Clase 3.0
Scripts, funciones y control de flujo

Marcos Rosetti y Luis Pacheco-Cobos

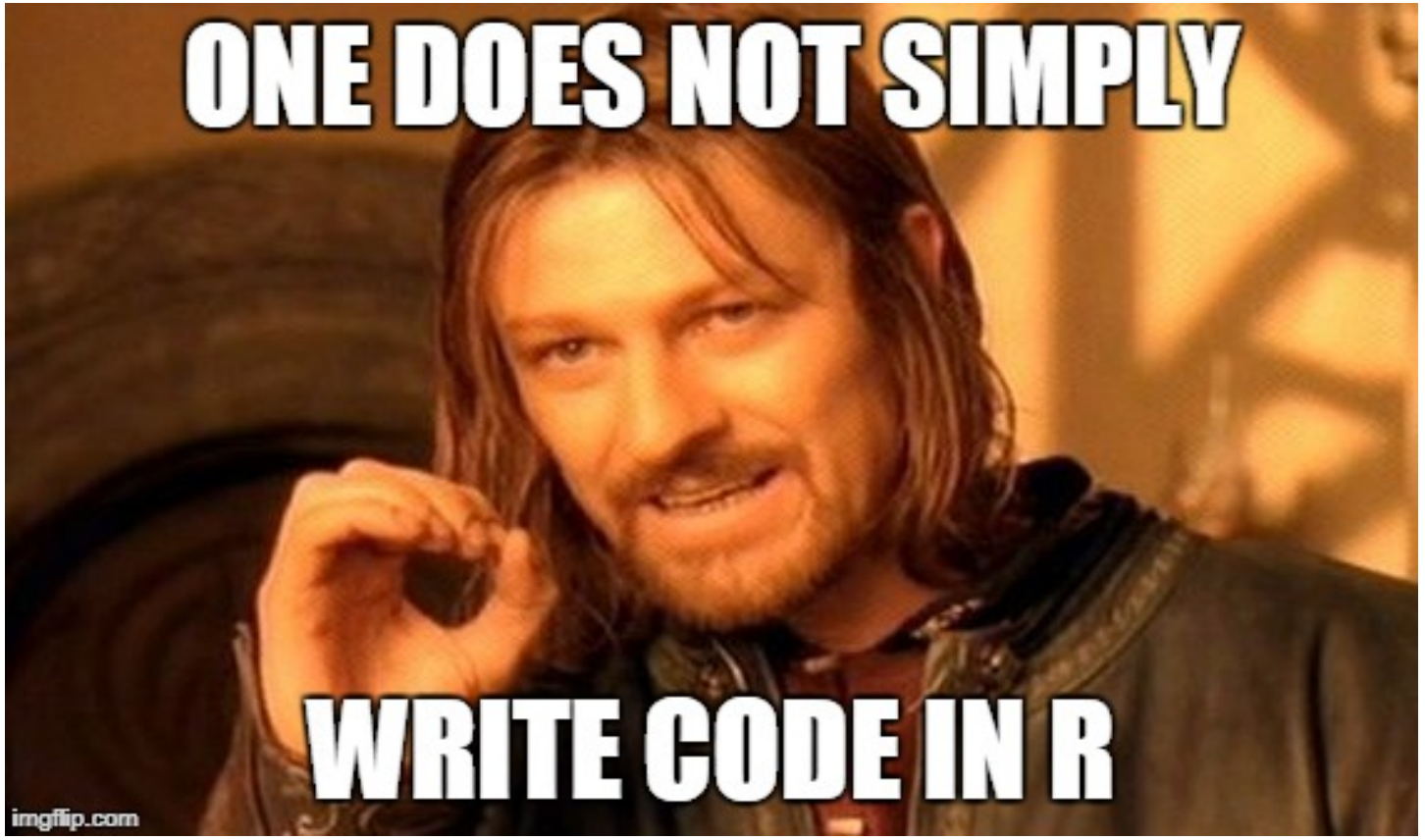
Estadística y Manejo de Datos con R (EMDR) — Virtual

Scripts

Scripts

- ¿Para qué un *script*?
  - Automatizar un código que queremos correr múltiples veces
  - Crear y guardar código que sirve para múltipless propósitos
  - Organizar en modulos editables un proceso largo y complejo que rebasa las capacidades de la línea de comando

Scripts

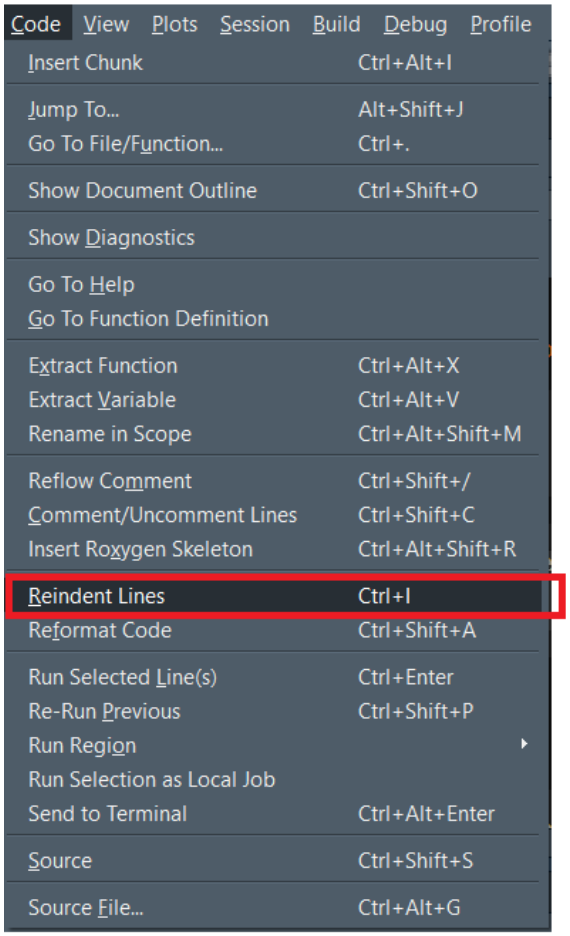


Scripts

- Un código debe estar bien indentado y no en una sola línea muy larga

```
long_function_name <- function(a = "a long string",
                               b = "another long string",
                               c = "an even longer string") {
}
```

Scripts


Code View Plots Session Build Debug Profile
Insert Chunk Ctrl+Alt+I
Jump To... Alt+Shift+J
Go To File/Function... Ctrl+.
Show Document Outline Ctrl+Shift+O
Show Diagnostics
Go To Help
Go To Function Definition
Extract Function Ctrl+Alt+X
Extract Variable Ctrl+Alt+V
Rename in Scope Ctrl+Alt+Shift+M
Reflow Comment Ctrl+Shift+/
Comment/Uncomment Lines Ctrl+Shift+C
Insert Roxygen Skeleton Ctrl+Alt+Shift+R
Reindent Lines Ctrl+I
Reformat Code Ctrl+Shift+A
Run Selected Line(s) Ctrl+Enter
Re-Run Previous Ctrl+Shift+P
Run Region
Run Selection as Local Job
Send to Terminal Ctrl+Alt+Enter
Source Ctrl+Shift+S
Source File... Ctrl+Alt+G

Scripts

```
points(Animals[rownames(Animals) == "African elephant", ],
     pch = 8, col = "red", cex = 2)

mPG <- PlantGrowth %>%
 group_by(group) %>%
 summarize(mw = mean(weight), sdw = sd(weight))

ggplot(data = diamonds , aes(x = price ,y = carat , color = color )) +
 geom_point() +
 facet_grid(.~ cut) +
 xlab("Precio") +
 ylab("Carats")
```

Scripts

- Nombres descriptivos para tus funciones

```
# Good

fit_models.R

utility_functions.R

# Bad

foo.r

stuff.r
```

Scripts

- Nombres descriptivos para tus objetos

```
# Good
day_one
day_1
# Bad
first_day_of_the_month
DayOne
dayone
djm1
```

Scripts


Untitled35.R
ME
Untitled34.R
Untitled33.R
Untitled32.R
Untitled31.R
.
.
.

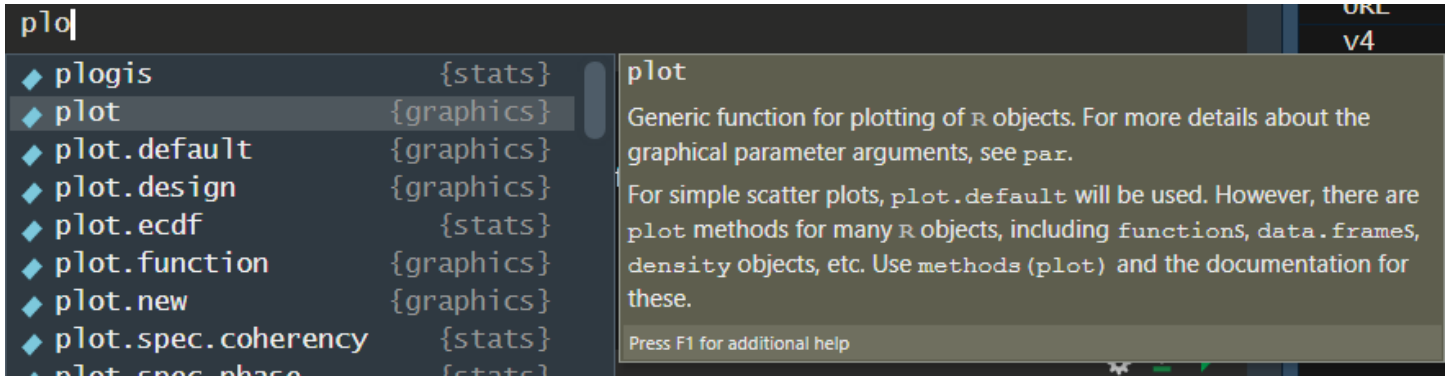
Scripts

- Evita usar/renombrar funciones existentes

```r
# Bad

T <- FALSE

c <- 10

mean <- function(x) { sum(x) }
```

Scripts

- Usa la función de autosugerencia para escribir los nombres de las funciones, variables, etc.



Scripts

- Espacios entre operadores y después de la coma hacen el código más legible

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

Scripts

- Excepciones

```
plot(x) # Good
plot (x) # Bad
plot( x ) # Bad

base::get # Good
base :: get # Bad
```

Scripts

- Conflictos entre paquetes
- Hay paquetes que tienen funciones con el mismo nombre
  - (p.e. `summarise` de `dplyr` y `summarise` de `MASS`)
- Cuando cargas un paquete, este anula la función del paquete previo
- Podemos hacer referencia a una función sin cargar la libreria con el operador `::`

```
dplyr::summarise()
MASS:summarise()
```

Scripts

- Asignación

```r
# Good
x <- 5

# Bad
x = 5
```

Scripts

Scripts

- Comentarios

```
# Load data --------------------------

# Run model --------------------------

# Plot data --------------------------
```

Scripts

- Ejemplo

```
# Marcos Rosetti, 31/12/ 2019 - Este codigo es un ejemplo!
rm(list =ls()) # remueve variables del workspace
graphics.off() # cierra todas las ventanas de graficos

# Paquetes (ya instalados, solo cargar)
library(dplyr)
library(tidyr)

# Cargo datos
datos <- read.table("mis_datos.csv", header =T , sep =",")

# Analisis -------
```

Funciones

Funciones

- Una función es una forma de describir una relación entre variables

```
y <- m * x + b
```

```
recta <- function(m, x, b) {
  y <- m * x + b
return(y)
}
y <- recta(m = 1, x = 1:10, b = 3)
plot(y)
```

Funciones

- Estructura básica

```
nombre <- function(arg.1, arg.2, arg.n) {
 contenido # va dentro de las llaves
return(resultado)
}
```

- Llamado

```
resultado <- nombre(arg.1, arg.2, arg.n) # argumentos van entre los parentesis
```

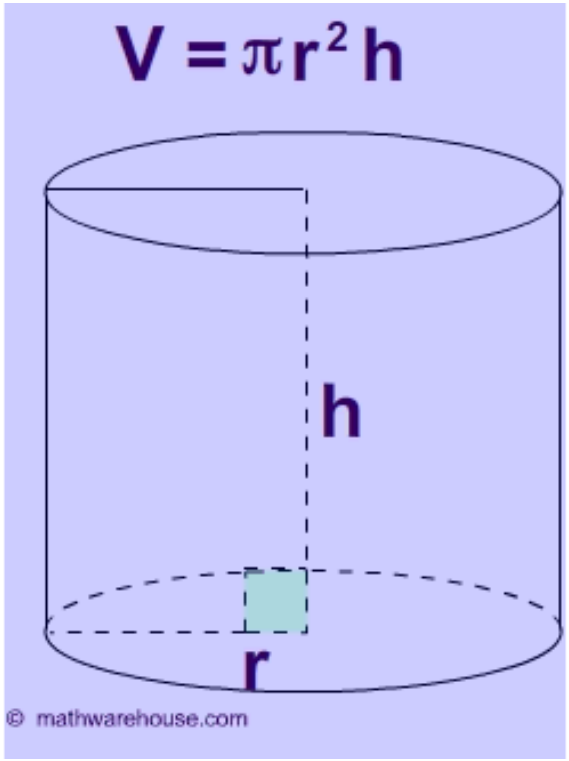
Funciones

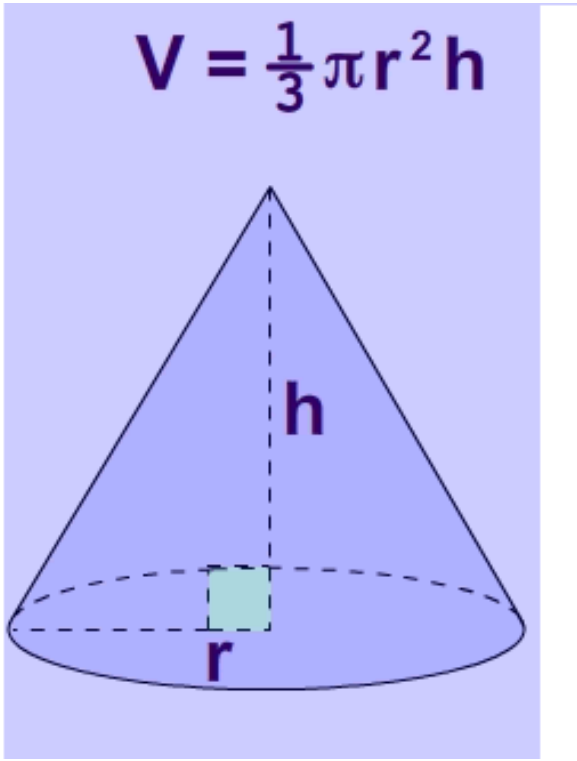
- Valores por defecto

```
suma <- function(a = 1, b = 1) {
  c = a + b
  return(c)
}
suma()
suma(2, 2)
```

Ejercicio: Funciones

- Escribe una función para calcular el (1) volumen de un cilindro y (2) el volumen de un cono





Ejercicio: Funciones

- Solución

```
vcilindro <- function(r, h) {
 v <- r^2 * pi * h
return(v)
}
vcilindro(10, 35)
```

```
## [1] 10995.57
```

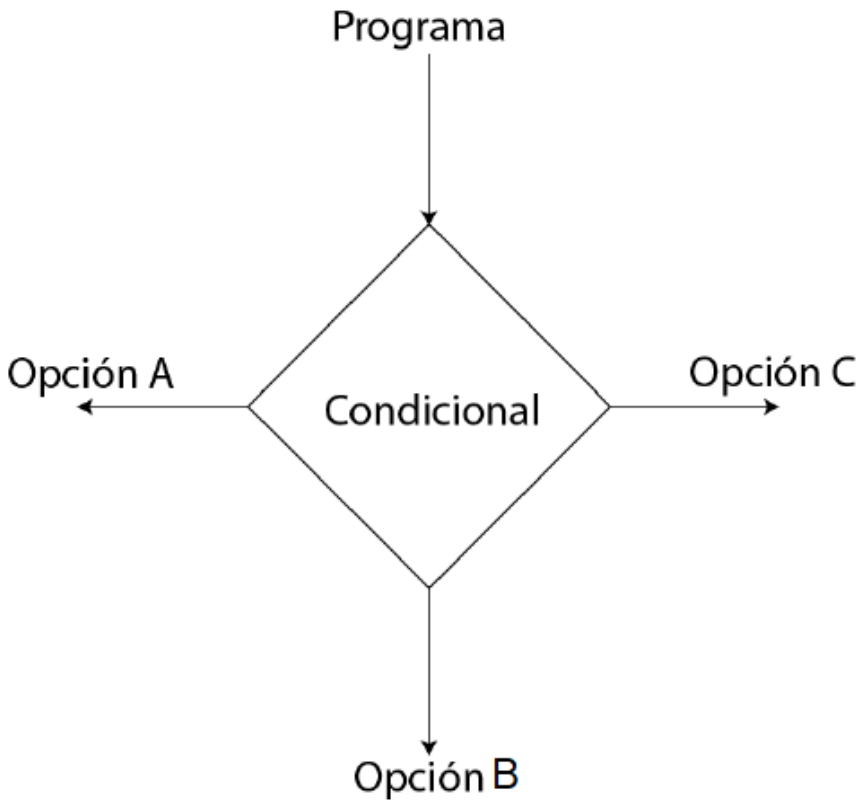
```
vcono <- function(r, h) {
  v <- vcilindro(r, h) * (1 / 3)
return(v)
}
vcono(52, 83)
```

```
## [1] 235024.6
```

Control condicional

Control condicional

- Condiciona la ejecución de cierto código a una condición lógica



Control condicional

- if

```
a <- 5 # intenta con a <- -5
if (a > 0) {
  print("el valor es mayor a zero")
}
```

```
## [1] "el valor es mayor a zero"
```

Control condicional

- `else`

```
a <- 5 # intenta con a <- -5
if (a > 0) {
  print("el valor es mayor a zero")
} else {
  print("el valor es menor a zero")
}
```

```
## [1] "el valor es mayor a zero"
```

Control condicional

- `else if`

```
a <- 0
if (a > 0) {
  print("el valor es positivo")
} else if (a == 0) {
  print("el valor es zero")
}
```

```
## [1] "el valor es zero"
```

Control condicional

- Comparativo de los caracteres boleanos

Símbolo	Operación
x == y	equivalencia
x != y	diferencia
!x	negación lógica
x & y	y
x | y	o

Control condicional

- Al comparar un escalar con un vector, se compara elemento por elemento

```
a <- 1
b <- c(1:5)
a == b
```

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

- Comprobar si un escalar está presente entre los elementos de un vector

```
a %in% b
```

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

Control condicional

- Para comparaciones entre vectores usamos `all()`:

```
v1 <- c("A", "B", "C", "D")
v2 <- v1

all(v1 == v2)
```

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

```
v3 <- c("A", "C", "C", "E")
all(v1 == v3)
```

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

Control condicional

- Cuidado con el reciclado:

```
v4 <- c("A","B","A","B")
v5 <- c("A","B")
all(v4 == v5)
```

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

```
all(length(v4) == length(v5)) & all(v4 == v5)
```

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

Control condicional

- ¿Son identicos?

```
v1 <- c("A", "B", "C", "D")
v3 <- c("A", "C", "C", "E")
identical(v1 ,v3)
```

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

- ¿Cuáles son distintos?

```
which(v1 != v3)
```

```
## [1] 2 4
```

Control condicional

- Y muchos otros

```
union(x, y)

intersect(x, y)

setdiff(y, x)

setequal(x, y)

duplicated(x)

unique(x)
```

Ejercicio: Control condicional

- Crea una función que imprima un vector si este tiene una longitud mayor de 3
  - tip: `length()`
- Crea una función que te imprima si un número es par o impar
  - tip: `round()`
- Crea una función que te diga si lo que se ingresa es un text o un número
  - tip: `is.numeric()`, `is.character()`

Ejercicio: Control condicional

- Soluciones (1era parte)

```
lthan3 <- function(v) {
  if(length(v) > 3) {
    print(v)
    }
  }
v2 <- c("a","b")
lthan3(v2)
v3 <- c("a","e","i","o","u")
lthan3(v3)
```

```
## [1] "a" "e" "i" "o" "u"
```

Ejercicio: Control condicional

- Soluciones (2nda parte)

```
is.pair <- function(i) {
  if((i / 2) - round(i / 2) == 0) {
    print(TRUE)
    }
  else {
    print(FALSE)
    }
}
i <- 11
is.pair(i)
```

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

Ejercicio: Control condicional

- Soluciones (3era parte)

```
what.is.i <- function(i){
  if(is.character(i)) {
    print("it's a string ")
    }
  else if(is.numeric(i)) {
    print("it's a number ")
    }
  }
i <- "Xalapa"
what.is.i(i)
```

```
## [1] "it's a string "
```

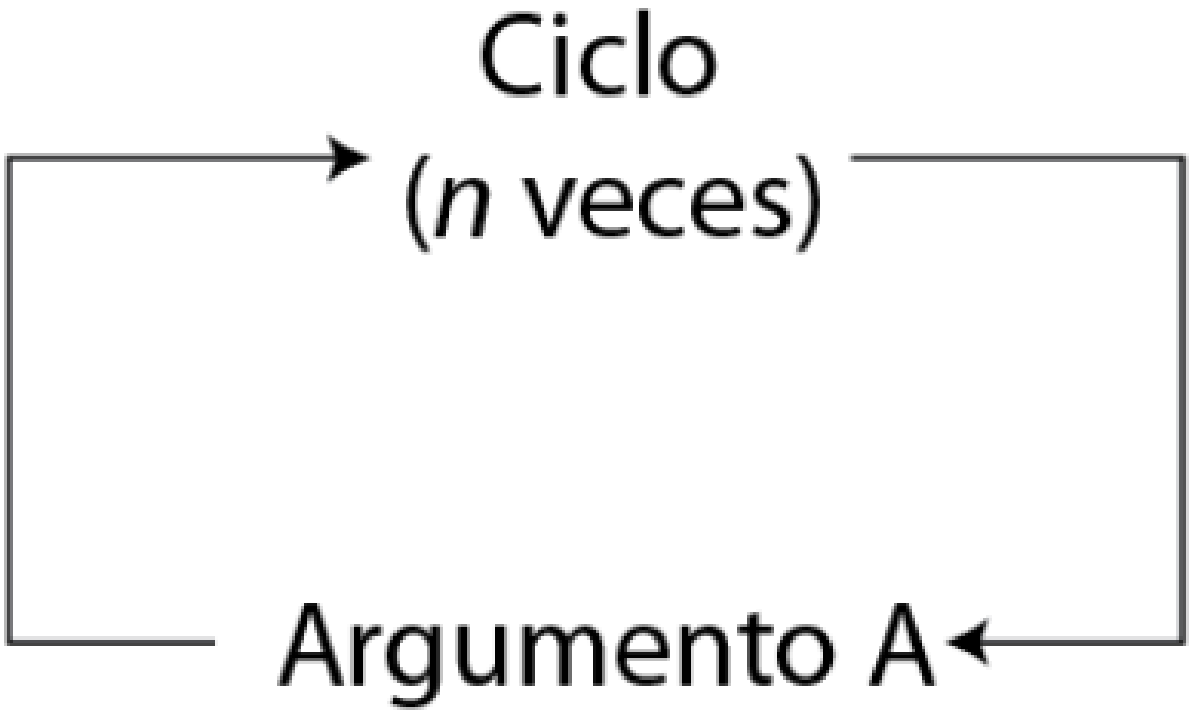
```
i <- 3.1416
what.is.i(i)
```

```
## [1] "it's a number "
```

Control iterativo

Control iterativo

- El control iterativo permite ejecutar una pieza de código un número determinado de veces



Control iterativo

```
for (i in inicio:fin) {
  # operaciones a realizar
  }
```

Control iterativo

- Uso

```
for (i in 1:10) {
  print(i)
  }
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

Control iterativo

- Uso

```r
abc <- seq(1 ,10 ,1) # con seq()
for (i in abc) {
  print(i)
  }
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

Control iterativo

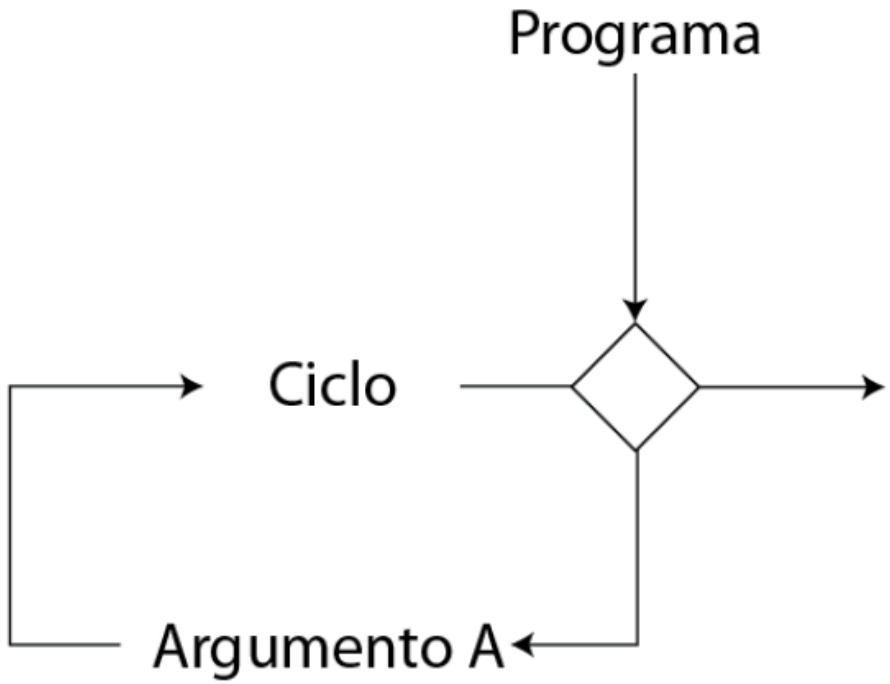
- Uso

```
abc <- letters [1:10] # con length()
for (i in 1:length(abc)) {
  print(i)
  }
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

Control iterativo

- El control iterativo revisa si se cumple una condición, ejecuta un código y vuelve a comenzar el ciclo



Control iterativo

- `while()`

```
while (condicion == TRUE) {
  # operaciones
  }
```

Control iterativo

- Se escapa usando un umbral

```
i <- 0
while (i < 10) {
  i <- i + 1 # el contador no incrementa autommaticamente
  print(i)
  }
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

Ejecicios: Control iterativo

- Escribe un loop que imprima todas las letras del abecedario, excepto las vocales - estas deben ser impresas en mayúsculas
    - tip: `letters`, `toupper()`

Ejecicios: Control iterativo

- Solución

```
for (i in letters) {
  if (i %in% c("a", "e", "i", "o", "u")) {
    print(toupper(i))
  } else {
    print(i)
  }
}
```

```
## [1] "A"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "E"
## [1] "f"
## [1] "g"
## [1] "h"
## [1] "I"
## [1] "j"
## [1] "k"
## [1] "l"
## [1] "m"
## [1] "n"
## [1] "O"
## [1] "p"
## [1] "q"
## [1] "r"
## [1] "s"
## [1] "t"
## [1] "U"
## [1] "v"
```

Programación funcional

Programación funcional

- Es un paradigma de programación declarativa basado en el uso de funciones matemáticas

Programación funcional

Programación funcional

- purrr

```
install.packages("tidyverse")
install.packages("purrr")
```

```
library(purrr)
library(tidyr)
library(dplyr)
library(broom)
```

Programación funcional

- `nest()` y `unnest()`

```
head(mtcars)
?mtcars
```

```
n_mtcars <- mtcars %>%
  nest(-cyl) # produce un df de listas
```

```
## Warning: All elements of `...` must be named.
## Did you want `data = c(mpg, disp, hp, drat, wt, qsec, vs, am, gear, carb)`?
```

```
n_mtcars
```

```
## # A tibble: 3 x 2
##     cyl data
##   <dbl> <list>
## 1     6 <tibble [7 × 10]>
## 2     4 <tibble [11 × 10]>
## 3     8 <tibble [14 × 10]>
```

Programación funcional

- `unnest()`

```
n_mtcars %>%
  unnest()
```

```
## Warning: `cols` is now required when using unnest().
## Please use `cols = c(data)`
```

```
## # A tibble: 32 x 11
##      cyl   mpg  disp    hp  drat    wt  qsec    vs    am  gear  carb
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1     6  21    160    110  3.9   2.62  16.5     0     1     4     4
##  2     6  21    160    110  3.9   2.88  17.0     0     1     4     4
##  3     6  21.4  258    110  3.08  3.22  19.4     1     0     3     1
##  4     6  18.1  225    105  2.76  3.46  20.2     1     0     3     1
##  5     6  19.2  168.   123  3.92  3.44  18.3     1     0     4     4
##  6     6  17.8  168.   123  3.92  3.44  18.9     1     0     4     4
##  7     6  19.7  145    175  3.62  2.77  15.5     0     1     5     6
##  8     4  22.8  108     93  3.85  2.32  18.6     1     1     4     1
##  9     4  24.4  147.    62  3.69  3.19  20       1     0     4     2
## 10     4  22.8  141.    95  3.92  3.15  22.9     1     0     4     2
## # … with 22 more rows
```

Programación funcional

- `map()`

```
my_test <- function(x) {
  lm(mpg ~ wt, data=x)
  }

mtcars %>%
  nest(-cyl) %>%
  mutate(res = map(data,my_test))
```

```
## Warning: All elements of `...` must be named.
## Did you want `data = c(mpg, disp, hp, drat, wt, qsec, vs, am, gear, carb)`?
```

```
## # A tibble: 3 x 3
##     cyl data               res
##   <dbl> <list>             <list>
## 1     6 <tibble [7 × 10]>  <lm>
## 2     4 <tibble [11 × 10]> <lm>
## 3     8 <tibble [14 × 10]> <lm>
```

Programación funcional

- `map()`

```
my_test <- function(x) {
  lm(mpg ~ wt, data=x)
  }

mtcars %>%
  nest(-cyl) %>%
  mutate(res = map(data,my_test)) %>%
  mutate(glance_lm = res %>% map(glance))
```

```
## Warning: All elements of `...` must be named.
## Did you want `data = c(mpg, disp, hp, drat, wt, qsec, vs, am, gear, carb)`?
```

```
## # A tibble: 3 x 4
##     cyl data               res    glance_lm
##   <dbl> <list>             <list> <list>
## 1     6 <tibble [7 × 10]>  <lm>   <tibble [1 × 12]>
## 2     4 <tibble [11 × 10]> <lm>   <tibble [1 × 12]>
## 3     8 <tibble [14 × 10]> <lm>   <tibble [1 × 12]>
```

Programación funcional

- `map()`

```
my_test <- function(x) {
  lm(mpg ~ wt, data=x)
  }

mtcars %>%
  nest(-cyl) %>%
  mutate(res = map(data,my_test)) %>%
  mutate(glance_lm = res %>% map(glance)) %>%
  unnest(glance_lm)
```

```
## Warning: All elements of `...` must be named.
## Did you want `data = c(mpg, disp, hp, drat, wt, qsec, vs, am, gear, carb)`?
```

```
## # A tibble: 3 x 15
##     cyl data  res   r.squared adj.r.squared sigma statistic p.value    df logLik
##   <dbl> <lis> <lis>     <dbl>         <dbl> <dbl>     <dbl>   <dbl> <dbl>  <dbl>
## 1     6 <tib… <lm>      0.465         0.357  1.17      4.34  0.0918     1  -9.83
## 2     4 <tib… <lm>      0.509         0.454  3.33      9.32  0.0137     1 -27.7
## 3     8 <tib… <lm>      0.423         0.375  2.02      8.80  0.0118     1 -28.7
## # … with 5 more variables: AIC <dbl>, BIC <dbl>, deviance <dbl>,
## #   df.residual <int>, nobs <int>
```

Programación funcional

- `map()`, otra versión

```
mtcars %>%
  split(.$cyl) # de R base
```

```
## $`4`
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
##
## $`6`
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C      17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
##
## $`8`
##                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
```

Programación funcional

- `map()`, otra versión

```
mtcars %>%
  split(.$cyl) %>%
  map(~ lm(mpg ~ wt, data = .))
```

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

Programación funcional

- `map()`, otra versión

```
mtcars %>%
  split(.$cyl) %>%
  map(~ lm(mpg ~ wt, data = .)) %>%
  map(summary)
```

```
## $`4`
##
## Call:
## lm(formula = mpg ~ wt, data = .)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.1513 -1.9795 -0.6272  1.9299  5.2523
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   39.571      4.347   9.104 7.77e-06 ***
## wt            -5.647      1.850  -3.052   0.0137 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.332 on 9 degrees of freedom
## Multiple R-squared:  0.5086, Adjusted R-squared:  0.454
## F-statistic: 9.316 on 1 and 9 DF,  p-value: 0.01374
##
##
## $`6`
##
## Call:
## lm(formula = mpg ~ wt, data = .)
```

Programación funcional

- `map()`, otra versión

```
mtcars %>%
  split(.$cyl) %>%
  map(~ lm(mpg ~ wt, data = .)) %>%
  map(summary) %>%
  map("r.squared")
```

```
## $`4`
## [1] 0.5086326
##
## $`6`
## [1] 0.4645102
##
## $`8`
## [1] 0.4229655
```

Programación funcional

- `map()`, otra versión

```
mtcars %>%
  split(.$cyl) %>%
  map(~ lm(mpg ~ wt, data = .)) %>%
  map(summary) %>%
  map_dbl("r.squared")
```

```
##         4         6         8
## 0.5086326 0.4645102 0.4229655
```

Programación funcional

- `map()`, otra versión

```
mtcars %>%
  split(.$cyl) %>%
  map(~ lm(mpg ~ wt, data = .)) %>%
  map(summary) %>%
  map_df("r.squared")
```

```
## # A tibble: 1 x 3
##     `4`   `6`   `8`
##   <dbl> <dbl> <dbl>
## 1 0.509 0.465 0.423
```

Programación funcional

Ejericicio: Programación funcional

- Usando `data(txhousing, package="ggplot2")` , escribe, mediante un modelo, la relación lineal entre `sales` y `listings` para cada categoria de `year` y extrae el valor de p
    - tips: `lm(sales ~ listings)`

Ejericicio: Programación funcional

- Solución

```
data(txhousing, package="ggplot2")
txhousing %>%
  split(.$year) %>%
  map(~lm(sales ~ listings,data=.)) %>%
  map(summary) %>%
  map_df(.,"coefficients") %>%
  slice(8) %>%
  pivot_longer(cols = '2000':'2015', names_to = "year", values_to = "p.value")
```

```
## # A tibble: 0 x 2
## # … with 2 variables: year <chr>, p.value <dbl[,4]>
```