Medición y diseño de investigación

Clase 1: Introducción y fundamentos de la programación en R

FCS-UdelaR - Martín Opertti y Fabricio Carneiro

June 14, 2023

Programación y ¿Qué es R?

Softwares estadísticos

Los softwares o paquetes estadísticos son programas informáticos diseñado para llevar a cabo análisis estadísticos. Algunos de los más utilizados son SPSS, Stata, SAS o R.

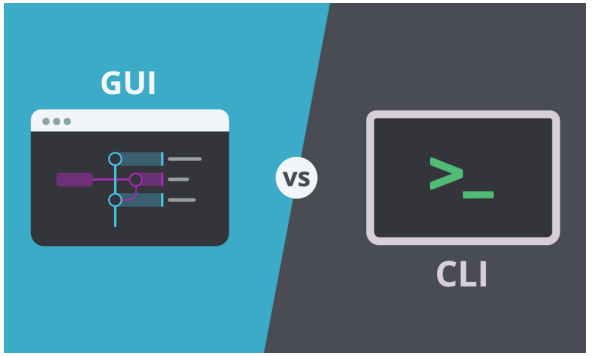
Fuente: R4DS

¿Por qué usar R?

- Es un software libre y gratuito
- Generar nuevas funciones es fácil, por lo que las está en constante desarrollo
- Tiene muchos usuarios de diversas disciplinas lo que genera una comunidad (particularmente mediante foros) que es de gran utilidad para la resolución de problemas de código
- Es uno de los programas más utilizados para técnicas innovadaras en estadística y visualización de datos
- Trabaja muy bien con otros programas/lenguajes (Excel, Latex, HTML, etc.)
- Es cada vez más usado tanto en el ámbito académico como profesional

Modalidades de uso de softwares estadísticos

- Point and click o GUI (Interfaz gráfica del usuario)
- Programación



¿Por qué programar?

- **Eficiencia:** Si bien programar llevará más tiempo al principio, rápidamente permite ahorrar mucho tiempo en comparación a tareas realizadas a través de una interfaz gráfica.
- **Más posibilidades:** En los softwares estadísticos tradicionales muchas operaciones (las más complejas o especificas) suelen no estar disponibles en la interfaz gráfica por lo que solo se pueden realizar mediante el uso de código.
- **Recolección de datos:** Saber programar abre la posibilidad para la recolección de datos que no es posible o es muy costosa manualmente.
- **Reproducibildiad:** El análisis de datos mediante programación mejora la transparencia y reproducibilidad en el proceso de generación de conocimiento
- **Colaboración:** La programación abre la puerta para una colaboración mucho más sencilla y eficiente.

R y R Studio

- R es un software y un lenguaje de programación gratuito enfocado en el análisis estadístico y la visualización de datos.
- R cuenta con gran potencia y flexibilidad, así como una númerosa -y creciente- comunidad de usuarios tanto académicos como profesionales.

- R Studio es un entorno de desarrollo integrado (IDE) . O en otras palabras... es una interfaz un poco (bastante) más amigable que usar R directamente.

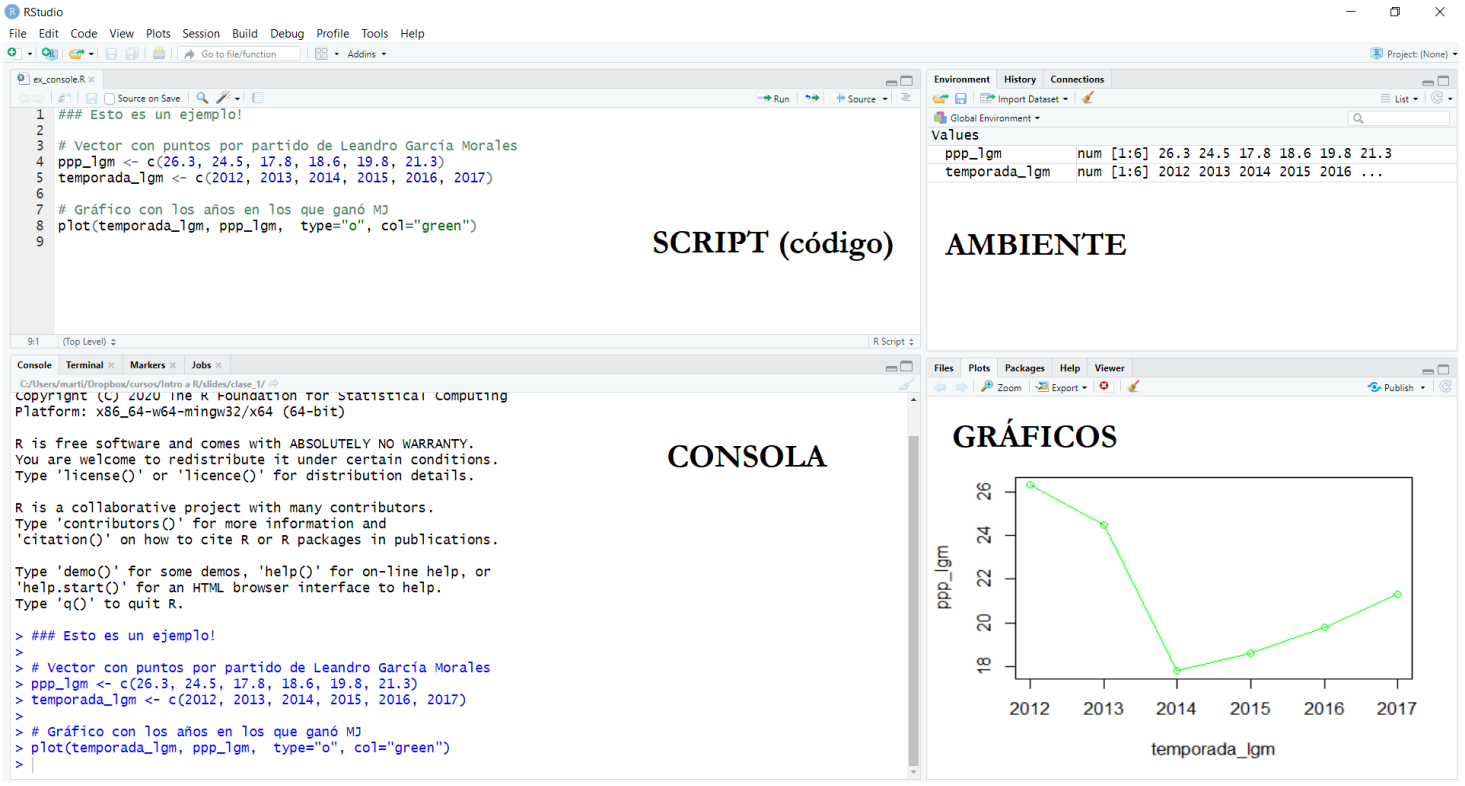
Recursos complementarios

La comunidad de usuarios de R es inmensa y muy abierta. Por esto hay muchísimos recursos para aprender de forma independiente y resolver problemas cuando nos estancamos:

- Libro "R for Data Science". Es muy completo y referencia en la mayoría de los cursos de R, pueden acceder a la versión online original y a una traducción
- R Bloggers y rpubs publican miles de tutoriales para temas específicos
- Existen foros -por ej. Stack Overflow- donde responden una infinidad preguntas de programación en R.
- IntRo es un excelente curso de R a cargo de Nicolás Schmidt de Facultad de Ciencias Sociales
- AnalizaR es un libro sobre análisis de datos en R con énfasis en Ciencia Política

Primeros pasos

Abrimos R Studio



R Studio

- **Source (editor):** es donde creamos y editamos los scripts, es decir, donde escribimos y almacenamos el código.
- **Console (consola):** imprime el código que corremos y la mayoría de los resultados. Podemos escribir código directamente aquí también, aunque si queremos guardarlo lo recomendable es hacerlo en el script.
- **Environment (ambiente):** Muestra todos los objetos que creaste en cada sesión.
- **Gráficos (y más):** Imprime los gráficos. En el mismo panel figuran otras pestañas como "Help" que sirve para buscar ayuda.

Scripts

- Es un archivo de texto con el código y anotaciones.
- Se crea arriba a la izquierda "file/New File/R Script" o `ctrl + shift + n`.
- Se guarda con `ctrl + s` y es un documento de texto como cualquier otro (word, txt). Esto nos permite reproducir paso a paso todo lo que hicimos durante nuestro análisis.
- Haciendo click luego en el script guardado se inicia R Studio.
- Para ejecutar una línea de código pueden usar el botón de "Run" arriba a la derecha o -más cómodo- `ctrl + enter`

Anotaciones

- Es importante ser prolijo y cuidadoso con el código que escribimos
- Los scripts nos dan la posibilidad de incluir comentarios, lo que es muy útil:

```
## Esta línea es una anotación.

## R ignora todo lo que está acá adentro (tiene que empezar con #)

## Podemos escribir nombres de funciones u objetos y R no las va a
# interpetar

## Usar anotaciones es clave para poder entender qué fue lo que
# hicimos anteriormente
```

- De esta forma, podemos comentar que fue lo que hicimos para acordarnos nosotros, y que los demás entiendan

Ayuda

- Obtener la ayuda correcta es fundamental al programar en R. Podemos obtener ayuda de todas las funciones que utilizamos con el comando `help()` (ej. `help(mean)`) o `?` (ej. `?mean`)
- Si no podemos solucionar un error con la documentación de las funciones/paquetes muchas veces sirve buscar en un navegador
- Muchas páginas contienen información relevante para solucionar problemas, entre las que se destaca stackoverflow
- En caso de no encontrar solución se puede consultar en páginas como stackoverflow mediante un ejemplo reproducible o reprex

Ayuda

```
help(mean)
```

mean {base} → Paquete

R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

→ Uso por defecto

Arguments

Argumentos

`x`
: An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.

`trim`
: the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.

`na.rm`
: a logical value indicating whether `NA` values should be stripped before the computation proceeds.

`...`
: further arguments passed to or from other methods.

Value

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), numeric (including integer) or complex, `NA_real_` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Lenguaje básico de R

R como calculadora

Para empezar, R sirve como calculadora. Se pueden realizar operaciones matemáticas, por ejemplo:

```
# Operaciones sencillas
2 + 2
```

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

```
20 - 10
```

```
## [1] 10
```

```
10 / 2
```

```
## [1] 5
```

```
10 * 10
```

```
## [1] 100
```

Objetos en R

En muchos programas estadísticos solemos solamente "imprimir" resultados (lo que llamamos expresiones). En R podemos utilizar este enfoque:

```
# Una operación sencilla:
17*17*7 # Se imprime el resultado
```

```
## [1] 2023
```

Sin embargo, en R también podemos almacenar los resultados en objetos. Creamos los objetos mediante asignaciones (`<-`). En este caso, guardemos el valor (a diferencia de imprimirlo).

```
anio <- 17*17*7 # Se crea un objeto
```

Si a esto lo ponemos entre paréntesis combinamos ambos enfoques: se guarda el objeto y se imprime el resultado

```
(anio <- 17*17*7) # Se crea un objeto y se imprime
```

```
## [1] 2023
```

Asignaciones

- El símbolo para crear un objeto es `<-` (alt + -) y se llama asignador
- Las asignaciones se crean de la siguiente manera: nombre_del_objeto `<-` valor.
- Como vimos, una vez que creo un objeto, R (por defecto) no imprime su valor. Este se puede obtener escribiendo simplemente el nombre del objeto o mediante la función `print()`:

```
anio <- 17*17*7 # Se crea un objeto

anio # Imprime el objeto anio
```

```
## [1] 2023
```

```
print(anio) # Imprime el objeto anio
```

```
## [1] 2023
```

Algunas funciones básicas

```
ls() # Lista los objetos en el ambiente

rm(year) # Borra objeto del ambiente

rm(list=ls()) # Borra todos los objetos del ambiente

help(ls) # Buscar ayuda sobre una función
```

Objetos

Clases y tipos de objetos

- En R utilizamos constantemente objetos. Cada objeto tiene una clase, tipo y atributos.
- Esto es importante porque las funciones que podemos aplicar a nuestros datos dependen del objeto en el que los definimos.
- El uso de objetos tiene muchos benificios como extraer parte de ellos para determinados usos, duplicarlos o realizar operaciones sin imprimir en la consola.

Tipos de objetos

El tipo de un objeto refiere a cuál es el tipo de los datos dentro del objeto. Los tipos más comunes son:

Nombre	Tipos	Ejemplo
integer	Númerico: valores enteros	10
double	Númerico: valores reales	10.5
character	Texto	"Diez"
logical	Lógico (TRUE or FALSE)	TRUE

Clases o estructura de datos

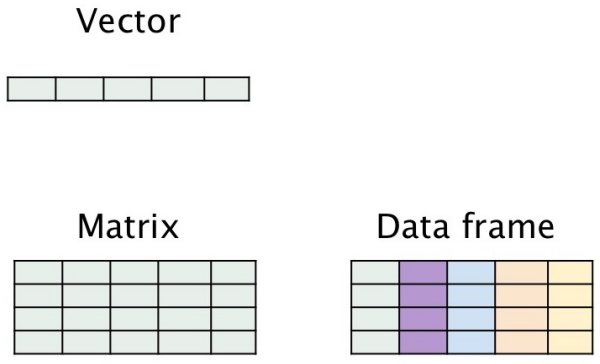
Las clases de objetos son formas de representar datos para usarlos de forma eficiente. Se dividen en cuántas dimensiones tienen y si poseen distintos tipos de datos o no. Las clases de datos más comunes en R son:

- `vector` (vectores): es la forma más simple, son unidimensionales y de un solo tipo
- `lists` (listas): son unidimensionales pero no están restringidas a un solo tipo de datos
- `matrix` (matrices): tienen dos dimensiones (filas y columnas) y un solo tipo de datos.
- `dataframes` (marcos de datos): son el tipo de estructura al que más acostumbrado estamos, con dos dimensiones (filas y columnas) y puede incluir distintos tipos de datos (uno por columna). Pueden considerarse como listas de vectores con el mismo tamaño.

En ocasiones podemos transformar objetos de una clase a otra.

Clases y tipos de objetos

Variables	Example
integer	100
numeric	0.05
character	“hello”
logical	TRUE
factor	“Green”



R variables and data types: Introduction to R Programming, Sydney-Informatics

Funciones para explorar objetos

R tiene funciones que nos permiten identificar la clase, el tipo, la estructura y los atributos de un objeto.

- `class()` - ¿Qué tipo de objeto es?
- `typeof()` - ¿Qué tipos de data tiene el objeto?
- `length()` - ¿Cuál es su tamaño?
- `attributes()` - ¿Tiene metadatos?

¿Por qué importan los tipos y clases?

Supongamos que creamos un objeto con el valor 10, al que luego le sumaremos otro objeto con el valor 20.

```
obj_1 <- "10"

typeof(obj_1)
```

```
## [1] "character"
```

```
obj_1 + 20 # Da error
```

```
## Error in obj_1 + 20: non-numeric argument to binary operator
```

¿Por qué importan los tipos y clases?

En cambio, si creamos el objeto de tipo número:

```
obj_1 <- 10

typeof(obj_1)
```

```
## [1] "double"
```

```
obj_1 + 20 # Funciona
```

```
## [1] 30
```

¿Por qué importan los tipos y clases?

Normalmente no trabajamos con objetos de un solo valor, y reescribirlos no es una opción. Para ellos tenemos coercionadores `as.logical()`, `as.integer()`, `as.double()`, o `as.character()`: funciones que transforman un objeto de un tipo a otro. En este caso:

```
obj_1 <- "10"

typeof(obj_1)
```

```
## [1] "character"
```

```
obj_1 <- as.numeric(obj_1)
typeof(obj_1)
```

```
## [1] "double"
```

```
is.numeric(obj_1) # Podemos verificarlo directamente también
```

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

Vectores

Vectores

Un vector es una colección de elementos. Hay 4 tipos de vectores: lógicos, character, integer y double (estos dos últimos son numéricos). Los elementos determinarán el tipo del objeto. Crear un vector es muy sencillo mediante la función `c()`:

```
mi_primer_vector <- c(1, 3, 5, 7, 143)
print(mi_primer_vector)
```

```
## [1]   1   3   5   7 143
```

Otras formas de crear vectores con `:` y `seq()`

```
v1 <- c(1:5) # Todos los números de 1 a 5
v1
```

```
## [1] 1 2 3 4 5
```

```
v2 <- seq(0, 50, 10) # De 0 a 50 de a 10 números
v2
```

```
## [1]  0 10 20 30 40 50
```

Indexación

Cuando queremos referirnos a uno o varios elementos dentro de un vector utilizamos `[]` (lo que se llama indexación).

```
## Indexación:
mi_primer_vector
```

```
## [1]   1   3   5   7 143
```

```
mi_primer_vector[1] # El primer elemento dentro del vector
```

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

```
# Nos sirve por ejemplo para extraer partes del vector:
v3 <- mi_primer_vector[1:3] # Creo nuevo vector con los elementos del 1 al 3
v3
```

```
## [1] 1 3 5
```

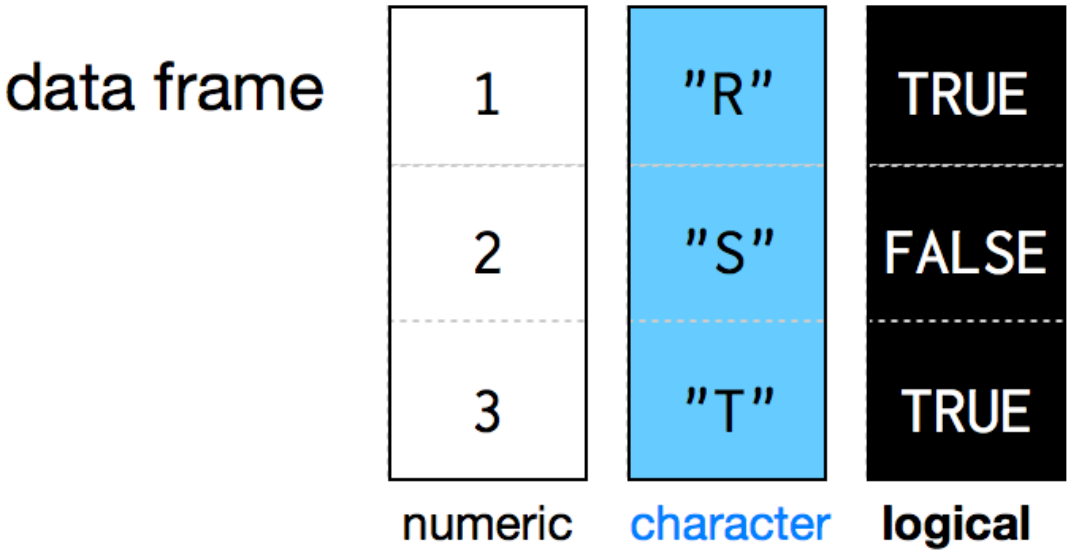
Dataframes

Dataframes

- Un dataframe o marco de datos (es lo que nos solemos referir como "base de datos").
- Es por lejos la estructura más usada y útil para almacenar y analizar datos
- Formalmente, son la conjunción de dos o más vectores (independientemente de su tipo) en una tabla con dimensiones (Grolemund, 2014)
- Cada vector se transforma en una columna.
- Es una forma de estructurar datos con filas y columnas. Las filas suelen ser las observaciones y las columnas las variables.
- Cada columna **debe** tener la misma longitud (número de observaciones)

Dataframes

Posible estructura de un dataframe o marco de datos:



(Grolemund, 2014)

Dataframes

Normalmente los dataframes con los que trabajamos los importamos desde otro formato (lo veremos más adelante), pero también podemos crearlos fácilmente en R.

Supongamos que queremos estructurar los resultados de una encuesta muy corta:

```
# Usamos la función data.frame
encuesta <- data.frame(edad = c(18,24,80,40,76),
                       ideologia = c("Izquierda", "Izquierda", "Derecha",
                                     "Centro", "Derecha"),
                       voto = c("Partido A", "Partido A", "Partido C",
                                "Partido B", "Partido B"))
class(encuesta)
```

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

```
encuesta
```

```
##   edad ideologia      voto
## 1   18 Izquierda Partido A
## 2   24 Izquierda Partido A
## 3   80   Derecha Partido C
## 4   40    Centro Partido B
## 5   76   Derecha Partido B
```

Dataframes: indexación

De forma similar a los vectores, la indexación `[]` nos permite acceder a datos dentro de nuestro dataframe. Como los dataframes tienen dos dimensiones (filas y columnas), tenemos que especificar cuáles valores queremos obtener. Para ello la indexación se divide en dos por una coma: antes de la coma nos referimos a las filas, y luego de la coma a las columnas:

```
# Valor de fila 1 y columna 1
encuesta[1, 1]
```

```
## [1] 18
```

```
# Valor de toda la columna 1 (no fijamos filas entonces nos devuelve todas)
encuesta[, 1]
```

```
## [1] 18 24 80 40 76
```

```
 # Valor de toda la fila 1 (no fijamos columnas entonces nos devuelve todas)
encuesta[1, ]
```

```
##   edad ideologia      voto
## 1   18 Izquierda Partido A
```

Dataframes

Una segunda manera para referirnos a datos dentro un dataframe es el usando el símbolo `$` . Es la manera más utilizada para refernirnos a una columna de un dataframe, y es muy sencillo de utilizar.

```
# Primero escribimos el nombre del dataframe, seguido por el símbolo $ y
# el nombre de la variable (sin comillas)
encuesta$edad # Esto imprime todos los valores de esa variable
```

```
## [1] 18 24 80 40 76
```

```
# En un dataframe cada variable es un vector y podemos fijarnos su clase
class(encuesta$edad)
```

```
## [1] "numeric"
```

```
mean(encuesta$edad) # Podemos aplicarle funciones (la media en este caso)
```

```
## [1] 47.6
```

Ejercicio

(1) Crear un vector con la edad de los integrantes de tu hogar

(2) Realizar una operación sobre ese vector para calcular la edad de cada integrante en 2030

(3) Crear un tercer vector con el nombre de cada uno y combinar los 3 vectores en un dataframe

Ejercicio

```r
# (1) Vector con la edad de los integrantes de tu hogar
edades <- c(26, 27)
edades
```

```
## [1] 26 27
```

```r
# (2) Calcular la edad para 2030
edades_2030 <- edades + 7
edades_2030
```

```
## [1] 33 34
```

```r
# (3) Tercer vector con el nombre de cada uno y crear dataframe
nombres <- c("Martín", "Leticia")
nombres
```

```
## [1] "Martín"  "Leticia"
```

```r
data.frame(edad_actual = edades,
           edades_2030 = edades_2030,
           nombres = nombres)
```

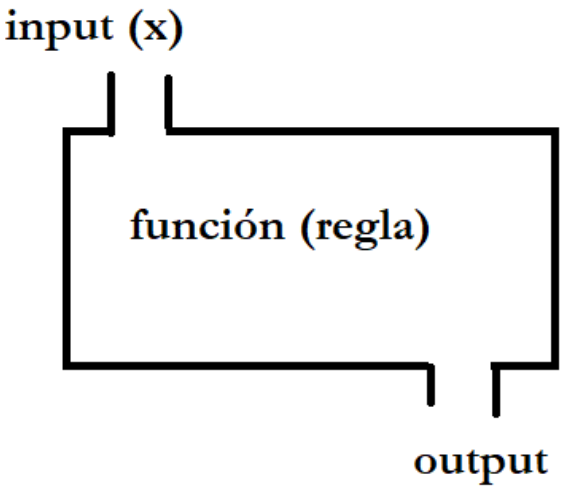
```
##   edad_actual edades_2030 nombres
## 1          26          33  Martín
## 2          27          34 Leticia
```

Funciones

Funciones

- Una función es una serie de instrucciones para realizar una tarea específica. La función suele necesitar un imput (generalmente datos) y suele devolver un output (generalmente datos luego de cierta regla)
- Los objetos **son** cosas, las funciones **hacen** cosas
- Por ejemplo, en el caso anterior, usamos la función `mean()` para calcular la media de la variable "edad" del dataframe "encuesta"
- Usar una función es sencillo: escribimos el nombre de la función, seguido de un paréntesis y dentro los datos a los que le queremos aplicar la función. Pueden ser objetos o directamente valores. Ej. `mean(c(10,20,30))` o `mean(objeto)`
- Dentro de la función se especifican los argumentos, que pueden ser divididos en dos tipos. El primero son los datos a los que se le aplica la función y el resto detalles de cómo se computa la función.

Funciones



Funciones (ejemplo)

```
# Supongamos que queremos calcular la media de: 12,24,36,48,60
(12 + 24 + 36 + 48 + 60)/5 # Calculo directamente la media
```

```
## [1] 36
```

```
data_ej <- c(12, 24, 36, 48, 60) # Genero el vector con los 5 números
sum(data_ej) / length(data_ej) # Calculo con dos funciones su media
```

```
## [1] 36
```

```
mean(data_ej) # Calculo la media directamente con la función mean()
```

```
## [1] 36
```

```
# También se puede ingresar data directamente en el argumento x
mean(c(12, 24, 36, 48, 60))
```

```
## [1] 36
```

Funciones: R Base

- R viene con un cojunto de funciones
- Las funciones que vienen "por defecto" son las que escribieron los creadores, al igual que en otros softwares no libres.
- La ventaja de R es que cualquiera puede crear nuevas funciones y publicarlas. Colecciones de funciones (generalmente relacionadas) se llaman "paquetes".

Funciones: argumentos

- Las funciones generalmente cuentan con argumentos que van dentro de los paréntesis.
- La mayoría de las funciones cuentan con el argumento "x" que suele ser el objeto al que le pasaremos la función. Al ser la mayoría de las veces el primer argumento, muchas veces no se explicita:

```
media_fun <- mean(data_ej) # Sin explicitar argumento x

media_fun_x <- mean(x = data_ej) # Explicitando argumento x

identical(media_fun, media_fun_x) # El mismo resultado
```

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

Funciones: argumentos

- Muchas funciones necesitan más de un argumento para funcionar de forma correcta.
- Por ejemplo, pensemos en la función `identical()`: "The safe and reliable way to test two objects for being exactly equal. It returns `TRUE` in this case, `FALSE` in every other case.".
- Por definición, `identical()` necesita dos conjuntos de datos distintos, para testear si son iguales.
- En la documentación `help(identical)` podemos ver que cuenta no solo con el argumento x, sino que también con y.

```
# Dos maneras de aplicar la función
identical(media_fun, media_fun_x) # por posición
```

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

```
identical(x = media_fun, y = media_fun_x) # por especificación
```

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

Funciones: argumentos

- A su vez, las funciones muchas veces cuentan con otros argumentos aparte de los datos que usan de insumo. Son detalles de cómo queremos aplicar la regla o el output que recibimos.
- Volvamos a la función `mean()`. Voy a crear un dataframe con la posición que obtuvo Uruguay en los últimos 5 mundiales de futbol masculino

```
# Dataframe con el resultado de Uruguuay en los últimos 5 mundiales
uru_mundial <- data.frame(year = c(2002, 2006, 2010, 2014, 2018),
                          posicion = c(26, NA, 4, 12, 5))
# Veamos la posición promedio:
mean(uru_mundial$posicion)
```

```
## [1] NA
```

```
# Como tenemos un dato perdido, la función nos devuelve NA

# Si especificamos el argumento na.rm (no tener en cuenta los datos perdidos):
mean(uru_mundial$posicion, na.rm = TRUE)
```

```
## [1] 11.75
```

Funciones: argumentos por defecto

- Es importante entender que la función `mean()` **por defecto** tiene el argumento `na.rm = FALSE`. De esta forma, si nosotros solamente le pasamos el argumento x, no quitará los datos perdidos.
- Leer la documentación de las funciones es fundamental, y sobretodo prestar atención a los argumentos por defecto.

Funciones: argumentos

```
help(mean)
```

mean {base} → Paquete

R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

→ Uso por defecto

Arguments

Argumentos

`x`
: An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.

`trim`
: the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.

`na.rm`
: a logical value indicating whether `NA` values should be stripped before the computation proceeds.

`...`
: further arguments passed to or from other methods.

Value

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), numeric (including integer) or complex, `NA_real_` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dialectos y paquetes

Ejercicio

Supongamos que tengo estos datos:

```
data
```

```
## # A tibble: 35 × 5
##     anio          pbi inflacion desempleo presidente
##    <dbl>        <dbl>     <dbl>     <dbl> <chr>
##  1  1985 249277574100      72.2     NA    Sanguinetti
##  2  1986 271238450500      76.4     NA    Sanguinetti
##  3  1987 292918912000      63.6     NA    Sanguinetti
##  4  1988 297256857900      62.2     NA    Sanguinetti
##  5  1989 300538279400      80.4     NA    Sanguinetti
##  6  1990 301431925100     113.      NA    Lacalle
##  7  1991 312099023700     102.       9.01 Lacalle
##  8  1992 336853433700      68.5      8.98 Lacalle
##  9  1993 345805469000      54.1      8.94 Lacalle
## 10  1994 370984750100      44.7      9    Lacalle
## # … with 25 more rows
```

Ejercicio

¿Qué quiero hacer con el código debajo?

```
as.data.frame(t(sapply(X = split(
  x = data[which(data$presidente %in% c("Vázquez", "Sanguinetti")),
           which(colnames(data) %in% c("pbi", "inflacion"))],
  f = data$presidente[which(data$presidente %in% c("Vázquez", "Sanguinetti"))],
  drop = TRUE),
  FUN = function(x) {apply(x, 2, mean)})))
```

Ejercicio

¿Qué quiero hacer con el código debajo?

```
data_dt <- data

setDT(data_dt)

data_dt[presidente %in% c("Vázquez", "Sanguinetti"),
       c("presidente", "pbi", "inflacion")  ][
         , lapply(.SD, mean), by = presidente]
```

Ejercicio

¿Qué quiero hacer con el código debajo?

```
data %>%
  filter(presidente %in% c("Vázquez", "Sanguinetti")) %>%
  select(presidente, pbi, inflacion) %>%
  group_by(presidente) %>%
  summarise_all(mean)
```

R Base

```
as.data.frame(t(sapply(X = split(
  x = data[which(data$presidente %in% c("Vázquez", "Sanguinetti")),
           which(colnames(data) %in% c("pbi", "inflacion"))],
  f = data$presidente[which(data$presidente %in% c("Vázquez", "Sanguinetti"))],
  drop = TRUE),
  FUN = function(x) {apply(x, 2, mean)})))
```

```
##                        pbi inflacion
## Sanguinetti 344923753631 46.168819
## Vázquez     584455715198  7.416316
```

Data.table

```
data_dt <- data
setDT(data_dt)
data_dt[presidente %in% c("Vázquez", "Sanguinetti"),
       c("presidente", "pbi", "inflacion")  ][
         , lapply(.SD, mean), by = presidente]
```

```
##     presidente          pbi inflacion
## 1: Sanguinetti 344923753631 46.168819
## 2:     Vázquez 584455715198  7.416316
```

Tidyverse

```
data %>%
  filter(presidente %in% c("Vázquez", "Sanguinetti")) %>%
  select(presidente, pbi, inflacion) %>%
  group_by(presidente) %>%
  summarise_all(mean)
```

```
## # A tibble: 2 × 3
##   presidente            pbi inflacion
##   <chr>               <dbl>     <dbl>
## 1 Sanguinetti 344923753631.     46.2
## 2 Vázquez     584455715198.      7.42
```

Dialectos

- Como vimos, en R podemos realizar una misma operación de muchas maneras distintas. Puesto de otra manera, R como lenguaje de programación tiene distintos "dialectos", esto es, paquetes (o conjuntos de paquetes) con sus propias funciones, sintaxis y comunidad de usuarios.
- Para la mayoría de las funciones requeridas para un análisis de datos estándar (importar datos, manipular, modelar y visualizar) existen -de forma muy simplificada- tres grandes dialectos: R Base, tidyverse y data.table.
- Tidyverse es una colección de paquetes diseñados para el análisis de datos. Este conjunto de paquetes comparte una filosofía de diseño, grámatica y estructura de datos.
- Las ventajas de Tidyverse están en su gramática (fácil de leer lo que invita a compartir y replicar), consistencia, alcance y su numerosa y creciente comunidad.

Tidyverse

Tidyverse

Tidyverse cuenta con varios paquetes que sirven para distintos tipos de tareas específicas. Podemos cargar todos los paquetes de forma conjunta:

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

install.packages("dplyr")
library(dplyr)
```

Tidyverse: paquetes

Paquetes que Tidyverse carga:

- readr: importar y exportar datos
- dplyr: manipulación de datos
- tidyr: manipulación de datos
- ggplot2: visualización de datos
- purr: programación avanzada
- tibble: estructura de datos
- forcats: factores
- stringr: variables de caracteres