Conceptos básicos de programación en R

Explicación Práctica 4 en R

¿Qué debes hacer para tener una buena práctica en R?

El objetivo de las recomendaciones es escribir código que sea más fácil de leer, entender y mantener. Aquí tienes una explicación más detallada:

1. **Nombres Descriptivos**:Utiliza nombres que expliquen claramente qué representan o hacen las variables y funciones, como numero_de_puntos en lugar de n_p o algo críptico.Esto facilita a otros (o a ti mismo en el futuro) comprender rápidamente el propósito del código, sin necesidad de adivinar o leer comentarios extensos.
2. **Estilo Consistente**:Asegúrate de usar una misma convención para todo el código, como la misma cantidad de espacios o tabulaciones para sangría. La consistencia evita confusión y facilita la lectura. Estructura el código de manera lógica: define las funciones antes de llamarlas, agrupa bloques relacionados, y sigue las mismas reglas para nombrar variables o funciones (por ejemplo, siempre en minúsculas con guiones bajos).
3. **Comentarios Claros y Útiles**: Los comentarios deben explicar el "por qué" detrás del código en lugar de repetir lo que ya es obvio. Por ejemplo, si tienes una fórmula matemática compleja, explica su propósito o el contexto.No es necesario comentar líneas evidentes como suma = a + b con # sumar dos números.
4. **Evitar Variables Globales**:Las variables globales pueden generar errores difíciles de rastrear, ya que pueden cambiar su valor desde cualquier parte del programa. En su lugar, pasa los valores necesarios como parámetros a las funciones. Esto las hace más autónomas y reutilizables, ya que no dependen de un contexto externo. Por ejemplo, una función que calcula un área debería recibir la base y la altura como parámetros, en lugar de depender de variables globales.
5. **Funciones Cortas y Específicas**:Cada función debe realizar una única tarea bien definida. Esto facilita la comprensión, el mantenimiento y la reutilización del código. Si una función empieza a abarcar demasiadas tareas o se vuelve difícil de leer, divídela en funciones más pequeñas y manejables.
6. **Control de Errores y Validación de Entrada**: Siempre verifica que los datos proporcionados a una función sean válidos y cumplan con las expectativas. Maneja errores con stop() para detener la ejecución en caso de problemas graves y utiliza warning() para alertar de situaciones no críticas.
7. **Uso Eficiente de Recursos**: Minimiza el uso de bucles cuando existan alternativas más rápidas y eficientes, como las funciones vectorizadas (apply, sapply, etc.). Esto no solo mejora el rendimiento, sino también la claridad del código.
8. **No Modificar el Entorno Global sin Necesidad**: Evita alterar variables globales desde dentro de las funciones. En lugar de ello, utiliza parámetros para pasar datos a las funciones y devuelve resultados explícitamente. Encapsula las operaciones dentro de funciones para mantener el código autónomo, modular y predecible. Esto también facilita la depuración y reduce el riesgo de errores.

FUNCIONES:

Síntesis de una función:

En R, una **función** es un bloque de código reutilizable diseñado para realizar una tarea específica. La sintaxis básica de una función en R es la siguiente:

```
nombre_funcion <- function(argumentos) {
    # Cuerpo de la función
}
```

1. Definición de una Función: Usamos el operador <- para asignar un bloque de código (la función) a un nombre. Este nombre (por ejemplo, nombre_funcion) es lo que usarás para llamar a la función más adelante.

2. La Palabra function: Indica que estás definiendo una función. Lo que sigue entre paréntesis (argumentos) son los valores que la función puede recibir como entrada cuando la usas.

3. Los Argumentos: Dentro de los paréntesis, puedes incluir uno o más nombres de variables, que representarán los datos que la función necesita para trabajar. Ejemplo: si quieres una función que sume dos números, podrías escribir function(a, b) donde a y b son los datos que la función usará.

4. El Cuerpo de la Función: Es el conjunto de instrucciones que la función ejecutará. Se escribe entre las llaves {}. Este código puede incluir cálculos, operaciones o cualquier tarea que desees automatizar. Puedes incluir un valor de retorno (es decir, un resultado que la función te devolverá) usando la palabra clave return().

5. Ejemplo Práctico:

```
# Definimos la función llamada "suma" con dos argumentos, a y b.

suma <- function(a, b) {

# Calculamos la suma de a y b y la guardamos en la variable "resultado".

  resultado <- a + b

# Devolvemos el valor de "resultado".

  return(resultado)

}
```

¿Cómo usar la función?

Para usar la función , simplemente escribes su nombre y le das los argumentos necesarios:

suma(3, 5)

Esto devolverá 8 porque la función sumará 3 + 5.

Beneficios de las Funciones:

- **Reutilización**: No tienes que escribir el mismo código varias veces.
- **Claridad**: Ayudan a organizar el código y hacerlo más fácil de leer.
- **Modularidad**: Dividen problemas grandes en piezas más pequeñas y manejables.

Función dentro de funciones:

```
greet_person <- function(name) {
    # Función interna para generar el saludo
    greet <- function(person_name)
    {
        return(paste("Hola,", person_name, "!"))
    }
    # Llamada a la función interna
    message <- greet(name)
    return(message)
}
# Llamada a la función externa
greet_person("Ana")
```

En este código, se define una función externa llamada greet_person, que puede ser llamada desde cualquier lugar del programa. Dentro de esta, se encuentra una función interna llamada greet, que solo es accesible dentro de greet_person. Aquí está la explicación más breve:

1. **greet_person (Función Externa)**:
   Es la función principal que toma un nombre como entrada (name) y genera un saludo utilizando una función interna. Esta función es accesible globalmente.
2. **greet (Función Interna)**:
   Vive dentro de greet_person y solo puede ser usada dentro de esta función. Se encarga de crear el mensaje de saludo utilizando el nombre recibido.
3. **Flujo del Código**:

- Cuando llamas a greet_person("Ana"), esta función pasa el nombre "Ana" a la función interna greet.
- greet genera el mensaje Hola, Ana! usando paste(), que combina texto.
- El mensaje se devuelve como salida de greet_person.

Nombres Asignados a los parámetros:

```r
# Variable global
x <- 5
# Función que toma un parámetro llamado 'y'
mi_funcion <- function(y) {
  y <- y * 2  # Cambiamos el valor de y en la función
  return(y)
}
# Variable global
x <- 5
# Función que toma un parámetro llamado 'y'
mi_funcion2 <- function(x) {
  x <- x * 2  # Cambiamos el valor de y en la función
  return(x)
}
```

```r
# Llamamos a la función pasando 'x' como parámetro,
#pero el parámetro en la función se llama 'y'
resultado <- mi_funcion(x)
print(resultado) # Resultado esperado: 10
print(x) # Resultado esperado: 5

# La variable global 'x' sigue siendo la misma
resultado <- mi_funcion2(x)
print(resultado) # Resultado esperado: 10
# Resultado esperado: 10
```

Paso de valor en R:

En **R**, cuando pasas una variable a una función, no estás pasando la **variable** en sí misma, sino que lo que realmente estás pasando es **su valor**. Esto significa que dentro de la función, trabajarás con una **copia** del valor de la variable, no con la variable original.

Por ejemplo:

r
Copiar código

```r
mi_variable <- 10
mi_funcion <- function(x) {
```

```
  x <- x + 5
  return(x)
}
resultado <- mi_funcion(mi_variable)
print(resultado)  # Esto imprimirá 15
print(mi_variable)  # Esto imprimirá 10
```

En este ejemplo, aunque dentro de la función mi_funcion, la variable x (que es una copia de mi_variable) se modificó, **la variable original mi_variable fuera de la función no se ve afectada**. Esto ocurre porque solo se pasó **el valor** de mi_variable a la función, no la **referencia** a la variable original.

Funciones y su relación con el entorno global:

Las funciones en **R** no pueden modificar directamente las variables que están en el **entorno global**. Esto se debe a que, como mencionamos antes, solo se pasa una copia del valor de la variable a la función. La función trabaja con esa copia y no tiene acceso directo a la variable original en el entorno global.

Siguiendo el ejemplo anterior:

r
Copiar código

```
mi_variable <- 10
mi_funcion <- function(x) {
  x <- x + 5
  return(x)
}
resultado <- mi_funcion(mi_variable)
print(resultado)  # Esto imprimirá 15
print(mi_variable)  # Esto imprimirá 10, ya que `mi_variable` no ha sido modificada en el entorno global
```

En este caso, mi_variable en el entorno global no se modifica porque solo se pasó su **valor** a la función.

Cambio de nombres de los parámetros:

Puedes nombrar los **parámetros** dentro de la función como quieras. Es decir, el nombre que le des al parámetro no tiene ningún impacto en el nombre de la variable global que se pasa como argumento a la función. Lo que importa es el **valor** que se le pasa.

Por ejemplo:

r
Copiar código

```
mi_variable <- 10
```

```
mi_funcion <- function(x) {  # Aquí 'x' es solo un nombre de parámetro, puede llamarlo como quiera
  return(x + 5)
}
resultado <- mi_funcion(mi_variable)  # 'mi_variable' es lo que se pasa, y 'x' lo recibe
print(resultado)  # Imprime 15
```

Aquí, aunque en la función el parámetro se llama x, el valor de mi_variable (que es 10) se pasa a ese parámetro. El nombre x dentro de la función no afecta el nombre de la variable global mi_variable.

Paso de parámetros de entrada por orden

```
#Definir una función sencilla que calcula el volumen
de un prisma rectangular

calcular_volumen <- function(largo, ancho, alto) {
  return(largo * ancho * alto)
}
# Llamada por orden: los valores se asignan en el
orden en el que se pasan
calcular_volumen(2, 3, 4)
# Largo = 2, Ancho = 3, Alto = 4 # Resultado: 24
```

Paso de parámetros de entrada por nombre

```
calcular_volumen(ancho = 3, alto = 4, largo = 2)
# Largo = 2, Ancho = 3, Alto = 4 # Resultado: 24
```

Usando ambos métodos

```
calcular_volumen(2, alto = 4, ancho = 2)
# Largo = 2, Ancho = 3, Alto = 4 # Resultado: 24
```

El 1º:

- Ventaja: Es breve y fácil cuando se conocen bien el orden y significado de los parámetros. Desventaja: Puede ser confuso si hay muchos argumentos o se olvida el orden exacto.

El 2º:

- Ventaja: Es claro y legible, especialmente cuando hay muchos parámetros o si no se conoce el orden exacto.
  Desventaja: Puede ser un poco más extenso que el paso por orden.

Función con parámetro por defecto:

```
# Función con un valor por defecto para el descuento
calcular_precio <- function(precio, cantidad,
descuento = 0.05)
{
    total <- precio * cantidad * (1 - descuento)
    return(total)
}
```

Ejemplo de usos:

```
# Llamada sin especificar el descuento
calcular_precio(precio = 100, cantidad = 3)
# Cálculo interno: 100 * 3 * (1 - 0.05) = 285#
Resultado: 285

# Llamada a la función especificando el descuento
calcular_precio(precio = 100, cantidad = 3, descuento
= 0.10) # Cálculo interno: 100 * 3 * (1 - 0.10) = 270
# Resultado: 270
```

Ventajas de Usar Valores por Defecto en las Funciones:

1. **Flexibilidad**:
    - Los **valores por defecto** proporcionan mayor **flexibilidad** al permitir que una función sea utilizada de manera más general. Esto significa que el usuario puede llamar a la función sin necesidad de proporcionar todos los parámetros si no lo desea.
    - Por ejemplo, si tienes una función para calcular el precio de un producto, puedes establecer un valor por defecto para el **descuento**. Si el usuario no lo especifica, se usará el valor por defecto, pero si el usuario lo quiere cambiar, puede introducir su propio descuento. Esto hace que la función sea más adaptable a diferentes situaciones sin necesidad de hacer múltiples versiones de la misma.

Ejemplo:
r
Copiar código

```
calcular_precio <- function(precio, cantidad, descuento = 0.1) {
  total <- precio * cantidad * (1 - descuento)
```

```
  return(total)
}

# Usar el valor por defecto de descuento (0.1)
resultado1 <- calcular_precio(100, 2)

# Usar un descuento diferente
resultado2 <- calcular_precio(100, 2, 0.2)

print(resultado1)  # Usando el descuento por defecto
print(resultado2)  # Usando un descuento personalizado
```

2. En este caso, el usuario puede decidir si quiere especificar un descuento, o si prefiere usar el valor por defecto (0.1).
3. **Código más limpio**:
    - Los valores por defecto hacen que tu código sea más **conciso y limpio**. No es necesario volver a definir los mismos parámetros en cada llamada a la función. Esto ayuda a evitar repeticiones innecesarias y mejora la legibilidad del código, ya que simplifica la interacción del usuario con la función.
4. **Ejemplo**:
   Si no usáramos valores por defecto, cada vez que quisiéramos llamar a `calcular_precio`, necesitaríamos incluir todos los parámetros, aunque no quisiéramos modificar el descuento. Usar un valor por defecto hace que la función sea más fácil de usar y mantener.
5. **Precauciones con los parámetros intermedios**:
    - Si el **argumento con valor por defecto** no es el último en la lista de parámetros, puede ser necesario especificar explícitamente los valores de los parámetros posteriores. De lo contrario, esto puede llevar a **errores** o a resultados inesperados si se omiten esos parámetros al llamar la función.

Esto ocurre porque **R** toma los valores de los parámetros por **orden**, y si omites un parámetro intermedio que tiene un valor por defecto, los valores posteriores pueden no funcionar como se espera.
Ejemplo de error potencial:
r
Copiar código

```
calcular_precio <- function(precio, cantidad, descuento = 0.1, impuesto = 0.05) {
  total <- precio * cantidad * (1 - descuento) * (1 + impuesto)
  return(total)
}

# Error: se omite 'descuento' y R no sabe qué hacer con 'impuesto'
resultado <- calcular_precio(100, 2)
```

6. Para evitar este tipo de problemas, es recomendable que los parámetros con valores por defecto estén al final de la lista de parámetros de la función. De esta manera, el orden de los parámetros no causará problemas cuando se omitan algunos.