BSM

Blockchain School
for Management

Tema 7

Módulo II: R. Manipulación de Datos II

Nicolás Forteza

2022-11-16

Existe una librería en R que se usa para manejar de manera correcta las fechas. Se llama lubridate.

Una serie temporal es un una varible cuyos valores están indexados por un índice temporal, en vez de por una observación, es decir, es una variable que cambia con el tiempo.

En realidad, estamos acostubmrados a manejar la dimensión temporal de manera frecuente. Puede parecer fácil programarlo, pero en realidad, pregúntate estas cosas:

- Tienen todos los años 365 días?

Existen otros factores, como las zonas horarias, o huso horario.

Vamos a usar una librería que se llama nycflights13 que tiene varios datasets de usolibre con series temporales.

```
library(lubridate)
library(tidyverse)
library(nycflights13)
```

Creando fechas/horas

Hay 3 tipos de dato

- **Fecha**. Tibbles lo reconoce como `<date>`.
- Una **hora** dentro del día. Tibbles lo reconoce como `<time>`.
- Una **fecha-hora** es una fecha mas una hora: identifica un instante preciso en una fecha determinada. Tibbles lo reconoce como `<dttm>`. En el resto de R, esto se llama POSIXct, pero no es un nombre muy útil.

Aquí vamos a centrarnos en fechas y fechas-horas y R no tiene una clase nativa. Si se necesita una, se puede usar el paquete **hms**.

Para saber qué dia y qué hora es:

```
today()
```

```
[1] "2022-11-16"
```

```
now()
```

```
[1] "2022-11-16 20:53:51 CET"
```

De otro modo, hay 3 maneras de crear fechas/horas

- Caracteres.
- De componentes fecha-hora individuales.
- de objetos fecha-hora existentes.

Funcionan de este modo

Para hacer con strings, lo podemos hacer de este modo:

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

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

```
mdy("Enero 31, 2017")
```

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

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

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

Para hacerlo con números, lo podemos hacer de este modo.Devuelve un string!

```
ymd(20170131)
```

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

Para crear una fecha hora, podemos añadir los prefijos que se ven a continuación. Siempre tienen que tener el orden natural temporal.

```
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"
```

También puedes forzar el huso temporal

```
ymd(20170131, tz = "UTC")
```

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

Puedes crear fechas horas de componentes individuales ya creados.

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

Usa el comando make_date para fusionar las columnas en una fecha.

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

Podemos crear funciones para ayudarnos y usar dplyr para tener sets de datos bien transformados.

```
make_datetime_100 <- function(year, month, day, time) {
  make_datetime(year, month, day, time %/% 100, time %% 100
}
flights_dt <- flights %>%
  filter(!is.na(dep_time), !is.na(arr_time)) %>%
  mutate(
    dep_time = make_datetime_100(year, month, day, dep_time
    arr_time = make_datetime_100(year, month, day, arr_time
    sched_dep_time = make_datetime_100(year, month, day, sc
    sched_arr_time = make_datetime_100(year, month, day, sc
  ) %>%
  select(origin, dest, ends_with("delay"), ends_with("time"
```

El tercer modo consiste en convertir fechas y horas entre sí.

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

```
[1] "2022-11-16 UTC"
```

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

```
[1] "2022-11-16"
```

A veces necesitas fechas/horas en offsets numéricas desde la primera fecha existente en el sistema de R, que es 1970-01-01. Si el offset está en segundos, usa `as_datetime()`; si está en días, usa `as_date()`.

```
as_datetime(60 * 60 * 10)
```

```
[1] "1970-01-01 10:00:00 UTC"
```

```
as_date(365 * 10 + 2)
```

Ejercicio

1. Qué pasa es si parseas una fecha que contiene fechas inválidas?
2. Qué hace la función `tzone` en `today()`? por qué es importante?
3. Usa la función de lubridate adecuada para parsear las siguientes fechas:

```
d1 <- "January 1, 2010"
d2 <- "2015-Mar-07"
d3 <- "06-Jun-2017"
d4 <- c("August 19 (2015)", "July 1 (2015)")
d5 <- "12/30/14" # Dec 30, 2014
```

Componentes fecha hora

Ahora podemos ver qué podemos hacer con las fechas y las horas. Vamos a ver ahora entonces cómo operar con fechas.

Puedes devolver partes individuales de las fechas accediendo con funciones como `year()`, `month()`, `mday()` (day of the month), `yday()` (day of the year), `wday()` (day of the week), `hour()`, `minute()`, y `second()`.

Componentes fecha hora

```
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
```

Para `month()` y `wday()` puedes fijar `label = TRUE` para devolver abreviaturas.

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

```
[1] jul
12 Levels: ene < feb < mar < abr < may < jun < jul < ago <
```

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

```
[1] viernes
7 Levels: domingo < lunes < martes < miércoles < jueves <
```

Podemos usar `wday()` para comparar días de fin de semana y fías de la semana.

```
flights_dt %>%
  mutate(wday = wday(dep_time, label = TRUE))
```

Y podemos analizar patrones interesantes

```
flights_dt %>%
  mutate(minute = minute(dep_time)) %>%
  group_by(minute) %>%
  summarise(
    avg_delay = mean(dep_delay, na.rm = TRUE),
    n = n()) %>%
  ggplot(aes(minute, avg_delay)) +
    geom_line()
```

Ejercicio

Igual que en la slide anterior, fíjate cómo varía el delay en función del schedule.

Ejercicio - solución

```
sched_dep <- flights_dt %>%
  mutate(minute = minute(sched_dep_time)) %>%
  group_by(minute) %>%
  summarise(
    avg_delay = mean(arr_delay, na.rm = TRUE),
    n = n())
ggplot(sched_dep, aes(minute, avg_delay)) +
  geom_line()
```

Redondeo

Existen funciones que hacen redondeos de fechas: `floor_date()`, `round_date()`, and `ceiling_date()`, a su unidad temporal más cercana

```
flights_dt %>%
  count(week = floor_date(dep_time, "week"))
```

Puede ser interesante para comparar períodos temporales

Puedes hacerlo del siguiente modo:

```
(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"
```

De manera alternativa:

```
update(datetime, year = 2020,
       month = 2, mday = 2, hour = 2)
```

```
[1] "2020-02-02 02:34:56 UTC"
```

Y con lubridate:

```
ymd("2015-02-01") %>%
  update(mday = 30)
```

```
[1] "2015-03-02"
```

```
ymd("2015-02-01") %>%
  update(hour = 400)
```

```
[1] "2015-02-17 16:00:00 UTC"
```

Exercises

1. Cómo cambia la distribución de las horas de salida en un día en el curso de un año?
2. Compara `dep_time`, `sched_dep_time` y `dep_delay`. Son consistentes?
3. Compara `air_time` con la duración entre salida y llegada.
4. Cómo cambia el retraso medio a lo largo de un día?
5. Qué dia de la semana presenta menores delays?

Ahora ya podemos operar con fechas y horas Hay 3 conceptos que hay que saber:

- **duraciones**, que son segundos.
- **períodos**, que representan unidades temporales, como semana o mes.
- **intervalos**, que representan una diferencia entre un principio y un fin.

Puedes restar fechas

```
h_age <- today() - ymd(19791014)
h_age
```

```
Time difference of 15739 days
```

Este objeto devuelve una diferencia en segundos, días, horas ...

Lubridate provee lo que necesitamos: **duration**.

```
as.duration(h_age)
```

```
[1] "1359849600s (~43.09 years)"
```

Tienen muchos constructores:

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

```
dweeks(3)
```

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

Puedes operar tal que así:

```
2 * dyears(1)
```

```
[1] "63115200s (~2 years)"
```

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

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

O así:

```
tomorrow <- today() + ddays(1)
last_year <- today() - dyears(1)
```

Lubridate provee de **periods**. Son spans temporales donde no hay un número fijo de segundos, pero funcionan con unidades temporales que el ser humano ha creado, como días o meses.

```
one_pm <- ymd_hms("2016-03-12 13:00:00", tz = "Europe/Madr
one_pm
```

```
[1] "2016-03-12 13:00:00 CET"
```

```
one_pm + days(1)
```

```
[1] "2016-03-13 13:00:00 CET"
```

Como en las duraciones, tenemos funciones muy fáciles de usar:

```
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"
```

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

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

Puedes operar también con ellos:

```
10 * (months(6) + days(1))
```

```
[1] "60m 10d 0H 0M 0S"
```

```
days(50) + hours(25) + minutes(2)
```

```
[1] "50d 25H 2M 0S"
```

y añadirlos a fechas.

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

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

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

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

```
one_pm + ddays(1)
```

```
[1] "2016-03-13 13:00:00 CET"
```

```
one_pm + days(1)
```

```
[1] "2016-03-13 13:00:00 CET"
```

Lubridate nos informa sobre los intervalos de este modo:

```
years(1) / days(1)
```

```
[1] 365.25
```

Si queremos hacerlo de forma más precisa, debemos usar **interval**.

```
next_year <- today() + years(1)
(today() %--% next_year) / ddays(1)
```

```
[1] 365
```

Exercises

1. Crea un vector de fechas devolviendo el primer día de cada mes de 2015.
2. Escribe una función que devuelve cuántos años tienes, en función del día que es hoy.

Zonas horarios

El primer challenge, es que dependiendo de donde vengas, estás acostumbradoa uno u otro. Por ejemplo, si uno es estadounidense, está acostumbrado a la zona horarioa EST, o Eastern Standard Time. Sin embargo, Canadá y Australia también tienen EST! Para evitar la confusión, R usa este tipado, homogéneo en cualquier parte del mundo: "America/New_York", "Europe/Paris", o "Pacific/Auckland".

Con este comando puedes saber qué opina R de tu zona horaria:

```
Sys.timezone()
```

```
[1] "Europe/Madrid"
```

O puedes ver todas las listas aquí:

```
length(OlsonNames())
```

```
[1] 596
```

```
head(OlsonNames())
```

```
[1] "Africa/Abidjan"     "Africa/Accra"       "Africa/Addis
[4] "Africa/Algiers"     "Africa/Asmara"      "Africa/Asmer
```

Puedes operar con ellos con normalidad.

```
(x1 <- ymd_hms("2015-06-01 12:00:00", tz = "America/New_Yo
```

```
[1] "2015-06-01 12:00:00 EDT"
```

```
(x2 <- ymd_hms("2015-06-01 18:00:00", tz = "Europe/Copenha
```

```
[1] "2015-06-01 18:00:00 CEST"
```

```
(x3 <- ymd_hms("2015-06-02 04:00:00", tz = "Pacific/Auckla
```

```
[1] "2015-06-02 04:00:00 NZST"
```

Puedes verificar que es lo mismo haciendo la resta

```
x1 - x2
```

```
Time difference of 0 secs
```

```
x1 - x3
```

```
Time difference of 0 secs
```

lubridate usa siempre UTC. **UTC (Coordinated Universal Time) is the standard time zone used by the scientific community and roughly equivalent to its predecessor GMT (Greenwich Mean Time).**

Cuando juntas fechas de diferentes zonas en un vector, se te transforman aj uso horario local.

```
x4 <- c(x1, x2, x3)
x4
```

```
[1] "2015-06-01 12:00:00 EDT" "2015-06-01 12:00:00 EDT"
[3] "2015-06-01 12:00:00 EDT"
```

Purrr es la librería de Tidyverse para aplicar funciones a nuestros Tibbles.

Básicamente, Purrr funciona con una función que se llama *map*, que vendría a ser el *apply* de los tibbles.

Hay que recordar primero los elementos básicos de R: vectores, dataframes, listas.

La función `map` aplica una función a cada elemento de un objeto. Existen muchas variantes:

- `map(.x, .f)` is the main mapping function and returns a list
- `map_df(.x, .f)` returns a data frame
- `map_dbl(.x, .f)` returns a numeric (double) vector
- `map_chr(.x, .f)` returns a character vector
- `map_lgl(.x, .f)` returns a logical vector

Purrr

El input de estas funciones puede ser un vector, dataframe o lista.

También lo podemos hacer con dataframes, números, strings ...

```
map(data.frame(a = 1, b = 4, c = 7), addTen)
map_dbl(c(1, 4, 7), addTen)
map_chr(c(1, 4, 7), addTen)
```

Estas sentencias devuelven diferentes cosas!

Para devolver dataframes, más “fancy”, podemos hacerlo de este modo:

```
map_df(c(1, 4, 7), function(.x) {
  return(data.frame(old_number = .x,
                    new_number = addTen(.x)))
})
```

Una función muy útil es `modify`. Te devuelve el mismo tipo de input.

```
modify(c(1, 4, 7), addTen)
```

Y además tiene unos primos-hermanos que ya conocemos:

```
modify_if(.x = list(1, 4, 7),
          .p = function(x) x > 5,
          .f = addTen)
```

Puedes usar funciones de manera “anónima”, del siguiente modo:

```
map_dbl(c(1, 4, 7), ~{.x + 10})
```

Vamos ahora a ver ejemplos con tibbles. Primero hay que descargar estos datos:

```
gapminder_orig <- read.csv("https://raw.githubusercontent.c
gapminder <- gapminder_orig
gapminder %>% map_chr(class)
```

```
    country        year         pop   continent     lifeExp
"character"   "integer"   "numeric" "character"   "numeric"
```

Hemos visto cuál es el tipo de cada columna!

Otros ejemplos:

```
gapminder %>% map_dbl(n_distinct)
```

```
  country      year       pop continent   lifeExp gdpPercap
      142        12      1704         5      1626      1704
```

```
gapminder %>% map_df(~(data.frame(n_distinct = n_distinct(.
                                  class = class(.x)))))
```

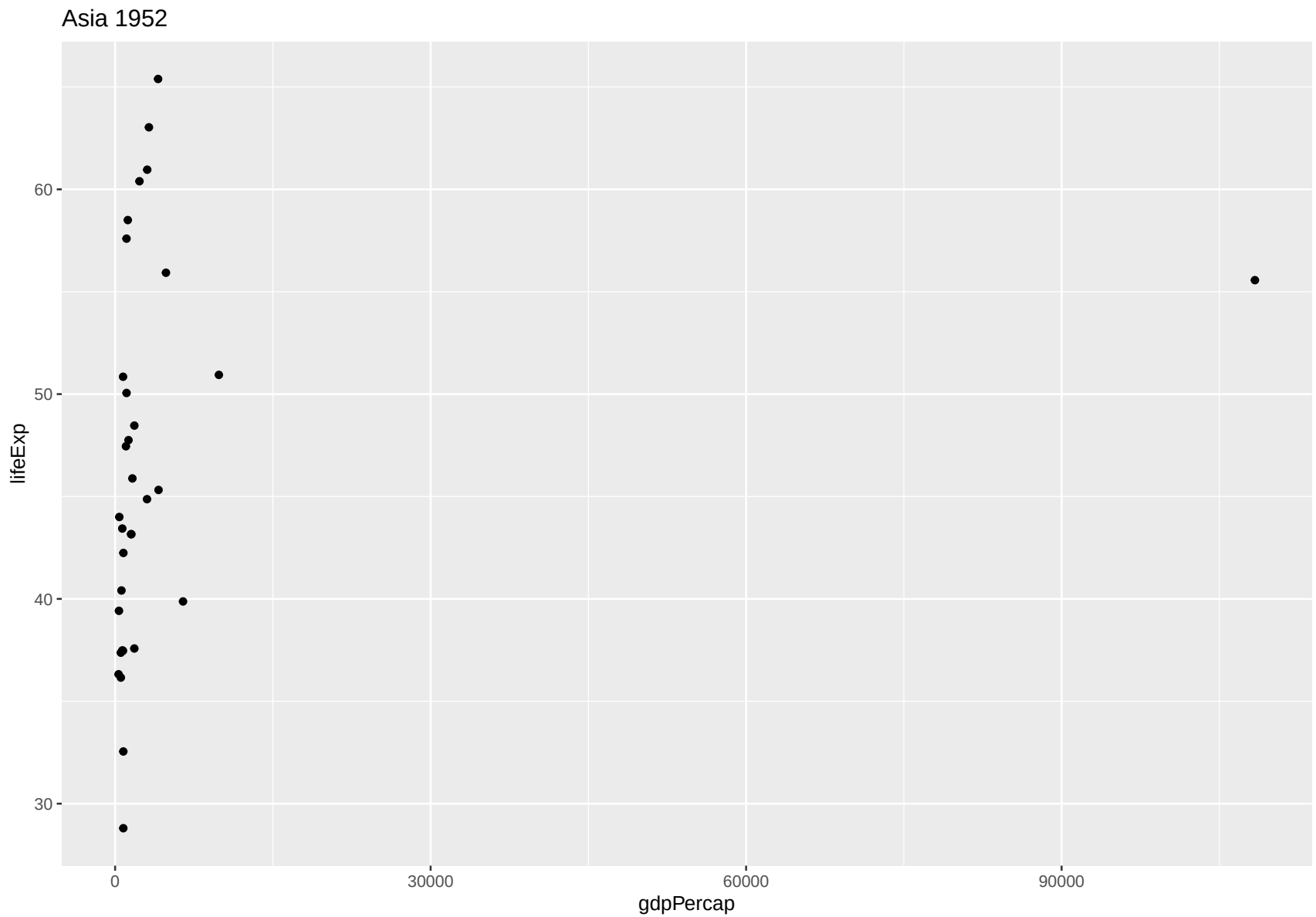
```
  n_distinct     class
1        142 character
2         12   integer
3       1704   numeric
4          5 character
5       1626   numeric
6       1704   numeric
```

```
gapminder %>% map_df(~(data.frame(n_distinct = n_distinct(.
                                  class = class(.x))),
```

Podemos iterar de manera doble con `map2`

```
continents <- gapminder %>% distinct(continent, year)%>% pu
years <- gapminder %>% distinct(continent, year)%>% pull(ye
plot_list <- map2(.x = continents,
                  .y = years,
                  .f = ~{
                    gapminder %>%
                      filter(continent == .x,
                             year == .y) %>%
                      ggplot() +
                      geom_point(aes(x = gdpPercap, y = li
                      ggtitle(glue::glue(.x, " ", .y))
                  })
```

```
plot_list[[1]]
```



Se puede tambieén hacer operaciones “nesteadas”.

```
gapminder_nested <- gapminder %>%
  group_by(continent) %>%
  nest()
gapminder_nested
```

```
# A tibble: 5 x 2
# Groups:   continent [5]
  continent data
  <chr>     <list>
1 Asia      <tibble [396 x 5]>
2 Europe    <tibble [360 x 5]>
3 Africa    <tibble [624 x 5]>
4 Americas  <tibble [300 x 5]>
5 Oceania   <tibble [24 x 5]>
```

¿Qué hay dentro de estas filas?

```
gapminder_nested$data[[1]]
```

```
# A tibble: 396 x 5
   country      year      pop lifeExp gdpPercap
   <chr>       <int>    <dbl>   <dbl>     <dbl>
 1 Afghanistan  1952  8425333    28.8      779.
 2 Afghanistan  1957  9240934    30.3      821.
 3 Afghanistan  1962 10267083    32.0      853.
 4 Afghanistan  1967 11537966    34.0      836.
 5 Afghanistan  1972 13079460    36.1      740.
 6 Afghanistan  1977 14880372    38.4      786.
 7 Afghanistan  1982 12881816    39.9      978.
 8 Afghanistan  1987 13867957    40.8      852.
 9 Afghanistan  1992 16317921    41.7      649.
10 Afghanistan  1997 22227415    41.8      635.
# ... with 386 more rows
```

Con la función Pluck lo podemos sacar de su fila

```
gapminder_nested %>%
  pluck("data", 1)
```

```
# A tibble: 396 x 5
   country      year      pop lifeExp gdpPercap
   <chr>       <int>    <dbl>   <dbl>     <dbl>
 1 Afghanistan  1952  8425333    28.8      779.
 2 Afghanistan  1957  9240934    30.3      821.
 3 Afghanistan  1962 10267083    32.0      853.
 4 Afghanistan  1967 11537966    34.0      836.
 5 Afghanistan  1972 13079460    36.1      740.
 6 Afghanistan  1977 14880372    38.4      786.
 7 Afghanistan  1982 12881816    39.9      978.
 8 Afghanistan  1987 13867957    40.8      852.
 9 Afghanistan  1992 16317921    41.7      649.
10 Afghanistan  1997 22227415    41.8      635.
# ... with 386 more rows
```

Ahora entonces podemos hacer operaciones más complejas cuando el dataframe está nesteado:

```
gapminder_nested %>%
  mutate(avg_lifeExp = map_dbl(data, ~{mean(.x$lifeExp)}))
```

```
# A tibble: 5 x 3
# Groups:   continent [5]
  continent data                avg_lifeExp
  <chr>     <list>                    <dbl>
1 Asia      <tibble [396 x 5]>         60.1
2 Europe    <tibble [360 x 5]>         71.9
3 Africa    <tibble [624 x 5]>         48.9
4 Americas  <tibble [300 x 5]>         64.7
5 Oceania   <tibble [24 x 5]>          74.3
```