

# Tema 0: Introducción a R y RStudio (Posit)

Pedro Albarrán

Dpto. de Fundamentos del Análisis Económico. Universidad de Alicante



# Contenidos I

1 Conceptos básicos

2 Objetos en R

3 Extendiendo R

4 R para análisis de datos

# Introducción

- Debéis tener instalados los programas gratuitos R y RStudio
- Nos familiarizaremos con los conceptos y comandos básicos de programación en R
- R es un lenguaje interpretado: ejecuta las instrucciones directamente en la consola
- RStudio es un entorno de desarrollo integrado (IDE) que combina varias herramientas para facilitar el uso de R: consola, **editor** para escribir comandos, ayuda, etc.

# R Studio

The screenshot shows the RStudio IDE interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. The toolbar contains icons for running, saving, and other actions. The main editor window shows a script named 'Example.R' with the following code:

```
1 ### Ejemplo de archivo de guión de R
2
3 x <- c(1, 2, 3)
4 y <- c(4, 5, 6)
5
6 examplenat <- cbind(x,y)
7
8 afunccion <- function(dataobj) {
9   K <- 48
10
11   for (k in 1:K) {
12     dataobj <- dataobj+k
13   }
14   return(dataobj)
15 }
```

Four red text boxes are overlaid on the image, highlighting key features:

- Escribir archivos de guión** (Writing script files) - points to the script editor.
- Información sobre datos, objetos, ...** (Information about data, objects, ...) - points to the Environment pane.
- Consola: ejecución directa de comandos, resultados de guiones** (Console: direct execution of commands, results of scripts) - points to the Console pane.
- Ver Ayuda, Ficheros, gráficos** (View Help, Files, plots) - points to the Files, Plots, and Help panes.

The Environment pane shows the following objects:

Object	Class	Value
examplenat	num	[1:3, 1:2] 1 2 3 4 5 6
x	num	[1:3] 1 2 3
y	num	[1:3] 4 5 6
afunccion	func	

The Console pane shows the following output:

```
> examplenat <- cbind(x,y)
>
> afunccion <- function(dataobj) {
+   K <- 48
+
+   for (k in 1:K) {
+     dataobj <- dataobj+k
+   }
+   return(dataobj)
+ }
>
> x
[1] 1 2 3
>
```

# Empezando con R



- Escribimos *comandos* en la **consola** y se ejecutan pulsando Enter:

- ▶ La tecla de tabulador  ofrece opciones de autocompletado
- ▶ Ejecutar algo que no es un **comando de R** devuelve un error

```
2 + 2
3 * (1 - 4)^2
sqrt(log(5/2))
pi
hola
```

- O en el **Editor de RStudio** y se envían a la consola la o las líneas seleccionadas para ser evaluadas con el icono  o con el atajo de teclado Ctrl+Enter
- NOTA: en MacOS, usad la tecla Command en lugar de Ctrl

# Archivos de guion (“scripts”)

- Es preferible incluir varios comandos en un archivo de texto para ejecutarlos
- Se puede replicar el proceso de cálculos paso a paso (no como Excel)
- Creamos un nuevo archivo con el icono  o en el menú *File > New File > R script* (atajo Ctrl + Mays + N)
- Guardamos el archivo con  o en *File > Save* (atajo Ctrl + S), eligiendo un directorio y nombre de extensión “.R” (por defecto) o “.r”
- En un archivo de guion (guardado), RStudio marca las líneas con error y muestra el mensaje de error al pasar el puntero



# Trabajar con ficheros de guion

- Cada comando es “una” línea y se ejecutan secuencialmente
- Un comando se puede extender visualmente más de una línea hasta completarse: p.e., hasta cerrar los paréntesis.
  - ▶ Escribimos `log(`, en otra línea y ejecutamos: la consola cambia de `>` a `+`
  - ▶ No hace “nada” esperando que completemos el comando.
- El carácter `#` marca el inicio de un **comentario**: lo que sigue se “ignora” (no se ejecuta) en R

```
# Pueden ir al principio de la línea  
2 + 2 # o después de una instrucción
```

- Comentar es un buen hábito: ayuda a entender/recordar qué hacemos
- Notad que RStudio tiene *resaltado de sintaxis*: distinto color para comentarios, números, funciones, etc.



# Directorio de Trabajo. Proyectos.

- Conviene **organizar los archivos** relacionados con un mismo tema en una estructura de (sub)directorios a partir de un **directorio de trabajo** principal
- RStudio permite definir **proyectos** para gestionarlo fácilmente a través del menú *File* o desplegando el icono en la parte superior derecha  Project: (None) ▾
- Desde el menú *File > New Project* o desde el icono, creamos un nuevo proyecto:
  - ▶ Podemos usar un *Nuevo Directorio* o elegir una ubicación ya existente
  - ▶ El **nombre del proyecto será el nombre del directorio**
  - ▶ También se crea un archivo con el mismo nombre y extensión “.Rproj”
- Al abrir RStudio, tenemos activo el último proyecto abierto: ej.,  Tema00 ▾
- Tanto desde el menú como desde el icono de gestión de proyectos, podemos
  - ▶ cerrar el proyecto actual, *File > Close Projects*,
  - ▶ abrir otros proyectos guardados: *File > Open Project* o *File > Recent Projects* 



# Proyectos (cont.)

- Para trabajar con un archivo, usamos la **ruta relativa** al directorio de trabajo:
  - ▶ si están en el raíz del directorio: `codigo.R`, `misdatos.Rdata`
  - ▶ si están en un subdirectorio, indicamos la ruta separando directorios por `/`:  
`datos/ventas.Rdata`, `datos/ano2020/ingresos.Rdata`
- La **pestaña**  en el cuadrante inferior-derecho ofrece una forma visual de abrir, crear, copiar, mover o eliminar archivos o directorios, etc.
- Evitad caracteres “raros” (acentos, espacios, etc.) en directorios y ficheros
- NOTA. El Explorador de Archivos de Windows y Finder de MacOS, no muestran defecto las extensiones de los archivos.
  - ▶ Puede ser confuso para distinguir entre dos archivos con el mismo nombre y diferente extensión: `proyecto.R` y `proyecto.Rproj`
  - ▶ Consultad cómo mostrarlas: p.e., para Windows y MacOS



# Funciones en R

- Las expresiones que aceptan **argumentos** se denominan funciones.

```
exp(2)
ceiling(5.2)
```

- Algunos argumentos son obligatorios, otros tienen valores por defecto que se pueden omitir
- Los argumentos se pueden especificar por nombre o por orden.

```
log(2, base=2)
log(2, 10)
log(base = 10, x = 2)
```

- ¿Cómo sabemos la manera de usar una función (ej. argumentos necesarios) o comando de R?

# Ayuda en R y RStudio

- RStudio tiene autocompletado y ayuda flotante para funciones y otros elementos de R
  - ▶ P.e., si empezamos a escribir la función `log`, se muestra la forma esperada de trabajar con esa función
- RStudio también tiene una **pestaña** para buscar ayuda
- Las búsquedas online o las IAs (como chatGPT, Gemini o Copilot) pueden ser útiles.
- PERO debemos tener un conocimiento mínimo para aprovechar realmente una solución
  - ▶ hay muchas formas de hacer lo mismo en R: una respuesta correcta puede no ajustarse a lo que ya sabemos
- NO uséis copiar-pegar **sin entender** el código: copiar-pensar-adaptar

# El operador de asignación

- El operador `<-` almacena un contenido en un *objeto* con un nombre,<sup>1</sup> que incluye letras, números y algunos caracteres especiales (“.”, “\_”)

```
x <- 2*3      # asignación, no muestra resultado
x             # ejecutamos mostrar la variable
print(x)
(x <- 2)      # asignación e impresión a la vez
```

- R es “case-sensitive”: `x` y `X` son dos objetos distintos
- Los objetos asignados pueden usarse posteriormente, p.e., para generar otros a partir de ellos

```
y <- x + 5    # asignamos y a partir del VALOR de x
(x <- x*3)    # re-asignamos x a partir de ella misma
y            # NO cambia (en Excel, sí)
```


---

<sup>1</sup>También se puede asignar con `=`

# El Espacio de Trabajo en R

- El espacio de trabajo es el conjunto de objetos activos en memoria, resultado de todos los comandos ejecutados previamente
- En RStudio, la pestaña **Environment** muestra los objetos y su valor
- Las funciones `ls()` y `rm()` muestran y eliminan respectivamente objetos del espacio de trabajo
- Borrarnos *todos* los objetos con  en el **Environment** o el comando

```
rm( list=ls() ) # eliminar todos los objetos
rm(y, x)       # eliminar solo algunos objetos
```

- Guardamos el contenido del entorno de trabajo con  (o al cerrar RStudio), pero es **innecesario**: ejecutando los comandos guardados en un archivo `.R` recuperamos los objetos

# Mensajes de “Error” y “Warning”

- En programación, cometer **errores** es **normal**
- En muchos errores, R se “quejará” mostrando **mensajes** en rojo
  - ▶ *Aviso*: R ofrece un resultado (y continuará al siguiente comando), PERO indica que puede haber algo “no deseado”
  - ▶ *Error*: para la ejecución, *sin resultado*, e “informa” de la razón
- Algunos mensajes son claros, pero otros requieren más investigación
- Peor que un mensaje de error: escribimos (copiamos) un código que funciona pero no hace lo que queremos...
- El ordenador NO se equivoca: hace lo que le pedimos según unas reglas bien definidas por R, que *debemos conocer*
  - ▶ Sed cuidadosos, pensad y **entended** cada paso del código

# Tipos de Objetos en R

- TODO en R es un objeto, cada uno con **distintas propiedades** y, por tanto, distintas formas de trabajar con él
- Además de las funciones, los principales objetos con los que trabajaremos son:
  - 1 vectores
  - 2 factores
  - 3 conjuntos de datos (“data frames”)
  - 4 listas
- Estos objetos pueden contener varios tipos de datos o variables:
  - ▶ entero
  - ▶ numérico (números reales)
  - ▶ lógico (valores verdadero/falso)
  - ▶ caracteres

# Vectores

- Un vector es una secuencia de datos elementales, creados con el operador “c()” (combinar)

```
x <- c(2.5, -4.1, 6.4, 8.2)      # vector numérico
y <- c(3, 0, -1, 2)             # vector de enteros
w <- c("hola", 'adios')         # vector de caracteres
z <- c(FALSE, TRUE, T, F)       # vector lógico
```

- Podemos crear vectores a partir de otros vectores o usando comandos

```
z <- c(x, y)
x <- rep(1, times=4)
y <- seq(from = 10, to = 1, by = -1)
z <- 1:10      # equivale a z <- seq(1,10,1)
```

- Un vector sólo puede contener objetos de un **único tipo** elemental, que podemos conocer en el Environment o con str()



## Vectores (cont.)

- Si se mezclan tipos distintos, R busca una clase que “acomode” a todos

```
vcr <- c("lunes", 2)
```

- Forzamos que un objeto sea tratado con una clase concreta, con `as.integer()`, `as.numeric()`, `as.character()` y `as.logical()`
  - ▶ Si no se puede convertir a número, devuelve NA (con un “warning”)
- NO se pueden realizar operaciones incompatibles entre clases
  - ▶ Cuidado con las comillas: NO es lo mismo un objeto (su contenido) que el carácter de su nombre

```
a <- 4  
c <- 'a' + 1
```

- Los vectores pueden tener *nombres* (una “etiqueta” única para cada elemento): un vector de caracteres de la misma longitud asignado con `names()`

# Aritmética de vectores

- La mayoría de los operadores se aplican *elemento-a-elemento*

```
a <- seq(1,3,1)
b <- seq(6,8,1)
```

```
a+b
a*b
```

- Con diferentes longitudes, se repite el vector corto cuanto sea necesario

```
b <- 6:9
a + b
a + 1 # lo que queremos!
```

- Algunas **funciones** relevantes

```
length(x) # longitud
sort(x)   # ordenar
max(x)    # máximo
min(x)    # mínimos
sum(x)    # suma
prod(x)   # producto
```

```
mean(x)   # media
var(x)    # varianza
table(x)  # frecuencias

summary(x) # estadísticos
```



# Vectores lógicos

- Obtenemos un objeto lógico enunciando una relación que puede ser cierta o falsa , como comparaciones básicas de igualdad o desigualdad

```
1 == 1 # TRUE
1 != 3 # TRUE
1 > 2  # FALSE
```

```
a <- 3
a >= 3 # TRUE
a + 1 <= 10 # FALSE
```

- Combinamos varios enunciados con operadores Y (&), O (|) y NO (!)
- Para conjuntos, `x %in% Y` es cierto cuando `x` es un elemento de `Y`

```
altura <- c(176, 165, 189, 155, 168)
```

```
altura >= c(175, 165, 195, 165, 168) # elemento a elemento
altura == 155                         # elemento a elemento
```

```
altura > 160 & altura <= 180
```

```
altura < 160 | altura >= 180
```

```
c(165,179) %in% altura
```

```
condicion <- !(altura <= 170)
```



# Acceso a los elementos de un vector

- Se utiliza el operador `[]` (paréntesis cuadrado)<sup>2</sup> y
- ❶ **Posiciones** de los elementos, usando un vector de enteros

```
altura[3]  
altura[c(1,3,5)]
```

- Con enteros negativos, indicamos posiciones que NO queremos

```
altura[-c(2,4)]
```

- ❷ **Condición** que satisfacen los elementos, usando un vector lógico

```
altura[altura > 180 | altura < 160]
```

- ❸ (Si lo tienen) **Nombres** de los elementos, usando un vector de caracteres

---

<sup>2</sup>También con `[[ ]]`

# Factores

- La **información cualitativa** se suele codificar como texto o números, pero NO tiene sentido numérico (ni de “texto”): *representan* clases o **categorías**
- Para destacar la naturaleza distinta de estos datos, existe un tipo de objeto específico en R: los **factores**
- Además de otras ventajas que veremos, permiten separar la representación original de las categorías (niveles) de cómo queremos mostrarlas (etiquetas)

```
genero    <- c(2, 1, 2, 2, 2)
genero_f  <- factor(genero, levels = c(1, 2),
                    labels = c("Mujer", "Hombre"))
```

- Se asocia nivel 1 con “Mujer”, 2 con “Hombre”, etc.
- Las operaciones con factores se realiza con las etiquetas, no con los niveles

```
genero_f == 1      # NO existe valor 1
genero_f == "Mujer"
```



## Factores (cont.)

- También podemos usar `as.factor()` para convertir un vector en un factor
- PERO es conveniente especificar los niveles y las categorías porque si no, R los asigna alfabéticamente

```
g <- factor(genero)  # as.factor(genero) hace lo mismo
g
```

- En este caso la etiqueta del primer nivel encontrado en los datos (el número 2) es “1” y la del siguiente nivel (el número 1) es “2”
- También podemos tener factores con orden con la opción `order = TRUE` y enumerando los niveles en orden

```
satisf  <- c("A", "B", "A", "B", "M")
satisf_f <- factor(satisf, order = TRUE,
                   levels = c("B", "M", "A"),
                   labels = c("Bajo", "Medio", "Alto"))
```



# “Resumiendo” un vector numérico o un factor

- La función `summary()` devuelve los principales estadísticos descriptivos de un vector numérico

```
summary(altura)
```

- Para información cualitativa, la media y otros estadísticos no tienen sentido

```
summary(genero)
genero <- c(1, 20, 20, 1, 1) # dos categorías igualmente
summary(genero)
```

- La función `summary()` ofrece resultados diferentes según el tipo de objeto (porque tiene distintas propiedades)

```
summary(genero_f)
```

# “Data Frames”

- Es un tipo de objetos específico para facilitar el análisis de datos: una colección de variables por columnas y observaciones por filas.
- Cada columna es un vector con un **nombre** y tipos de datos (quizás) diferentes

```
altura <- c(177, 178, 168, 164, 186, 162, 160)
peso   <- c(75, 85, 70, 60, 80, 65, 54)
genero <- c(2, 1, 2, 2, 2, 1, 1)
genero_f <- factor(genero, levels = c(1, 2),
                  labels = c("Mujer", "Hombre"))
datos <- data.frame("Altura"=altura, "Peso"=peso, "Genero"= genero_f)
```

- Se visualizan con View(datos) o en “Enviroment” o una parte con head()
- Seleccionamos columnas por nombre **con \$** o por nombre o posición **con [[ ]]**

```
vectAltura <- datos$Altura    # objeto resultante = vector
datos[[2]] == datos[["Peso"]]
```





## “Data Frames” (cont.)

- También podemos usar `[]` para seleccionar filas y columnas por posición, nombre y/o condición lógica

```
datos1 <- datos[datos$Genero == "Hombre", 1:2] # Altura y Peso de hombres
```

- Suele ser mejor usar `subset()` que devuelve siempre un “data frame”

```
D1 <- subset(datos, Altura > 165)
D2 <- subset(datos, subset = Altura %in% c(177,178),
              select = c(Altura, Peso))
```

- Generamos nuevas variables con el vector de asignación

```
datos$Altura_m <- datos$Altura / 100
```

- Se pueden añadir filas y columnas a un *data frame* con `rbind()` y `cbind()`, respectivamente, a partir de vectores u otros *data frames*

# Listas

- Una lista es una colección de objetos de distinto tipo (a diferencia de un vector)
- Los elementos de una lista suelen tener nombres

```
miLista <- list(saludo="hola", vec=x, lista=list(1:4, "X"),
               datos=datos)
```

- Con `[[ ]]` (por posición o por nombre) o con `$` (solo por nombre) extraemos los elementos en su clase original

```
miLista$vec
miLista[[2]] + 3
```

- También podemos usar `[]`, pero devuelve una lista
- `unlist()` convierte una lista en vector, usando la clase que pueda ajustarse a todos los objetos (elementales)

# Bibliotecas (“libraries”)

- Una biblioteca contiene nuevos objetos de R: funciones, datos, etc.
- Para instalar una nueva biblioteca (se hace **una vez**), en *Tools* > *Install packages* o en **Packages** o con el comando

```
install.packages("AER")
```

- Mantenemos actualizados los paquetes, en el menú *Tools* o en **Packages**
- La biblioteca solo está disponible si se carga en la *sesión actual*

```
library(AER)
```

- Nota: en adelante, la bibliotecas que carguemos se suponen instaladas
- En **Packages** vemos las bibliotecas instaladas y las cargadas aparecen marcadas

System Library

☒ base☐ boot

## Bibliotecas (cont.)

- El nombre completo de un objeto es `biblioteca::nombre`
  - ▶ La nombre de la biblioteca solo es necesaria si no se ha cargado o dos objetos diferentes tienen el mismo nombre en bibliotecas distintas

```
base::log(1)
log(1)
```

```
library(Hmisc)
find("units")
```

- Para mostrar todos los datos de las bibliotecas cargadas

```
data()
```

- Los podemos cargar en el “Environment” y obtener información detallada en la ayuda (ej., nombre de variables)

```
data("Affairs")
help("Affairs")
```

# Datos “nativos” en R

- Guardamos objetos del espacio de trabajo con `save()` (en una ruta relativa al directorio de trabajo)


```
x <- 1:20
y <- 2 * x ^ 2 + 1
save(x, file="x.RData")          # un objeto, o varios
save(x, y, file="data/xy.RData") # separados por comas
```

- Para cargar datos al espacio de trabajo, con `load()` (= icono )

```
load("data/xy.RData")
```

- En la pestaña de **Files**: doble-clic carga un archivo de datos
- Nota: este tipo de archivo puede contener varios objetos, incluidas varios conjuntos de datos

# Datos en otros formatos externos a R

- Varias bibliotecas permiten trabajar con distintos formatos de datos, p.e.:
  - 1 Texto, con delimitadores o de ancho fijo: `utils` (R base), `readr`
  - 2 Hojas de cálculo: `readxl`, `openxlsx`
  - 3 Formatos de software estadístico: `haven`, `foreign`
- Descargad estos ejemplos (UA cloud): `renta.txt`, `sex_data.csv`, `beauty.xls`, `nsw.dta`
- En  Import de RStudio, tenemos acceso visual para cargar algunos formatos (con la biblioteca necesaria instalada)
- `rio` es un meta-paquete (instala otras bibliotecas) para importar y exportar varios formatos de datos de forma sencilla
  - ▶ A partir de la extensión del archivo, detecta el formato y, por tanto, la biblioteca necesaria

# Importar y exportar con rio

- rio permite trabajar con el **mismo comando** para todos los formatos, pero las opciones por defecto pueden no ser adecuadas
  - ▶ En la Ayuda se incluye una presentación completa del paquete
- El comando `import()` se usa para leer los datos

```
library(rio)
sex    <- import("data/sex_data.csv")
renta  <- import("data/renta.txt")
beauty <- import("data/beauty.xls")
nsw    <- import("data/nsw.dta")      # formato Stata
```

- Podemos exportar datos a un tipo de formato con `export()`

```
export(nsw, "data/nsw.csv")
```

- O convertir un archivo del disco a otro formato con `convert()`

## Otras fuentes de datos

- Bibliotecas con datos muy utilizados: `pwt` (“Penn World Tables”)
- Bibliotecas con funciones para obtener datos online (con APIs públicas)
  - ▶ datos de las OECD y eurostat (incluye datos del INE español)
  - ▶ `rdbnomics` para los datos gratuitos de <https://db.nomics.world/>
  - ▶ datos económicos y financieros con `quantmod` y `tidyquant`
  - ▶ `quandl` (de pago)
- `qualtRics` trabaja con software de encuestas `qualtrics`
- Descarga de páginas web y *webscraping* con las bibliotecas `rvest` y `httr` (función `GET()`)
- `googlesheets4`
- DBI accede con Bases de Datos relaciones (SQL)





# Gráficos Básicos

- Podemos representar gráficos de dos variables o funciones

```
x <- c(3, 4, 5, 6, 7, 8)
y <- c(5, 3, 7, 7, 5, 10)
```

```
plot(x,y)
```

- El resultado aparece en la pestaña *Plots* de RStudio
- Podemos cambiar opciones (ver Ayuda de `plot.default`) como *type* (puntos, líneas, etc.), símbolo de punto (*pch*), tipo de línea (*lty*), ancho de línea (*lwd*), color (*col*), título, etiquetas de los ejes, etc.

```
plot(x, y, type="b", pch=3)
plot(x, y, type="l", lty=2, lwd=2)
plot(x, y, xlab="Eje X", ylab="Eje Y", main="Mi título")
```

- Se pueden cambiar más opciones con `par()`, combinar gráficos, añadir líneas, texto, etc. y exportar los gráficos

# Estadísticos descriptivos: variables discretas

- Para variables discretas (factores), `table()` calcula distribuciones de frecuencias de una variable o conjuntas: el resultado es un **objeto**

```
data("PSID1982", package = "AER")  
(frec <- table(PSID1982$occupation) )  
(frec2 <- table(PSID1982$occupation, PSID1982$ethnicity))
```

- Podemos mostrar frecuencias relativas con `prop.table()`

```
prop.table(frec)  
prop.table(frec2)
```

```
prop.table(frec2, margin = 1)  
prop.table(frec2, margin = 2)
```

- También es informativa su representación con gráficos de barras

```
barplot(frec, horiz = T)
```

```
barplot(prop.table(frec2), beside = T)
```

# Estadísticos descriptivos: variables continuas

```
data(ceosal1, package='wooldridge')
```

- Ya hemos visto funciones de estadísticos como `mean()`, `var()`, etc.

```
median(ceosal1$salary)
```

```
var(ceosal1$salary)
```

```
quantile(ceosal1$salary,  
         probs=c(0.25, 0.75) ) # 1er y 3er cuartil
```

```
summary(ceosal1$salary) # de una variable (vector)
```

```
summary(ceosal1)        # de todo el conjunto de datos
```

```
cov(ceosal1$salary, ceosal1$roe) # covarianza
```

```
cor(ceosal1$salary, ceosal1$roe) # correlación
```

# Estadísticos descriptivos: variables continuas (cont.)

- Para variables continuas, las frecuencias de valores en un *intervalo* se pueden tabular o graficar en un histograma

```
hist(ceosal1$roe)    # intervalos automáticos
```

```
hist(ceosal1$roe, freq=F,          # densidad, no casos  
     breaks=c(0,5,10,20,30,60)) # intervalos explícitos
```

- O la densidad (versión suavizada del histograma)

```
plot(density(ceosal1$roe))
```

- Un gráfico de caja ofrece información resumida de la distribución: mediana, 1er y 3er cuartil, y valores “extremos”

```
boxplot(ceosal1$roe, horizontal=T)
```

```
boxplot(ceosal1$roe~ceosal1$consprod)
```

# Valores ausentes (“missing values”): NA

- Muchos conjuntos de datos tienen valores ausentes de ciertas observaciones para algunas variables: ej., descargad (UA Cloud) `earn.RData`
- Sabemos si una observación es NA y la frecuencia total:

```
load("data/earn.RData")  
x <- earn$earnings
```

```
is.na(x)  
table(is.na(x))
```

- Por defecto en R, un cálculo con NAs es NA: debemos decir que los elimine explícitamente (y ser conscientes de lo que implica)

```
mean(x)
```

```
mean(x, na.rm=TRUE)
```

- `na.omit()` elimina observaciones con NAs de una o varias variables

```
earn2 <- na.omit(earn)
```

- ¿Cómo tratar los NAs? Eliminarlos implica selección muestral y la alternativa de imputar valores implica supuestos sobre éstos

# Nota sobre programación “avanzada”

- Como en todo lenguaje de programación R, tiene funciones para
  - ▶ Ejecución condicional `if()`: una parte del código se ejecuta solo si se cumple una condición
  - ▶ Bucles `for()`: se repite un mismo bloque de código mientras se itera por los valores de vector
  - ▶ Crear funciones propias con `function()`
- Una variante de la ejecución condicional, solo para crear variables según una condición

```
data("Affairs", package = "AER")  
Affairs$univers <- ifelse(Affairs$education>15, 1, 0)
```