Parte III

ronny hdez-mora

5/25/2019

Capítulo 15

Funciones

##2 Hacer función propia para la varianza

```
secuencia <- c(10, 15, 30, 28, NA, 8)

varianza <- function(x) {
  media <- mean(secuencia, na.rm = TRUE)
  total <- length(secuencia) - 1
  sumatoria <- sum(cuadrado <- (x - media)^2)
  return(sumatoria / total)
}
```

##5 Hacer funcion para tomar dos vectores y

```
secuencia_a <- c(10, 15, 30, 28, NA, 8)
secuencia_b <- c(10, NA, 30, 28, 17, 8)

both_na <- function(x, y) {
  return(paste("La posición de x es:", which(is.na(x)), ", y",
               "la posición de y es:", which(is.na(y))))
}

both_na(secuencia_a, secuencia_b)
```

```
## [1] "La posición de x es: 5 , y la posición de y es: 2"
```

##6 ¿Qué hacen las siguientes funciones?

```
is_directory <- function(x) file.info(x)$is.dir
is_readable <- function(x) file.access(x, 4) == 0
```

Functions are for humans and computers

##1 Leer el código de las funciones y hacer nombres

```
# Tomar en cuenta lo que hace doble o simple & |
a <- c(1:5)
b <- c(1:5)

c <- c(6:10)
d <- c(6:10)

a == b & c == d
```

```
## [1] TRUE TRUE TRUE TRUE TRUE
```

```
a == b && c == d
```

```
## [1] TRUE
```

```
5 > 4 & 6 > 3
```

```
## [1] TRUE
```

```
5 > 4 && 6 > 3
```

```
## [1] TRUE
```

```
5 < 4 & 6 > 3
```

```
## [1] FALSE
```

```
5 < 4 && 6 > 3
```

```
## [1] FALSE
```

```
5 < 4 | 6 > 3
```

```
## [1] TRUE
```

```
5 < 4 || 6 > 3
```

```
## [1] TRUE
```

```
a <- function(x, y, op) {
  switch (op,
          plus = x + y,
          minus = x - y,
          times = x * y,
          divide = x / y,
          stop("Unknown op!")
  )
}
```

Escribir funcion que diga buenos dias, buenas tardes o buenas noches de acuerdo a la hora del dia.

```
saludo <- function(hora = lubridate::now()){
  if (hour(hora) <= 12) {
    print("¡Buenos días!")
  } else if (hour(hora) > 12) {
    print("¡Buenas tardes!")
  } else {
    print("¡Buenas noches!")
  }
}
```

##3 Escribir funcion que si numero es divisible por 3 imprime ## fizz, si es divisible por tres y cinco imprime fizzbuzz. Lo ## demas imprime el numero

```
fizzbuzz <- function(x) {
  if(x %% 3 == 0 & x %% 5 == 0) {
    print("fizzbuzz")
```

```
  } else if(x %% 3 == 0) {
    print("fizz")
  } else {
    return(x)
  }
}
```

Una funcion de ejemplo con dot-dot-dot

```
comas <- function(...) {
  stringr::str_c(..., collapse = ",")
}

comas(letters[1:5])
```

```
## [1] "a,b,c,d,e"
```

Capítulo 17: Iteraciones con purr

Introducción

Uno de los primeros ejemplos de loops:

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

output <- vector("double", ncol(df))

for (i in seq_along(df)) {
  output[[i]] <- median(df[[i]])
}
```

Ejercicios

- 1 Generar loops para:

1.a Obtener media de cada columna en mtcars

```
# Estructura para almacenar valores
medias <- vector("double", ncol(mtcars))

# Loop
for (i in seq_along(mtcars)) {
  medias[[i]] <- mean(mtcars[[i]])
```

```
}

# Mostrar resultados
print(medias)
```

```
##  [1]  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250
##  [7]  17.848750   0.437500   0.406250   3.687500   2.812500
```

1.b Determinar el tipo de cada columna en `fligths13::fligths`

```
# Crear objeto con datos
vuelos <- nycflights13::flights

# Estructura para almancenar datos
# tipo_columnas <- vector("character", ncol(vuelos))

# Hay que hacer estructura de lista porque clase de columna
# time_hour tiene dos: "POSIXct" "POSIXt" y eso causa problemas
# de espacio en la estructura de vector
tipo_columnas <- list()

# Loop
for (i in seq_along(vuelos)) {
  tipo_columnas[[i]] <- class(vuelos[[i]])
}

# Mostrar resultados
tipo_columnas
```

```
## [[1]]
## [1] "integer"
##
## [[2]]
## [1] "integer"
##
## [[3]]
## [1] "integer"
##
## [[4]]
## [1] "integer"
##
## [[5]]
## [1] "integer"
##
## [[6]]
## [1] "numeric"
##
## [[7]]
## [1] "integer"
##
## [[8]]
## [1] "integer"
##
```

```
## [[9]]
## [1] "numeric"
##
## [[10]]
## [1] "character"
##
## [[11]]
## [1] "integer"
##
## [[12]]
## [1] "character"
##
## [[13]]
## [1] "character"
##
## [[14]]
## [1] "character"
##
## [[15]]
## [1] "numeric"
##
## [[16]]
## [1] "numeric"
##
## [[17]]
## [1] "numeric"
##
## [[18]]
## [1] "numeric"
##
## [[19]]
## [1] "POSIXct" "POSIXt"
```

1.c Computar el número de valores únicos en cada columna de iris

```
# Estructura para almacenar datos
valores_unicos <- vector("double", ncol(iris))

# Loop
for (i in seq_along(iris)) {
  valores_unicos[[i]] <- length(unique(iris[[i]]))
}

# Mostrar resultados
valores_unicos
```

```
## [1] 35 23 43 22  3
```

- 4. Es común observar loops que no pre-alocan el output y en lugar, incrementan el tamaño del vector en cada paso. ¿Cómo esto afecta el rendimiento? Diseñe y ejecute un experimento

```
output <- vector("integer", 0)

x <- iris
```

```
for (i in seq_along(x)) {
  output <- c(output, lengths(x[[i]]))
}

output
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [71] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [106] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [141] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [176] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [211] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [246] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [281] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [316] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [351] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [386] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [421] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [456] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [491] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [526] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [561] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [596] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [631] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [666] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [701] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [736] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

El experimento con tiempo:

```
# Output sin asginar
x <- vuelos

output <- vector("integer", 0)

system.time(for (i in seq_along(x)) {
  output <- c(output, lengths(x[[i]]))
})
```

```
##    user  system elapsed
##   0.131   0.060   0.191
```

```
# Output asignado
x <- vuelos

output <- vector("integer", ncol(x))

system.time(for (i in seq_along(x)) {
  output <- c(output, lengths(x[[i]]))
})
```

```
##    user  system elapsed
##   0.119   0.036   0.155
```

Midiendo el tiempo vemos que tenemos una diferencia de `0.265 - 0.153` con respecto a una estructura con

espacios creados a una sin haber sido asignada.

Variaciones de for loops:

- Modificar un objeto existente.
- Looping sobre nombres o valores.
- Manejar outputs de tamaño desconocido.
- Manejar secuencias de tamaño desconocido.

Ejercicios

- 1. Imagine tiene un directorio lleno de archivos .csv que quiere leer Escriba el loop necesario para cargarlos en un solo dataframe.
- 2. ¿Qué pasa si usa `for (nm in names(x))` y x no tiene nombres? ¿Qué pasa si sólo algunos de los elementos tienen nombre? ¿

For loops vs Functionals

- En R loops pueden envolver otros loops

La ventaja se ve en el siguiente ejemplo: Tenemos un conjunto de datos en el cual queremos obtener la media de cada columna:

```
# Crear tibble ejemplo
datos <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

# Crear loop para tener media de cada columna
output <- vector("double", ncol(datos))

for (i in seq_along(datos)) {
  output[[i]] <- mean(datos[[i]])
}

output
```

```
## [1] -0.06551589 -0.31323293 -0.07810630 -0.56244543
```

Pero luego nos damos cuenta que sería útil obtner la mediana y la desviación estándar. Por lo que copiamos y pegamos el código cambiando la función mean por median y sd respectivamente:

```
for (i in seq_along(datos)) {
  output[[i]] <- median(datos[[i]])
}

output
```

```
## [1]  0.1248113 -0.6212426 -0.4022746 -0.1786687
```

```
for (i in seq_along(datos)) {
  output[[i]] <- sd(datos[[i]])
}

output
```

```
## [1] 1.0485102 1.1881737 1.1226644 0.7608834
```

Hasta este punto hemos violado la regla: hemos copiado y pegado más de dos veces el mismo código. Por ende debemos de tener una manera de generalizar esto dentro de una función. La única diferencia que encontramos es el nombre de la función, todo lo demás se encuentra igual.

En una función podemos agregar un argumento que indique la función:

```
resumen_columnas <- function(x, fun) {
  out <- vector("double", ncol(x))
  for (i in seq_along(x)) {
    out[[i]] <- fun(x[[i]])
  }
  return(out)
}
```

Con la función podríamos jugar así:

```
resumen_columnas(datos, mean)
```

```
## [1] -0.06551589 -0.31323293 -0.07810630 -0.56244543
```

```
resumen_columnas(datos, median)
```

```
## [1]  0.1248113 -0.6212426 -0.4022746 -0.1786687
```

```
resumen_columnas(datos, sd)
```

```
## [1] 1.0485102 1.1881737 1.1226644 0.7608834
```

Funciones map()

Con estas funciones tenemos la posibilidad de "saltarnos" el crear un loop y usar una función directamente que hace énfasis en la operación que se genera. Hay una función por cada tipo de output

- `map()` crea una lista
- `map_lgl()` crea un vector logico
- `map_int()` crea un vector entero
- `map_dbl()` crea un vector doble
- `map_chr()` crear un vetor caracteres

En el caso del ejemplo anterior, podríamos sustituir el loop con:

```
map_dbl(datos, mean)
```

```
##           a           b           c           d
## -0.06551589 -0.31323293 -0.07810630 -0.56244543
```

```
map_dbl(datos, median)
```

```
##          a          b          c          d
##  0.1248113 -0.6212426 -0.4022746 -0.1786687
```

```
map_dbl(datos, sd)
```

```
##         a         b         c         d
## 1.0485102 1.1881737 1.1226644 0.7608834
```

Atajos

Con **purrr** y sus funciones `map_()` tenemos:

- Un segundo argumento, `.f` que es la función a aplicar puede ser una fórmula, un vector de caracteres o un vector de enteros.
- map_ usa `...` para pasar argumentos adicionales a `.f` cada vez que es llamado
- Las funciones de map preservan los nombres.

En este ejemplo podemos dividir los datos de **mtcars** en tres piezas (por cilindro) y ajustar el mismo modelo linear para cada pieza:

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(
    function(datos) {
      lm(mpg ~ wt, data = datos)
    }
  )

models
```

```
## $`4`
##
## Call:
## lm(formula = mpg ~ wt, data = datos)
##
## Coefficients:
## (Intercept)           wt
##      39.571       -5.647
##
##
## $`6`
##
## Call:
## lm(formula = mpg ~ wt, data = datos)
##
## Coefficients:
## (Intercept)           wt
##       28.41        -2.78
##
##
## $`8`
##
## Call:
## lm(formula = mpg ~ wt, data = datos)
##
## Coefficients:
## (Intercept)           wt
##      23.868       -2.192
```