 4 y resto 2, porque el 3 "entra" 4 veces en el 14 y todavía sobran 2. Es decir, a 14 le podemos restar el 3 cuatro veces hasta que ya no se lo podamos restar más, quedando un resto de 2. Entonces la idea es empezar diciendo que el *resto* es el dividendo (al principio, nos *resta* todo el dividendo) y restarle iterativamente el valor del divisor hasta que el resto se haga menor que el divisor. Mientras tanto, tenemos que ir contando cuántas restas se hacen, puesto que eso será el valor del cociente. Ejemplo:

- `dividendo = 14`, `divisor = 3`, `resto = 14`.
- `14 - 3 = 11`, digo que el `resto` es 11 y cuento que ya hice una resta con `cociente = 1`.
- `11 - 3 = 8`, digo que el `resto` es 8 y cuento que ya hice dos restas con `cociente = 2`.
- `8 - 3 = 5`, digo que el `resto` es 5 y cuento que ya hice tres restas con `cociente = 3`.
- `5 - 3 = 2`, digo que el `resto` es 2, cuento que ya hice cuatro restas con `cociente = 4`.
- Como ya obtuve un `resto` menor que el `divisor`, me detengo, con este resultado: `cociente = 4` y `resto = 2`.

Como a priori no sabemos cuántas iteraciones de este proceso tenemos que hacer, empleamos un `while`.

```
#' Cociente de la división de números naturales
#'
#' @description
#' Obtiene el cociente entero y el resto en la división de dos números naturales
#'
#' @details
#' La función imprime un mensaje con todos los detalles de la división, con el
#' dividendo, divisor, cociente y resto. Sin embargo, la función devuelve solo el
#' cociente entero.
#'
```

```r
#' @param dividendo,divisor Números naturales.
#'
#' @return El cociente entero de la división de ambos números.
#'
#' @examples
#' cociente(1253, 4)
#' cociente(3, 4)
#'
cociente <- function(dividendo, divisor) {
    resto <- dividendo
    cociente <- 0
    while (resto >= divisor) {
        cociente <- cociente + 1
        resto <- resto - divisor
    }
    cat("Dividendo:", dividendo, "\n")
    cat("Divisor:", divisor, "\n")
    cat("Cociente:", cociente, "\n")
    cat("Resto:", resto, "\n")
    return(cociente)
}
```

Ejemplos de uso:

```r
cociente(1253, 4)
```

```
Dividendo: 1253
Divisor: 4
Cociente: 313
Resto: 1
```

```
[1] 313
```

```r
cociente(3, 4)
```

```
Dividendo: 3
Divisor: 4
Cociente: 0
Resto: 3
```

```
[1] 0
```

26.10 Ejercicio 10

```r
#' Cálculo del máximo común divisor
#'
#' @description
#' Calcula el máximo común divisor entre dos números naturales
#'
#' @details
#' Aplica el algoritmo de Euclides para encontrar el MCD. Dividir al mayor por el
#' menor y registrar el resto. Si el resto es cero, el divisor es el MCD. Si el
#' resto no es cero, hay que dividir el divisor por el resto y hacer la misma
#' evaluación. Es decir, el divisor pasa a ser el nuevo dividendo y el resto, el
#' nuevo divisor. Así hasta obtener el resto cero y tener el MCD.
#'
#' @param a,b Números naturales.
#'
#' @return El máximo común divisor de ambos números.
#'
#' @examples
#' max_com_div(100, 24)
#' max_com_div(25, 100)
#' max_com_div(24, 24)
#'
max_com_div <- function(a, b) {

    # Establecer como dividendo al mayor y como divisor al menor
    if (a > b) {
        dividendo <- a
        divisor <- b
    } else {
        dividendo <- b
        divisor <- a
    }

    # Iniciar al resto como igual al dividendo
    resto <- dividendo

    # Aplicar algortimo de Euclides
    while (resto != 0) {

      # Dividir al mayor por el menor y registrar el resto
        resto <- dividendo %% divisor

        # Si el resto es cero (lo chequearemos en la condición del while), el
```

```
        # divisor es el mcd, por las dudas nos preparamos para entregarlo.
        mcd <- divisor

        # Si el resto no es cero (lo chequearemos en la condición del while), hay
        # que dividir el divisor por el resto y hacer la misma evaluación. Es decir,
        # el divisor pasa a ser el nuevo dividendo y el resto, el nuevo divisor
        dividendo <- divisor
        divisor <- resto
    }

    # Devolver mcd
    return(mcd)
}
```

Ejemplos de uso:

```
max_com_div(100, 24)
```

```
[1] 4
```

```
max_com_div(25, 100)
```

```
[1] 25
```

```
max_com_div(24, 24)
```

```
[1] 24
```

Otra forma:

```
max_com_div <- function(a, b) {
    # Establecer como dividendo al mayor y como divisor al menor
    if (a > b) {
        dividendo <- a
        divisor <- b
    } else {
        dividendo <- b
        divisor <- a
    }
    # Iniciar al resto como igual al dividendo
    resto <- dividendo
    # Aplicar algortimo de Euclides
```

```
    while (resto > 0) {
        resto <- dividendo %% divisor
        if (resto == 0) return(divisor)
        dividendo <- divisor
        divisor <- resto
    }
}
```

26.11 Ejercicio 11

b. En el script `funciones_unidad3.R` se implementa la función `combinatorio(m,n)` y se guarda junto con `fact()`. Su contenido debe ser:

```
# ------------------------------------------------------------------
# DEFINICIÓN DE FUNCIONES (funciones.R)
# ------------------------------------------------------------------

#' Cálculo de factoriales
#'
#' @description
#' Calcula el factorial de números enteros no negativos.
#'
#' @details
#' Produce un error si se quiere calcular el factorial de un número negativo.
#'
#' @param n Número entero no negativo para el cual se calcula el factorial.
#'
#' @return El factorial de n.
#'
#' @examples
#' fact(5)
#' fact(0)
#'
fact <- function(n) {
  if (n < 0 || n != floor(n)) {
    stop("n debe ser entero no negativo.")
  }
  resultado <- 1
  if (n > 0) {
      for (i in 1:n) {
          resultado <- resultado * i
      }
  }
  return(resultado)
```

```
}

#' Cálculo de números combinatorios
#'
#' @description
#' Calcula números combinatorios invocando a la función fact()
#'
#' @param m,n números naturales
#'
#' @return El número combinatorio m en n
#'
#' @examples
#' combinatorio(3, 1)
#'
combinatorio <- function(m, n) {
    return(fact(m) / (fact(m - n) * fact(n)))
}
```

c. El contenido del script `ejercicio11.R` puede ser:

```
# ------------------------------------------------------------------
# PROGRAMA: "Ejemplificar propiedades de los nros combinatorios"
# ------------------------------------------------------------------

# Cargar funciones
source("funciones_unidad3.R")
```

```
cat("Propiedad 1: C(m, 0) = 1. Ejemplo:\n")
res1 <- combinatorio(5, 0)
cat("C(5, 0) =", res1, "\n\n")

cat("Propiedad 2: C(m, m) = 1. Ejemplo:\n")
res1 <- combinatorio(5, 5)
cat("C(5, 5) =", res1, "\n\n")

cat("Propiedad 3: C(m, 1) = m. Ejemplo:\n")
res1 <- combinatorio(5, 1)
cat("C(5, 1) =", res1, "\n\n")

cat("Propiedad 4: C(m, n) = C(m, m-n). Ejemplo:\n")
res1 <- combinatorio(5, 2)
res2 <- combinatorio(5, 3)
cat("C(5, 2) =", res1, "\n")
cat("C(5, 3) =", res2, "\n\n")

cat("Propiedad 5: C(m, n) = C(m-1, n-1) + C(m-1, n). Ejemplo:\n")
```

```r
res1 <- combinatorio(5, 2)
res2 <- combinatorio(4, 1) + combinatorio(4, 2)
cat("C(5, 2) =", res1, "\n")
cat("C(4, 1) + C(4, 2) =", res2)
```

```
Propiedad 1: C(m, 0) = 1. Ejemplo:
C(5, 0) = 1

Propiedad 2: C(m, m) = 1. Ejemplo:
C(5, 5) = 1

Propiedad 3: C(m, 1) = m. Ejemplo:
C(5, 1) = 5

Propiedad 4: C(m, n) = C(m, m-n). Ejemplo:
C(5, 2) = 10
C(5, 3) = 10

Propiedad 5: C(m, n) = C(m-1, n-1) + C(m-1, n). Ejemplo:
C(5, 2) = 10
C(4, 1) + C(4, 2) = 10
```

d. A continuación, se prueba pasando un valor de `n` mayor que `m`:

```r
combinatorio(4, 5)
```

```
Error in fact(m - n): n debe ser entero no negativo.
```

Como se observa, la función falla porque internamente se invoca a la función `fact()` con un número negativo, y la misma fue programada para disparar un error en ese caso. El mensaje `n debe ser entero no negativo` no es demasiado claro, dado que es emitido por la función `fact()` y se refiere a su argumento `n`, no al `n` de `combinatorio()`. Podríamos mejorar esta situación de varias formas. Una de ellas podría ser disparar un error antes, dentro de la función `combinatorio()`, para evitar cualquier procesamiento si $n > m$:

```r
combinatorio <- function(m, n) {
  if (n > m) {
    stop("m debe ser menor o igual que n ")
  }
    return(fact(m) / (fact(m - n) * fact(n)))
}
```

e. Se generaliza la función `combinatorio(m, n)` para calcular números combinatorios con y sin reposición. Para esto se incluye el argumento `r`, que toma el valor lógico `TRUE` si el cálculo es con reposición o `FALSE` en caso contrario.

Agregamos lo siguiente al archivo `unidad3_funciones.R`. Notar que esta vez no usamos `return()`, pero igualmetne se devuelve el resultado deseado por ser lo último que se evalúa:

```r
#' Cálculo de números combinatorios
#'
#' @description
#' Calcula el número combinatorio m en n, con o sin reposición según r
#'
#' @param m,n números naturales
#' @param r valor lógico, sobre si el cálculo es con o sin repetición, F por defecto
#'
#' @return El número combinatorio m en n, con o sin reposición
#'
#' @examples
#' combinatorio2(10, 2, TRUE)
#' combinatorio2(10, 2)
#'
combinatorio2 <- function(m, n, r = FALSE) {
  if (r) {
    combinatorio(m + n - 1, n)
  } else {
    combinatorio(m, n)
  }
}
```

En el script `ejercicio_11.R` probamos la nueva función:

```r
# Números combinatorios con y sin reposición

m <- 5
n <- 2
cat(m, "tomados de a", n, "sin resposición:", combinatorio2(m, n), "\n")
cat(m, "tomados de a", n, "con resposición:", combinatorio2(m, n, TRUE))
```

```
5 tomados de a 2 sin resposición: 10
5 tomados de a 2 con resposición: 15
```

La cantidad de combinaciones con reposición siempre es mayor, salvo para `n = 1` donde ambas coinciden.

26.12 Ejercicio 12

La documentación se ha incluido en cada una de las respuestas anteriores.

Capítulo 27

Soluciones de la Práctica de la Unidad 4

Los scripts propuestos como solución a los problemas de esta práctica pueden ser descargados desde este archivo comprimido.

27.1 Ejercicio 1

a. Dirigirse a la carpeta `unidad_4\`, crear allí la carpeta principal `organizacion_usuarios` y entrar en ella:

```
cd ~tu_ruta\unidad_4
mkdir organizacion_usuarios
cd organizacion_usuarios
```

b. Crear las tres subcarpetas:

```
mkdir registrados temporales_abril historial
```

c. Para crear usuarios registrados, primero moverse a la carpeta `registrados` usando `cd` y luego crear los archivos:

```
cd registrados

echo IP: 192.168.1.1 > ABC1234A_R.txt
echo Usuario: juanperez >> ABC1234A_R.txt
echo Email: juanperez@email.com >> ABC1234A_R.txt

echo IP: 192.168.1.2 > DEF5678B_R.txt
echo Usuario: marialopez >> DEF5678B_R.txt
echo Email: marialopez@email.com >> DEF5678B_R.txt
```

```
echo IP: 192.168.1.3 > GHI9012C_R.txt
echo Usuario: carlosgomez >> GHI9012C_R.txt
echo Email: carlosgomez@email.com >> GHI9012C_R.txt
```

Notar el uso de `>>` para que el contenido a escribir se agregue como una línea nueva en el archivo, sin sobreescribir lo anterior.

Para crear usuarios temporales primero usar `cd ..` para volver a la carpeta `organizacion_usuarios` y de allí meterse a `temporales_abril`, para crear los archivos correspondientes:

```
cd ..\temporales_abril

echo IP: 192.168.2.1 > JKL3456D_T.txt
echo Nombre: invitado01 >> JKL3456D_T.txt

echo IP: 192.168.2.2 > MNO7890E_T.txt
echo Nombre: invitado02 >> MNO7890E_T.txt

echo IP: 192.168.2.3 > PQR1234F_T.txt
echo Nombre: invitado03 >> PQR1234F_T.txt
```

d. Supongamos que el usuario temporal `MN34OP56_T.txt` se ha registrado, debemos copiar el archivo del usuario a la carpeta `registrados`. Para eso primero volvemos a la carpeta principal `organizacion_usuarios` con `cd ..` y luego copiamos el archivo del usuario desde la carpeta `temporales_abril` hacia la carpeta `registrados`:

```
cd ..
copy temporales_abril\MN34OP56_T.txt registrados
```

Eliminar el archivo original de la carpeta`temporales_abril`:

```
del temporales_abril\MN34OP56_T.txt
```

Editar el archivo copiado en `registrados` para agregar datos (nombre de usuario e email):

```
echo Usuario: federicoruiz >> registrados/MN34OP56_T.txt
echo Email: federico@email.com >> registrados/MN34OP56_T.txt
```

Renombrar el archivo con sufijo `_R`:

```
rename registrados\MN34OP56_T.txt MN34OP56_R.txt
```

Mostrar contenido actualizado:

```
type registrados\MN34OP56_R.txt
```

e. Copiar a la carpeta historial `historial` cada uno de los archivos de la carpeta `temporales_abril`. Usamos `*` para copiar todos los archivos que existen (otra ocpión es hacerlo uno por uno, escribiendo su nombre y usando el comando `copy` varias veces):

```
copy temporales_abril\* historial
```

Eliminar la carpeta `temporales_abril` y crear `temporales_mayo`:

```
rmdir /s /q temporales\temporales_abril
mkdir temporales\temporales_mayo
```

Nota: en el comando `rmdir` la opción `/s` elimina la carpeta y todos sus contenidos (subcarpetas y archivos), mientras que `/q` omite la confirmación de eliminación, lo que hace que el comando se ejecute sin preguntarle al usuario si está seguro de eliminar.

27.2 Ejercicio 2

Una posible solución para este problema es incuir el siguiente contenido en el archivo `menu.R`:

```
opcion <- 0

while (opcion != 3) {
  cat("\n=== MENÚ PRINCIPAL ===\n\n")
  cat("1. Saludar\n")
  cat("2. Mostrar hora actual\n")
  cat("3. Salir\n\n")
  cat("Elegí una opción: ")

  opcion <- scan(file = "stdin", what = integer(), n = 1, quiet = TRUE)

  if (opcion == 1) {
    cat("\nIngresá tu nombre: ")
    nombre <- scan(file = "stdin", what = character(), n = 1, quiet = TRUE)
    cat("¡Hola, ", nombre, "!\n", sep = "")

  } else if (opcion == 2) {
    hora <- format(Sys.time(), "%H:%M:%S")
    cat("\nLa hora actual es:", hora, "\n")

  } else if (opcion == 3) {
    cat("\nIngresá tu nombre para despedirte: ")
    nombre <- scan(file = "stdin", what = character(), n = 1, quiet = TRUE)
    cat("¡Chau, ", nombre, "!\n", sep = "")

  } else {
    cat("\nOpción inválida. Probá de nuevo.\n")
  }
}
```

y ejecutarlo con:

```
Rscript menu.R
```

27.3 Ejercicio 3

El contenido del script `evaluar_pelicula.R` puede ser:

```
cat("\n=============================================\n")
cat("\n      SISTEMA DE EVALUACIÓN DE PELÍCULAS\n")
cat("\n=============================================\n\n")

cat("Ingrese la cantidad de jueces en el grupo:\n")
n <- scan(file = "stdin", what = integer(), n = 1, quiet = TRUE)

if (n < 3 || n > 6) {
  cat("Cantidad inválida de jueces. Debe ser entre 3 y 6.\n")
} else {
  cat("\nIngrese el nombre de la película:\n")
  nombre <- scan(file = "stdin", what = character(), n = 1, quiet = TRUE, sep = "*+-")
  # Nota: scan separa el input donde ve espacios en blanco, para que no lo haga
  # cambiamos el valor que tiene en su argumento sep, poniendo cualquier cosa
  # que esperamos que nunca aparezca en un nombre de película, para que ningún
  # nombre sea nunca separado.
  suma <- 0
  for (i in 1:n) {
    cat("\nIngrese la calificación del juez ", i, ":\n", sep = "")
    nota <- scan(file = "stdin", what = numeric(), n = 1, quiet = TRUE)
    while (nota < 0 || nota > 10) {
      cat("Calificación inválida. Las calificaciones deben ser entre 1 y 10.\nIngrésela nuevamente:\
      nota <- scan(file = "stdin", what = numeric(), n = 1, quiet = TRUE)
    }
    suma <- suma + nota
  }
  promedio <- suma / n
  cat(
    "\nLa clasificación promedio para la película <", nombre, "> es ",
    round(promedio, 2), " puntos.\n\n", sep = ""
  )
}
```

Debe ser ejecutado con:

```
Rscript evaluar_pelicula.R
```

27.4 Ejercicio 4

a. **Versión con intentos ilimitados**. El contenido del script puede ser:

```
cat("\n================================================\n")
cat("\n  BIENVENIDO/A AL AULA VIRTUAL DE PROGRAMACIÓN 1  \n")
cat("\n================================================\n\n")

contrasenia_correcta <- "amoprogramar"
ingresada <- ""

while (ingresada != contrasenia_correcta) {
  cat("\nIngrese la contraseña:\n")
  ingresada <- scan(file = "stdin", what = character(), n = 1, quiet = TRUE)
  if (ingresada != contrasenia_correcta) {
    cat("Contraseña incorrecta. Ingrésela nuevamente.\n")
  }
}

cat("\n¡Contraseña correcta! Puede ingresar y continuar con sus estudios.\n\n")
```

b. **Versión con intentos limitados (5 intentos)**. El contenido del script puede ser:

```
cat("\n================================================\n")
cat("\n  BIENVENIDO/A AL AULA VIRTUAL DE PROGRAMACIÓN 1  \n")
cat("\n================================================\n\n")

contrasenia_correcta <- "amoprogramar"
ingresada <- ""
intentos_restantes <- 5

while (intentos_restantes > 0 && ingresada != contrasenia_correcta) {
  cat("\nIngrese la contraseña:\n")
  ingresada <- scan(file = "stdin", what = character(), nmax = 1, quiet = TRUE)

  if (ingresada != contrasenia_correcta) {
    intentos_restantes <- intentos_restantes - 1
    if (intentos_restantes > 0) {
      cat("Contraseña incorrecta. Intentos restantes:", intentos_restantes, "\n")
    } else {
      cat("\nDemasiados intentos fallidos. Su cuenta ha sido bloqueada.\n\n")
    }
  }
```

```
}

if (ingresada == contrasenia_correcta) {
  cat("\n¡Contraseña correcta! Puede ingresar y continuar con sus estudios.\n\n")
}
```

27.5 Ejercicio 5

El contenido del script `resolvente.R` puede ser:

```
# Mensaje de bienvenida
cat("========================================\n")
cat("  Calculadora de ecuaciones cuadráticas\n")
cat("========================================\n\n")

# Leer coeficientes desde la terminal
cat("Ingrese el valor del coeficiente 'a':\n")
a <- scan(file = "stdin", n = 1, quiet = TRUE)
if (a == 0) {
  stop("El coeficiente 'a' debe ser distinto de cero. Fin del programa.\n")
}
cat("Ingrese el valor del coeficiente 'b':\n")
b <- scan(file = "stdin", n = 1, quiet = TRUE)
cat("Ingrese el valor del coeficiente 'c':\n")
c <- scan(file = "stdin", n = 1, quiet = TRUE)

# Mostrar la ecuación de forma legible
ecuacion <- paste0(a, "x²", ifelse(b < 0, "", "+"), b, "x", ifelse(c < 0, "", "+"), c, "=0")
cat("\nEcuación ingresada: ", ecuacion, "\n")

# Calcular el discriminante
D <- b^2 - 4 * a * c
cat("Discriminante (D):", round(D, 3), "\n")

# Calcular soluciones
if (D > 0) {
  x1 <- (-b - sqrt(D)) / (2 * a)
  x2 <- (-b + sqrt(D)) / (2 * a)
  cat("Existen dos soluciones reales distintas:\n")
  cat("x  =", round(x1, 3), ", x  =", round(x2, 3), "\n")
} else if (D == 0) {
  x <- -b / (2 * a)
  cat("Existe una única solución real (raíz doble):\n")
  cat("x =", round(x, 3), "\n")
```

```
} else {
  cat("Las soluciones son complejas (no reales).\n")
}

# Mensaje de despedida
cat("\nGracias por usar la calculadora de ecuaciones cuadráticas.\n")
```

Observaciones:

- La función `ifelse()` permite implementar en una sola línea una estructura *if-else* sencilla. El primer argumento es la condición lógica a evaluar, el segundo es el valor devuelto en caso de que la condición sea `TRUE` y el tercero, el valor devuelto en caso de que sea `FALSE`. Esta función se usó para agregar un signo `+` delante de los coeficientes `b` y `c` si son positivos.
- Se puede pensar en formas más elaboradas para escribir la expresión de la ecuación, que hagan un mejor manejo de espacios o que omitan los términos cuyos coeficientes son nulos.
- Se usó la función `stop()` para detener la ejecución del script si el coeficiente asociado al término de segundo grado es 0.

27.6 Ejercicio 6 (opcional)

El contenido del script `resolvente_args.R` puede ser:

```
# Mensaje de bienvenida
cat("========================================\n")
cat("  Calculadora de ecuaciones cuadráticas\n")
cat("========================================\n")

# Capturar los argumentos pasados desde la terminal en un vector
args <- commandArgs(trailingOnly = TRUE)

# Si no proveyó suficientes argumentos, generar un error y que se detenga el programa
if (length(args) < 3) {
  stop("Debe proveer tres valores para los coeficientes de la ecuación.")
}

# Convertir a numérico todos los argumentos
a <- as.numeric(args[1])
b <- as.numeric(args[2])
c <- as.numeric(args[3])

# Si alguno no era numérico, no se pudo convertir y es NA
if (is.na(a) || is.na(b) || is.na(c)) {
  stop("Los argumentos ingresados no son valores numéricos.")
}
```

```r
# Si a = 0, genera un error y se detiene
if (a == 0) {
  stop("Error: el coeficiente 'a' debe ser distinto de cero. Fin del programa.\n")
}

# Mostrar la ecuación de forma legible
ecuacion <- paste0(a, "x²", ifelse(b < 0, "", "+"), b, "x", ifelse(c < 0, "", "+"), c, "=0")
cat("\nEcuación ingresada: ", ecuacion, "\n")

# Calcular el discriminante
D <- b^2 - 4 * a * c
cat("Discriminante (D):", round(D, 3), "\n")

# Calcular soluciones
if (D > 0) {
  x1 <- (-b - sqrt(D)) / (2 * a)
  x2 <- (-b + sqrt(D)) / (2 * a)
  cat("Existen dos soluciones reales distintas:\n")
  cat("x  =", round(x1, 3), ", x  =", round(x2, 3), "\n")
} else if (D == 0) {
  x <- -b / (2 * a)
  cat("Existe una única solución real (raíz doble):\n")
  cat("x =", round(x, 3), "\n")
} else {
  cat("Las soluciones son complejas (no reales).\n")
}

# Mensaje de despedida
cat("\nGracias por usar la calculadora de ecuaciones cuadráticas.\n")
```

Trabajo Práctico

Introducción

El “Juego de la Memoria”, también conocido como “Memotest”, es un clásico juego de mesa que pone a prueba la capacidad de atención y la memoria visual de los participantes. Su dinámica es simple pero atrapante: se colocan fichas o cartas boca abajo, con imágenes ocultas, y los jugadores deben voltearlas de a dos por turno, intentando encontrar pares coincidentes. Si las dos fichas tienen la misma imagen, el jugador las retira del tablero y puede seguir jugando. Si no coinciden, se vuelven a colocar boca abajo, y es turno del siguiente jugador.

Este juego ha sido disfrutado por generaciones tanto en el ámbito familiar como en actividades educativas, ya que además de ser entretenido, fomenta habilidades cognitivas importantes. En este trabajo práctico, vamos a programar en R una versión digital y simplificada del juego, que se podrá jugar en solitario desde la terminal de la computadora.

Objetivo

El objetivo de este trabajo práctico es desarrollar una versión interactiva del Juego de la Memoria en R, que funcione desde la terminal. El programa permitirá jugar con un tablero de fichas ocultas, revelarlas de a pares, y buscar coincidencias hasta completar todos los pares disponibles. El juego debe mostrar el estado del tablero en cada momento, registrar la cantidad de intentos realizados y permitir al jugador reiniciar la partida si lo desea.

Dinámica del juego

Para comprender cómo debe funcionar el programa, deben mirar el video grabado con una *demo*. El tablero estará compuesto por una cantidad par de fichas organizadas en una grilla rectangular (por ejemplo, 4 filas por 5 columnas). Cada ficha tiene un símbolo, letra o texto oculto, y hay exactamente dos fichas con cada símbolo.

En cada turno, el jugador debe ingresar la posición de dos fichas para descubrir. El programa debe mostrar el tablero con las dos fichas seleccionadas visibles. Si las fichas coinciden, permanecen visibles. Si no coinciden, el jugador debe memorizar su ubicación y el programa espera hasta que el jugador le diga que está listo para continuar. El juego continúa hasta que el jugador haya encontrado todos los

pares. Al finalizar, se muestra la cantidad de intentos realizados y se ofrece la posibilidad de jugar otra partida.

La terminal

El programa es interactivo y requiere que el usuario ingrese datos en tiempo real: configurar el tablero, elegir posiciones del tablero para descubrir, decidir si desea volver a jugar, etc. Por ello, el programa debe ejecutarse desde la **terminal** de la computadora, donde pueden escribirse comandos y visualizarse respuestas del sistema. Es importante familiarizarse con la Unidad 4 de la asignatura para lograr este funcionamiento del programa.

Materiales provistos

Para realizar este trabajo práctico cuentan con los siguientes archivos de apoyo:

- **Script** `jugar.R`: archivo principal donde deben escribir su programa. Contiene una plantilla con instrucciones y espacio para agregar los datos del equipo y el código del juego.
- **Paquete** `memoria`: este paquete creado por la cátedra ofrece funciones auxiliares que simplifican muchas partes de la solución de este trabajo y serán de extrema utilidad. Se recomienda leer la documentación del paquete con atención y correr los ejemplos. Las funciones disponibles son: `nuevo_tablero`, `validar_fichas`, `mostrar_tablero`, `texto_lento`, `limpiar_consola`, `leer_eleccion`, `mostrar_info`, `posiciones_disponibles` y `mostrar_pares`. En el script `jugar.R` se incluyen instrucciones de instalación.

Descomposición algorítmica

Para organizar el código de manera clara y modular, se recomienda aplicar el principio de descomposición algorítmica, creando funciones propias para distintas partes del juego. Estas funciones pueden definirse en `jugar.R` o, mejor aún, en scripts adicionales. El o los nuevos scripts deberán estar guardados en el mismo directorio que `jugar.R`, desde donde se los debe invocar con la función `source()`. Desde dicha carpeta, deben abrir la terminal y ejecutar el programa con `Rscript jugar.R`, sin que se produzcan errores.

Todas las funciones creadas deben estar documentadas bajo el sistema **Roxygen**. Algunas podrán tomar argumentos, otras no, y algunas devolverán valores mientras que otras sólo imprimirán mensajes en pantalla. Lo importante es que cada función cumpla una tarea bien definida y facilite la lectura del programa principal.

Otras indicaciones y sugerencias

- Es conveniente diseñar el algoritmo antes de programar, es decir, tomarse tiempo para pensar cómo es la lógica de la estructura del juego, qué pequeñas partes se pueden ir programando y probando de a poco, etc.
- No dejar para último momento. Empezar con inmediatez, ya que no se resuelve de un día para el otro y es posible que necesiten realizar consultas con los docentes. Si dejan para último momento ya no habrá tiempo de organizar consultas, no podrán entregar el trabajo práctico y quedarán libres en la asignatura.
- Probar el programa por partes, para asegurarse que cada cosa que van agregando funcione. Tratar de armar todo y luego evaluar puede hacer muy difícil a la tarea de detectar errores.
- Utilicen comentarios para identificar distintas partes de su programa y para describir qué se prentende realizar en cada sección.
- El objetivo estará cumplido si logran un programa similar al presentado en la *demo*. Opcionalmente, pueden incluir otros agregados que les resulte de interés.
- Si no logran programar todo lo pedido, igualmente realicen la entrega de lo que hayan podido hacer y los docentes evaluarán si es suficiente o no para aprobar.
- No intentar resolver todo de a una vez. Agregar pequeñas partes de a poco. Utilizar la emisión de mensajes temporarios para ir probando el código.

Equipos

El trabajo práctico se resuelve en grupos de **exactamente CUATRO integrantes**, de cualquier comisión. Hay tiempo hasta el **lunes 19/5/25** para informar la composición de cada equipo en este foro de Comunidades. Asimismo, si no tenés equipo o faltan integrantes en tu grupo, hay tiempo hasta esa fecha para comentar la situación en ese mismo foro. El día martes 20/5 se tomarán del foro los nombres de aquellos que hayan manifestado interés de hacer el TP pero no pudieron formar equipo, y entre ellos se crearán al azar nuevos equipos y/o se completarán también al azar los grupos incompletos. El armado de estos equipos será inapelable.

El script `jugar.R` tiene un espacio destinado para escribir los nombres de los integrantes del equipo. **Deben colocar allí los nombres de quienes hayan efectivamente participado del trabajo**. Comunidades les permite a todos los integrantes de un equipo ver los archivos que se han entregado, de modo que cada estudiante sabe si su nombre figura o no en el script. Si en el mismo falta el nombre de un estudiante y los docentes no son contactados al respecto, se entenderá que dicha persona no realizó el trabajo y quedará libre.

Código de conducta

- La discusión entre integrantes de distintos grupos está permitida, no así el intercambio de código. Se puede debatir en el foro de Comunidades o en otros medios, pero no publicar partes de código.
- Las entregas son sometidas a un software de detección de plagio, si se detectan similaridades sospechosas en el trabajo de distintos grupos, podrán quedar descalificados.

- Cuando se detecta un uso llamativo de herramientas de Inteligencia Artificial o estructuras de programación que no se corresponde con los usos y funciones básicas vistas en este curso introductorio, los estudiantes son convocados por los docentes para defender y explicar su entrega de forma oral y responder preguntas sobre la resolución entregada.

Entrega

Deben entregar el archivo `jugar.R` y cualquier otro script que generen, en la sección destinada para tal fin del aula virtual. Es suficiente con que un integrante del equipo adjunte los archivos. El resto de los integrantes podrá ver la entrega y, más adelante, la devolución y calificación por parte de los docentes.

Fecha límite de entrega: martes 10/06 a las 08:00 No se reciben trabajos fuera de este horario límite, sin ningún tipo de excepción.

Evaluación

La evaluación tendrá en cuenta:

- Si el programa funciona y permite jugar.
- Si el programa implementa el juego con el comportamiento presentado en la demo.
- Si hay prolijidad, claridad y buen uso de comentarios en el código.
- Si el trabajo muestra ser original y no copia de otros recursos.

Bibliografía

- Grolemund, G. (2014). Hands-On Programming with R. O'Reilly. Disponible online: https://rstudio-education.github.io/hopr/.
- Moeller, J. (2013). The Windows Command Line Beginner's Guide. Azure Flame Media. 2nd Ed.
- Paradis, E. (2005). R para Principiantes. Universit´e Montpellier II. Disponible online: https://cran.r-project.org/doc/contrib/rdebuts_es.pdf.
- Peng, R.; Kross, S.; Anderson, B. (2020). Mastering Software Development in R. Leanpub. Disponible online: https://bookdown.org/rdpeng/RProgDA/.
- Santana, S.; Mateos Farfán, E. (2014). El arte de programar en R: un lenguaje para la estadística. Instituto Mexicano de Tecnología del Agua, UNESCO. Disponible online: https://cran.r-project.org/doc/contrib/Santana_El_arte_de_programar_en_R.pdf.
- Shotts, W. (2024). The Linux Command Line: A Complete Introduction. No Starch Press. 6th Internet Edition. Disponible online: https://sourceforge.net/projects/linuxcommand/files/TLCL/24.11/TLCL-24.11.pdf/download.
- Wickham, H. (2019). Advanced R. Chapman and Hall/CRC. 2nd Ed. Disponible online: https://adv-r.hadley.nz/.
- Wickham, H.; Bryan, J. (2023). R Packages. O'Reilly. 2nd Ed. Disponible online: https://r-pkgs.org/.
- Wickham, H.; Cetinkaya-Rundel, M.; Grolemund, G. (2023). R para Ciencia de Datos. O'Reilly. 2da Ed. Disponible online: https://es.r4ds.hadley.nz/.

Íconos creados por Smashicons and by juicy_fish y tomados de Flaticon.