Guía Breve: R para Ciencias de la Vida

Coordinador: Dr. Fernando Hernández

Docentes: Dr. Fernando Hernández, Lic. Rafael Puche

Este documento es una guía resumida de los temas clave del curso "INM-211 R para Ciencias de la Vida". Cada sección presenta una introducción teórica, ejemplos aplicados y ejercicios.

IVIC, Junio de 2025

Módulo 1: Fundamentos de R y RStudio

1. Introducción a R

- **Fundamento Teórico:** R es un lenguaje de programación y un entorno de software libre enfocado en el cálculo estadístico y la generación de gráficos. Es ampliamente utilizado en la investigación científica, especialmente en bioestadística, bioinformática y ecología.
    - **Ventajas:** Gratuito, código abierto, gran comunidad de usuarios, miles de paquetes especializados (CRAN, Bioconductor), potente para manipulación de datos y gráficos de alta calidad.
    - **Desventajas:** Curva de aprendizaje inicial puede ser pronunciada, manejo de memoria puede ser un desafío con datos muy grandes (aunque mejora constantemente).
    - **Aplicaciones en Ciencias de la Vida:** Análisis de expresión génica, modelado de enfermedades, análisis de secuencias de ADN/proteínas, estudios epidemiológicos, visualización de datos biológicos complejos.
- **Ejemplo:**
    - Un investigador quiere analizar si un nuevo fármaco reduce la presión arterial en pacientes. Usaría R para importar los datos de presión arterial antes y después del tratamiento, realizar pruebas estadísticas (como una prueba t), y generar gráficos para visualizar los resultados.
- **Ejercicio en Clase:**
    - Discusión grupal: ¿Qué tipo de análisis de datos biológicos o de salud les interesa realizar? ¿Cómo creen que R podría ayudarles?
- **Ejercicio para Casa:**
    - Investigar y nombrar tres paquetes de R específicos para el análisis de datos en su área de interés dentro de las ciencias de la vida (ej. genómica, epidemiología, ecología médica). Describir brevemente para qué sirve cada uno.

2. Instalación de R y RStudio

- **Fundamento Teórico:** R es el motor, y RStudio es un Entorno de Desarrollo Integrado (IDE) que facilita enormemente el trabajo con R. RStudio provee una consola, un editor de scripts, herramientas para visualización, depuración y gestión de proyectos.
    - **Instalación:** Se debe instalar R primero, y luego RStudio. Ambos están disponibles para Windows, macOS y Linux. (Referirse a la guía de instalación detallada previamente proporcionada).
- **Ejercicio en Clase**

- Abrir RStudio. Identificar las 4 ventanas principales:
    1. **Editor/Script:** Donde se escriben y guardan los comandos.
    2. **Consola:** Donde se ejecutan los comandos y se ven salidas inmediatas.
    3. **Entorno/Historial:** Muestra los objetos creados y el historial de comandos.
    4. **Archivos/Gráficos/Paquetes/Ayuda/Visor:** Pestañas multifunción.
- Escribir `1+1` en la consola y presionar Enter.
- Crear un nuevo script de R (File > New File > R Script). Escribir `print("Hola Mundo Biológico")` en el script y ejecutar la línea (Ctrl+Enter o Cmd+Enter).

- **Ejercicio para Casa:**
    - Explorar el menú "Tools" > "Global Options" en RStudio. Identificar al menos tres opciones de personalización del apariencia o comportamiento de RStudio que les parezcan útiles.

3. Primeros Pasos: Consola, Editor, Ayuda, Paquetes

- **Fundamento Teórico:**
    - **Consola:** Para ejecución interactiva de comandos.
    - **Editor:** Para escribir y guardar scripts (secuencias de comandos) que pueden ser ejecutados en su totalidad o por partes. Es la forma recomendada de trabajar para asegurar la reproducibilidad.
    - **Ayuda:** R tiene un sistema de ayuda robusto.
        - `?nombre_funcion` o `help(nombre_funcion)`: Muestra la ayuda de una función específica.
        - `?? "palabra clave"` o `help.search("palabra clave")`: Busca funciones relacionadas con una palabra clave.
    - **Paquetes (Packages):** Colecciones de funciones, datos y documentación que extienden las capacidades de R.
        - `install.packages("nombre_paquete")`: Instala un paquete desde CRAN.
        - `library(nombre_paquete)`: Carga un paquete en la sesión actual para poder usar sus funciones.
- **Ejemplo (Biología/Salud):**
    - Supongamos que queremos analizar datos de diversidad de especies. Podríamos necesitar el paquete `vegan`.

```
# Instalar el paquete (solo se hace una vez)
# install.packages("vegan")

# Cargar el paquete (cada vez que se inicia una nueva sesión de R y se quiere usar)
library(vegan)

# Buscar ayuda para una función del paquete vegan, por ejemplo, para análisis de diversidad
?diversity
```

- **Ejercicio en Clase:**

1. Usar la consola para calcular `5 * (10 - 3)`.
2. En un script, asignar el resultado anterior a una variable llamada `resultado_calculo`.
3. Imprimir el valor de `resultado_calculo` en la consola usando la función `print()`.
4. Buscar ayuda para la función `mean` (calcula el promedio).
5. Instalar el paquete `ggplot2` (si hay tiempo y conexión a internet). Si no, discutir los pasos.

- **Ejercicio para Casa:**
  1. Crear un script de R que realice las siguientes operaciones:
     - Asignar su edad a una variable.
     - Asignar el número de cromosomas humanos (46) a otra variable.
     - Calcular la suma de estas dos variables y guardarla en una tercera variable.
     - Imprimir el valor de la tercera variable.
  2. Buscar ayuda para la función `sum()`. ¿Cuál es su propósito principal?

Módulo 2: Tipos y Estructuras de Datos en R

4. Tipos de Datos

- **Fundamento Teórico:** R maneja diferentes tipos de datos.
  - **Atómicos (elementales):**
    - `integer`: Números enteros (ej. `10L`, donde `L` indica que es un entero).
    - `numeric` (o `double`): Números reales/decimales (ej. `3.1416`, `10`).
    - `complex`: Números complejos (ej. `2+3i`).
    - `logical`: Valores booleanos (`TRUE` o `FALSE`, o sus abreviaciones `T` o `F`).
    - `character`: Cadenas de texto (ej. `"ADN"`, `'proteina'`).
  - **Casos Especiales:**
    - `NULL`: Representa un objeto vacío o la ausencia de valor.
    - `NA` (Not Available): Indica un valor perdido o ausente. Muy común en datos reales.
    - `NaN` (Not a Number): Resultado de operaciones matemáticas indefinidas (ej. `0/0`).
    - `Inf`, `-Inf`: Infinito positivo o negativo (ej. `1/0`).
  - **Datos Heterogéneos (pueden contener diferentes tipos de datos atómicos):**
    - `list`: Colecciones ordenadas de objetos, donde cada objeto puede ser de un tipo diferente.
    - `data.frame` (Marco de datos): Estructura tabular (filas y columnas) donde cada columna puede ser de un tipo diferente, pero dentro de una misma columna todos los elementos deben ser del mismo tipo. Es la estructura más usada para análisis de datos.
- **Ejemplo (Biología/Salud):**

```
# Tipos atómicos
numero_pacientes <- 25L       # integer
concentracion_farmaco <- 0.5  # numeric
```

```r
gen_activo <- TRUE            # logical
nombre_bacteria <- "E. coli"  # character

# Casos especiales
datos_glucosa_faltantes <- NA

# Lista (heterogénea)
info_experimento <- list(
  id_experimento = "EXP001",
  fecha = "2025-06-04",
  numero_muestras = 10L,
  tratamiento_aplicado = TRUE,
  notas = "Resultados preliminares"
)

# DataFrame (se verá en detalle más adelante)
datos_pacientes <- data.frame(
  id_paciente = c("P01", "P02", "P03"),
  edad = c(45L, 52L, 38L),
  sexo = c("Masculino", "Femenino", "Masculino"),
  glucosa_mg_dl = c(102.5, NA, 98.0)
)
```

- **Ejercicio en Clase:**
  1. Crear variables en R para los siguientes datos y usar la función `class()` para verificar su tipo:
     - El número de aminoácidos en una proteína corta (ej. 150). (Pista: usar `L` si se quiere entero)
     - El pH de una solución (ej. 7.4).
     - Si un gen está mutado o no (ej. `TRUE` o `FALSE`).
     - El nombre de un virus (ej. "Influenza A").
  2. Crear una lista llamada `info_muestra` que contenga: un ID de muestra (texto), la fecha de recolección (texto), y el volumen en ml (numérico).
- **Ejercicio para Casa:**
  1. Investigar la diferencia entre `NA` y `NULL` en R. Proporcionar un ejemplo de cuándo se podría usar cada uno en un contexto biológico.
  2. Crear un vector con los siguientes valores: `10, 20, NA, 40, 50`. Calcular la media de este vector usando `mean()`. ¿Qué ocurre? Buscar cómo la función `mean()` maneja los `NA` (pista: ver su ayuda con `?mean`).

5.a. Estructuras de Datos: Vectores

- **Fundamento Teórico:** Un vector es una secuencia ordenada de elementos del **mismo tipo atómico**. Son la estructura de datos más fundamental en R.
  - **Creación:**
    - `c()`: Función para combinar/concatenar elementos. Ej: `pesos <- c(60.5, 72.1, 55.8)`
    - `:`: Para secuencias de enteros. Ej: `1:5` crea `1 2 3 4 5`.
    - `seq()`: Para secuencias más complejas. Ej: `seq(from=0, to=1, by=0.2)`

        - `rep()`: Para replicar elementos. Ej: `rep("A", 3)` crea `"A" "A" "A"`.
    - Nombres: Se pueden asignar nombres a los elementos de un vector.

      names(vector) <- c("nombre1", "nombre2", ...)
    - **Operaciones:**
        - **Wise-element (elemento a elemento):** Operaciones aritméticas y lógicas entre vectores de igual longitud se realizan elemento a elemento. Ej: `v1 + v2`.
        - **Coerción:** Si se combinan diferentes tipos de datos en un vector, R los "coercerá" al tipo más general (logical -> integer -> numeric -> complex -> character). Ej: `c(1, "dos", TRUE)` se convertirá en un vector de caracteres: `"1", "dos", "TRUE"`.
        - **Reciclaje:** Si se opera con vectores de diferente longitud, R "recicla" los elementos del vector más corto hasta igualar la longitud del más largo (esto puede generar advertencias si la longitud del más largo no es múltiplo de la del más corto).
    - **Filtrado (Subsetting):** Acceder a elementos específicos.
        - Por índice numérico: `vector[3]`, `vector[c(1, 5)]`, `vector[1:3]`
        - Por índice lógico: `vector[vector > valor]`, `vector[condicion_logica]`
        - Por nombres (si los tiene): `vector["nombre_elemento"]`
    - Factores: Tipo especial de vector para almacenar variables categóricas (nominales u ordinales). Internamente, R almacena los factores como enteros, pero muestra las etiquetas de texto. Útil para modelos estadísticos.

      factor(vector_caracteres, levels = c(...), ordered = TRUE/FALSE)
- **Ejemplo (Biología/Salud):**

```r
# Vector de alturas de plantas en cm
alturas_plantas <- c(10.2, 11.5, 9.8, 10.5, 12.1)

# Vector de tipos sanguíneos (ejemplo de factor)
tipos_sanguineos_pacientes <- c("A", "O", "B", "A", "AB", "O", "O")
factor_tipos_sanguineos <- factor(tipos_sanguineos_pacientes)
print(factor_tipos_sanguineos)
summary(factor_tipos_sanguineos) # Muestra la frecuencia de cada nivel

# Vector de expresión génica (log2 fold change)
expresion_gen <- c(1.5, -0.8, 2.1, 0.3, -1.2)
names(expresion_gen) <- c("GenA", "GenB", "GenC", "GenD", "GenE")

# Filtrado: genes con sobreexpresión (log2 fold change > 1)
genes_sobreexpresados <- expresion_gen[expresion_gen > 1]
print(genes_sobreexpresados)

# Operaciones: convertir alturas a metros
alturas_metros <- alturas_plantas / 100
print(alturas_metros)
```

- **Ejercicio en Clase:**
    1. Crear un vector llamado `temperaturas_corporales` con los siguientes valores (en °C): `36.5, 37.0, 36.8, 37.2, 36.9`.
    2. Calcular la temperatura promedio usando `mean()`.

3. Crear un vector lógico que indique cuáles temperaturas son mayores a 37.0 °C.
4. Usar el vector lógico para filtrar y mostrar solo las temperaturas mayores a 37.0 °C.
5. Crear un vector de caracteres con los días de la semana de lunes a viernes. Asignar estos nombres al vector `temperaturas_corporales`. Mostrar el vector con nombres.

- **Ejercicio para Casa:**
  1. Crear un vector con los pesos (en kg) de 5 ratones de laboratorio.
  2. Crear otro vector con los pesos de los mismos 5 ratones una semana después tras un tratamiento.
  3. Calcular la diferencia de peso para cada ratón (peso_después - peso_antes).
  4. Investigar la función `length()`. ¿Para qué sirve? Aplicarla a uno de los vectores de peso.
  5. Crear un factor a partir de un vector que represente el estado de salud de 10 pacientes ("Sano", "Enfermo", "En Recuperación"). Asegurarse de que los niveles del factor tengan un orden lógico si aplica (ej. "Sano" < "En Recuperación" < "Enfermo"). Usar `summary()` para ver la distribución.

5.b. Estructuras de Datos: Matrices

- **Fundamento Teórico:** Una matriz es una estructura de datos bidimensional (filas y columnas) donde **todos los elementos deben ser del mismo tipo atómico**.
  - **Creación:**
    - `matrix(data, nrow, ncol, byrow = FALSE)`: `data` es un vector con los datos, `nrow` el número de filas, `ncol` el número de columnas. `byrow = TRUE` llena la matriz por filas, `FALSE` (por defecto) por columnas.
  - **Operaciones con Matrices:**
    - Operaciones aritméticas elemento a elemento (si las dimensiones son compatibles).
    - Funciones como `t()` (transpuesta), `dim()` (dimensiones), `nrow()`, `ncol()`.
  - **Operaciones Matriciales de Álgebra Lineal:**
    - `%*%`: Multiplicación de matrices.
    - `solve()`: Inversa de una matriz (si existe), o resuelve sistemas de ecuaciones lineales.
    - `eigen()`: Autovalores y autovectores.
  - **Filtrado:** `matriz[fila, columna]`. Se pueden usar vectores para seleccionar múltiples filas/columnas. `matriz[1,]` selecciona toda la fila 1. `matriz[,2]` selecciona toda la columna 2.
- **Ejemplo (Biología/Salud):**

```
# Matriz de conteo de células (3 muestras, 2 tipos de células)
# Datos: Muestra1(TipoA=100, TipoB=50), Muestra2(TipoA=120, TipoB=60),
Muestra3(TipoA=90, TipoB=45)
datos_celulas <- c(100, 120, 90, 50, 60, 45)
matriz_celulas <- matrix(datos_celulas, nrow = 3, ncol = 2, byrow = FALSE,
                         dimnames = list(c("Muestra1", "Muestra2",
"Muestra3"),
                                         c("CelulaTipoA", "CelulaTipoB")))
```

```r
print(matriz_celulas)

# Acceder al conteo de CelulaTipoB en la Muestra2
print(matriz_celulas["Muestra2", "CelulaTipoB"])

# Suma de células por muestra (suma por filas)
total_por_muestra <- rowSums(matriz_celulas)
print(total_por_muestra)

# Promedio de células por tipo (promedio por columnas)
promedio_por_tipo <- colMeans(matriz_celulas)
print(promedio_por_tipo)
```

- **Ejercicio en Clase:**
  1. Crear una matriz 2x3 llamada `exp_genica` con los siguientes datos (que representan niveles de expresión): `1.2, 0.8, 2.5, 0.5, 1.9, 3.1`. Llenarla por filas.
  2. Asignar nombres a las filas ("Gen1", "Gen2") y a las columnas ("Control", "Tratamiento1", "Tratamiento2").
  3. Mostrar la matriz.
  4. Extraer el valor de expresión del "Gen1" bajo "Tratamiento1".
  5. Calcular la expresión promedio para cada gen (promedio por filas).
- **Ejercicio para Casa:**
  1. Crear una matriz 3x3 con datos numéricos de su elección (ej. concentraciones de metabolitos).
  2. Calcular su transpuesta.
  3. Crear otra matriz 3x3. Intentar la multiplicación matricial (`%*%`) entre ambas.
  4. Investigar las funciones `rownames()` y `colnames()`. ¿Cómo se usan para asignar o cambiar los nombres de filas y columnas de una matriz existente?

5.c. Estructuras de Datos: Listas

- **Fundamento Teórico:** Una lista es una colección ordenada de componentes (elementos) donde **cada componente puede ser de un tipo y estructura diferente**. Puede contener vectores, matrices, otras listas, funciones, etc.
  - **Creación:** `list(nombre1 = componente1, nombre2 = componente2, ...)`
  - **Acceso a datos:**
    - `$` : Para acceder a un componente por su nombre. Ej: `mi_lista$nombre_componente`. Es el más común y recomendado para acceso interactivo.
    - `[[ ]]`: Para acceder a un componente por su nombre (como cadena) o por su índice numérico. Devuelve el componente en sí mismo. Ej: `mi_lista[["nombre_componente"]]` o `mi_lista[[1]]`.
    - `[ ]`: Para extraer una sub-lista (es decir, devuelve una lista, incluso si es de un solo elemento). Ej: `mi_lista[1]` devuelve una lista con el primer componente.
  - **Funciones `apply` (familia):**

    - `lapply(X, FUN, ...)`: Aplica una función `FUN` a cada elemento de una lista `X` y devuelve una lista de resultados.
    - `sapply(X, FUN, ..., simplify = TRUE)`: Similar a `lapply`, pero intenta simplificar el resultado a un vector o matriz si es posible.
- **Ejemplo (Biología/Salud):**

```
# Lista para almacenar información de un paciente
paciente_info <- list(
  id = "P007",
  datos_demograficos = c(edad = 45, sexo_cod = 1), # 1=Masculino, 2=Femenino
  historial_medico = c("Hipertensión", "Diabetes tipo 2"),
  resultados_lab = data.frame( # Un data frame dentro de la lista
    test = c("Glucosa", "Colesterol HDL", "Triglicéridos"),
    valor = c(130, 45, 210),
    unidad = c("mg/dL", "mg/dL", "mg/dL")
  ),
  notas_adicionales = "Paciente refiere fatiga"
)

# Acceder al ID del paciente
print(paciente_info$id)

# Acceder a los resultados de laboratorio (que es un data.frame)
print(paciente_info$resultados_lab)

# Acceder al valor del test de Glucosa
print(paciente_info$resultados_lab[paciente_info$resultados_lab$test ==
"Glucosa", "valor"])
# O usando [[ ]]
print(paciente_info[["resultados_lab"]][1, "valor"])

# Ejemplo con lapply: obtener la clase de cada elemento de la lista
clases_elementos <- lapply(paciente_info, class)
print(clases_elementos)
```

- **Ejercicio en Clase:**
    1. Crear una lista llamada `experimento_genoma` con los siguientes componentes:
        - `organismo`: "Saccharomyces cerevisiae" (carácter)
        - `num_cromosomas`: 16L (entero)
        - `genes_estudiados`: un vector de caracteres con tres nombres de genes (ej. "ADH1", "GAL4", "CDC28")
        - `es_eucariota`: TRUE (lógico)
    2. Mostrar la lista completa.
    3. Acceder y mostrar solo el número de cromosomas usando `$`.
    4. Acceder y mostrar el segundo gen estudiado usando `[[ ]]` y el índice numérico del vector.
- **Ejercicio para Casa:**
    1. Crear una lista que contenga:

    - Un vector numérico de 5 elementos.
    - Una matriz de 2x2.
    - Un valor lógico.
2. Usar `lapply()` para calcular la longitud (`length()`) de cada componente de la lista.
3. Modificar el valor lógico de la lista.
4. Añadir un nuevo componente a la lista (ej. un vector de caracteres).

5.d. Estructuras de Datos: Marcos de Datos (DataFrames)

- **Fundamento Teórico:** Un `data.frame` es la estructura de datos más importante y comúnmente usada en R para almacenar datos tabulares (como hojas de cálculo).
    - Es una lista de vectores de igual longitud, donde cada vector representa una columna.
    - Cada columna debe contener elementos del mismo tipo, pero diferentes columnas pueden tener diferentes tipos.
    - Tiene dos dimensiones: filas (observaciones) y columnas (variables).
    - **Creación:**
        - `data.frame(columna1 = vector1, columna2 = vector2, ...)`
    - **Funciones útiles:** `head()`, `tail()`, `str()`, `summary()`, `dim()`, `nrow()`, `ncol()`, `names()` o `colnames()`, `rownames()`.
    - `cbind()`: Combina data frames por columnas (deben tener el mismo número de filas).
    - `rbind()`: Combina data frames por filas (deben tener las mismas columnas y nombres de columna).
    - **Acceso a subconjuntos:** Similar a las matrices y listas.
        - `mi_df[fila_indice, columna_indice]`
        - `mi_df[fila_indice, ]` (todas las columnas de esas filas)
        - `mi_df[, columna_indice]` (todas las filas de esas columnas)
        - `mi_df$nombre_columna` (devuelve la columna como un vector)
        - `mi_df[["nombre_columna"]]` (devuelve la columna como un vector)
        - `mi_df["nombre_columna"]` (devuelve un data.frame con esa columna)
        - Uso de condiciones lógicas para filtrar filas: `mi_df[mi_df$columna_numerica > 10, ]`
- **Ejemplo (Biología/Salud):**

```
# Crear un data frame de datos de pacientes
pacientes_df <- data.frame(
  ID_Paciente = c("P01", "P02", "P03", "P04"),
  Edad = c(25, 42, 31, 56),
  Sexo = factor(c("Femenino", "Masculino", "Femenino", "Masculino")),
  Tratamiento = factor(c("A", "B", "A", "B")),
  PresionArterialSistolica = c(120, 135, 118, 140),
  stringsAsFactors = FALSE # Buena práctica para evitar conversión automática
a factores
)
```

```r
print(pacientes_df)
str(pacientes_df) # Muestra la estructura del data frame
summary(pacientes_df) # Resumen estadístico

# Acceder a la columna Edad
print(pacientes_df$Edad)

# Acceder a las primeras 2 filas
print(head(pacientes_df, 2))

# Filtrar pacientes con PresionArterialSistolica > 130
pacientes_hipertensos_leves <-
pacientes_df[pacientes_df$PresionArterialSistolica > 130, ]
print(pacientes_hipertensos_leves)

# Seleccionar solo las columnas ID_Paciente y Sexo
id_sexo_df <- pacientes_df[, c("ID_Paciente", "Sexo")]
print(id_sexo_df)
```

- **Ejercicio en Clase:**
  1. Crear un data frame llamado `datos_cultivo` con la siguiente información para 3 muestras:
     - `ID_Muestra`: "M1", "M2", "M3"
     - `Tipo_Bacteria`: "E.coli", "S.aureus", "E.coli"
     - `Tiempo_Incubacion_hrs`: 24, 48, 24
     - `Densidad_Optica_600nm`: 0.85, 1.23, 0.92
  2. Mostrar la estructura del data frame usando `str()`.
  3. Mostrar las primeras 2 filas usando `head()`.
  4. Seleccionar y mostrar solo la columna `Densidad_Optica_600nm`.
  5. Filtrar y mostrar las muestras donde el `Tipo_Bacteria` es "E.coli".
- **Ejercicio para Casa:**
  1. Crear un data frame con información sobre 5 genes: Nombre del Gen, Longitud (en pares de bases), Número de Exones, y si está asociado a una enfermedad (TRUE/FALSE).
  2. Calcular la longitud promedio de los genes en el data frame.
  3. Filtrar los genes que tienen más de 10 exones.
  4. Añadir una nueva columna al data frame que sea la longitud dividida por el número de exones (longitud promedio por exón).
  5. Investigar la función `subset()`. ¿Cómo se puede usar para filtrar data frames? Replicar el filtrado del punto 3 usando `subset()`.

Módulo 3: Programación y Análisis de Datos

6. Programación en R: Estructuras de Control

- **Fundamento Teórico:** Las estructuras de control permiten dirigir el flujo de ejecución de un script.
  - **Condicionales:**
    - `if (condicion) { expresion1 }`: Si la condición es `TRUE`, se ejecuta `expresion1`.
    - `if (condicion) { expresion1 } else { expresion2 }`: Si es `TRUE`, se ejecuta `expresion1`; si no, `expresion2`.
    - `ifelse(test, yes, no)`: Función vectorizada. Para cada elemento en `test`, si es `TRUE` devuelve el elemento correspondiente de `yes`, si no, el de `no`.
  - **Bucles (Ciclos):**
    - `for (variable in secuencia) { expresion }`: Repite `expresion` para cada valor de `variable` en `secuencia`.
    - `while (condicion) { expresion }`: Repite `expresion` mientras `condicion` sea `TRUE`. ¡Cuidado con los bucles infinitos!
- **Ejemplo (Biología/Salud):**

```r
# if-else: Clasificar nivel de glucosa
glucosa_paciente <- 145 # mg/dL
if (glucosa_paciente < 100) {
  categoria_glucosa <- "Normal"
} else if (glucosa_paciente < 126) {
  categoria_glucosa <- "Pre-diabetes"
} else {
  categoria_glucosa <- "Diabetes"
}
print(paste("El paciente tiene:", categoria_glucosa))

# ifelse: Clasificar múltiples valores de pH
ph_muestras <- c(6.8, 7.1, 7.4, 6.5, 7.0)
clasificacion_ph <- ifelse(ph_muestras < 7.0, "Ácido",
                           ifelse(ph_muestras > 7.0, "Alcalino", "Neutro"))
print(clasificacion_ph)

# for: Procesar una lista de nombres de genes
genes_interes <- c("BRCA1", "TP53", "EGFR")
for (gen in genes_interes) {
  print(paste("Procesando información para el gen:", gen))
  # Aquí iría el código para buscar en base de datos, etc.
}

# while: Simular crecimiento poblacional hasta un límite (ejemplo simple)
poblacion_bacterias <- 100
limite_poblacion <- 1000
generaciones <- 0
while (poblacion_bacterias < limite_poblacion) {
  poblacion_bacterias <- poblacion_bacterias * 1.5 # Tasa de crecimiento
```

```
  generaciones <- generaciones + 1
}
print(paste("Población alcanzó", poblacion_bacterias, "en", generaciones,
"generaciones."))
```

- **Ejercicio en Clase:**
  1. Escribir un `if-else` que evalúe una variable `temperatura_cultivo`. Si es mayor a 30°C, imprimir "Temperatura alta"; si no, "Temperatura óptima".
  2. Crear un vector numérico de 5 concentraciones de un fármaco. Usar un bucle `for` para imprimir cada concentración.
  3. Usar `ifelse` para crear un nuevo vector que clasifique las concentraciones: si es > 0.5, "Alta"; si no, "Baja".
- **Ejercicio para Casa:**
  1. Crear un data frame con una columna de "Nivel_Expresion" (numérica) y otra de "Tipo_Celular" (carácter).
  2. Usar un bucle `for` para iterar sobre las filas del data frame. Dentro del bucle, usar un `if-else` para imprimir un mensaje diferente según el nivel de expresión (ej. "Sobreexpresado", "Subexpresado", "Normal") y el tipo celular.
  3. Simular un experimento donde se mide el pH de una solución cada hora. Empezar con pH = 7.0. En cada hora, el pH disminuye en 0.1. Usar un bucle `while` para determinar cuántas horas toma para que el pH sea menor a 6.0. Imprimir el número de horas y el pH final.

7. Automatización: Creación y Uso de Funciones

- **Fundamento Teórico:** Las funciones permiten empaquetar un conjunto de operaciones en un bloque de código reutilizable. Esto mejora la organización, reduce la redundancia y facilita la depuración.
  - **Principios para crear funciones:**
    1. Darle un nombre descriptivo.
    2. Definir los argumentos (inputs) que necesita.
    3. Escribir el cuerpo de la función (las operaciones).
    4. Especificar qué debe devolver (output) usando `return()` (aunque R devuelve implícitamente el resultado de la última expresión evaluada).
  - **Sintaxis:**

```
nombre_de_la_funcion <- function(argumento1, argumento2, ...) {
  # Cuerpo de la función: operaciones
  resultado <- argumento1 + argumento2 # Ejemplo
  return(resultado)
}
```

  - **Llamada a la función:** `nombre_de_la_funcion(valor1, valor2)`
- **Ejemplo (Biología/Salud):**

```
# Función para calcular el Índice de Masa Corporal (IMC)
# IMC = peso (kg) / altura (m)^2
```

```r
calcular_imc <- function(peso_kg, altura_m) {
  if (!is.numeric(peso_kg) || !is.numeric(altura_m)) {
    stop("El peso y la altura deben ser numéricos.") # stop() detiene la
ejecución con error
  }
  if (peso_kg <= 0 || altura_m <= 0) {
    stop("El peso y la altura deben ser positivos.")
  }

  imc <- peso_kg / (altura_m^2)
  return(imc)
}

# Usar la función
imc_paciente1 <- calcular_imc(peso_kg = 70, altura_m = 1.75)
print(paste("IMC Paciente 1:", round(imc_paciente1, 2))) # round() para
redondear

# Función para clasificar IMC
clasificar_imc <- function(imc) {
  if (imc < 18.5) {
    categoria <- "Bajo peso"
  } else if (imc < 25) {
    categoria <- "Normal"
  } else if (imc < 30) {
    categoria <- "Sobrepeso"
  } else {
    categoria <- "Obesidad"
  }
  return(categoria)
}

categoria_paciente1 <- clasificar_imc(imc_paciente1)
print(paste("Categoría Paciente 1:", categoria_paciente1))
```

- **Ejercicio en Clase:**
  1. Crear una función llamada `convertir_C_a_F` que tome una temperatura en grados Celsius como argumento y devuelva la temperatura en grados Fahrenheit. Fórmula: F = (C * 9/5) + 32.
  2. Probar la función con 0°C, 25°C y 100°C.
  3. Crear una función simple llamada `saludar_investigador` que tome un nombre como argumento y devuelva el string "Hola, [nombre]! Bienvenido a R."
- **Ejercicio para Casa:**
  1. Crear una función que tome un vector de valores de pH como argumento y devuelva un vector con la clasificación ("Ácido", "Neutro", "Alcalino") para cada valor. Usar la función `ifelse` dentro de su función.
  2. Crear una función que calcule la dilución necesaria para alcanzar una concentración final. Debe tomar como argumentos: concentración inicial, volumen inicial, y concentración final deseada. Debe devolver el volumen final total y el volumen de diluyente a añadir.
     - Pista: C1V1 = C2V2.

8. Análisis de Datos: Lectura, Guardado y Tidyverse

- **Fundamento Teórico (Lectura/Guardado):**
  - **Lectura de datos:**
    - `scan()`: Lee datos directamente desde la consola o un archivo, útil para vectores.
    - `read.table(file, header = FALSE, sep = "", ...)`: Función genérica para leer archivos tabulares. `header` indica si la primera fila son nombres de columna. `sep` es el separador de campos (ej. `" "`, `"\t"`, `","`).
    - `read.csv(file, header = TRUE, sep = ",", ...)`: Especializada para archivos CSV (valores separados por comas). Por defecto `header=TRUE` y `sep=","`.
    - `read.csv2(file, header = TRUE, sep = ";", dec = ",", ...)`: Para CSVs donde el separador es punto y coma y el decimal es coma (común en Europa).
    - Otros paquetes para leer formatos específicos: `readxl` (para Excel .xls, .xlsx), `foreign` (SPSS, Stata), `XML`, `jsonlite` (JSON).
  - **Guardado de datos:**
    - `write.table(x, file, sep = " ", row.names = TRUE, col.names = NA, ...)`
    - `write.csv(x, file, row.names = TRUE, ...)`
    - `save(objeto1, objeto2, ..., file = "datos.RData")`: Guarda objetos de R en un archivo binario `.RData`.
    - `load("datos.RData")`: Carga objetos desde un archivo `.RData`.
- **Fundamento Teórico (Tidyverse):**
  - **Qué es Tidyverse:** Una colección de paquetes de R diseñados para la ciencia de datos que comparten una filosofía de diseño, gramática y estructuras de datos subyacentes. Facilita la manipulación y visualización de datos.
  - **Principales paquetes:** `dplyr` (manipulación de datos), `ggplot2` (gráficos), `tidyr` (ordenar datos), `readr` (leer datos rectangulares), `purrr` (programación funcional), `tibble` (data frames modernos).
  - **Tidy Data:** Un estándar para organizar datos tabulares:
    1. Cada variable forma una columna.
    2. Cada observación forma una fila.
    3. Cada tipo de unidad observacional forma una tabla.
  - `dplyr` **- Verbos básicos (funciones):**
    - `filter()`: Selecciona filas basadas en condiciones.
    - `select()`: Selecciona columnas por nombre.
    - `mutate()`: Crea nuevas columnas o modifica existentes.
    - `arrange()`: Ordena filas.
    - `summarize()` (o `summarise()`): Reduce múltiples valores a un solo resumen (ej. media, suma).
    - `group_by()`: Agrupa el data frame por una o más variables para que las operaciones subsiguientes se realicen por grupo.

    - El operador "pipe" `%>%` (o `|>` en R >= 4.1): Pasa el resultado de una operación como el primer argumento de la siguiente, facilitando cadenas de operaciones legibles.
- **Ejemplo (Biología/Salud - Tidyverse):**

```r
# Primero, instalar y cargar tidyverse
# install.packages("tidyverse") # Solo una vez
library(tidyverse)

# Supongamos que tenemos un data frame 'datos_expresion_genica' con columnas:
# Gen, MuestraID, NivelExpresion, TipoTejido (ej. "Tumor", "Normal")

# Crear datos de ejemplo
set.seed(123) # Para reproducibilidad
datos_expresion_genica <- data.frame(
  Gen = rep(paste0("Gen", 1:5), each = 4),
  MuestraID = paste0("M", 1:20),
  NivelExpresion = rnorm(20, mean = 5, sd = 2), # Datos aleatorios normales
  TipoTejido = rep(c("Tumor", "Normal"), times = 10)
)

# Usando dplyr:
# 1. Filtrar por el Gen1
# 2. Seleccionar solo las columnas Gen, NivelExpresion, TipoTejido
# 3. Crear una nueva columna 'ExpresionAlta' (TRUE si NivelExpresion > 6)
# 4. Agrupar por TipoTejido y calcular la expresión promedio

analisis_expresion <- datos_expresion_genica %>%
  filter(Gen == "Gen1") %>%
  select(Gen, NivelExpresion, TipoTejido) %>%
  mutate(ExpresionAlta = NivelExpresion > 6) %>%
  group_by(TipoTejido) %>%
  summarize(ExpresionPromedio = mean(NivelExpresion),
            NumeroMuestras = n()) # n() cuenta el número de observaciones por
grupo

print(analisis_expresion)
```

- **Ejercicio en Clase:**
    1. Crear un pequeño archivo CSV (ej. en un editor de texto simple, guardarlo como `datos_bio.csv`) con:

```
ID,Especie,Peso_gr,Longitud_cm
1,Mus musculus,25,8
2,Rattus norvegicus,250,20
3,Mus musculus,22,7.5
```

    2. Leer este archivo en R usando `read.csv()` y guardarlo en un data frame. Mostrar el data frame.
    3. Si `tidyverse` está instalado:
        - Filtrar para obtener solo los "Mus musculus".
        - Calcular el peso promedio de los "Mus musculus".

- **Ejercicio para Casa:**
    1. Buscar un conjunto de datos biológicos simple en formato CSV en internet (ej. datos de iris, datos de pingüinos de `palmerpenguins`, datos de expresión génica simplificados).
    2. Leerlo en R.
    3. Usando `dplyr`:
        - Mostrar las primeras 10 filas.
        - Seleccionar 3 columnas de interés.
        - Filtrar las filas basándose en algún criterio de una columna numérica (ej. > valor) y una categórica (ej. == "especie_X").
        - Crear una nueva columna que sea el resultado de una operación entre otras dos columnas.
        - Agrupar por una variable categórica y calcular estadísticas resumen (media, mediana, desviación estándar) de una variable numérica para cada grupo.
    4. Guardar el data frame procesado como un nuevo archivo CSV.

Módulo 4: Visualización de Datos

9. Graficando con R

- **Fundamento Teórico (Gráficos Base de R):**
    - R tiene un sistema gráfico base potente y flexible.
    - `plot(x, y, type="p", ...)`: Función genérica para gráficos. `type` define el tipo: "p" (puntos), "l" (líneas), "b" (ambos), "h" (histograma), etc.
        - Muchos argumentos para personalizar: `main` (título), `xlab`, `ylab` (etiquetas ejes), `col` (color), `pch` (símbolo de punto), `lty` (tipo de línea), `xlim`, `ylim` (límites ejes).
    - `hist(x, ...)`: Histograma.
    - `barplot(height, names.arg, ...)`: Gráfico de barras. `height` es un vector o matriz.
    - `boxplot(formula, data, ...)`: Diagrama de caja y bigotes. Ej: `boxplot(valor ~ categoria, data=mi_df)`
    - **Personalización:** Se puede añadir elementos a un gráfico existente (ej. `points()`, `lines()`, `text()`, `legend()`).
    - **Múltiples gráficos:**
        - `par(mfrow = c(nr, nc))` o `par(mfcol = c(nr, nc))`: Divide la ventana gráfica en `nr` filas y `nc` columnas para mostrar múltiples gráficos.
        - `layout(matrix(...))`: Para diseños más complejos.
        - Paquete `lattice`: Para gráficos multivariados condicionados (trellis graphics).
- **Fundamento Teórico (ggplot2):**
    - Parte del `tidyverse`. Basado en la "Gramática de los Gráficos".
    - Construye gráficos por capas (layers).
    - **Componentes principales:**
        1. `data`: El data frame.

2. `aes()` (aesthetics): Cómo las variables se mapean a propiedades visuales (ej. `x`, `y`, `color`, `shape`, `size`, `fill`).
3. `geom_` (geometries): El tipo de objeto geométrico que representa los datos (ej. `geom_point()`, `geom_line()`, `geom_bar()`, `geom_boxplot()`, `geom_histogram()`, `geom_smooth()` para líneas de tendencia).
4. `labs()`: Para etiquetas (título, ejes, leyenda).
5. `theme()`: Para personalizar la apariencia no relacionada con los datos (fondo, fuentes, etc.).
6. `facet_wrap()` o `facet_grid()`: Para crear subgráficos (paneles) basados en variables categóricas.

- **Sintaxis básica:** `ggplot(data, aes(x=var_x, y=var_y)) + geom_funcion()`

- **Ejemplo (Biología/Salud - ggplot2):**

```
library(ggplot2)

# Usaremos el data frame 'pacientes_df' creado anteriormente
pacientes_df <- data.frame(
  ID_Paciente = c("P01", "P02", "P03", "P04", "P05", "P06"),
  Edad = c(25, 42, 31, 56, 33, 48),
  Sexo = factor(c("Femenino", "Masculino", "Femenino", "Masculino",
"Femenino", "Masculino")),
  Tratamiento = factor(rep(c("A", "B"), times=3)),
  PresionArterialSistolica = c(120, 135, 118, 140, 122, 128),
  ColesterolTotal = c(180, 220, 190, 240, 185, 210)
)

# Gráfico de dispersión: Edad vs Presión Arterial, coloreado por Sexo
ggplot(pacientes_df, aes(x = Edad, y = PresionArterialSistolica, color =
Sexo)) +
  geom_point(size = 3) + # Añade puntos, size controla el tamaño
  geom_smooth(method = "lm", se = FALSE) + # Añade línea de regresión lineal
sin intervalo de confianza
  labs(title = "Presión Arterial vs Edad por Sexo",
       x = "Edad (años)",
       y = "Presión Arterial Sistólica (mmHg)",
       color = "Sexo del Paciente") +
  theme_minimal() # Un tema limpio

# Boxplot: Colesterol Total por Tratamiento
ggplot(pacientes_df, aes(x = Tratamiento, y = ColesterolTotal, fill =
Tratamiento)) +
  geom_boxplot() +
  labs(title = "Distribución del Colesterol Total por Tratamiento",
       x = "Grupo de Tratamiento",
       y = "Colesterol Total (mg/dL)") +
  theme_light()

# Histograma: Distribución de Edades
ggplot(pacientes_df, aes(x = Edad)) +
  geom_histogram(binwidth = 5, fill = "skyblue", color = "black") +
  labs(title = "Distribución de Edades de los Pacientes",
       x = "Edad (años)",
```

```
        y = "Frecuencia") +
  theme_classic()
```

- **Ejercicio en Clase:**
  1. Usar el data frame `datos_cultivo` del ejercicio anterior (o crear uno similar con `ID_Muestra`, `Tiempo_Incubacion_hrs`, `Densidad_Optica_600nm`).
  2. Con gráficos base de R:
     - Crear un gráfico de dispersión de `Tiempo_Incubacion_hrs` vs `Densidad_Optica_600nm`. Añadir título y etiquetas a los ejes.
  3. Si `ggplot2` está disponible:
     - Replicar el gráfico de dispersión anterior usando `ggplot2`.
     - Si tienen una columna categórica (ej. `Tipo_Bacteria`), colorear los puntos según esa categoría.
- **Ejercicio para Casa:**
  1. Usar el conjunto de datos que investigaron y leyeron en el ejercicio de `dplyr`.
  2. Con gráficos base de R:
     - Crear un histograma de una variable numérica.
     - Crear un boxplot comparando una variable numérica entre diferentes grupos de una variable categórica.
  3. Con `ggplot2`:
     - Crear un gráfico de dispersión entre dos variables numéricas, coloreando los puntos por una tercera variable categórica y cambiando la forma de los puntos por otra variable categórica (si la tienen).
     - Crear un gráfico de barras que muestre la frecuencia (o promedio de una variable numérica) para cada nivel de una variable categórica.
     - Personalizar uno de los gráficos de `ggplot2` cambiando el tema (`theme_...()`), las etiquetas (`labs()`) y los colores.
     - Intentar usar `facet_wrap()` para crear subgráficos.

10. Casos de Estudio

- **Fundamento Teórico:** Esta sección del curso se enfoca en la aplicación práctica de R en problemas de investigación reales, presentados por especialistas. El objetivo es ver cómo se integran todos los conceptos aprendidos (manipulación de datos, programación, estadística, visualización) para resolver preguntas biológicas o de salud.
- **Posibles áreas de casos de estudio:**
  - Análisis de datos de secuenciación de nueva generación (NGS): RNA-Seq (expresión diferencial), ChIP-Seq, análisis de variantes.
  - Estudios de asociación del genoma completo (GWAS).
  - Análisis filogenéticos.
  - Modelado epidemiológico.
  - Análisis de datos de proteómica o metabolómica.
  - Bioestadística aplicada a ensayos clínicos.

- **Ejercicio (Conceptual - No requiere R aún):**
    - Pensar en un problema de investigación en su área de interés. ¿Qué tipo de datos se necesitarían? ¿Qué preguntas específicas se podrían responder usando R? ¿Qué tipo de gráficos serían útiles para comunicar los resultados?

Bibliografía Recomendada

1. **"R for Data Science (2nd edition)"** por Hadley Wickham, Mine Çetinkaya-Rundel, y Garrett Grolemund.
    - **Descripción:** Considerado el libro de referencia para aprender a realizar ciencia de datos con R y el Tidyverse. Cubre desde la importación y limpieza de datos hasta la visualización y comunicación de resultados. La segunda edición está actualizada con las últimas versiones de los paquetes.
    - **Acceso:** https://r4ds.hadley.nz/
2. **"ModernDive: Statistical Inference via Data Science"** por Chester Ismay y Albert Y. Kim.
    - **Descripción:** Un excelente recurso para aprender R en el contexto de la inferencia estadística y la ciencia de datos, utilizando el Tidyverse. Muy didáctico y con muchos ejemplos.
    - **Acceso:** https://moderndive.com/
3. **"Advanced R (2nd edition)"** por Hadley Wickham.
    - **Descripción:** Para aquellos que ya tienen una base en R y quieren profundizar en los aspectos más técnicos del lenguaje, como programación funcional, metaprogramación y optimización de rendimiento.
    - **Acceso:** https://adv-r.hadley.nz/
4. **"R Packages (2nd edition)"** por Hadley Wickham y Jennifer Bryan.
    - **Descripción:** Guía completa para aprender a crear paquetes de R, desde lo más básico hasta prácticas avanzadas de desarrollo de software. Esencial si se quiere compartir código de forma estructurada.
    - **Acceso:** https://r-pkgs.org/
5. **"Cookbook for R"** por Winston Chang.
    - **Descripción:** Ofrece soluciones prácticas ("recetas") a problemas comunes en el análisis de datos y la creación de gráficos con R. Muy útil para consultas rápidas y ejemplos concretos.
    - **Acceso:** http://www.cookbook-r.com/