Pipes y Funciones

Alejandro Bellati

25 de mayo de 2021

Pipes

Hora de analizarlo un poco más...

- Herramienta poderosa para expresar claramente una secuencia de múltiples operaciones.

Pipes

Hora de analizarlo un poco más...

- Herramienta poderosa para expresar claramente una secuencia de múltiples operaciones.
- Alternativas
- Cuándo no usarlo
- Herramientas útiles relacionadas

Alternativas

El pipe viene del siguiente paquete, tidyverse lo carga automáticamente.

```
library(magrittr)
```

El objetivo del pipe es ayudarnos a escribir código más legible y entendible. Para analizar las alternativas:

El pequeño conejito Foo Foo saltando por el bosque rocogiendo ratones del campo y golpeándolos en la cabeza.

Supondremos que tenemos un objeto de R llamado *Foo_Foo*, y una función para cada verbo: *saltar()*, *recoger()* y *golpear()*.

Alternativas

Alternativa 1: guardar cada paso intermedio como un nuevo objeto.

```
foo_foo_1 <- saltar(foo_foo, a_traves = bosque)
foo_foo_2 <- recoger(foo_foo_1, que = ratones_del_campo)
foo_foo_3 <- golpear(foo_foo_2, en = cabeza)
```

Desventaja: obliga a nombrar cada paso intermedio. No son nombres naturales:

Alternativas

Alternativa 1: guardar cada paso intermedio como un nuevo objeto.

```
foo_foo_1 <- saltar(foo_foo, a_traves = bosque)
foo_foo_2 <- recoger(foo_foo_1, que = ratones_del_campo)
foo_foo_3 <- golpear(foo_foo_2, en = cabeza)
```

Desventaja: obliga a nombrar cada paso intermedio. No son nombres naturales:

1. El código está abarrotado con nombres poco importantes

Alternativas

Alternativa 1: guardar cada paso intermedio como un nuevo objeto.

```
foo_foo_1 <- saltar(foo_foo, a_traves = bosque)
foo_foo_2 <- recoger(foo_foo_1, que = ratones_del_campo)
foo_foo_3 <- golpear(foo_foo_2, en = cabeza)
```

Desventaja: obliga a nombrar cada paso intermedio. No son nombres naturales:

1. El código está abarrotado con nombres poco importantes
2. Hay que tener cuidado con los sufijos!, es fácil equivocarse.

Alternativas

Alternativa 1: guardar cada paso intermedio como un nuevo objeto.

```
foo_foo_1 <- saltar(foo_foo, a_traves = bosque)
foo_foo_2 <- recoger(foo_foo_1, que = ratones_del_campo)
foo_foo_3 <- golpear(foo_foo_2, en = cabeza)
```

Desventaja: obliga a nombrar cada paso intermedio. No son nombres naturales:

1. El código está abarrotado con nombres poco importantes
2. Hay que tener cuidado con los sufijos!, es fácil equivocarse.

No hay problemas de memoria. R guarda las columnas comunes de dataframes en el mismo lugar de memoria.

Alternativas

No hay problemas de memoria. R guarda las columnas comunes de dataframes en el mismo lugar de memoria.

```
diamantes2 <- diamantes %>%
  dplyr::mutate(precio_por_quilate = precio / quilate)

pryr::object_size(diamantes)

## Registered S3 method overwritten by 'pryr':
##  method      from
##  print.bytes Rcpp

## 3.46 MB

pryr::object_size(diamantes2)

## 3.89 MB

pryr::object_size(diamantes, diamantes2)

## 3.89 MB
```

Alternativas

Alternativa 2: sobreescribir cada paso intermedio:

```
foo_foo <- saltar(foo_foo, a_traves = bosque)
foo_foo <- recoger(foo_foo, que = ratones_del_campo)
foo_foo <- golpear(foo_foo, en = cabeza)
```

Alternativas

Alternativa 2: sobreescribir cada paso intermedio:

```
foo_foo <- saltar(foo_foo, a_traves = bosque)
foo_foo <- recoger(foo_foo, que = ratones_del_campo)
foo_foo <- golpear(foo_foo, en = cabeza)
```

1. Menos tipeo, menos que pensar por lo tanto menos error.

Alternativas

Alternativa 2: sobreescribir cada paso intermedio:

```
foo_foo <- saltar(foo_foo, a_traves = bosque)
foo_foo <- recoger(foo_foo, que = ratones_del_campo)
foo_foo <- golpear(foo_foo, en = cabeza)
```

1. Menos tipeo, menos que pensar por lo tanto menos error.
2. Depurar no es fácil

Alternativas

Alternativa 2: sobreescribir cada paso intermedio:

```
foo_foo <- saltar(foo_foo, a_traves = bosque)
foo_foo <- recoger(foo_foo, que = ratones_del_campo)
foo_foo <- golpear(foo_foo, en = cabeza)
```

1. Menos tipeo, menos que pensar por lo tanto menos error.
2. Depurar no es fácil
3. Repetimos el objeto 6 veces, es poco transparente lo que esta siendo cambiado.

Alternativas

Alternativa 3: Componer funciones

```
golpear(
  recoger(
    saltar(foo_foo, en = la_cabeza),
    arriba = raton_de_campo
  ),
  por_el = bosque
)
```

Se debe leer de adentro hacia afuera! muy confuso

Usando el Pipe

```
foo_foo %>%
  saltar(a_través = bosque) %>%
  recoger(que = ratones_campo) %>%
  golpear(en = cabeza)
```

Usando el Pipe

```
foo_foo %>%
  saltar(a_través = bosque) %>%
  recoger(que = ratones_campo) %>%
  golpear(en = cabeza)
```

1. Se enfoca en los verbos. Es una secuencia de acciones imperativas

Usando el Pipe

```
foo_foo %>%
  saltar(a_través = bosque) %>%
  recoger(que = ratones_campo) %>%
  golpear(en = cabeza)
```

1. Se enfoca en los verbos. Es una secuencia de acciones imperativas
2. Desventaja: hay que conocer %>%

Funcionamiento

¿Cómo funciona?

```
mi_pipe <- function(.) {
  . <- saltar(., a_traves = bosque)
  . <- recoger(., que = ratones_campo)
  golpear(., en = la_cabeza)
}
mi_pipe(foo_foo)
```

Funcionamiento

¿Cómo funciona?

```
mi_pipe <- function(.) {
  . <- saltar(., a_traves = bosque)
  . <- recoger(., que = ratones_campo)
  golpear(., en = la_cabeza)
}
mi_pipe(foo_foo)
```

Esto no funciona con cualquier función!!

Problemas

1. Funciones que usan el entorno actual!

```
assign("x",10)
x

## [1] 10

rm(list=ls())

"x" %>% assign(10)
x

## Error in eval(expr, envir, enclos): objeto 'x' no encontrado
```

```
rm(list=ls())

env <- environment()
"x" %>% assign(100, envir = env)
x

## [1] 100
```

Problemas

2. Lazy evaluation/Evaluación diferida o "peresosa": los argumentos solo se computan cuando la función los usa.

```
stop("!!!", call. = FALSE)

## Error: !!!

tryCatch(stop("!!!", call. = FALSE), error = function(e) "Un error")

## [1] "Un error"

stop("!") %>%
  tryCatch(error = function(e) "Un error")

## [1] "Un error"
```

```
sqrt(-1)

## Warning in sqrt(-1): Se han producido NaNs

## [1] NaN

sqrt(-1) %>%
  tryCatch(error = function(e) "Flores", warning = function(w) "Advertencia uiui")

## [1] "Advertencia uiui"
```

Cuándo no usar Pipe

Idealmente son usados para reescribir una secuencia lineal bastante corta. NO deberías usar pipes si:

Cuándo no usar Pipe

Idealmente son usados para reescribir una secuencia lineal bastante corta. NO deberías usar pipes si:

1. Pipes de más de 10 pasos: mejor crear objetos intermedios con nombres significativos. Más fácil depurar, más fácil de entender.

Cuándo no usar Pipe

Idealmente son usados para reescribir una secuencia lineal bastante corta. NO deberías usar pipes si:

1. Pipes de más de 10 pasos: mejor crear objetos intermedios con nombres significativos. Más fácil depurar, más fácil de entender.
2. Múltiples inputos y outputs: no hay un objeto principal a ser transformado.

Cuándo no usar Pipe

Idealmente son usados para reescribir una secuencia lineal bastante corta. NO deberías usar pipes si:

1. Pipes de más de 10 pasos: mejor crear objetos intermedios con nombres significativos. Más fácil depurar, más fácil de entender.
2. Múltiples inputos y outputs: no hay un objeto principal a ser transformado.
3. Un grafo dirigido con estructura compleja. El pipe es lineal.

Otras herramientas de magrittr

El pipe T, *%T>%*, funciona igual que el pipe común pero devuelve lo anterior.

```
x %>% f()
x %T>% f()
```

Ambos ejecutan *f(x)* pero la primera retorna *f(x)* mientras que la segunda retorna *x*.

```
rnorm(100) %>%
  matrix(ncol = 2) %>%
  plot() %>%
  str()

##  NULL
```

```
rnorm(100) %>%
  matrix(ncol = 2) %T>%
  plot() %>%
  str()

##  num [1:50, 1:2] -1.048 -0.583 -1.128 0.176 0.38 ...
```

Otras herramientas de magrittr

Para funciones que no aceptan dataframes, el siguiente operador me permite referirme a las variables del dataframe de manera explícita.

```
mtautos %$%
  cor(cilindrada, millas)

## [1] -0.8475514
```

Para asignaciones se tiene:

```
mtautos %<>% transform(cilindros = cilindros * 2)
```

No se recomienda, pues el libro es defensor de que una asignación es una operación especial que debe ser clara cuando sucede.

Funciones

Las funciones permiten automatizar algunas tareas comunes de manera más poderosa que copiar y pegar:

Funciones

Las funciones permiten automatizar algunas tareas comunes de manera más poderosa que copiar y pegar:

1. Puedes dar a la función un nombre evocador que hará tu código más fácil de leer

Funciones

Las funciones permiten automatizar algunas tareas comunes de manera más poderosa que copiar y pegar:

1. Puedes dar a la función un nombre evocador que hará tu código más fácil de leer
2. A medida que cambien los rquerimientos, solo necesitarás cambiar tu código en un solo lugar, en vez de en varios lugares.

Funciones

Las funciones permiten automatizar algunas tareas comunes de manera más poderosa que copiar y pegar:

1. Puedes dar a la función un nombre evocador que hará tu código más fácil de leer
2. A medida que cambien los rquerimientos, solo necesitarás cambiar tu código en un solo lugar, en vez de en varios lugares.
3. Eliminas las probabilidades de errores accidentales cuando copias y pegas.

Funciones

Las funciones permiten automatizar algunas tareas comunes de manera más poderosa que copiar y pegar:

1. Puedes dar a la función un nombre evocador que hará tu código más fácil de leer
2. A medida que cambien los rquerimientos, solo necesitarás cambiar tu código en un solo lugar, en vez de en varios lugares.
3. Eliminas las probabilidades de errores accidentales cuando copias y pegas.

Objetivo: introducirse al tema con consejos prácticos y consejos de estilo.

¿Cuándo utilizar?

Deberías considerar escribir una función siempre que has copiado y pegado un bloque de código más de dos veces.

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

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
 (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
 (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
 (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
 (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

¿Cuándo utilizar?

Deberías considerar escribir una función siempre que has copiado y pegado un bloque de código más de dos veces.

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

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
 (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
 (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
 (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
 (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

Repetir código lleva a errores. Extraer el código repetido en una función es buena idea.

Escribiendo una función

```
y <- (x - min(x, na.rm = TRUE)) /
 (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
```

Estamos calculando el rango de los datos tres veces. Sacar cálculos intermedio en variables nombradas es una buena práctica.

```
rescale01 <- function(x) {
 rng <- range(x, na.rm = TRUE)
 (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(0,5,10))

## [1] 0.0 0.5 1.0
```

1. Necesitas elegir un nombre para la función, aquí *rescale01*.

Escribiendo una función

```
y <- (x - min(x, na.rm = TRUE)) /
 (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
```

Estamos calculando el rango de los datos tres veces. Sacar cálculos intermedio en variables nombradas es una buena práctica.

```
rescale01 <- function(x) {
 rng <- range(x, na.rm = TRUE)
 (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(0,5,10))

## [1] 0.0 0.5 1.0
```

1. Necesitas elegir un nombre para la función, aquí *rescale01*.
2. Listar los inputs, o argumentos, dentro de *function*.

Escribiendo una función

```
y <- (x - min(x, na.rm = TRUE)) /
 (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
```

Estamos calculando el rango de los datos tres veces. Sacar cálculos intermedio en variables nombradas es una buena práctica.

```
rescale01 <- function(x) {
 rng <- range(x, na.rm = TRUE)
 (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(0,5,10))

## [1] 0.0 0.5 1.0
```

1. Necesitas elegir un nombre para la función, aquí *rescale01*.
2. Listar los inputs, o argumentos, dentro de *function*.
3. Situar el código que has creado en el cuerpo de la función.

Escribiendo una función

```
y <- (x - min(x, na.rm = TRUE)) /
 (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
```

Estamos calculando el rango de los datos tres veces. Sacar cálculos intermedio en variables nombradas es una buena práctica.

```
rescale01 <- function(x) {
 rng <- range(x, na.rm = TRUE)
 (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(0,5,10))

## [1] 0.0 0.5 1.0
```

1. Necesitas elegir un nombre para la función, aquí *rescale01*.
2. Listar los inputs, o argumentos, dentro de *function*.
3. Situar el código que has creado en el cuerpo de la función.

1. No modifica la variable de entrada *x*.

Escribiendo una función

```
y <- (x - min(x, na.rm = TRUE)) /
 (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
```

Estamos calculando el rango de los datos tres veces. Sacar cálculos intermedio en variables nombradas es una buena práctica.

```
rescale01 <- function(x) {
 rng <- range(x, na.rm = TRUE)
 (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(0,5,10))

## [1] 0.0 0.5 1.0
```

1. Necesitas elegir un nombre para la función, aquí *rescale01*.
2. Listar los inputs, o argumentos, dentro de *function*.
3. Situar el código que has creado en el cuerpo de la función.

1. No modifica la variable de entrada *x*.
2. Regla de retorno estándar: una función devuelve el último valor que calculó.

Más comentarios

El código se simplifica:

```
df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

Aún tiene cierta repetición el código, esto se atacará en iteración.

Es buena idea chequear la función:

```
rescale01(c(-10,0,10))

## [1] 0.0 0.5 1.0

rescale01(c(1,2,3,NA,5))

## [1] 0.00 0.25 0.50   NA 1.00
```

Esto es un test interactivo informal. Hay test formales y automatizados (*unit testing*).

Otras ventajas

```
x <- c(1:10, Inf)
rescale01(x)

##  [1]   0   0   0   0   0   0   0   0   0   0 NaN
```

Si nuestros requerimientos sobre al función cambian, solo tenemos que hacer cambio en un lugar sólo.

```
rescale01 <- function(x) {
 rng <- range(x, na.rm = TRUE, finite = TRUE)
 (x - rng[1]) / (rng[2] - rng[1])
}
```

Esta es una importante parte del principio de "no repetirse a uno mismo". Cuanta más repetición tengas en tu código, más lugares tendrás que recordar de actualizar cuando las cosas cambien (¡y eso siempre sucede!), y es más probable que crees errores (bugs) a lo largo del tiempo.

Consejos de estílo

Las funciones son para las computadores y para los seres humanos. A R no le importa qué nombre le des a tu función pero a los lectores sí.

1. El nombre de una función idealmente debería ser corto pero que transmita claramente que es lo que hace. Mejor claro que corto
2. En general deberían ser verbos, pero hay excepciones, sobre todo cuando el verbo es muy amplio: *obtener*, *calcular*, *hallar*.

```
# Muy corto
f()

# No es un verbo y es poco descriptivo
my_awesome_function()

# Largos, pero descriptivos
imputar_faltantes()
colapsar_anios()
```

3. Nombres con múltiples palabras con *snake_case*: *reemplazar_valores_faltantes*. Lo importante es ser consistentes, otro formato es *camelCase*.

Consejos de estílo

4. Si tienes familia de funciones que hacen cosas similiares hace que tengan nombres ya argumentes consistentes. Utiliza un prefijo común para indicar que estan conectadas. (Mejor que sufijo común)

```
# Bien
input_select()
input_checkbox()
input_text()

# No tan bien
select_input()
checkbox_input()
text_input()
```

5. No sobrescribas funciones y variables ya existentes.
6. Usa comentarios para explicar el *porqué*, no el *qué* ni el *cómo*. Si no se entiende el código mejor reescribirlo. Los comentarios capturan la razón de tus decisiones.
7. Comentario como encabezados para dividir tu archivo en partes (Ctrl + Shift + R):

```
# Cargar los datos ------------------------------------

# Graficar los datos ----------------------------------
```

Ejecución condicional

Una sentencia *if* permite ejecutar código condicional:

```
tiene_nombre <- function(x) {
 nms <- names(x)
 if (is.null(nms)) {
 rep(FALSE, length(x))
 } else {
 !is.na(nms) & nms != ""
 }
}
```

```
 if((b != 0) && (a/b \n´eq 7 )){
   ...
 }
```

1. La condición debe evaluar como TRUE o FALSE, longitud 1.
2. Hay que usar // y *&&*, no la versión vectorizada. Estas expresiones hacen cortocircuito.
3. Cuidado con == que esta vectorizada. Considerar *identical()* (cuidado tipos y punto flotante) o *near()*. También *all()* y *any()*.

Condiciones múltiples

Encadenar múltiplas condiciones juntas:

```
if (this) {
 # haz aquello
} else if (that) {
 # haz otra cosa
} else {
 #
}
```

Considerar también usar switch:

```
operar<-function(x,y,op){
 switch(op,
   plus = x+y,
   minus = x-y,
   times = x*y,
   divide = x/y,
   stop("!Operación desconocida")
 )
}
operar(7,21,"divide")

## [1] 0.3333333
```

Estílo del código

1. Tanto *if* como function deberían ir casi siempre con las llaves *{}*, con el contenido una sangría de dos espacios.
2. La llave de apertura nunca debe ir en su propia línea y siempre debe ser seguida de una nueva línea. La llave de cierre debe siempre ir en su propia línea, a menos que sea seguida por else.

```
# Bien
if (y < 0 && debug) {
 message("Y es negativo")
}
if (y == 0) {
 log(x)
} else {
 y ^ x
}
# Mal ------------------------------------
if (y < 0 && debug)
message("Y is negative")
if (y == 0) {
 log(x)
}
else {
 y ^ x
}
#Esta bien evitar las llaves {} si es muy corto el if. ----------------------------------
x <- if (y < 20) "Too low" else "Too high"
```

Argumentos de funciones

Se dividen en dos, los datos y los argumentos que controlan los detalles. En general los datos deben ir primero y los detalles después, o no colocar (asumen valores por defecto).

Al programar una función los valores por defecto se indican de la misma manera que se llama a una función con un argumento nombrado.

```
mean_ci <- function(x, conf = 0.95) {
 se <- sd(x) / sqrt(length(x))
 alpha <- 1 - conf
 mean(x) + se * qnorm(c(alpha / 2, 1 - alpha / 2))
}

x <- runif(100)
mean_ci(x)

## [1] 0.4666133 0.5748691

mean_ci(x, conf = 0.99)

## [1] 0.4496051 0.5918773
```

En general los valores por defecto deben ser el valor más común de la variable, hay excepciones: $na.rm = TRUE$ es mala idea.

Argumentos de funciones

1. Al llamar a una función generalmente se omiten los nombres de las variables de datos.
2. Si quieres usar los argumentos de detalle debes usarlos con nombre, o prefijo si este es único.
3. Es buena práctica dejar un espacio alrededor de = y un espacio luego de la coma. Ayuda a leer.

```
# Bien
mean(1:10, na.rm = TRUE)
mean(1:10, n = FALSE)

# Mal
mean(x = 1:10, FALSE)
```

En general es preferible usar nombres largos y descriptivos pero hay algunos cortos que son típicos:

- x,y,z: vectores
- w: un vector de pesos
- df: un data frame.
- i,j: índices numéricos
- n: longitud, o número de filas.
- p: número de columnas.
- na.rm: eliminar o no los faltantes.

Argumentos de funciones

Es fácil llamar a una función con entradas inválidas. Para evitar esto es útil verificar las condiciones previas importantes y arrojar un error (con stop(), parar), si estas no son verdaderas:

```
wt_mean <- function(x, w) {
 sum(x * w) / sum(w)
}
wt_mean(1:6,1:3)

## [1] 7.666667

wt_mean <- function(x, w) {
 if (length(x) != length(w)) {
 stop("`x` y `w` deben tener la misma extensión", call. = FALSE)
 }
 sum(w * x) / sum(w)
}

wt_mean(1:6,1:3)

## Error: ‘x‘ y ‘w‘ deben tener la misma extensión
```

Argumentos de funciones

Una utilidad para esto es *stopifnot()*:

```
wt_mean <- function(x, w, na.rm = FALSE) {
 stopifnot(is.logical(na.rm), length(na.rm) == 1)
 stopifnot(length(x) == length(w))

 if (na.rm) {
 miss <- is.na(x) | is.na(w)
 x <- x[!miss]
 w <- w[!miss]
 }
 sum(w * x) / sum(w)
}
wt_mean(1:6, 6:1, na.rm = "foo")

## Error in wt_mean(1:6, 6:1, na.rm = "foo"): is.logical(na.rm) is not TRUE
```

Argumentos de funciones

- Muchas funciones en R admiten una cantidad arbitraria de entradas. Para eso usan el argumento especial ..., llamado punto-punto-punto o ellipsis.
- Captura cualquier número de argumentos que no estén contemplados de otra forma.
- Se puede pasar los argumentos de ... a otra función.
- Cuidado con los argumentos mál escritos.

```
commas <- function(...){
  stringr::str_c(..., collapse = ", ")
}
commas(letters[1:10])

## [1] "a, b, c, d, e, f, g, h, i, j"

rule <- function(..., pad = "-") {
 title <- paste0(...)
 width <- getOption("width") - nchar(title) - 5
 cat(title, " ", stringr::str_dup(pad, width), "\n", sep = "")
}
rule("Important output")

## Important output ----------------------------------------------------------
```

Argumentos de funciones

```
(z<-c(a=3,b=2,4,5,6,7))

## a b
## 3 2 4 5 6 7

names(z)

## [1] "a" "b" ""  ""  ""  ""

#Cuidado:
x <- c(1, 2)
sum(x, na.mr = TRUE)

## [1] 4
```

- Podemos darles nombres a las variables.
- Ojo, si escribímos mal el nombre de un argumento de la función pensará que es un argumento nuevo y no lo detectara como error.
- Si solo quieres capturar los valores de ..., entonces utiliza *list(...)*.

Argumentos de funciones

```
(z<-c(a=3,b=2,4,5,6,7))

## a b
## 3 2 4 5 6 7

names(z)

## [1] "a" "b" ""  ""  ""  ""

#Cuidado:
x <- c(1, 2)
sum(x, na.mr = TRUE)

## [1] 4
```

- Podemos darles nombres a las variables.
- Ojo, si escribímos mal el nombre de un argumento de la función pensará que es un argumento nuevo y no lo detectara como error.
- Si solo quieres capturar los valores de ..., entonces utiliza *list(...)*.

Recordar evaluación diferida! Los arguments que nunca se usan ni se llaman.

Valores de Retorno

Hay cosas importantes a considerar al retornar un valor.

1. ¿Cuándo retornar un valor? El valor devuelto es la última sentencia que la función evalua. Pero, se puede optar por retornar algo anticipadamente, y esto puede hacer el código más legible. Por ejemplo si los argumentos estan vacíos:

```
complicated_function <- function(x, y, z) {
 if (length(x) == 0 || length(y) == 0) {
 return(0)
 }
 # Código complicado aquí
}
```

Valores de Retorno

O si hay un if con un bloque complicado y uno sencillo.

```
f <- function() {
 if (x) {
 # Haz
 # algo
 # que
 # tome
 # muchas
 # líneas
 } else {
 # retorna algo corto
 }
}

# ES PREFERIBLE:

f <- function() {
 if (!x) {
 return(algo_corto)
 }

 # Haz
 # algo
 # que
 # tome
 # muchas
 # líneas
}
```

Valores de retorno

2. Escribir funciones aptas para Pipes.

Para lograr esto es importante ver el tipo de objeto de tu valor de retorno.

Hay dos tipos básicos de funciones aptas para pipes:

1. **Transformaciones**: se ingresa un objeto como primer argumento, se devuelve una versión modificada del mismo.
2. **Efectos secundarios**: el objeto ingresado no se modifica. La función realiza una acción sobre el objeto (como graficar). Estas funciones deben retornar invisiblemente el primer argumento para que puedan ser utilizados en una secuencia de pipes.

```
mostrar_faltantes <- function(df) {
 n <- sum(is.na(df))
 cat("Valores faltantes: ", n, "\n", sep = "")
 invisible(df)
}
mostrar_faltantes(mtautos)

## Valores faltantes: 0

x<-mostrar_faltantes(mtautos)

## Valores faltantes: 0

class(x)

## [1] "data.frame"
```

Entorno

Un componente importante de una función es su entorno. El entorno de una función controla cómo R encuentra el valor asociado a un nombre.

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

En muchos lenguajes esto sería un error, porque *y* no esta definida dentro de la función. Sin embargo R sigue la reglas de *ámbito léxico* para encontrar el valor asociado a un nombre. Como *y* no esta definida entonces R busca en el entorno donde la función esta definida.

```
y <- 100
f(10)

## [1] 110

y <- 1000
f(10)

## [1] 1010
```

Entorno

Es una receta para bugs. La ventaja es desde el punto de vista del lenguaje, permite que R sea muy consistente. Cada nombre es buscado usando el mismo conjunto de reglas. Eso incluye hasta la suma!

```
`+` <- function(x, y) {
 if (runif(1) < 0.1) {
 sum(x, y)
 } else {
 sum(x, y) * 1.1
 }
}
table(replicate(1000, 1 + 2))

##
##   3 3.3
## 109 891

rm(`+`)
```

R pone pocos límites. Podes hacer cosas que son extremadamente desaconsejables. Pero esta flexibilidad es lo que hace que herramientas como *ggplot2* y *dplyr* sean posibles.