

Análisis de Datos

Tema 3 - Data Wrangling

3.3 Organización de Datos

Roi Naveiro

Data Wrangling

Objetivo: dejar los datos listos para su posterior exploración y modelización

Convertir datos crudos en datos procesados

Datos crudos

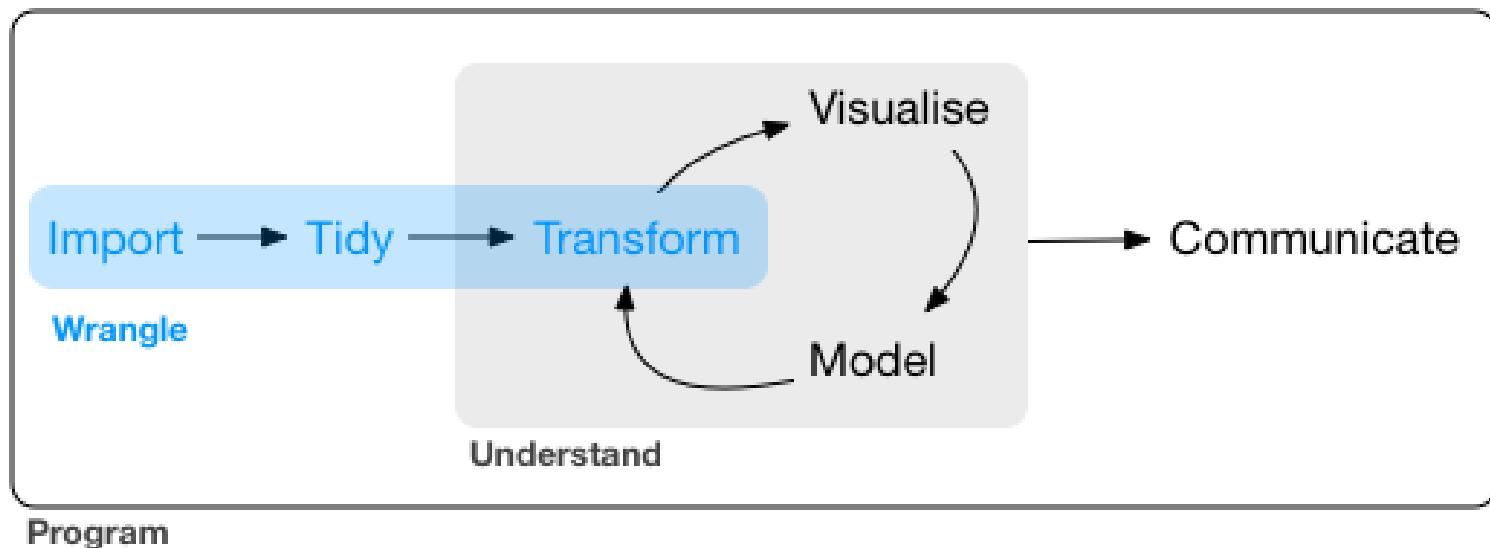
- Los datos tal cual aparecen en la fuente de origen
- No han sufrido ninguna manipulación

Datos procesados

- Cada variable es una columna
- Cada observación una fila
- Cada unidad observacional es una celda
- Datos más complejos, en varias tablas interconectadas

Data Wrangling

- Importación de los datos
- Organización de los datos
- Transformación de los datos



Transformación de datos

Transformación de datos

- Strings
- Factores
- Fechas y horas

Strings

Strings

- Veremos lo básico de manipulación de strings
- Datos en formato texto

```
"Esto es un string"
```

```
## [1] "Esto es un string"
```

```
c("una", "dos y tres")
```

```
## [1] "una"           "dos y tres"
```

- Expresiones regulares (regular expressions): permiten identificar patrones en texto
- Usaremos **stringr** de tidyverse para manipular strings
- Todas sus funciones empiezan con **str_**

Strings: longitud

Podemos calcular la longitud de una string usando **str_length()**

```
str_length("Hola")
```

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

Las operaciones están vectorizadas

```
str_length( c("Hola", "Adiós") )
```

```
## [1] 4 5
```

Pregunta: ¿Qué pasa si hay espacios?

Strings: combinación

Combinamos strings con **str_c**

```
str_c("Ma", "drid")  
  
## [1] "Madrid"  
  
str_c("Un", "dos", sep=",")  
  
## [1] "Un,dos"
```

Pregunta: ¿Cómo combinarías dos strings con una coma seguida de un espacio?

Strings: combinación

También vectorizado!

```
str_c("prefijo-", c("X", "Y", "Z"), "-sufijo")
```

```
## [1] "prefijo-X-sufijo" "prefijo-Y-sufijo" "prefijo-Z-sufijo"
```

Para colapsar un texto largo en una única string

```
str_c(c("En", "un", "lugar", "de", "la", "Mancha"), collapse = " ")
```

```
## [1] "En un lugar de la Mancha"
```

Strings: subsetting

Para extraer partes de una string usamos **str_sub**

```
x <- c("Liberty", "La Nuit")  
str_sub(x, 1, 3)
```

```
## [1] "Lib" "La "
```

```
# Con números negativos empieza desde el final  
str_sub(x, -3, -1)
```

```
## [1] "rty" "uit"
```

Strings: cosas útiles

```
x <- c("Liberty", "La nuit")  
str_to_lower(x)
```

```
## [1] "liberty" "la nuit"
```

```
str_to_upper(x)
```

```
## [1] "LIBERTY" "LA NUIT"
```

```
str_to_title(x)
```

```
## [1] "Liberty" "La Nuit"
```

Strings: expresiones regulares

Permiten describir patrones en strings. Se pueden hacer varias cosas con los patrones

- Determinar que strings cumplen el patrón.
- Determinar en qué posición aparece el patrón
- Extraer el patrón
- Reemplazarlo

E.g. patrón "contiene ES seguido de dos números"

```
x1 <- "ES49"  
x2 <- "Mi número de cuenta es ES35 30..."  
x3 <- "Resido en España (ES)"
```

Strings: expresiones regulares

Vamos aprender a jugar con patrones muy sencillos. Primero, coincidencia exacta (ojo mayúsculas y minúsculas!)

```
x <- c("Cats", "Uñas Chung Lee", "Capital")
str_view(x, "Le")
```

Cats

Uñas Chung Lee

Capital

Strings: expresiones regulares

Para identificar cualquier carácter usamos .

```
x <- c("Cats", "Uñas Chung Lee", "Capital")
str_view(x, ".a.")
```

Cats

Uñas Chung Lee

Capital

Strings: expresiones regulares

Empieza por ^ y termina por \$

```
x <- c("Cats", "Uñas Chung Lee", "Capital")
str_view(x, "^C")
```

Cats

Uñas Chung Lee

Capital

Strings: expresiones regulares

\d: cualquier dígito \s: espacio en blanco [abc]: a, b, o c.

```
x = c("B12", "Eslava", "Ocho y Medio")
str_view_all(x, "\\d")
```

Strings: expresiones regulares

```
x = c("ES12", "Mi cuenta es ES46", "Resido en España (ES)")  
str_view(x, "ES\\d\\d")
```

Strings: expresiones regulares

Ejercicio: Detecta las autovías correctas

```
x <- c("A1", "A12", "A14", "A6", "EA5", "A3", "F2")
```

Strings: detectar patrones

Con esto estamos preparados para detectar patrones

```
x = c("ES12", "Mi cuenta es ES46", "Resido en España (ES)")  
str_detect(x, "ES\\d\\d")
```

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

Uso común, seleccionar los elementos que cumplen el patrón

```
words[str_detect(words, "x$")]
```

```
## [1] "box" "sex" "six" "tax"
```

Strings: detectar patrones

También con dataframes

```
df <- tibble(  
  word = words,  
  i = seq_along(word)  
)  
df %>%  
  filter(str_detect(word, "x$"))
```

```
## # A tibble: 4 × 2  
##   word     i  
##   <chr> <int>  
## 1 box     108  
## 2 sex     747  
## 3 six     772  
## 4 tax     841
```

Strings: contar patrones

¿Cuántas veces ocurre el patrón?

```
x <- c("apple", "banana", "pear")
str_count(x, "a")  
  
## [1] 1 3 1
```

Ejercicio: calcula la media de vocales por palabra en **words**.

Strings: contar patrones

Se usa mucho con mutate

```
df %>%
  mutate(
    vocales = str_count(word, "[aeiou])",
  )

## # A tibble: 980 × 3
##   word      i vocales
##   <chr>     <int>    <int>
## 1 a          1        1
## 2 able       2        2
## 3 about      3        3
## 4 absolute   4        4
## 5 accept     5        2
## 6 account    6        3
## 7 achieve    7        4
## 8 across     8        2
## 9 act         9        1
## 10 active    10       3
## # ... with 970 more rows
```

Factores

Factores

- Para trabajar con variables categóricas con número fijo de valores
- Trabajaremos con **forcats** de tidyverse
- Crear factores

Factores: creación

```
x1 <- c("Dec", "Apr", "Jan", "Mar")
```

Usar una string para almacenar esta variable tiene dos problemas

- Los valores son fijos y nada previene de errores

```
x1 <- c("Dec", "Apr", "Jan", "Mer")
```

- No ordena de manera útil

```
sort(x1)
```

```
## [1] "Apr" "Dec" "Jan" "Mer"
```

Factores: creación

Esto se arregla creando un factor

```
month_levels <- c(  
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",  
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"  
)  
y1 <- factor(x1, levels = month_levels)  
sort(y1)
```

```
## [1] Jan Apr Dec  
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Factores: creación

Con pipes

```
f2 <- x1 %>% factor(levels = month_levels)  
f2  
  
## [1] Dec Apr Jan <NA>  
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Factores

Trabajaremos con los datos **gss_cat**

```
glimpse(gss_cat)
```

```
## Rows: 21,483
## Columns: 9
## $ year      <int> 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 20...
## $ marital   <fct> Never married, Divorced, Widowed, Never married, Divorced, Mar...
## $ age       <int> 26, 48, 67, 39, 25, 25, 36, 44, 44, 47, 53, 52, 52, 51, 52, 40...
## $ race      <fct> White, White, White, White, White, White, White, White, White, ...
## $ rincome   <fct> $8000 to 9999, $8000 to 9999, Not applicable, Not applicable, ...
## $ partyid   <fct> "Ind,near rep", "Not str republican", "Independent", "Ind,near...
## $ relig     <fct> Protestant, Protestant, Protestant, Orthodox-christian, None, ...
## $ denom     <fct> "Southern baptist", "Baptist-dk which", "No denomination", "No...
## $ tvhours   <int> 12, NA, 2, 4, 1, NA, 3, NA, 0, 3, 2, NA, 1, NA, 1, 7, NA, 3, 3...
```

Factores

Pregunta: ¿Cómo identificarías los niveles posibles de la variable **race**?

```
gss_cat %>% count(race)
```

```
## # A tibble: 3 × 2
##   race     n
##   <fct> <int>
## 1 Other    1959
## 2 Black    3129
## 3 White   16395
```

Factores

- Reordenar niveles
- Modificar niveles

Factores - Reordenar niveles

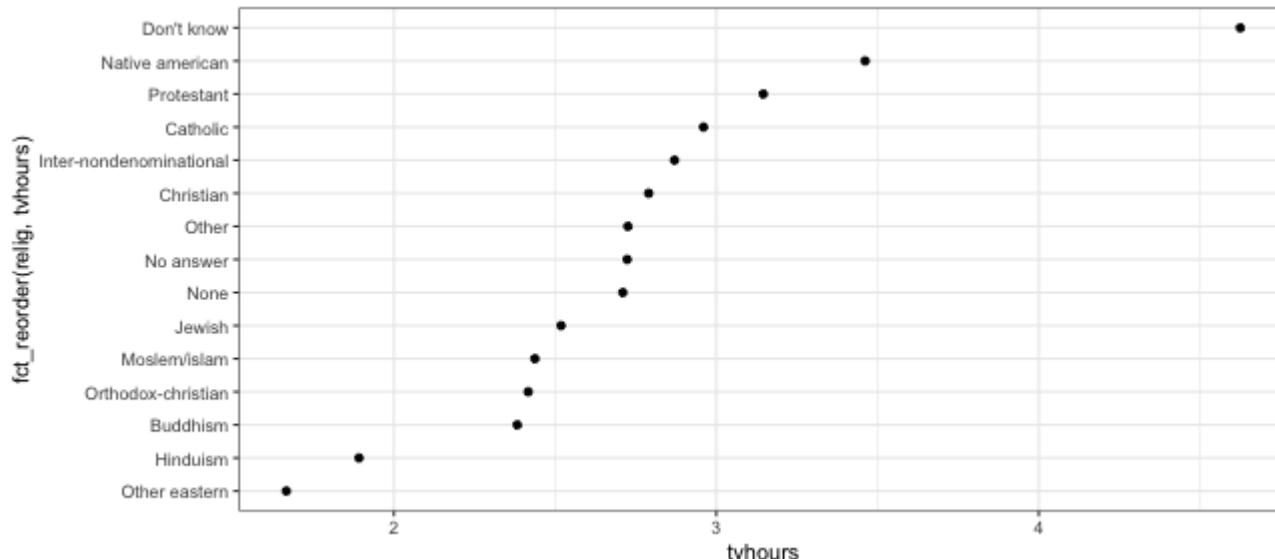
Con **fct_reorder()**. Toma tres argumentos

- **f**, el factor que queremos reordenar
- **x**, vector numérico para ordenar
- Opcionalmente, **fun**, en caso de que haya varios valores de **x** para la misma **f**. (Por defecto, **fun** es la mediana)

Factores - Reordenar niveles

Tres opciones

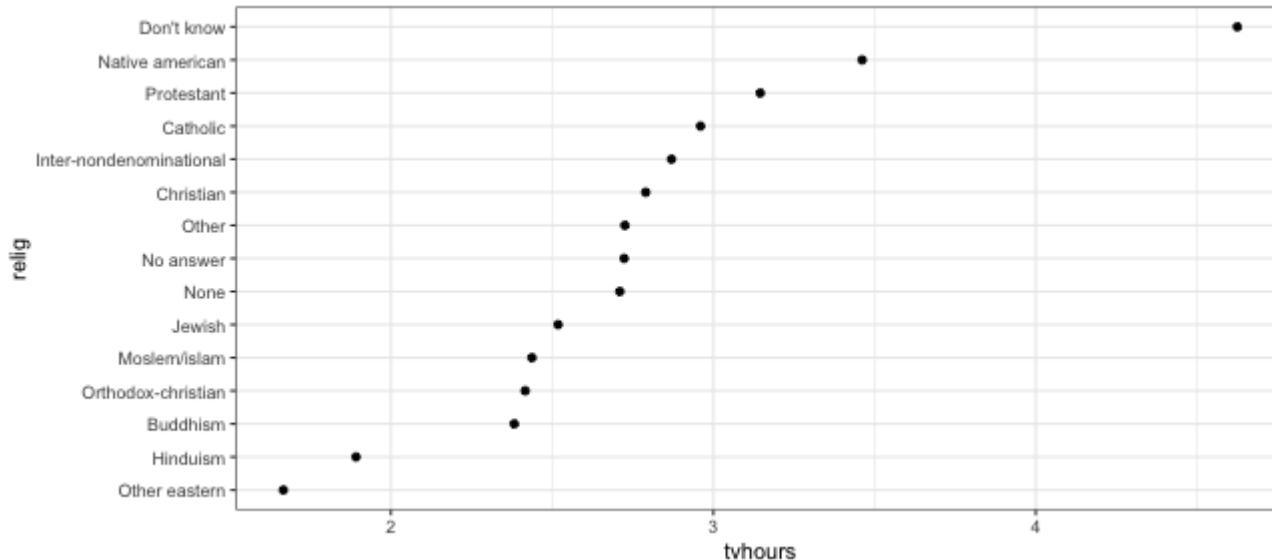
```
gss_cat %>%
  group_by(relig) %>%
  summarise(tvhours = mean(tvhours, na.rm = TRUE)) %>%
  ggplot(aes(tvhours, fct_reorder(relig, tvhours))) +
  geom_point() + theme_bw()
```



Factores - Reordenar niveles

Tres opciones

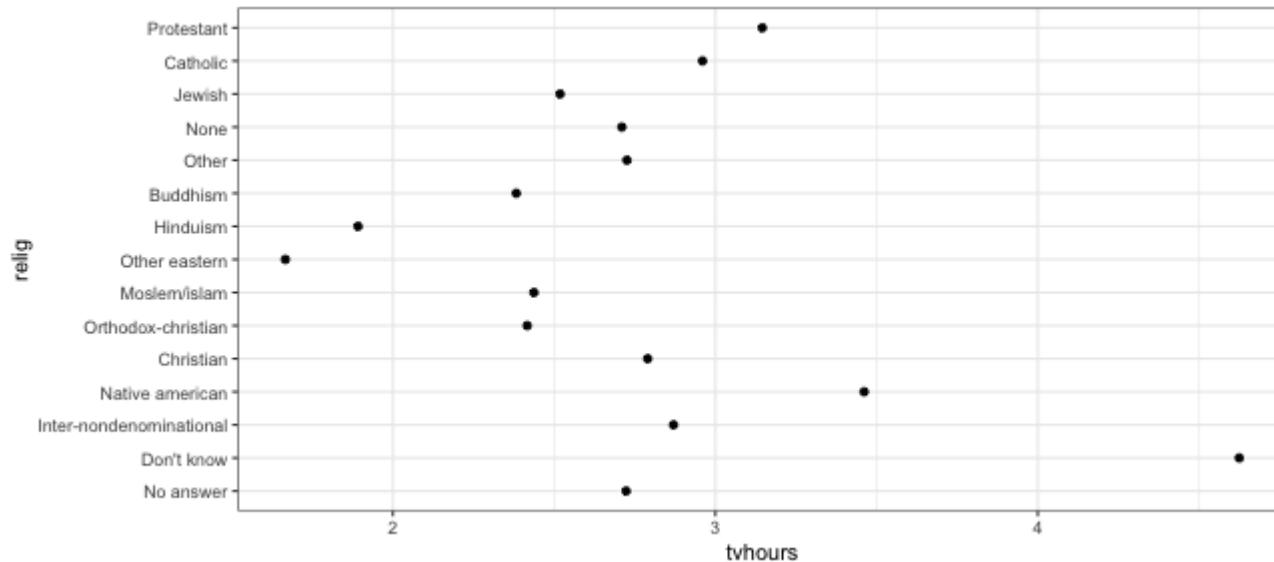
```
gss_cat %>%
  group_by(relig) %>%
  summarise(tvhours = mean(tvhours, na.rm = TRUE)) %>%
  mutate (relig = fct_reorder(relig, tvhours)) %>%
  ggplot(aes(tvhours, relig)) +
  geom_point() + theme_bw()
```



Factores - Reordenar niveles

Tres opciones

```
relig_summary <- gss_cat %>%
  group_by(relig) %>%
  summarise(
    age = mean(age, na.rm = TRUE),
    tvhours = mean(tvhours, na.rm = TRUE),
    n = n()
  )
ggplot(relig_summary, aes(tvhours, relig)) + geom_point() + theme_bw()
```



Factores - Modificar niveles

Usamos `fct_recode()`

```
gss_cat %>% count(partyid)

## # A tibble: 10 × 2
##   partyid      n
##   <fct>     <int>
## 1 No answer    154
## 2 Don't know     1
## 3 Other party   393
## 4 Strong republican  2314
## 5 Not str republican 3032
## 6 Ind,near rep   1791
## 7 Independent    4119
## 8 Ind,near dem    2499
## 9 Not str democrat 3690
## 10 Strong democrat 3490
```

Factores - Modificar niveles

```
gss_cat %>%
  mutate(partyid = fct_recode(partyid,
    "Republican, strong"      = "Strong republican",
    "Republican, weak"        = "Not str republican",
    "Independent, near rep"  = "Ind,near rep",
    "Independent, near dem"  = "Ind,near dem",
    "Democrat, weak"          = "Not str democrat",
    "Democrat, strong"        = "Strong democrat"
  )) %>%
  count(partyid)
```

```
## # A tibble: 10 × 2
##   partyid             n
##   <fct>           <int>
## 1 No answer       154
## 2 Don't know      1
## 3 Other party     393
## 4 Republican, strong 2314
## 5 Republican, weak 3032
## 6 Independent, near rep 1791
## 7 Independent      4119
## 8 Independent, near dem 2499
## 9 Democrat, weak   3690
## 10 Democrat, strong 3490
```

Factores - Modificar niveles

A veces útil colapsar niveles

```
gss_cat %>%
  mutate(partyid = fct_collapse(partyid,
    other = c("No answer", "Don't know", "Other party"),
    rep = c("Strong republican", "Not str republican"),
    ind = c("Ind,near rep", "Independent", "Ind,near dem"),
    dem = c("Not str democrat", "Strong democrat"))
  ) %>%
  count(partyid)
```

```
## # A tibble: 4 × 2
##   partyid     n
##   <fct>   <int>
## 1 other      548
## 2 rep        5346
## 3 ind       8409
## 4 dem       7180
```

Fechas y horas

Fechas y horas

Para trabajar con fechas y horas usamos **lubridate** que NO es parte tidyverse. Descárgalo y cárgalo en R

```
library(lubridate)
```

Fechas y horas

Tres tipos

- Fechas, referidas como en los tibble
- Hora, referida como en los tibble
- Fecha y hora, identifica únicamente un tiempo. Referidas como en los tibble (en R, POSIXct)

Fechas y horas

```
today()
```

```
## [1] "2022-11-13"
```

```
str(today())
```

```
## Date[1:1], format: "2022-11-13"
```

```
now()
```

```
## [1] "2022-11-13 20:46:44 CET"
```

```
str(now())
```

```
## POSIXct[1:1], format: "2022-11-13 20:46:44"
```

Fechas y horas

En general proceden de strings.

Pregunta: ¿Cómo convertir strings en objetos de fecha o fecha-hora?

Fechas y horas

- Otra forma de parsear más fácil con funciones de lubridate
- Solo especificar orden usando "y", "m", y "d"

```
ymd("2017-01-31")
```

```
## [1] "2017-01-31"
```

```
ymd("17-Jan-31")
```

```
## [1] "2017-01-31"
```

Fechas y horas

```
mdy("January 31st, 2017")
```

```
## [1] "2017-01-31"
```

```
dmy("31 of January of 2017")
```

```
## [1] "2017-01-31"
```

```
ymd_hms("2017-01-31 20:11:59")
```

```
## [1] "2017-01-31 20:11:59 UTC"
```

```
mdy_hm("01/31/2017 08:01")
```

```
## [1] "2017-01-31 08:01:00 UTC"
```

Fechas y horas

A veces nos dan las componentes individuales por separado

```
library(nycflights13)
flights %>%
  select(year, month, day, hour, minute)

## # A tibble: 336,776 × 5
##       year   month   day   hour   minute
##   <int> <int> <int> <dbl>   <dbl>
## 1  2013     1     1     5     15
## 2  2013     1     1     5     29
## 3  2013     1     1     5     40
## 4  2013     1     1     5     45
## 5  2013     1     1     6     0
## 6  2013     1     1     5     58
## 7  2013     1     1     6     0
## 8  2013     1     1     6     0
## 9  2013     1     1     6     0
## 10 2013     1     1     6     0
## # ... with 336,766 more rows
```

Fechas y horas

Para esto **make_datetime()**. Ojo orden de argumentos!!

```
flights %>%  
  select(year, month, day, hour, minute) %>%  
  mutate(departure = make_datetime(year, month, day, hour, minute))
```

```
## # A tibble: 336,776 × 6  
##   year month   day hour minute departure  
##   <int> <int> <int> <dbl> <dbl> <dttm>  
## 1 2013     1     1     5     15 2013-01-01 05:15:00  
## 2 2013     1     1     5     29 2013-01-01 05:29:00  
## 3 2013     1     1     5     40 2013-01-01 05:40:00  
## 4 2013     1     1     5     45 2013-01-01 05:45:00  
## 5 2013     1     1     6      0 2013-01-01 06:00:00  
## 6 2013     1     1     5     58 2013-01-01 05:58:00  
## 7 2013     1     1     6      0 2013-01-01 06:00:00  
## 8 2013     1     1     6      0 2013-01-01 06:00:00  
## 9 2013     1     1     6      0 2013-01-01 06:00:00  
## 10 2013    1     1     6      0 2013-01-01 06:00:00  
## # ... with 336,766 more rows
```

Fechas y horas

Cambiar entre formatos

```
as_datetime(today())
```

```
## [1] "2022-11-13 UTC"
```

```
as_date(now())
```

```
## [1] "2022-11-13"
```

Fechas y horas

¿Qué hacer con fechas y horas?

- Extraer componentes
- Realizar operaciones

Fechas y horas - Componentes

```
datetime <- ymd_hms("2016-07-08 12:34:56")  
  
year(datetime)
```

```
## [1] 2016
```

```
month(datetime)
```

```
## [1] 7
```

```
mday(datetime)
```

```
## [1] 8
```

```
yday(datetime)
```

```
## [1] 190
```

```
wday(datetime)
```

```
## [1] 6
```

Fechas y horas - Componentes

```
datetime <- ymd_hms("2016-07-08 12:34:56")  
hour(datetime)
```

```
## [1] 12
```

```
minute(datetime)
```

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

```
second(datetime)
```

```
## [1] 56
```

Fechas y horas - Componentes

```
datetime <- ymd_hms("2016-07-08 12:34:56")
```

```
month(datetime, label = TRUE)
```

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

```
## 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
```

```
wday(datetime, label = TRUE, abbr = FALSE)
```

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

```
## 7 Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < ... < Saturday
```

Fechas y horas - Componentes

Podemos modificar componentes usando las mismas funciones

```
(datetime <- ymd_hms("2016-07-08 12:34:56"))

## [1] "2016-07-08 12:34:56 UTC"

year(datetime) <- 2020
datetime

## [1] "2020-07-08 12:34:56 UTC"

month(datetime) <- 01
datetime

## [1] "2020-01-08 12:34:56 UTC"

hour(datetime) <- hour(datetime) + 1
datetime

## [1] "2020-01-08 13:34:56 UTC"
```

Fechas y horas - Componentes

También podemos modificar componentes con **update**

```
update(datetime, year = 2023)  
  
## [1] "2023-01-08 13:34:56 UTC"
```

Fechas y horas - Operaciones

Para operar con fechas, necesitaremos:

- **Duraciones:** número exacto de segundos
- **Períodos:** otras unidades como meses y semanas

Fechas y horas - Operaciones

Al restar dos fechas, obtenemos un objeto de clase **difftime**

```
mi_edad <- today() - ymd("1993 May 17")  
mi_edad
```

```
## Time difference of 10772 days
```

Este objeto alberga tiempos en segundos, minutos, horas, días, etc.

Ambiguo

Fechas y horas - Operaciones con duraciones

Alternativa, trabajar con duraciones (en segundos)

```
as.duration(mi_edad)
```

```
## [1] "930700800s (~29.49 years)"
```

Fechas y horas - Operaciones con duraciones

Funciones útiles. Siempre en segundos!!

```
dseconds(15)
```

```
## [1] "15s"
```

```
dminutes(10)
```

```
## [1] "600s (~10 minutes)"
```

```
dhours(c(12, 24))
```

```
## [1] "43200s (~12 hours)" "86400s (~1 days)"
```

```
ddays(0:5)
```

```
## [1] "0s"           "86400s (~1 days)" "172800s (~2 days)"
```

```
## [4] "259200s (~3 days)" "345600s (~4 days)" "432000s (~5 days)"
```

Fechas y horas - Operaciones con duraciones

Funciones útiles. Siempre en segundos!!

```
dweeks(3)
```

```
## [1] "1814400s (~3 weeks)"
```

```
dyears(1)
```

```
## [1] "31557600s (~1 years)"
```

Fechas y horas - Operaciones con duraciones

```
dyears(1) + dweeks(12) + dhours(15)
```

```
## [1] "38869200s (~1.23 years)"
```

```
2*dweeks(12)
```

```
## [1] "14515200s (~24 weeks)"
```

```
mañana <- today() + ddays(1)  
año_pasado <- today() - dyears(1)
```

Fechas y horas - Operaciones con períodos

Más intuitivo

```
seconds(15)
```

```
## [1] "15S"
```

```
minutes(10)
```

```
## [1] "10M 0S"
```

```
hours(c(12, 24))
```

```
## [1] "12H 0M 0S" "24H 0M 0S"
```

```
days(7)
```

```
## [1] "7d 0H 0M 0S"
```

Fechas y horas - Operaciones con períodos

Más intuitivo

```
months(1:6)
```

```
## [1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S" "4m 0d 0H 0M 0S"  
## [5] "5m 0d 0H 0M 0S" "6m 0d 0H 0M 0S"
```

```
weeks(3)
```

```
## [1] "21d 0H 0M 0S"
```

```
years(1)
```

```
## [1] "1y 0m 0d 0H 0M 0S"
```

Fechas y horas - Operaciones con períodos

Operaciones con fechas

```
ymd("2017-01-01") + years(1)
```

```
## [1] "2018-01-01"
```

```
ymd("2017-01-01") + months(1)
```

```
## [1] "2017-02-01"
```

```
ymd("2017-01-01") + days(1)
```

```
## [1] "2017-01-02"
```

Fechas y horas - Operaciones con períodos

¿Qué está pasando?

```
ymd("2016-01-01") + dyears(1)
```

```
## [1] "2016-12-31 06:00:00 UTC"
```

```
ymd("2016-01-01") + years(1)
```

```
## [1] "2017-01-01"
```

Bibliografía

Este tema está fundamentalmente basado en [R for Data Science](#), Wickham and Grolemund (2016)