Redacción de artículos académicos con R

03. Operaciones de control y funciones

Bajaña Alex

Chanatasig Evelyn

Heredia Aracely

2021-06-28

Operaciones de control y funciones

Muestras aleatorias cuantitativas

Muestras aleatorias

En la clase anterior describimos la función `sample()` que toma un vector y crea una muestra aleatoria de un tamaño deseado.

```
edades <- 18:25
sample(edades)
```

> Si no declaramos un tamaño de población, la función `sample` reordena el vector aleatorio.

Reproducibilidad

R maneja un sistema de números `pseudo aleatorios` por lo que el uso de una semilla (`seed`) permite la reproducción de resultados, especialmente útil en simulaciones:

```
set.seed(1984)    #es la semilla
sample(edades, size = 100, replace = T)
```

> Ejecuta dos veces esta línea y observarás que el resultado es el mismo: una muestra aleatoria de tamaño 100 con reemplazo.

Distribuciones de probabilidad

Funciones para distribuciones de probabilidad:

- `rnorm`: genera variables normales aleatorias con una media desviación estándar dadas
- `dnorm`: evalúa la densidad de probabilidad Normal (con media y desviación estándar dadas) en un punto (o vector de puntos)
- `pnorm`: evalúa la función de distribución acumulativa para una distribución Normal
- `rpois`: genera variaciones aleatorias de Poisson con una tasa dada

Variantes de la distribución normal:

- `d`: Función de densidad
- `r`: Generación de números aleatorios
- `p`: Función de distribución acumulada $\Phi(q)$
- `q`: Cuantiles $\Phi^{-1}(p)$

Muestras aleatorias y distribuciones de probabilidad

Ejemplos

- Se producen 100 cervezas al día. Existe un 10% de probabilidad de encontrar un error ¿Cuantas se deben arreglar al día?

```
set.seed(1)
rbinom(n = 7,size = 100,prob = 0.10)

set.seed(5)
rbinom(n = 7,size = 100,prob = 0.10)

set.seed(1)
rbinom(n = 7,size = 100,prob = 0.10)
```

Muestras aleatorias y distribuciones de probabilidad

Ejemplos

- Probabilidad de obtener 8 errores en la linea de producción:

```
dbinom(x = 8,size = 100,prob = 0.10)
```

- Probabilidad de obtener 8 o menos errores en la linea de producción:

```
 pbinom(q = 8,size = 100,prob = 0.10)
```

Funciones

Funciones

Una de las mejores maneras de lograr tener mayor alcance haciendo ciencia de datos es escribir funciones. Las funciones te permitirán automatizar algunas tareas comunes de una forma más poderosa y general que `copiar-y-pegar` además tiene ventajas sobre `copiar-y-pegar` como:

- **Nombrar tu función:** hará tu código más fácil de entender.
- A medida que cambien los requerimientos, solo necesitarás cambiar tu código en un solo lugar, en vez de en varios lugares.
- Eliminas las probabilidades de errores accidentales cuando copias y pegas (por ej., al actualizar el nombre de una variable en un lugar).

> Para escribir funciones en R base, no necesitarás ningún paquete extra.

¿Cuándo escribir una función?

Deberías considerar escribir una función cuando has copiado y pegado un bloque de código más de dos veces (es decir, ahora tienes tres copias del mismo)

```
# Datos
a <- 3
b <- 5
c <- 8
d <- 2

#Ejemplo del mismo código 3 veces
cuadrado_1 <- a*a + 2*a*b + b*b
cuadrado_2 <- b*b + 2*b*c + c*c
cuadrado_3 <- c*c + 2*c*d + d*d
```

Consejos

El nombre de una función es importante. Idealmente, debería ser corto, pero que evoque claramente lo que la función hace.

Usa comentarios, esto es, líneas que comienzan con `#`, para explicar el "porqué" de tu código. En general deberías evitar comentarios que expliquen el "qué" y el "cómo". Si no se entiende qué es lo que hace el código leyéndolo, deberías pensar cómo reescribirlo de manera que sea más claro.

Otro uso importante de los comentarios es para dividir tu archivo en partes, de modo que resulte más fácil de leer. Utiliza líneas largas de `-` y `=` para que resulte más fácil detectar los fragmentos.

```
#--------------------------------------------------------------------------
# Esta sección es para.....
#--------------------------------------------------------------------------

#==========================================================================
# Esta sección es para.....
#==========================================================================
```

Pasos para crear una nueva función:

1. Necesitas elegir un nombre para la función.
2. Listar los inputs, o argumentos de la función dentro de `function`. Ej: `function(x, y, z)`.
3. En el cuerpo de la función debes escribir el código que has creado. Este va dentro de llaves `{}` y se ubica a continuación de `function(...)`.

```
nombre_funcion <- function(x,y,z){
  ...
  ...
}
```

> Para escribir una función primero debes darte cuenta cómo funciona, el paso a paso y luego probarla con un solo input para verificar que funcione.

Estilo del código

`function` debe ir siempre entre llaves (`{}`) y el contenido debería tener una sangría de dos espacios. Esto hace que sea más fácil distinguir la jerarquía dentro de tu código al mirar el margen izquierdo.

La llave de apertura nunca debe ir en su propia línea y siempre debe ir seguida de una línea nueva. Una llave de cierre siempre debe ir en su propia línea. Siempre ponle sangría al código que va dentro de las llaves.

```
#Función bien escrita
cuadrado_bien <- function (x,y){
  resultado <- x*x + 2*x*y + y*y
  return(resultado)
}

#Función mal escrita
cuadrado_mal <- function (x,y)
{                    #La llave de apertura nunca debe ir en su propia línea
resultado <- x*x + 2*x*y + y*y   #el código debe tener sangría
return(resultado)}  #la llave de cierre siempre debe ir en su propia línea
```

Argumentos de funciones

Los argumentos de las funciones normalmente están dentro de dos conjuntos amplios: un conjunto provee los datos a computar y el otro los argumentos que controlan los detalles de la computación. Por ejemplo:

En `log()`, los datos son `x`, y los detalles son `la base del algoritmo`.

Elección de nombres

Los nombres de los argumentos también son importantes. A R no le importa, pero sí a quienes leen tu código (¡incluyéndo a tu futuro-yo!). En general, deberías preferir nombres largos y más descriptivos, aunque hay un puñado de nombres muy comunes y muy cortos. Vale la pena memorizar estos:

- `x, y, z:` vectores.
- `w:` un vector de pesos.
- `df:` un data frame.
- `i, j:` índices numéricos (usualmente filas y columnas).
- `n:` longitud, o número de filas.
- `p:` número de columnas.

Valores de retorno

Darse cuenta qué es lo que tu función debería devolver suele ser bastante directo: ¡es el porqué de crear la función en primer lugar!

Sentencias de retorno explícitas

El valor devuelto por una función suele ser la última sentencia que esta evalúa.

```
cuadrado_bien2 <- function (x,y){
  x*x + 2*x*y + y*y
}

cuadrado_bien3 <- function (x,y){
  resultado <- x*x + 2*x*y + y*y
  resultado
}
```

Puedes optar por devolver algo haciendo uso de la función `return()` (retornar o devolver en inglés).

```
cuadrado_bien4 <- function (x,y){
  resultado <- x*x + 2*x*y + y*y
  return(resultado)
}
```

Entorno

El entorno de una función controla cómo R encuentra el valor asociado a un nombre. Por ejemplo, toma la siguiente función:

```
f <- function(x) {
 x + y
}
```

En muchos lenguajes de programación, esto sería un error, porque `y` no está definida dentro de la función. En `R`, esto es un código válido ya que `R` usa reglas llamadas de `ámbito léxico` (lexical scoping) para encontrar el valor asociado a un nombre. Como `y` no está definida dentro de la función, R mirará dentro del entorno donde la función fue definida:

```
y <- 100
f(10)
# [1] 110
```

Este comportamiento parece una receta para errores (bugs) y, debes evitar las funciones así. Pero en líneas generales no causa demasiados problemas.

Ejemplo

R pone pocos límites a tu poder, de hecho, puedes escribir una función de adición manualmente.

```
suma <- function(x,y,z){
  x+y+z
}
```

Punto-punto-punto (...)

Muchas funciones en R tienen un número arbitrario de inputs:

```
reemplaza_a_por_b <- function(mensaje,...){
  str_replace_all(... ) %>%
    str_c("Mensaje: ",mensaje," ",.)
}

reemplaza_a_por_b(string = c("banana","araña"),
                  pattern ="a" ,
                  replacement ="c",
                  mensaje = "Hola")
```

Ejecución condicional

Ejecución condicional

En R la sentencia `if` evalúa una condición, y, si se cumple con la condición, se ejecuta el tramo de código que sigue, caso contario no se realiza acción alguna.

```
if(condicion){
  código a ejecutar
}
```

También existe la posibilidad de declarar una sentencia que se ha de ejecutar en el caso en que no se cumpla con la condición, para ello empleamos la sentencia else:

```
if(condicion){
  código a ejecutar si se cumple la condicion
}else{
  código a ejecutar si no se cumple la condicion
}
```

El codigo que se declara dentro de las sentencias `if` y `else` puede ser tan simple o complejo que se desee.

Ejemplo

El coeficiente de Gini es una medida de la desigualdad, normalmente se utiliza para medir la desigualdad en los ingresos dentro de un país.

```
gini <- 40
gini1 <- 52

#Ejemplo simple
if(gini > 60){
  print("El país es desigual")
}
```

Ejemplo

El coeficiente de Gini es una medida de la desigualdad, normalmente se utiliza para medir la desigualdad en los ingresos dentro de un país.

```
#Ejemplo complejo
if(gini1 < 30){
  print("El país muestra igualdad")
} else if(gini1 > 30 & gini1 < 70){
  print("El país debería tomar precauciones acerca de la desigualdad")
}else{
  print("El país es desigual")
}
```

> La función `print()` imprime en consola el contenido de un objeto, puede ser un mensaje en forma de character o cualquier objeto.

Ejecución iterativa

Bucles, for loops o ejecución iterativa

El bucle `for` se utiliza para repetir una o más instrucciones un determinado número de veces, este optimiza el uso de memoria, el tiempo de procesamiento y el tiempo codificando.

De entre todos los bucles, el `for` se suele utilizar cuando sabemos seguro el número de veces que queremos que se ejecute.

```
# Estructura:
for (indice in desde:hasta) {
  sentencias a ejecutar en cada iteración
}
```

- **índice:** se coloca la variable que utilizaremos para llevar la cuenta de las veces que se ejecuta el bucle
- **desde:hasta:** indica cuándo ha de comenzar y cuándo se ha de detener el bucle, también puede contener una condición que cumplir para que continúe la ejecución del bucle.

Bucles, for loops o ejecución iterativa

Ejemplos

```r
# Ejemplo: imprimir la suma de sí mismo con el número que sigue
# desde el 1 hasta el 5
for(i in 1:5){
  suma <- i+(i+1)
  print(suma)
}

# Mostrar los nombres de los asistente con un saludo
asistentes <- c("Melody","Fausto","Daniela")
for(nombre in asistentes){
  saludo <- paste("Hola!",nombre)
  print(saludo)
}
```

> `paste` es una función que une todos los vectores de caracteres que se le suministran y construye una sola cadena de caracteres

Ejemplo de una función compleja: con `for` e `if`

Función para identificar un número primo

```
for(num in 5:25 ){
  flag <-  0
  if(num > 1) {                 # Sí el valor es mayor 1:
    flag <-  1
    for(i in 2:(num-1)) {       # Reviso la división:
      if ((num %% i) == 0) {
        # Aqui rompemos el loop si al menos un número antes de
        # num hace una división entera
        flag <-  0
        break
      }
    }
  }
  if(num == 2)    flag <-  1
  if(flag == 1) {
    print(paste(num,"Es un número primo"))
  } else {            # Si no se cumple la condición paso a la siguiente iterac
    next
  }
}
```


THE
END

Recursos

- Programming with R: Creating Functions
- Advance R: functions
- R for Data Science: Functions