Sesión 3

Manipulación de variables en R

Campamento de invierno EGOB | UC | 04 de agosto, 2023

José D. Conejeros | jdconejeros@uc.cl

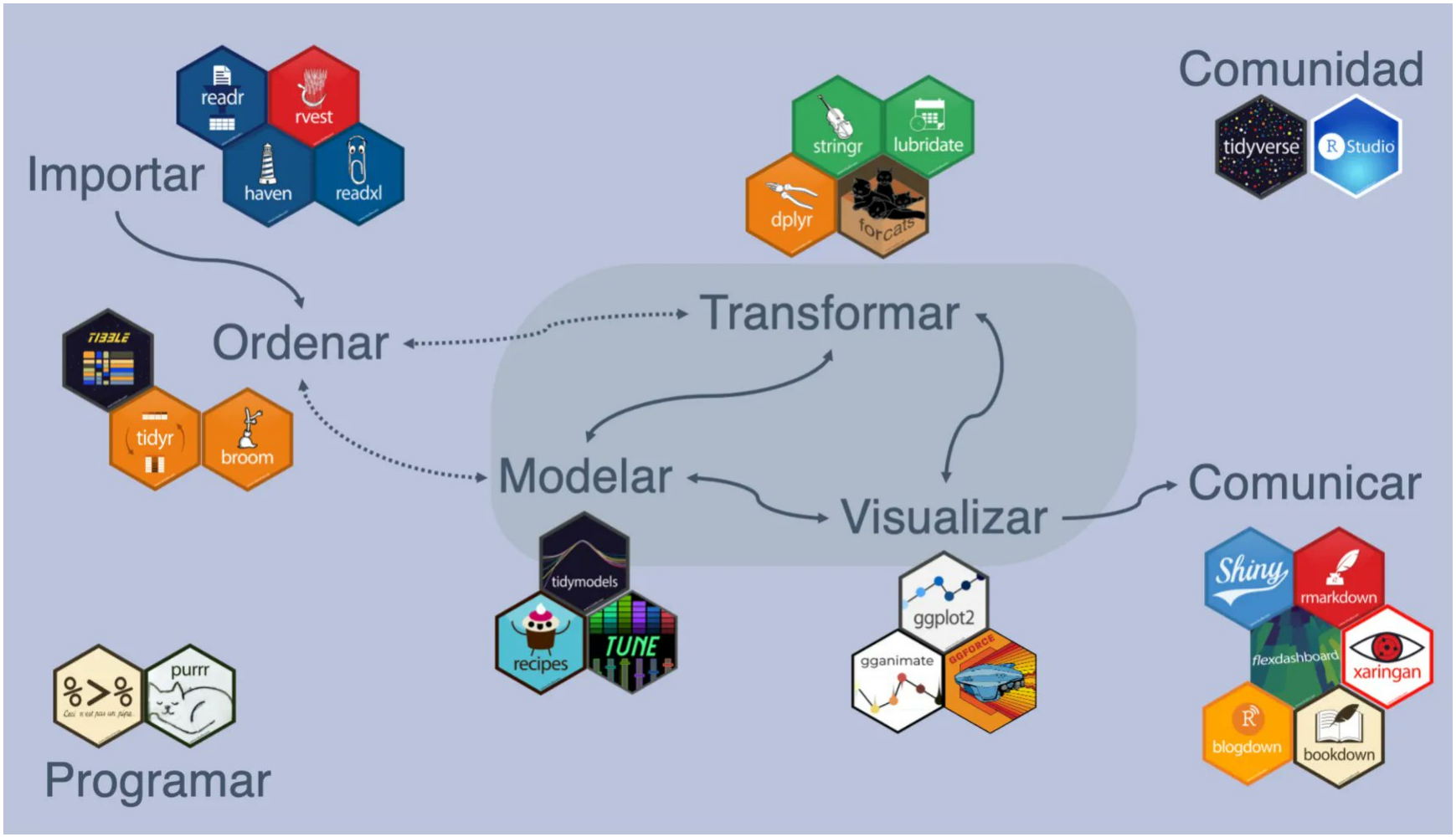
Guía

1. Tidyverse
2. Recodificación de variables
3. Variables numéricas
4. Variables tipo factor
5. Fechas

Entonces, ¿qué es tidyverse?

- Un universo de paquentes de R
- Diseñado para las tareas de programación, importación, ordenación, manipulación y visualización de datos.
- Tienen una misma filosofía de diseño, gramática y estructura de datos. Cuenta con alta consistencia interna.
- Resuelve problemas complejos combinando diferentes piezas consistentes unas con otras.

Las librerías de tidyverse



El corazón son las librerías

- `readr`: lectura de tablas de datos en formato csv, txt, entre otros.
- `dplyr`: procesamiento de variables y datos.
- `tidyr`: trabajo con bases de datos ordenadas.
- `ggplot2`: visualización de datos.
- `purrr`: Herramienta para trabajar funciones, vectores e iteraciones.
- `tibble`: gestión de marcos de datos.
- `stringr`: trabajo con variables de tipo caracter (textos).
- `forcats`: trabajo con variables de tipo factor (variables cualitativas).

Recodificación de variables

Una condición: `if_else()` o `ifelse()`

```
data ← data %>%
  mutate(nueva_variable=
           if_else(condificion,
                   accion_verdadero,
                   accion_falsa))
```

Múltiples condiciones `case_when()`

```
data ← data %>%
  mutate(nueva_variable=case_when(
    condificion1 ~ accion1,
    condificion2 ~ accion2,
    condificion3 ~ accion3,
    .,
    .,
    condificion_N ~ accion_N,
    TRUE ~ NA
    ))
# Si estoy generando numeric es Na_real
# Si estoy generando texto es Na_characte
```

Variables numéricas en R

Los vectores numéricos son la columna vertebral de la ciencia de datos. Veamos el siguiente vector:

```
x ← c("1.2", "5.6", "1e3")
class(x)
```

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

Podemos transformalo de dos formas:

R base

```
x_num ← as.numeric(x)
class(x_num); x_num
```

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

```
[1]    1.2    5.6 1000.0
```

Tidyverse

```
x_num ← parse_double(x)
class(x_num); x_num
```

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

```
[1]    1.2    5.6 1000.0
```

Variables numéricas en R

Ahora veamos el siguiente ejemplo:

```
x ← c("$1,234", "USD 3,513", "59%")
class(x)
```

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

R base

```
x_num ← as.numeric(x)
class(x_num); x_num
```

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

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

Tidyverse

```
x_num ← parse_number(x)
class(x_num); x_num
```

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

```
[1] 1234 3513   59
```

Algunas modificaciones sobre variables numéricas

Redondeo

```
x ← 123.456

floor(x) # Hacia abajo
```

```
[1] 123
```

```
ceiling(x) # Hacia arriba
```

```
[1] 124
```

```
round(x, 1)
```

```
[1] 123.5
```

Algunas modificaciones sobre variables numéricas

Sumas acumuladas

```
x ← 1:10
cumsum(x)
```

```
[1]  1  3  6 10 15 21 28 36 45 55
```

Y los clásicos resumenes descriptivos que hemos visto anteriormente.

Factores en R

Los factores se utilizan para variables categóricas, variables que tienen un conjunto fijo y conocido de valores posibles. También para tener vectores de caracteres en un orden no alfabético.

Veamos el siguiente ejemplo:

```
x1 ← c("Dic", "Abr", "Ene", "Mar") # Variable de interés

mes_agno ← c(
  "Ene", "Feb", "Mar", "Abr", "May", "Jun",
  "Jul", "Ago", "Sep", "Oct", "Nov", "Dic"
)
class(mes_agno); mes_agno
```

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

```
 [1] "Ene" "Feb" "Mar" "Abr" "May" "Jun" "Jul" "Ago" "Sep" "Oct" "Nov" "Dic"
```

Apliquemos un ordenamiento:

```
sort(x1) # Ordenamos
```

```
[1] "Abr" "Dic" "Ene" "Mar"
```

Factores en R

Ahora definamos la variable como factor:

```
y1 ← factor(x1, levels = mes_agno)
class(y1); y1
```

```
[1] "factor"
```

```
[1] Dic Abr Ene Mar
Levels: Ene Feb Mar Abr May Jun Jul Ago Sep Oct Nov Dic
```

Ordenemos nuevamente:

```
sort(y1)
```

```
[1] Ene Mar Abr Dic
Levels: Ene Feb Mar Abr May Jun Jul Ago Sep Oct Nov Dic
```

Veamos un ejemplo

General Social Survey es una encuesta estadounidense de larga duración realizada por la organización de investigación independiente NORC en la Universidad de Chicago.

```
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…
```

Veamos un ejemplo

```
gss_cat ▷
  count(race)
```

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

Veamos un ejemplo

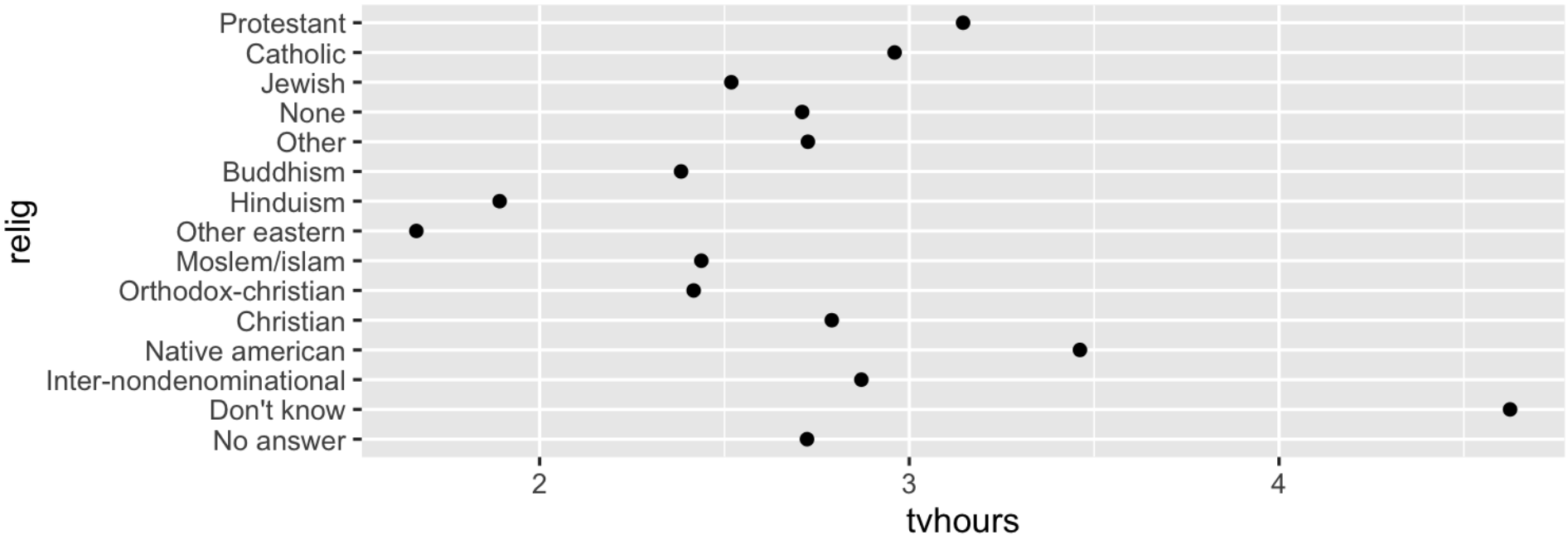
```
relig_summary ← gss_cat ▷
  group_by(relig) ▷
  summarize(
    tvhours = mean(tvhours, na.rm = TRUE),
    n = n()
  )

relig_summary
```

```
# A tibble: 15 × 3
   relig                   tvhours     n
   <fct>                     <dbl> <int>
 1 No answer                  2.72    93
 2 Don't know                 4.62    15
 3 Inter-nondenominational    2.87   109
 4 Native american            3.46    23
 5 Christian                  2.79   689
 6 Orthodox-christian         2.42    95
 7 Moslem/islam               2.44   104
 8 Other eastern              1.67    32
 9 Hinduism                   1.89    71
10 Buddhism                   2.38   147
```

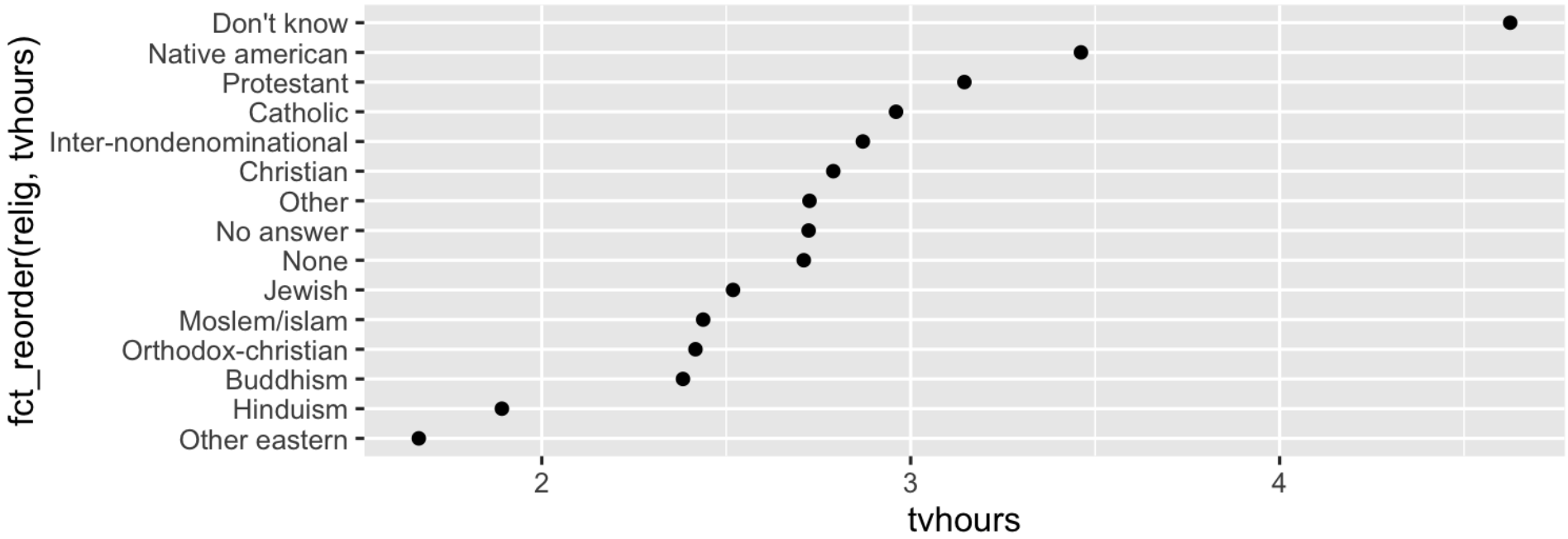
Veamos un ejemplo

```
ggplot(relig_summary, aes(x = tvhours, y = relig)) +
  geom_point()
```



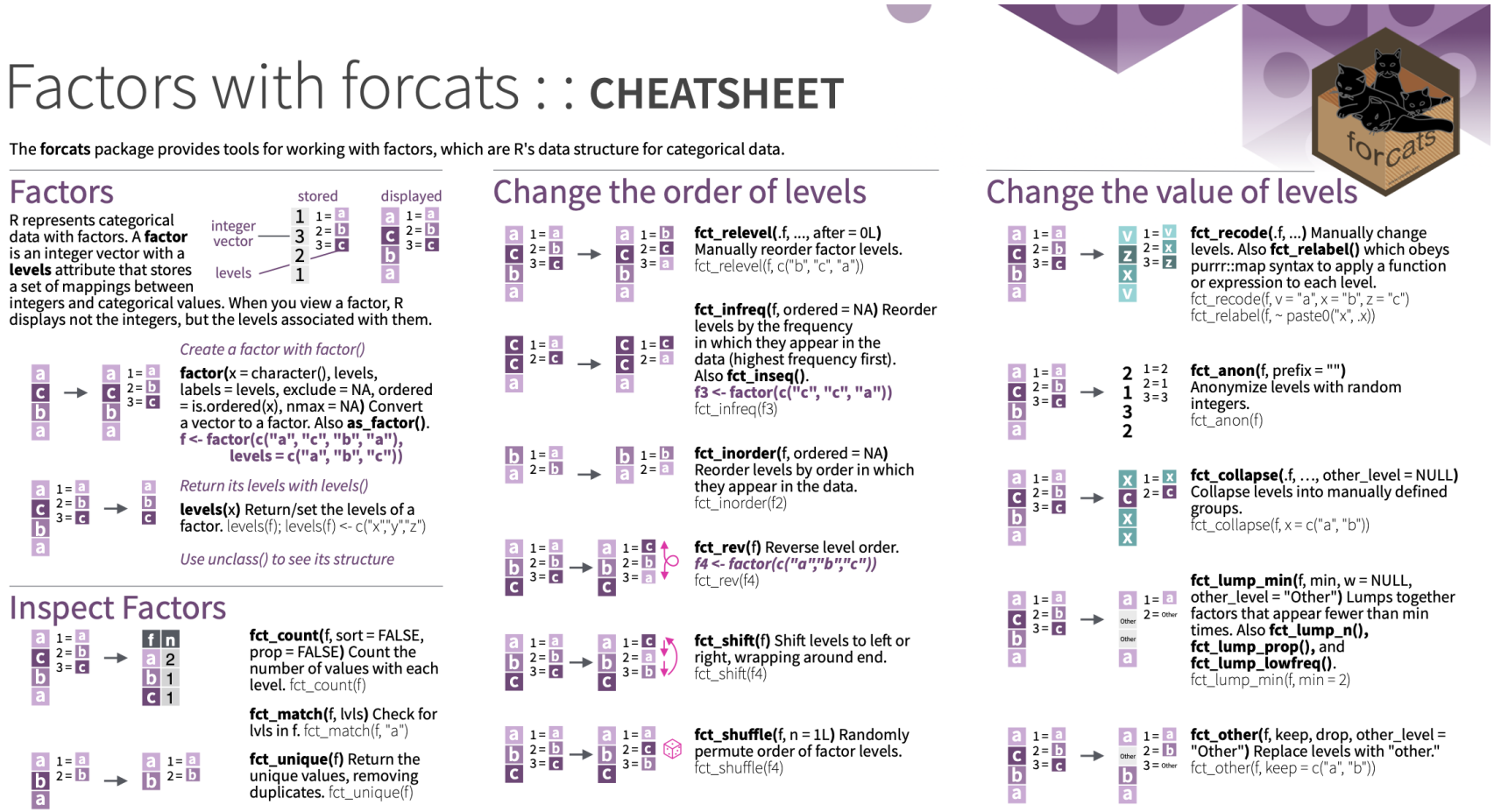
Veamos un ejemplo

```
ggplot(relig_summary, aes(x = tvhours, y = fct_reorder(relig, tvhours))) +
  geom_point()
```



Forcats

El paquete forcats proporciona herramientas para trabajar con factores, que son la estructura de datos de R para datos categóricos. Más info en: < https://forcats.tidyverse.org/ >



Fechas en R

A primera vista, las fechas y las horas parecen simples. Sin embargo, cuanto más aprendes sobre las fechas y las horas, ¡más complicadas parecen volverse!

```
today()
```

```
[1] "2023-08-04"
```

```
now()
```

```
[1] "2023-08-04 02:20:12 -04"
```

Fechas en R

A primera vista, las fechas y las horas parecen simples. Sin embargo, cuanto más aprendes sobre las fechas y las horas, ¡más complicadas parecen volverse!

Tabla 18.1: Todos los formatos de fecha entendidos por readr

Tipo	Código	Significado	Ejemplo
Año	%Y	año de 4 dígitos	2021
	%y	año de 2 dígitos	21
Mes	%m	Número	2
	%b	Nombre abreviado	Feb
	%B	Nombre completo	Febrero
Día	%d	Dos dígitos	02
	%e	Uno o dos dígitos	2
Tiempo	%H	hora de 24 horas	13
	%I	hora de 12 horas	1
	%p	AM PM	pm
	%M	Minutos	35
	%S	Segundos	45
	%0S	Segundos con componente decimal	45.35
	%Z	Nombre de la zona horaria	América/Chicago
	%z	Desplazamiento de UTC	+0800
Otro	%.	Omitir uno que no sea un dígito	:
	%*	Saltar cualquier número de no dígitos	

Estructura de fechas

Definiendo fechas desde cadenas de caracteres:

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

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

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

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

```
dmy("31-Jan-2017")
```

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

Estructura de fechas

Definiendo fechas desde cadenas de caracteres:

```
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"
```

Estructura de fechas

```
library(nycflights13)
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
# ℹ 336,766 more rows
```

Extraer componentes de una fecha

```
datetime ← ymd_hms("2026-07-08 12:34:56"); class(datetime)
```

```
[1] "POSIXct" "POSIXt"
```

```
year(datetime)
```

```
[1] 2026
```

```
month(datetime)
```

```
[1] 7
```

```
mday(datetime)
```

```
[1] 8
```

Extraer componentes de una fecha

```
yday(datetime)
```

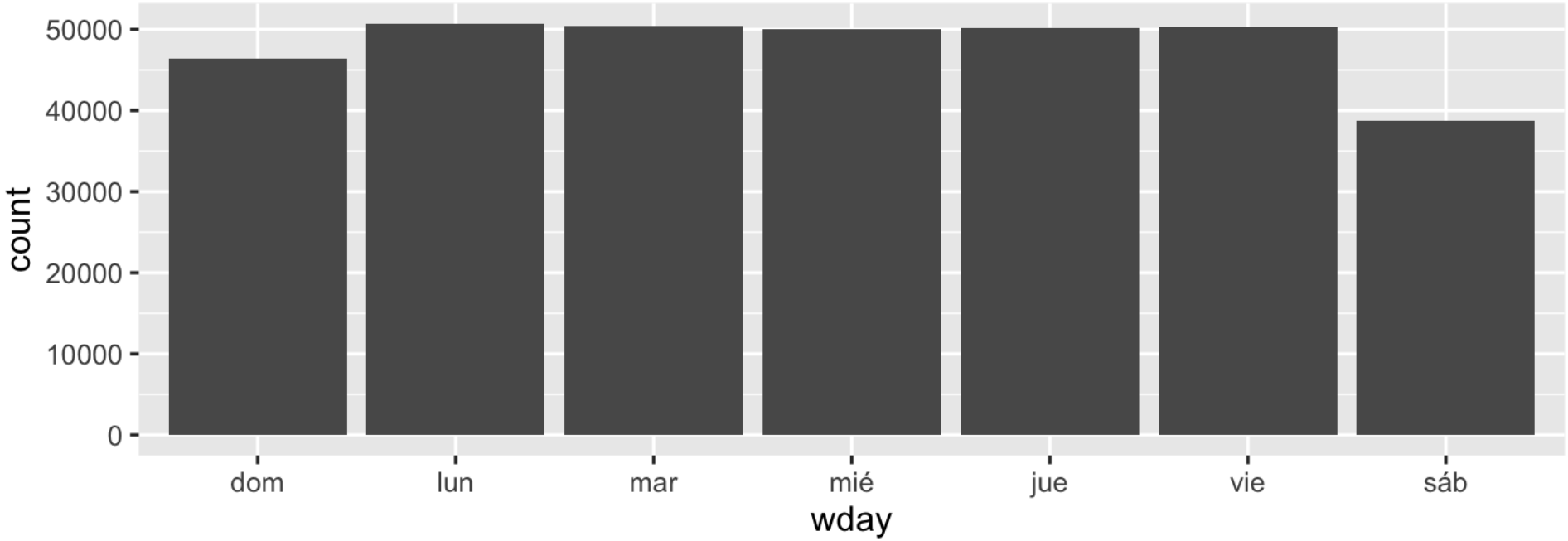
```
[1] 189
```

```
wday(datetime)
```

```
[1] 4
```

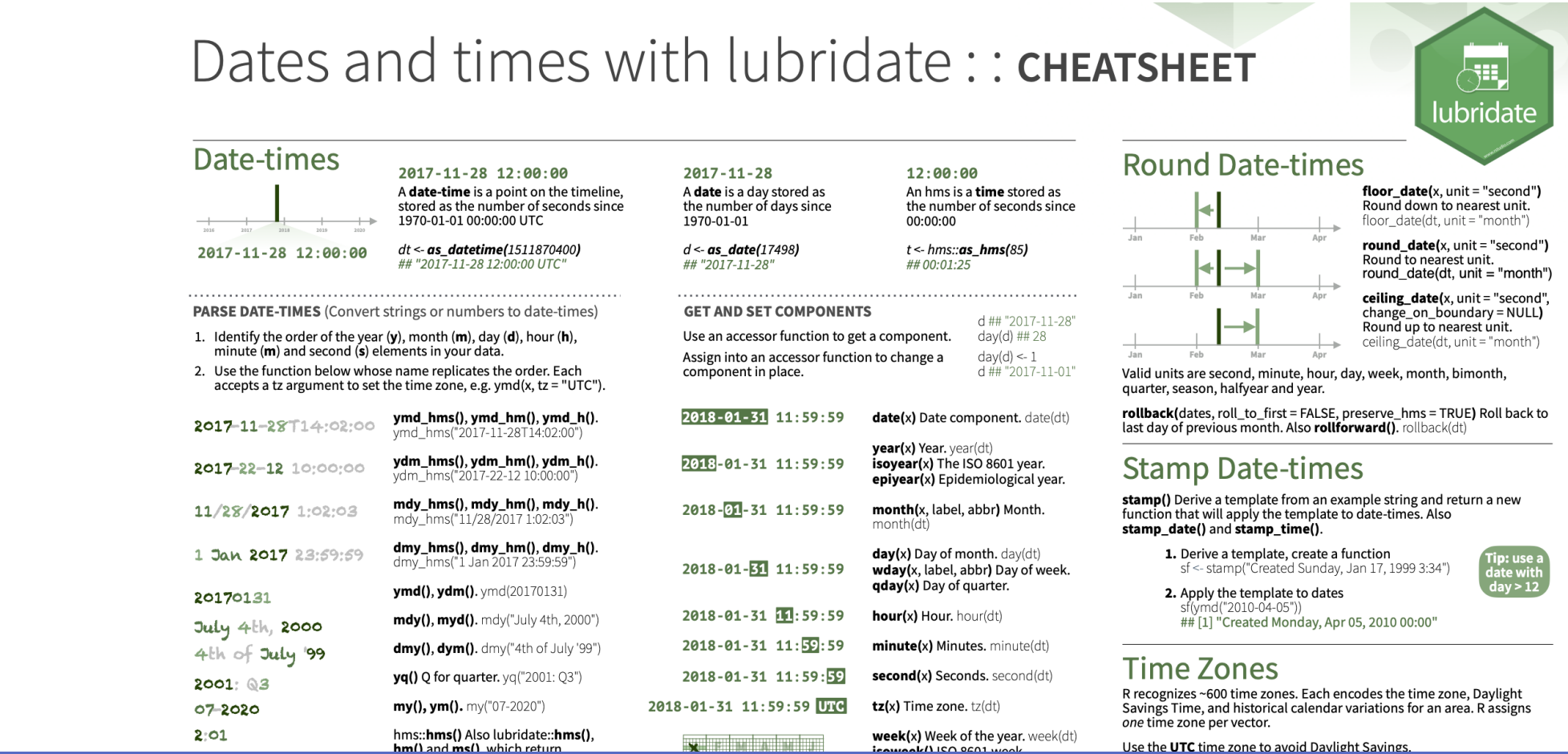
Generar piezas de información de una fecha

```
flights ▷
  mutate(departure = make_datetime(year, month, day, hour, minute)) ▷
  mutate(wday = wday(departure, label = TRUE)) ▷
  ggplot(aes(x = wday)) +
  geom_bar()
```



Lubridate

El paquete lubridate proporciona herramientas para trabajar con fechas. Más info en: < https://lubridate.tidyverse.org/ >



Dates and times with lubridate : : CHEATSHEET

lubridate

Date-times

2017-11-28 12:00:00

A **date-time** is a point on the timeline, stored as the number of seconds since 1970-01-01 00:00:00 UTC

dt <- ***as_datetime****(1511870400)*
"2017-11-28 12:00:00 UTC"

2017-11-28

A **date** is a day stored as the number of days since 1970-01-01

d <- ***as_date****(17498)*
"2017-11-28"

12:00:00

An hms is a **time** stored as the number of seconds since 00:00:00

*t <- hms::****as_hms****(85)*
00:01:25

PARSE DATE-TIMES (Convert strings or numbers to date-times)

1. Identify the order of the year (**y**), month (**m**), day (**d**), hour (**h**), minute (**m**) and second (**s**) elements in your data.
2. Use the function below whose name replicates the order. Each accepts a tz argument to set the time zone, e.g. ymd(x, tz = "UTC").

2017-11-28T14:02:00 — **ymd_hms(), ymd_hm(), ymd_h().** ymd_hms("2017-11-28T14:02:00")

2017-22-12 10:00:00 — **ydm_hms(), ydm_hm(), ydm_h().** ydm_hms("2017-22-12 10:00:00")

11/28/2017 1:02:03 — **mdy_hms(), mdy_hm(), mdy_h().** mdy_hms("11/28/2017 1:02:03")

1 Jan 2017 23:59:59 — **dmy_hms(), dmy_hm(), dmy_h().** dmy_hms("1 Jan 2017 23:59:59")

20170131 — **ymd(), ydm().** ymd(20170131)

July 4th, 2000 — **mdy(), myd().** mdy("July 4th, 2000")

4th of July '99 — **dmy(), dym().** dmy("4th of July '99")

2001: Q3 — **yq()** Q for quarter. yq("2001: Q3")

07-2020 — **my(), ym().** my("07-2020")

2:01 — hms::**hms()** Also lubridate::**hms(), hm()** and **ms()**, which return

GET AND SET COMPONENTS

Use an accessor function to get a component. — d ## "2017-11-28" / day(d) ## 28

Assign into an accessor function to change a component in place. — day(d) <- 1 / d ## "2017-11-01"

2018-01-31 11:59:59 — **date(x)** Date component. date(dt)

2018-01-31 11:59:59 — **year(x)** Year. year(dt) **isoyear(x)** The ISO 8601 year. **epiyear(x)** Epidemiological year.

2018-01-31 11:59:59 — **month(x, label, abbr)** Month. month(dt)

2018-01-31 11:59:59 — **day(x)** Day of month. day(dt) **wday(x, label, abbr)** Day of week. **qday(x)** Day of quarter.

2018-01-31 11:59:59 — **hour(x)** Hour. hour(dt)

2018-01-31 11:59:59 — **minute(x)** Minutes. minute(dt)

2018-01-31 11:59:59 — **second(x)** Seconds. second(dt)

2018-01-31 11:59:59 UTC — **tz(x)** Time zone. tz(dt)

week(x) Week of the year. week(dt)

Round Date-times

Jan Feb Mar Apr

floor_date(x, unit = "second") Round down to nearest unit. floor_date(dt, unit = "month")

round_date(x, unit = "second") Round to nearest unit. round_date(dt, unit = "month")

ceiling_date(x, unit = "second", change_on_boundary = NULL) Round up to nearest unit. ceiling_date(dt, unit = "month")

Valid units are second, minute, hour, day, week, month, bimonth, quarter, season, halfyear and year.

rollback(dates, roll_to_first = FALSE, preserve_hms = TRUE**)** Roll back to last day of previous month. Also **rollforward()**. rollback(dt)

Stamp Date-times

stamp() Derive a template from an example string and return a new function that will apply the template to date-times. Also **stamp_date()** and **stamp_time()**.

1. Derive a template, create a function
   sf <- stamp("Created Sunday, Jan 17, 1999 3:34")
2. Apply the template to dates
   sf(ymd("2010-04-05"))
   ## [1] "Created Monday, Apr 05, 2010 00:00"

Tip: use a date with day > 12

Time Zones

R recognizes ~600 time zones. Each encodes the time zone, Daylight Savings Time, and historical calendar variations for an area. R assigns *one* time zone per vector.

Use the **UTC** time zone to avoid Daylight Savings.

Variables tipo texto en R

Para crear una cadena se puede utilizar comillas simples (') o comillas dobles ("). No hay diferencia en el comportamiento entre los dos, por lo que en aras de la coherencia, la guía de estilo de tidyverse recomienda usar ", a menos que la cadena contenga múltiples ". Por ejemplo:

```
string1 ← "Taller de R"; class(string1); string1
```

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

```
[1] "Taller de R"
```

```
string2 ← "Taller de R EGOB auspiciado por 'Super 8 company'"; class(string2); string2
```

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

```
[1] "Taller de R EGOB auspiciado por 'Super 8 company'"
```

Existe una serie de caracteres especiales que uno podría especificar, por ejemplo:

```
"\U0001f604"
```

```
[1] "😄"
```

Operaciones con caracteres

- `str_c()`

```
# Con vectores
str_c("Hello ", c("John", "Susan"))
```

```
[1] "Hello John"  "Hello Susan"
```

```
# Con datos
df ← tibble(name = c("Flora", "David", "Terra", NA))
df ▷ mutate(greeting = str_c("Hi ", name, "!"))
```

```
# A tibble: 4 × 2
  name  greeting
  <chr> <chr>
1 Flora Hi Flora!
2 David Hi David!
3 Terra Hi Terra!
4 <NA>  <NA>
```

Operaciones con caracteres

- `str_glue()`

```
df ▷ mutate(greeting = str_glue("Hi {name}!"))
```

```
# A tibble: 4 × 2
  name  greeting
  <chr> <glue>
1 Flora Hi Flora!
2 David Hi David!
3 Terra Hi Terra!
4 <NA>  Hi NA!
```

Operaciones con caracteres

- `str_flatten()`:

```
df ← tribble(
  ~ name, ~ fruit,
  "Carmen", "banana",
  "Carmen", "apple",
  "Marvin", "nectarine",
  "Terence", "cantaloupe",
  "Terence", "papaya",
  "Terence", "mandarin"
)
df ▷
  group_by(name) ▷
  summarize(fruits = str_flatten(fruit, ", "))
```

```
# A tibble: 3 × 2
  name    fruits
  <chr>   <chr>
1 Carmen  banana, apple
2 Marvin  nectarine
3 Terence cantaloupe, papaya, mandarin
```

Operaciones con caracteres

- `str_sub()`:

```
library(babynames)
babynames ▷
  mutate(
    first = str_sub(name, 1, 1),
    last = str_sub(name, -1, -1)
  )
```

```
# A tibble: 1,924,665 × 7
    year sex   name           n   prop first last
   <dbl> <chr> <chr>      <int>  <dbl> <chr> <chr>
 1  1880 F     Mary        7065 0.0724 M     y
 2  1880 F     Anna        2604 0.0267 A     a
 3  1880 F     Emma        2003 0.0205 E     a
 4  1880 F     Elizabeth   1939 0.0199 E     h
 5  1880 F     Minnie      1746 0.0179 M     e
 6  1880 F     Margaret    1578 0.0162 M     t
 7  1880 F     Ida         1472 0.0151 I     a
 8  1880 F     Alice       1414 0.0145 A     e
 9  1880 F     Bertha      1320 0.0135 B     a
10  1880 F     Sarah       1288 0.0132 S     h
```

Uso de expresiones regulares

Un lenguaje conciso y poderoso para describir patrones dentro de cadenas de caracteres:

Veamos un ejemplo:

```
fruit
```

```
 [1] "apple"          "apricot"        "avocado"
 [4] "banana"         "bell pepper"    "bilberry"
 [7] "blackberry"     "blackcurrant"   "blood orange"
[10] "blueberry"      "boysenberry"    "breadfruit"
[13] "canary melon"   "cantaloupe"     "cherimoya"
[16] "cherry"         "chili pepper"   "clementine"
[19] "cloudberry"     "coconut"        "cranberry"
[22] "cucumber"       "currant"        "damson"
[25] "date"           "dragonfruit"    "durian"
[28] "eggplant"       "elderberry"     "feijoa"
[31] "fig"            "goji berry"     "gooseberry"
[34] "grape"          "grapefruit"     "guava"
[37] "honeydew"       "huckleberry"    "jackfruit"
[40] "jambul"         "jujube"         "kiwi fruit"
[43] "kumquat"        "lemon"          "lime"
[46] "loquat"         "lychee"         "mandarine"
```

Uso de expresiones regulares

Veamos un ejemplo:

```
str_view(fruit, "apple|melon|nut")
```

```
 [1] │ <apple>
[13] │ canary <melon>
[20] │ coco<nut>
[52] │ <nut>
[62] │ pine<apple>
[72] │ rock <melon>
[80] │ water<melon>
```

Algunas expresiones regulares

- `\\d`: Dígito del 1 al 9 0,1,2 ... 9
- `\\D`: Distinto de dígito A, a, $,)
- `\\s`: Espacio
- `\\S`: Distinto de espacio
- `\\w`: Palabra A, B, C, d, e, f, ...
- `\\W`: Distinto de palabra _, &, #, ...
- `\\t`: Tabulador
- `\\n`: Salto de linea
- `^`: Comienzo de cadena ^C -> Casa, Coche, ...
- `$`: Fin de cadena s$ -> Casas, coches, ...
- `\`: Caracteres especiales. \, +
- `|`: O lógico OR (v|b)aca -> vaca o baca
- `[ab]`: O lógico OR a, b
- `[^ab]`: Distintos de ab c, d, e, f, ...
- `[0-9]`: Todos los dígitos 0, 1, 2, 3, ...
- `[A-Z]`: Todas las letras mayúsculas A, B, C, ...
- `[a-z]`: Todas las letras minúsculas a, b, c, ...

Algunas expresiones regulares

- `a+`: Letra «a» al menos una vez a, aa, aaa, ...
- `a*`: Letra «a» cero o más veces a, , aa, aaa, ...
- `a?`: Letra «a» cero o una vez a
- `a{4}`: Buscar 4 «a» seguidas aaaa
- `a{2,4}`: Buscar entre 2 y 4 «a» seguidas aa, aaa, aaaa
- `a{2,4}?`: Buscar entre 2 y 4 «a» seguidas como mucho una vez aa, aaaa, ...
- `a{2,}`: Busca a partir de 2 «a» seguidas aa, aaa, aaaa, aaaa, ...

Algunas expresiones regulares

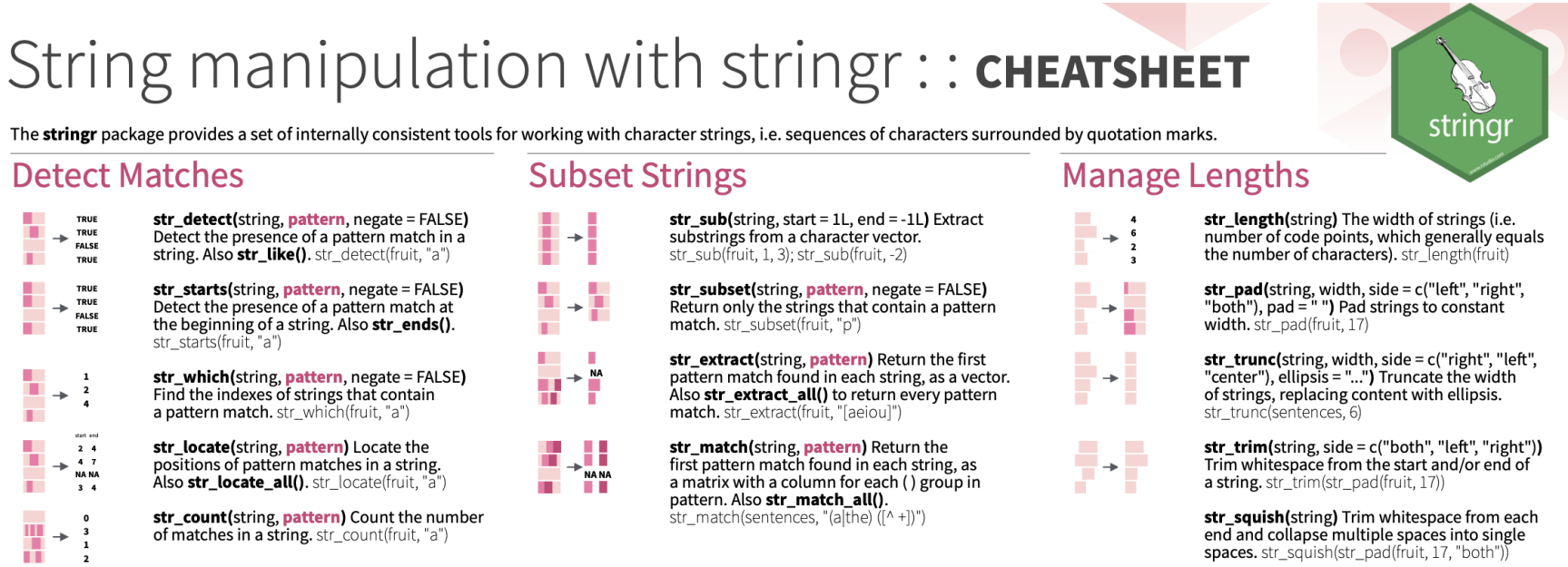
- `[:alnum:]`: Caracteres alfanuméricos [:alpha:] y [:digit:] A, B, c, d, 1, 2, ...
- `[:alpha:]`: Caracteres: [:lower:] y [:upper:] A, B, c, ...
- `[:blank:]`: Caracteres blancos Espacio, Tabulador, ...
- `[:cntrl:]`: Caracteres de control
- `[:digit:]`: Dígitos 0, 1, 2, 3, ...
- `[:graph:]`: Caracteres gráficos [:alnum:] y [:punct:] A, B, c, d, 1, 2, #, %, ...
- `[:lower:]`: Todas las letras minúsculas a, b, c, ...
- `[:print:]`: Caracteres gráficos [:alnum:] y [:punct:] A, B, c, d, 1, 2, #, %, ...
- `[:punct:]`: Caracteres de puntuación ! » # $ % & ‘ ( ) * + , – . / : ; < = > ? @ [\ ] ^ _ ` { | } ~
- `[:space:]`: Caracteres de espaciado Espacio, tabulador, nueva linea, ...
- `[:upper:]`: Todas las letras mayúsculas A, B, C, ...
- `[:xdigit:]`: Dígitos hexadecimales 0, 1, 2, 3, A, B, e, f, ...

Fuente: < https://www.diegocalvo.es/expresiones-regulares-en-r/ >

Nota: ChatGPT ayuda bastante a la construcción de expresiones regulares.

Stringr

El paquete forcats proporciona herramientas para trabajar con caracteres. Más info en: < https://stringr.tidyverse.org/ >



String manipulation with stringr : : CHEATSHEET

The **stringr** package provides a set of internally consistent tools for working with character strings, i.e. sequences of characters surrounded by quotation marks.

Detect Matches

str_detect(string, **pattern**, negate = FALSE**)** Detect the presence of a pattern match in a string. Also **str_like()**. str_detect(fruit, "a")

str_starts(string, **pattern**, negate = FALSE**)** Detect the presence of a pattern match at the beginning of a string. Also **str_ends()**. str_starts(fruit, "a")

str_which(string, **pattern**, negate = FALSE**)** Find the indexes of strings that contain a pattern match. str_which(fruit, "a")

str_locate(string, **pattern**) Locate the positions of pattern matches in a string. Also **str_locate_all()**. str_locate(fruit, "a")

str_count(string, **pattern**) Count the number of matches in a string. str_count(fruit, "a")

Subset Strings

str_sub(string, start = 1L, end = -1L**)** Extract substrings from a character vector. str_sub(fruit, 1, 3); str_sub(fruit, -2)

str_subset(string, **pattern**, negate = FALSE**)** Return only the strings that contain a pattern match. str_subset(fruit, "p")

str_extract(string, **pattern**) Return the first pattern match found in each string, as a vector. Also **str_extract_all()** to return every pattern match. str_extract(fruit, "[aeiou]")

str_match(string, **pattern**) Return the first pattern match found in each string, as a matrix with a column for each ( ) group in pattern. Also **str_match_all()**. str_match(sentences, "(a|the) ([^ +])")

Manage Lengths

str_length(string**)** The width of strings (i.e. number of code points, which generally equals the number of characters). str_length(fruit)

str_pad(string, width, side = c("left", "right", "both"), pad = " "**)** Pad strings to constant width. str_pad(fruit, 17)

str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "..."**)** Truncate the width of strings, replacing content with ellipsis. str_trunc(sentences, 6)

str_trim(string, side = c("both", "left", "right"**))** Trim whitespace from the start and/or end of a string. str_trim(str_pad(fruit, 17))

str_squish(string**)** Trim whitespace from each end and collapse multiple spaces into single spaces. str_squish(str_pad(fruit, 17, "both"))

Stringr

Mutate Strings

str_sub() <- value. Replace substrings by identifying the substrings with str_sub() and assigning into the results.
str_sub(fruit, 1, 3) <- "str"

str_replace(string, **pattern**, replacement**)** Replace the first matched pattern in each string. Also **str_remove()**.
str_replace(fruit, "p", "-")

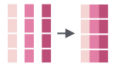
str_replace_all(string, **pattern**, replacement**)** Replace all matched patterns in each string. Also **str_remove_all()**.
str_replace_all(fruit, "p", "-")

str_to_lower(string, locale = "en"**)**[1] Convert strings to lower case.
str_to_lower(sentences)

str_to_upper(string, locale = "en"**)**[1] Convert strings to upper case.
str_to_upper(sentences)

str_to_title(string, locale = "en"**)**[1] Convert strings to title case. Also **str_to_sentence()**.
str_to_title(sentences)

Join and Split

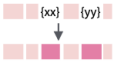


str_c(..., sep = "", collapse = NULL**)** Join multiple strings into a single string.
str_c(letters, LETTERS)

str_flatten(string, collapse = ""**)** Combines into a single string, separated by collapse.
str_flatten(fruit, ", ")

str_dup(string, times**)** Repeat strings times times. Also **str_unique()** to remove duplicates.
str_dup(fruit, times = 2)

str_split_fixed(string, **pattern**, n**)** Split a vector of strings into a matrix of substrings (splitting at occurrences of a pattern match). Also **str_split()** to return a list of substrings and **str_split_n()** to return the nth substring.
str_split_fixed(sentences, " ", n=3)



str_glue(..., .sep = "", .envir = parent.frame()**)** Create a string from strings and {expressions} to evaluate. str_glue("Pi is {pi}")

str_glue_data(.x, ..., .sep = "", .envir = parent.frame(), .na = "NA"**)** Use a data frame, list, or environment to create a string from strings and {expressions} to evaluate.
str_glue_data(mtcars, "{rownames(mtcars)} has {hp} hp")

Order Strings

str_order(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...**)**[1] Return the vector of indexes that sorts a character vector. fruit[str_order(fruit)]

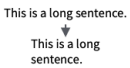
str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...**)**[1] Sort a character vector. str_sort(fruit)

Helpers

str_conv(string, encoding**)** Override the encoding of a string. str_conv(fruit,"ISO-8859-1")

str_view(string, **pattern**, match = NA**)** View HTML rendering of all regex matches.
str_view(sentences, "[aeiou]")

str_equal(x, y, locale = "en", ignore_case = FALSE, ...**)**[1] Determine if two strings are equivalent. str_equal(c("a", "b"), c("a", "c"))



str_wrap(string, width = 80, indent = 0, exdent = 0**)** Wrap strings into nicely formatted paragraphs. str_wrap(sentences, 20)

[1] See **bit.ly/ISO639-1** for a complete list of locales.

posit™

Referencias

Wickham, H., & Grolemund, G. (2016). R for data science: import, tidy, transform, visualize, and model data. " O'Reilly Media, Inc.". Cap. 13 al 19. Recurso en línea: https://r4ds.hadley.nz/

Urdinez, F., & Cruz, A. (2020). R for Political Data Science: A Practical Guide. CRC Press. Cap. 2. Recurso en línea en español: https://arcruz0.github.io/libroadp/

Posit Cheatsheets ("hojas de trucos"): https://posit.co/resources/cheatsheets/?type=posit-cheatsheets/

Página oficial de Tidyverse: https://www.tidyverse.org/

Sesión 3

Manipulación de variables

04 de agosto, 2023

José D. Conejeros | jdconejeros@uc.cl | JDConejeros