



Capacitación en R y herramientas de productividad



Proyecto Estratégico Servicios Compartidos para la Producción Estadística

Procesamiento de bases de datos (3)

Abril 2021

Sesión 3: Procesamiento de BBDD (3)



- Unión de bases de datos (joins, binds) 
- Manejo de variables de tiempo (fechas) 

Unión de *data frames*



¿Para qué tareas creen que es
útil conocer sobre uniones de
bases de datos?



¿Alguna vez han tenido que unir bases de datos?

¿O en su **vida cotidiana**, han notado que en diversas situaciones ocurren uniones de bases de datos?



La mayoría de las funciones para **análisis** y **transformación** de datos que utilizaremos en este curso están diseñadas para operar sobre una tabla o *data frame*.

R incluye en sus paquetes base una función multipropósito para unir datos llamada `merge()`.

Vamos a ignorar `merge()` en esta ocasión y enfocarnos en algunas funciones de `dplyr` que tienen el mismo objetivo.

¿Por qué?

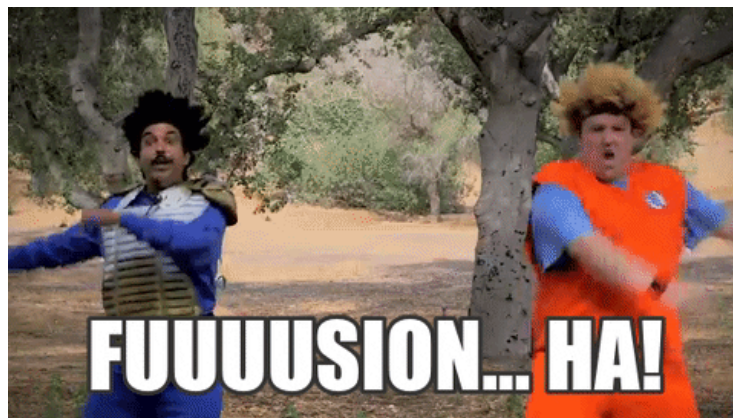
- Los **joins** de `dplyr` ofrecen más control sobre el proceso, dado que existen distintas funciones para generar uniones *ad hoc* a nuestros requerimientos.
- Los **joins** mantienen el orden de las filas (no así necesariamente `merge()`)

Unión de *data frames*

No queremos que nos vaya a pasar...



O esto...



Los **joins** son un set de funciones que son parte de **dplyr**.

Y al igual que las funciones de **dplyr** que aprendimos previamente, comparten su **gramática** y **simpleza**.

Como **intuición básica**, los joins **combinan** dos data frames agregando columnas de uno al otro.

Si bien existen **6 tipos de joins básicos**, nos concentraremos acá en los **2 que más se utilizan**.







- `left_join()`
- `inner_join()`

Que son 2 de los 4 joins denominados como **mutating** joins.

Y también veremos 2 funciones muy sencillas de **ensamblaje** de *data frames* con `dplyr`.

- `bind_rows()` (su simil en lenguaje R base es `rbind()`)
- `bind_cols()` (su simil en lenguaje R base es `cbind()`)

Entonces, **a modo de contexto**, veamos los tipos de **joins** que existen.

- `left_join()` 
- `right_join()` 
- `inner_join()` 
- `full_join()` 
- `semi_join()` 
- `anti_join()` 

Como pueden ver, **son bastantes** (y no están todos ahí 🤖🤖). Esto es porque son **muy específicos**.

Poder usar cada uno de forma precisa es todo un arte 🧠🖋️.

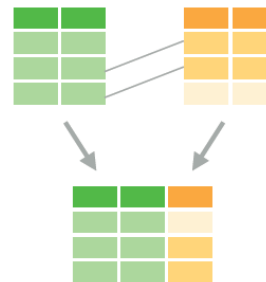
Tipos de uniones (no solo joins)

Hay *diferentes tipos de uniones* de *data frames*:

Mutating joins: para agregar columnas.

- `left_join()`
- `right_join()`
- `inner_join()`
- `full_join()`

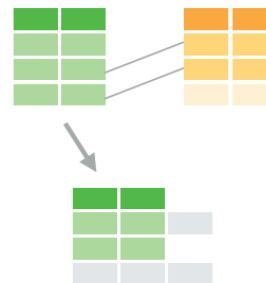
Mutating joins



Joins de filtrado: para extraer o filtrar filas.

- `semi_join()`
- `anti_join()`

Filtering joins



funciones de ensamblaje: para "pegar" *data frames*

- `bind_rows()`
- `bind_cols()`

dplyr::left_join()

Es la función de unión más básica y **más utilizada** entre los **joins**.

¿Qué hace `left_join()`?

(En adelante, llamaremos al primer *data frame* **X** y al segundo **Y**)

- Retorna **todas las filas de X** y **todas las columnas de X e Y**.
- Las filas de X que no tienen *match* en Y, tendrán **NAs** en las nuevas variables.
- Las filas de Y que no tienen *match* en X, **son ignoradas por completo**.



`right_join()` hace exactamente lo mismo pero al revés.

Veamos un ejemplo...

dplyr::left_join()

```
library(dplyr)
band_members
```

```
## # A tibble: 3 x 2
##   name band
##   <chr> <chr>
## 1 Mick  Stones
## 2 John  Beatles
## 3 Paul  Beatles
```

```
band_instruments
```

```
## # A tibble: 3 x 2
##   name plays
##   <chr> <chr>
## 1 John  guitar
## 2 Paul  bass
## 3 Keith guitar
```

```
left_join(band_members, band_instruments, by = "name")
```

```
## # A tibble: 3 x 3
##   name band plays
##   <chr> <chr> <chr>
## 1 Mick  Stones <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
```

Se puede hacer lo mismo con *pipes*.

```
band_members %>% left_join(band_instruments, by = "name")
```

¿Cuál es la llave o *key* que conecta los dos *data frames*?

Sobre las llaves (keys)

Aunque en problemas de pocas variables no es necesario indicarla (podríamos no haber escrito `by = "name"`).

La llave en general es muy importante, y sirve para **individualizar sin ambigüedades** a cada observación.

Es por eso que puede estar compuesta por una, dos, o más variables si es necesario.

Keys

primary
key

```
> names
  name  band
1 Mick  Stones
2 John  Beatles
3 Paul  Beatles
```

foreign key

```
> plays
  name plays
1 John Guitar
2 Paul  Bass
3 Keith Guitar
```

```
# Example join output
  name  band plays
1 Mick  Stones <NA>
2 John Beatles Guitar
3 Paul Beatles  Bass
4 Keith  <NA> Guitar
```

Keys

primary
key

```
> names2
  name  surname  band
1 John  Coltrane  NA
2 John  Lennon   Beatles
3 Paul  McCartney Beatles
```

foreign key

```
> plays2
  name  surname plays
1 John  Lennon  Guitar
2 Paul  McCartney Bass
3 Keith  Richards Guitar
```

```
# Example join output
  name  surname  band plays
1 John  Coltrane <NA> <NA>
2 John  Lennon  Beatles Guitar
3 Paul  McCartney Beatles Bass
4 Keith  Richards <NA> Guitar
```

base X: *primary key* / **base Y:** *foreign key*.

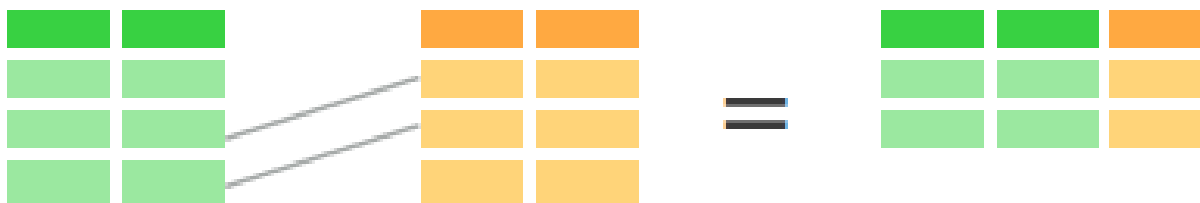
dplyr::inner_join()

Retorna **SOLO** las filas de X donde hay *match* con Y y todas las columnas de X e Y.

Las filas de X que no tienen *match* en Y, *son ignoradas por completo*.

Las filas de Y que no tienen *match* en X, *son ignoradas por completo*.

Esta función generalmente reduce filas de un *data frame*.



Veamos un ejemplo...

dplyr::inner_join()

Tomemos este código y creemos `songs` y `albums`.

```
songs <- tibble(song = c("Come Together", "Dream On", "Hello, Goodbye", "It's Not  
album = c("Abbey Road", "Aerosmith", "Magical Mystery Tour", "A  
first = c("John", "Steven", "Paul", "Tom"),  
last = c("Lennon", "Tyler", "McCartney", "Jones"))  
  
albums <- tibble( album = c("A Hard Day's Night", "Magical Mystery Tour", "Begg  
"Abbey Road", "Led Zeppelin IV", "The Dark Side of th  
"Aerosmith", "Rumours", "Hotel California"),  
band = c("The Beatles", "The Beatles", "The Rolling Stones",  
"The Beatles", "Led Zeppelin", "Pink Floyd", "Aerosmi  
"Fleetwood Mac", "Eagles"),  
year = c(1964, 1967, 1968, 1969, 1971, 1973, 1973, 1977, 1982))
```

¿Cuál es la posible llave entre estos dos *data frames*?



¿Qué pasará si hacemos un `inner_join()` sin especificar una llave?

Vamos a ver...

dplyr::inner_join()

```
songs %>% inner_join(albums)
```

```
## Joining, by = "album"
```

```
## # A tibble: 3 x 6
```

	song	album	first	last	band	year
	<chr>	<chr>	<chr>	<chr>	<chr>	<dbl>
## 1	Come Together	Abbey Road	John	Lennon	The Beatles	1969
## 2	Dream On	Aerosmith	Steven	Tyler	Aerosmith	1973
## 3	Hello, Goodbye	Magical Mystery Tour	Paul	McCartney	The Beatles	1967

En este caso **no hay ambigüedad**, por lo que R entiende cuál es la llave.

Puedo restringir la cantidad de columnas que quiero unir de cualquiera de los dos *data frames*.

```
songs[c(1,2)] %>% inner_join(albums[-3], by = "album")
```

```
## # A tibble: 3 x 3
```

	song	album	band
	<chr>	<chr>	<chr>
## 1	Come Together	Abbey Road	The Beatles
## 2	Dream On	Aerosmith	Aerosmith
## 3	Hello, Goodbye	Magical Mystery Tour	The Beatles

dplyr::inner_join()

Como pueden notar, la dificultad de los `joins` no está en el código.

Se encuentra en decidir cómo, cuándo y cuál usar.

Ejercicio express 1

Ahora ustedes...

Creemos estas dos *tibbles* de `artists` y `bands`.

```
artists <- tibble(first = c("Jimmy", "George", "Mick", "Tom", "Davy",  
                           "John", "Paul", "Jimmy", "Joe", "Elvis", "Keith",  
                           "Paul", "Ringo", "Joe", "Brian", "Nancy"),  
                 last = c("Buffett", "Harrison", "Jagger", "Jones", "Jones",  
                           "Lennon", "McCartney", "Page", "Perry", "Presley",  
                           "Richards", "Simon", "Starr", "Walsh", "Wilson", "Wils  
                 instrument = c("Guitar", "Guitar", "Vocals", "Vocals", "Vocals",  
                               "Guitar", "Bass", "Guitar", "Guitar", "Vocals",  
                               "Guitar", "Guitar", "Drums", "Guitar", "Vocals")  
  
bands <- tibble(first = c("John", "John Paul", "Jimmy", "Robert", "George", "John  
                           "Paul", "Ringo", "Jimmy", "Mick", "Keith", "Charlie",  
                           "Bonham", "Jones", "Page", "Plant", "Harrison", "Lennon",  
                           "Starr", "Buffett", "Jagger", "Richards", "Watts", "Wood  
                 band = c("Led Zeppelin", "Led Zeppelin", "Led Zeppelin", "Led Zepp  
                           "The Beatles", "The Beatles", "The Beatles", "The Beatle  
                           "The Coral Reefers", "The Rolling Stones", "The Rolling  
                           "The Rolling Stones", "The Rolling Stones"))
```

Ejercicio express 1

Utilizando las bases `artists` y `bands`.

1- Generar una base de datos que contenga *first* (nombre), *last* (apellido) y la banda (*band*), para **todos** los artistas presentes en la base `artists`.

¿Qué pueden observar sobre la base resultante?

2- Generar una base de datos, a partir *artist* y *bands*, que contenga las variables *first* (nombre), *last* (apellido), la banda (*band*) e instrumento (*instrument*), que incluya solo a los artistas que tienen información en las 4 variables.

¿Cuántas observaciones tiene la base creada?

Ejercicio express 1: solución

Ejercicio 1:

```
artists %>% left_join(bands, by = c("first", "last")) %>% head()
```

```
## # A tibble: 6 x 4
##   first last   instrument band
##   <chr> <chr>   <chr>      <chr>
## 1 Jimmy Buffett Guitar    The Coral Reefers
## 2 George Harrison Guitar    The Beatles
## 3 Mick Jagger Vocals    The Rolling Stones
## 4 Tom Jones Vocals    <NA>
## 5 Davy Jones Vocals    <NA>
## 6 John Lennon Guitar    The Beatles
```

Ejercicio 2:

```
artists %>% inner_join(bands, by = c("first", "last")) %>% head(5)
```

```
## # A tibble: 5 x 4
##   first last   instrument band
##   <chr> <chr>   <chr>      <chr>
## 1 Jimmy Buffett Guitar    The Coral Reefers
## 2 George Harrison Guitar    The Beatles
## 3 Mick Jagger Vocals    The Rolling Stones
## 4 John Lennon Guitar    The Beatles
## 5 Paul McCartney Bass      The Beatles
```

Funciones de ensamblaje

hasta ahora no se ha mencionado la forma más sencilla de unir 2 *data frames*.

Si tenemos dos *data frames* que tienen la misma estructura de columnas o de filas...

Por ejemplo: submuestras de una encuesta, distintos años de un RRAA, etc.

Podemos simplemente **ensamblarlas**.

R base trae por defecto `rbind()` para pegar filas y `cbind()` para columnas.

Recomendamos usar `bind_rows()` y `bind_cols()` de `dplyr()`.

Importante: `cbind`, pero también `bind_cols()` no usan una llave, **suponen que las filas están en el mismo orden**.

bind_rows() y bind_cols()

¿por qué preferir `bind_rows()` y `bind_cols()` en vez de `rbind()` y `cbind()`?

- Son más rápidas
- Siempre retornan una `tibble`, que es una versión mejorada de un *data frame*.
- Tienen una sintaxis más flexible.
- **Y lo más importante:** el argumento `".id"` de `bind_rows()`, permite identificar el origen de cada fila. Esto hace más fácil su tratamiento.

Generemos un ejemplo de juguete solo para ejemplificar el uso de `bind_rows`.

```
artists_2 = artists #duplicamos este data frame solo con un fin pedagógico  
ensamble <- bind_rows(original = artists, duplic = artists_2, .id= "base_datos")
```



Variables de tiempo en R

Las **fechas y horas** ⌚, como información, presentan una **gran versatilidad** para el análisis de diversos fenómenos.

En este módulo **solo veremos fechas** 📅, pero para horas, minutos y segundos la lógica es la misma.

Sin embargo, si bien es posible trabajar variables de tiempo sin herramientas dedicadas, sería **extremadamente engorroso**, y se requerirían **herramientas medianamente sofisticadas** para transformar estas variables en objetos con los que podamos operar.

Para eso **R** ofrece herramientas dedicadas especialmente a solucionarnos la vida. Podrían separarse en 2 tipos:

- Herramientas para **organizar fechas en un formato reconocible**.
- Herramientas que **traduzcan** estas fechas de formato estandarizado a **números**.

Variables de tiempo en R

Y con números se pueden hacer muchas cosas: **operaciones matemáticas**, **gráficos**, etc. Mientras R por detrás trabaja con números, nosotros seguimos viendo sencillas y amigables fechas ☺.

```
# Pueden reemplazar esta fecha por sus cumpleaños  
mi_cumple <- ("1985-06-12")  
str(mi_cumple)
```

```
## chr "1985-06-12"
```

```
mi_cumple <- as.Date(mi_cumple)  
str(mi_cumple)
```

```
## Date[1:1], format: "1985-06-12"
```

```
as.numeric(mi_cumple)
```

```
## [1] 5641
```

La función `as.Date()` recibió una cadena de caracteres y la transformó en un objeto fecha (`date`), y esa fecha aloja un número. ¿Qué representa ese número?

Variables de tiempo en R

```
as.numeric(as.Date("1970-01-01"))
```

```
## [1] 0
```

Es la distancia desde un momento **escogido de manera arbitraria**: el **1 de enero de 1970**.

`as.Date()` es una función sencilla de usar, **pero no es muy robusta para el trabajo con fechas**.

```
mi_cumple <- as.Date("12-06-1985"); str(mi_cumple)
```

```
## Date[1:1], format: "0012-06-19"
```

No soluciona el problema del ordenamiento. Requiere asistentes para hacerlo.

```
library(anytime)
```

```
## Warning: package 'anytime' was built under R version 4.0.3
```

```
mi_cumple <- as.Date(anydate("12-06-1985")); str(mi_cumple)
```

Variables de tiempo en R

Aún así, **no hay que descartarla**, es muy útil cuando el formato es la norma ISO 8661. Esta indica un formato **YYYY-MM-DD** y una cantidad de dígitos por parámetro (4-2-2).

Dentro del universo de **tidyverse** existe una **librería especializada para el tratamiento de fechas y horas**.

Se llama **lubridate** y su objetivo es hacer más intuitiva la manipulación y análisis de este tipo de variables.

Veamos algunos operadores básicos muy útiles.

R base tiene funciones para extraer la fecha y hora en el momento de la consulta.

```
Sys.Date() # La fecha de hoy
```

```
## [1] "2021-04-09"
```

```
Sys.time() # el momento exacto, con fecha, horas, minutos y segundos
```

```
## [1] "2021-04-09 22:35:33 -04"
```

Manejo de fechas con **lubridate**

lubridate tiene funciones que hacen lo mismo, pero con un lenguaje más intuitivo.

```
library(lubridate) # cargamos lubridate
today()
```

```
## [1] "2021-04-09"
```

```
now()
```

```
## [1] "2021-04-09 22:35:33 -04"
```

Hay 2 formas principales para crear una fecha.

- Desde una cadena de caracteres o numeros.
- Desde componentes *date-time* individuales.

1. Desde cadenas de caracteres

- La más habitual es a partir de cadenas de caracteres.
- Existen helpers en lubridate que automáticamente ordenan el formato de una variable fecha.
- Solo hay que ordenarlos de acuerdo al input.
- Se aceptan diferentes tipos de separadores.

```
ymd("1985-06-12")
```

```
## [1] "1985-06-12"
```

```
# No asimila bien el mes en español  
mdy("Jun 12, 1985")
```

```
## [1] "1985-06-12"
```

```
dmy("12/jun/1985")
```

```
## [1] "1985-06-12"
```

También se pueden crear fechas a partir de variables numéricas. Siempre y cuando respeten el orden y cantidad de dígitos.

```
ymd(20190322)
```

```
## [1] "2019-03-22"
```

```
dmy(22032019)
```

```
## [1] "2019-03-22"
```

2. Creación desde componentes *date-time* individuales

A veces las fechas nos llegan en un *data frame* separadas en día, mes, año.

Manejo de fechas con **lubridate**

Debemos unirlos para operarlas como objetos *date*. **Usaremos la base de nacimientos de EEVV 2017.**

Para crear un objeto *date* que llamaremos *fecha_nac* usamos la función *make_date()* de *lubridate*.

```
# cargamos la base
library(readxl)
library(lubridate)

nac2017 <- read_excel("data/nac_2017.xlsx")

# seleccionamos día, mes, año de nacimiento y creamos una fecha
nac2017 %>%
  select(dia_nac, mes_nac, ano_nac) %>%
  mutate(fecha_nac = make_date(ano_nac, mes_nac, dia_nac)) %>%
  head(5)
```

```
## # A tibble: 5 x 4
##   dia_nac mes_nac ano_nac fecha_nac
##   <dbl>   <dbl>   <dbl> <date>
## 1     27     11    2017 2017-11-27
## 2     27      1    2017 2017-01-27
## 3     21      3    2017 2017-03-21
## 4     28      6    2017 2017-06-28
## 5     10      4    2017 2017-04-10
```

Ejercicio express 2

Utilizando la base de datos "nac2017" que acabamos de cargar en nuestro entorno de trabajo.

1- Generar dentro de la base de datos (en la misma o un objeto nuevo) una variable llamada "fecha_nac" que contenga en un solo campo la fecha de nacimiento completa de cada nacido.

2- Generar además una variable llamada "fecha_ins" que contenga en un solo campo la fecha de inscripción completa de cada nacido.

Ejercicio express 2: solución

Es posible crear ambas variables en un solo paso.

```
nac2017 <- nac2017 %>%  
  mutate(fecha_nac = make_date(ano_nac, mes_nac, dia_nac),  
         fecha_ins = make_date(ano_ins, mes_ins, dia_ins))
```

```
nac2017 %>% select(ano_nac, mes_nac, dia_nac, ano_ins, mes_ins,  
                  dia_ins, fecha_nac, fecha_ins) %>%  
  head()
```

```
## # A tibble: 6 x 8  
##   ano_nac mes_nac dia_nac ano_ins mes_ins dia_ins fecha_nac fecha_ins  
##   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <date>   <date>  
## 1    2017     11     27    2017     11     30 2017-11-27 2017-11-30  
## 2    2017      1     27    2017      2      2 2017-01-27 2017-02-02  
## 3    2017      3     21    2017      3     23 2017-03-21 2017-03-23  
## 4    2017      6     28    2017      7      3 2017-06-28 2017-07-03  
## 5    2017      4     10    2017      4     13 2017-04-10 2017-04-13  
## 6    2017     10     14    2017     10     16 2017-10-14 2017-10-16
```

Manejo de fechas con lubridate

Así como podemos componer una fecha, también podemos descomponerla.

```
mi_cumple <- dmy("12-06-1985")  
year(mi_cumple)
```

```
## [1] 1985
```

```
month(mi_cumple, label = T) # con label se pide la etiqueta
```

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

```
## Levels: ene < feb < mar < abr < may < jun < jul < ago < sep < oct < nov < dic
```

```
mday(mi_cumple)
```

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

```
wday(mi_cumple) # considera que el día 1 es el domingo
```

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

Operaciones aritméticas

Por ejemplo, pueden saber **cuántos días de vida tienen**.

```
today() - ymd("1985-06-12")
```

```
## Time difference of 13085 days
```

Existe un **set de funciones** que sirven para operar sobre periodos de tiempo de una manera **intuitiva y versatil**: se llaman **periods** y algunos de ellos son:

```
days(1)
weeks(1)
months(1) # esta función es de R base
years(1)
```

¿Qué podemos hacer con ellos?

```
# ¿que fecha es en un año y un mes más?
today() + years(1) + months(1)
```

```
## [1] "2022-05-09"
```

Operaciones aritméticas

Podemos, por ejemplo, crear una variable *deadline* que indique cuándo es un mes después de un punto inicial.

```
inicio <- as.Date("2020-08-30")  
inicio + months(1)
```

```
## [1] "2020-09-30"
```

Pero no es una función tan robusta. ¿Qué pasa con los meses de 31 días?

```
inicio <- ymd("2020-08-31") # esta otra función es parecida a as.Date  
inicio + months(1)
```

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

No sabe qué hacer y entrega un **NA**. Pero `lubridate()` contiene operadores **robustos** para solucionarlo.

```
inicio %m+% months(1)
```

```
## [1] "2020-09-30"
```

`%m+%` también funciona con años y días. También existe `%m-%` para restar periodos.

```
bisiesto <- ymd("2020-02-29")
bisiesto %m+% years(1)
```

```
## [1] "2021-02-28"
```

```
bisiesto %m+% days(1)
```

```
## [1] "2020-03-01"
```

Además se pueden generar automáticamente varios periodos. Esto puede ser muy útil para validar datos 🧐

```
inicio <- ymd("2020-08-31")
inicio %m+% months(1:6)
```

```
## [1] "2020-09-30" "2020-10-31" "2020-11-30" "2020-12-31" "2021-01-31" "2021-02-28"
```

Operaciones aritméticas

También podemos calcular **intervalos de tiempo** entre dos momentos de manera consistente.

Para eso utilizamos el operador `%--%`.

```
siguiente_año <- today() + years(1)
(today() %--% siguiente_año) / days(1) # diferencia en días
```

```
## [1] 365
```

Para encontrar cuántos períodos caen dentro de un intervalo, con `%/%` pueden obtener la división entera:

```
(today() %--% siguiente_año) / weeks(1)
```

```
## [1] 52.14286
```

Ahora con `%/%`.

```
(today() %--% siguiente_año) %/% weeks(1)
```

```
## [1] 52
```

Ejercicio express 3

Vamos a utilizar la base de datos donde creamos "fecha_nac" y "fecha_ins".

La variable "fecha_nac" refiere a la **fecha de nacimiento** de un nacido durante el año estadístico 2017 y "fecha_ins" indica la **fecha en que el nacido fue inscrito**.

1- Genera una variable llamada "dif_days" que indique la **diferencia en días** entre que los nacidos nacieron y fueron inscritos.

2- Genera una variable llamada "dif_weeks" que indique la **diferencia en semanas enteras** (sin decimales) entre que los nacidos nacieron y fueron inscritos.

3- Escoje una de las dos variables creadas y **genera una tabla de resumen** que contenga la mínima diferencia, la máxima, diferencia media y la mediana.

Para resolver el punto 3 tendrán que googlear un poco



Ejercicio express 3

1- Generar "dif_days".

```
nac2017 <- nac2017 %>%  
  mutate(dif_days = (fecha_nac %--% fecha_ins) / days(1))
```

2- Generar "dif_weeks".

```
nac2017 <- nac2017 %>%  
  mutate(dif_weeks = (fecha_nac %--% fecha_ins) %/% weeks(1))
```

3- Generación tabla de resumen de estadísticos.

```
resumen_dif <- nac2017 %>% summarise(min_dif = min(dif_days),  
                                     max_dif = max(dif_days),  
                                     media_dif = mean(dif_days),  
                                     median_dif = median(dif_days))
```


Manejo de fechas con `lubridate`

El uso de `lubridate` puede generar **cierta dificultad** en un principio.

Esto debido a la cantidad de operadores nuevos que ofrece (`%--%`, `%m+%`, `%m-%`, etc.).

Pero si trabajamos habitualmente con fechas u horas, y son un aspecto importante de nuestro trabajo, vale mucho la pena estudiarlos bien.

Pues `lubridate` ofrece herramientas **precisas** y **robustas** para el trabajo con datos temporales.

Que además son absolutamente compatibles con las librerías de `tidyverse`.

Pueden encontrar muchísima más información [aquí](#).

Tarea para la

En el archivo "nacimientos.rar" que se encuentra disponible en la carpeta '**data/tarea**' en la **sesion 4 de nuestro canal en Teams**, encontrarán las siguientes bases de datos:

- **nac2017_j1.xlsx**
- **nac2017_j2.xlsx**
- **nac2017_j3.xlsx**

Estas bases de datos **fueron creadas a partir de la base de datos oficial de nacimientos del 2017.**

- 1- En primer lugar, **carga estos 3 objetos en su entorno y explórelos.** ¿Qué son y cómo se relacionan estos objetos?
- 2- Genera en **nac2017_j2** y **nac2017_j3** las variables "fecha_nac", "fecha_ins" y "dif_days", tal como lo hiciste en el ejercicio anterior.
- 3- Ahora, une **nac2017_j1** con **nac2017_j2**, conservando todos los registros de **nac2017_j1** y solo las variables "fecha_nac", "fecha_ins" y "dif_days" de **nac2017_j2**, que acabas de crear.

Tarea para la

4- ¿Qué sucedió al unir **nac2017_j1** con **nac2017_j2**? ¿Se unieron todos los registros? Si no lo hicieron, ¿por qué pasó eso?

5- Ahora **ensambla** **nac2017_j2** y **nac2017_j3** y este nuevo objeto únelo con **nac2017_j1**, conservando solo "fecha_nac", "fecha_ins" y "dif_days" del objeto ensamblado.

6- ¿Qué pasó ahora? 😊

Nada de esto sería posible sin:

- [R for Data Science](#), de Hadley Wickham
- [Advanced R](#), de Hadley Wickham
- [Data wrangling, exploration, and analysis with R](#), de Jenny Bryan
- [Introduction to R](#), de Data Carpentry
- [Xaringan: Presentation Ninja](#), de Yihui Xie. Para generar esta presentación con la plantilla ninja ✂
- [Tutorial de lubridate](#)

R for Data Science tiene una traducción al español realizada por la comunidad hispana de R:

- [R para ciencia de datos](#), de Hadley Wickham



Capacitación en R y herramientas de productividad

Proyecto Estratégico Servicios Compartidos para la Producción Estadística

Procesamiento de bases de datos (3)

Abril 2021