

Menos for, más purrr: Programación funcional con R



Nacho Evangelista
evangelistaignacio@gmail.com

Grupo de Usuarios de R en Rosario

18 de febrero de 2021

6 Referencias

- Buscamos evitar duplicación de código. La duplicación hace que los errores y *bugs* sean más frecuentes. También se hace más difícil modificar el código.
- Un estilo funcional implica descomponer un problema grande en partes y resolver cada una de estas partes con una función o combinación de funciones.
- La idea es arrancar con porciones de código pequeños y fáciles de entender (funciones). Combinar estos bloques en estructuras más complejas.
- El estilo funcional permite obtener soluciones eficientes y elegantes a problemas cotidianos (preprocesamiento, modelización, visualización)

Motivación

- Buscamos evitar duplicación de código. La duplicación hace que los errores y *bugs* sean más frecuentes. También se hace más difícil modificar el código.
- Un estilo funcional implica descomponer un problema grande en partes y resolver cada una de estas partes con una función o combinación de funciones.
- La idea es arrancar con porciones de código pequeños y fáciles de entender (funciones). Combinar estos bloques en estructuras más complejas.
- El estilo funcional permite obtener soluciones eficientes y elegantes a problemas cotidianos (preprocesamiento, modelización, visualización)

Motivación

- Buscamos evitar duplicación de código. La duplicación hace que los errores y *bugs* sean más frecuentes. También se hace más difícil modificar el código.
- Un estilo funcional implica descomponer un problema grande en partes y resolver cada una de estas partes con una función o combinación de funciones.
- La idea es arrancar con porciones de código pequeños y fáciles de entender (funciones). Combinar estos bloques en estructuras más complejas.
- El estilo funcional permite obtener soluciones eficientes y elegantes a problemas cotidianos (preprocesamiento, modelización, visualización)

Motivación

- Buscamos evitar duplicación de código. La duplicación hace que los errores y *bugs* sean más frecuentes. También se hace más difícil modificar el código.
- Un estilo funcional implica descomponer un problema grande en partes y resolver cada una de estas partes con una función o combinación de funciones.
- La idea es arrancar con porciones de código pequeños y fáciles de entender (funciones). Combinar estos bloques en estructuras más complejas.
- El estilo funcional permite obtener soluciones eficientes y elegantes a problemas cotidianos (preprocesamiento, modelización, visualización)

- Nuestro objetivo es reemplazar los *for loops* o estructuras de repetición.
- En lugar de eso, trataremos de trabajar con operaciones vectorizadas.
- Por ejemplo: en una estructura funcional, no es necesario crear una lista vacía para ir guardando resultados ni ir llevando control de un índice. El código es más conciso.
- En R base existen las funciones de la familia `apply`. En el paquete `purrr`, estas funciones se reemplazan por la familia `map`, más fáciles de usar. Se llaman funcionales: reciben una función como argumento.

- Nuestro objetivo es reemplazar los *for loops* o estructuras de repetición.
- En lugar de eso, trataremos de trabajar con operaciones vectorizadas.
- Por ejemplo: en una estructura funcional, no es necesario crear una lista vacía para ir guardando resultados ni ir llevando control de un índice. El código es más conciso.
- En R base existen las funciones de la familia `apply`. En el paquete `purrr`, estas funciones se reemplazan por la familia `map`, más fáciles de usar. Se llaman funcionales: reciben una función como argumento.

- Nuestro objetivo es reemplazar los *for loops* o estructuras de repetición.
- En lugar de eso, trataremos de trabajar con operaciones vectorizadas.
- Por ejemplo: en una estructura funcional, no es necesario crear una lista vacía para ir guardando resultados ni ir llevando control de un índice. El código es más conciso.
- En R base existen las funciones de la familia `apply`. En el paquete `purrr`, estas funciones se reemplazan por la familia `map`, más fáciles de usar. Se llaman funcionales: reciben una función como argumento.

- Un vector es un objeto que guarda elementos individuales del mismo tipo
- Un dataframe es una estructura que guarda varios vectores de la misma longitud pero de distinto tipo.
- Una lista es una estructura que permite guardar objetos de distinto tipo y longitud.

¿Cómo se definen?

```
animalitos <- list(
  animales = c("perro", "gato", "elefante", "vaca"),
  acciones = c("ladrar", "maullar", "barritar", "mugir"),
  nombres = list("Dalmata" = "Pongo", "Marley"),
  5)
```

List of 4

¿Cómo se definen?

```
animalitos <- list(
  animales = c("perro", "gato", "elefante", "vaca"),
  acciones = c("ladrar", "maullar", "barritar", "mugir"),
  nombres = list("Dalmata" = "Pongo", "Marley"),
  5)
```

```
str(animalitos)
```

List of 4

```
$ animales: chr [1:4] "perro" "gato" "elefante" "vaca"
$ acciones: chr [1:4] "ladrar" "maullar" "barritar" "mugir"
$ nombres :List of 2
  ..$ Dalmata: chr "Pongo"
  ..$       : chr "Marley"
$          : num 5
```

```
names(animalitos)
```

```
[1] "animales" "acciones" "nombres"  ""
```

¿Cómo se definen?

```
animalitos <- list(
  animales = c("perro", "gato", "elefante", "vaca"),
  acciones = c("ladrar", "maullar", "barritar", "mugir"),
  nombres = list("Dalmata" = "Pongo", "Marley"),
  5)
```

```
str(animais)
```

List of 4

```
$ animales: chr [1:4] "perro" "gato" "elefante" "vaca"
$ acciones: chr [1:4] "ladrar" "maullar" "barritar" "mugir"
$ nombres :List of 2
..$ Dalmata: chr "Pongo"
..$       : chr "Marley"
$       : num 5
```

```
names(animaitos)
```

```
[1] "animales" "acciones" "nombres" ""
```

¿Cómo se accede a sus elementos?

```
animalitos[["animales"]]
```

```
[1] "perro"      "gato"      "elefante"  "vaca"
```

\$Dalmata

[1] "Pongo"

[[2]]

[1] "Marley"

¿Cómo se accede a sus elementos?

```
animalitos[["animales"]]
```

```
[1] "perro"      "gato"      "elefante"  "vaca"
```

```
animalitos$acciones
```

```
[1] "ladrar"    "maullar"   "barritar"  "mugir"
```

\$Dalmata

[1] "Pongo"

[[2]]

[1] "Marley"

¿Cómo se accede a sus elementos?

```
animalitos[["animales"]]
```

```
[1] "perro"      "gato"      "elefante"  "vaca"
```

```
animalitos$acciones
```

```
[1] "ladrar"    "maullar"   "barritar"  "mugir"
```

```
animalitos$"nombres"
```

\$Dalmata

[1] "Pongo"

[[2]]

```
[1] "Marley"
```

```
animalitos[[1]]
```

```
[1] "perro"    "gato"     "elefante" "vaca"
```

```
animalitos[[4]]
```

```
[1] 5
```

```
animalitos[[1]]
```

```
[1] "perro"    "gato"     "elefante" "vaca"
```

```
animalitos[[4]]
```

```
[1] 5
```


El paquete purrr

- Muchas operaciones de R funcionan en forma vectorizada; cuando se aplican a vectores, se ejecutan elemento a elemento (una suerte de iteración)

```
v ← c(2,5,7)
exp(v)
```

```
[1] 7.389056 148.413159 1096.633158
```

- Muchas funciones no tienen esa capacidad

```
meses ← c("Ene", "Feb", "Mar", "Abr", "May", "Jun",  
          "Jul", "Ago", "Sep", "Oct", "Nov", "Dic")  
which(meses=="Sep")  
which(meses=c("Sep", "Ene"))
```

[1] 9

[1] 9

El paquete `purrr` sirve para vectorizar operaciones y por lo tanto, para iterar

La función map

- La función (funcional) más básica de `purrr` es `map`:

- 1 Toma un vector y una función,
- 2 aplica la función a cada elemento del vector,
- 3 devuelve los resultados en una lista.

```
map( .x, .f, ... )
```

La función map

- La función (funcional) más básica de `purrr` es `map`:

- 1 Toma un vector y una función,
- 2 aplica la función a cada elemento del vector,
- 3 devuelve los resultados en una lista.

```
map(.x, .f, ...)
```


La función map

- La función (funcional) más básica de `purrr` es `map`:
 - 1 Toma un vector y una función,
 - 2 aplica la función a cada elemento del vector,
 - 3 devuelve los resultados en una lista.

```
map(.x, .f, ...)
```

La función map

- La función (funcional) más básica de `purrr` es `map`:

- 1 Toma un vector y una función,
- 2 aplica la función a cada elemento del vector,
- 3 devuelve los resultados en una lista.

```
map(.x, .f, ...)
```

La función map

- La función (funcional) más básica de `purrr` es `map`:
 - 1 Toma un vector y una función,
 - 2 aplica la función a cada elemento del vector,
 - 3 devuelve los resultados en una lista.

Función map

```
map(.x, .f, ...)
```

- `.x` es la lista (o el vector) sobre la que aplicaremos la función,
- `.f` es la función en cuestión,
- `...` son argumentos adicionales de `.f`

La función map

- La función (funcional) más básica de `purrr` es `map`:

- 1 Toma un vector y una función,
- 2 aplica la función a cada elemento del vector,
- 3 devuelve los resultados en una lista.

Función map

```
map(.x, .f, ...)
```

- `.x` es la lista (o el vector) sobre la que aplicaremos la función,
- `.f` es la función en cuestión,
- `...` son argumentos adicionales de `.f`

- `map(1:3,f)` es equivalente a `list(f(1), f(2), f(3))`

Ejemplos

Apliquemos algunas funciones a nuestra lista de animalitos

```
map(animales,typeof)
```

```
$animales
[1] "character"
```

```
$acciones
[1] "character"
```

```
$nombres
[1] "list"
```

```
[[4]]
[1] "double"
```


También podemos usar `purrr` con vectores

```
v ← c(1,3)
map(v, rnorm)
```

```
[[1]]
[1] -0.3077209
```

```
[[2]]
[1] -0.69040612 -0.53197638  0.09848098
```

```
# agregamos parámetros adicionales de la función rnorm
v <- c(1,5)
map(v,rnorm,mean=4,sd=2)
```

```
[[1]]
[1] 5.42703
```

```
[[2]]
[1] 2.706652 6.992548 6.430643 4.013287 6.662737
```


Especificando el tipo de salida

Si la función que aplicamos devuelve un único elemento, podemos usar las variantes `map_lgl`, `map_int`, `map_dbl` o `map_chr` y obtener un vector como resultado

```
map_chr(animales,typeof)
```

animales	acciones	nombres	
"character"	"character"	"list"	"double"

Especificando el tipo de salida

Si la función que aplicamos devuelve un único elemento, podemos usar las variantes `map_lgl`, `map_int`, `map_dbl` o `map_chr` y obtener un vector como resultado

```
map_chr(animales,typeof)
```

animales	acciones	nombres	
"character"	"character"	"list"	"double"

```
map_int(animales, length)
```

```
animales acciones nombres
      4         4         2         1
```

Funciones propias

Además de usar funciones predefinidas, podemos definir nuestras propias funciones

```
sumar_diez ← function(x) return(x + 10)
map_dbl(c(-10,4,7), sumar_diez)
```

```
[1] 0 14 17
```

```
[1] 0 14 17
```

```
[1] 0 14 17
```

Funciones propias

Además de usar funciones predefinidas, podemos definir nuestras propias funciones

```
sumar_diez ← function(x) return(x + 10)
map_dbl(c(-10,4,7), sumar_diez)
```

[1] 0 14 17

```
map_dbl(c(-10,4,7), function(x) return(x + 10))
```

[1] 0 14 17

```
[1] 0 14 17
```

```
sumar_diez ← function(x) return(x + 10)
map_dbl(c(-10,4,7), sumar_diez)
```

$$[1] \quad 0 \ 14 \ 17$$

```
map_dbl(c(-10,4,7), function(x) return(x + 10))
```

[1] 0 14 17

```
# función anónima (el argumento es siempre .x)
map_dbl(c(-10,4,7), ~ .x + 10)
```

[1] 0 14 17

La definición de una función anónima permite ser más explícito en el pasaje de parámetros

```
v <- c(1,5)
map(v, ~ rnorm(.x,mean=4,sd=2)) # el argumento es siempre .x!
```

[[1]]

[1] 2.368587

[[2]]

```
[1] 5.235575 7.642995 7.042945 4.588525 4.049139
```

Comparar con

También podemos hacer

[[1]]

[1] 2.2987216 0.9204782

La definición de una función anónima permite ser más explícito en el pasaje de parámetros

```
v ← c(1,5)
map(v, ~ rnorm(.x,mean=4,sd=2)) # el argumento es siempre .x!
```

[[1]]

[1] 2.368587

[[2]]

```
[1] 5.235575 7.642995 7.042945 4.588525 4.049139
```

Comparar con

```
v ← c(1,5)
map(v, rnorm, mean=4, sd=2)
```

También podemos hacer

[[1]]

[1] 2.2987216 0.9204782

La definición de una función anónima permite ser más explícito en el pasaje de parámetros

```
v ← c(1,5)
map(v, ~ rnorm(.x,mean=4,sd=2)) # el argumento es siempre .x!
```

[[1]]

[1] 2.368587

[[2]]

```
[1] 5.235575 7.642995 7.042945 4.588525 4.049139
```

Comparar con

```
v ← c(1,5)
map(v, rnorm, mean=4, sd=2)
```

También podemos hacer

```
v ← c(1,8)
map(v, ~ rnorm(n=2,mean=.x,sd=1)) # el argumento es siempre .x!
```

[[1]]

```
[1] 2.2987216 0.9204782
```


[1] "2019-03-06" "2019-03-07" "2019-03-08"

Pero yo nunca usé listas...

```
pelis <- tibble(
  cancion = c("Strange Things", "Life is a Highway", "I'm a Believer"),
  autor = c("Randy Newman", "Rascal Flatts", "Smash Mouth"),
  pelicula = c("Toy Story", "Cars", NA)
)
```

```
[1] "list"
```

Pero yo nunca usé listas...

```
pelis <- tibble(
  cancion = c("Strange Things", "Life is a Highway", "I'm a Believer"),
  autor = c("Randy Newman", "Rascal Flatts", "Smash Mouth"),
  pelicula = c("Toy Story", "Cars", NA)
)
```

```
typeof(pelis)
```

```
[1] "list"
```



```
map_chr(pelis,typeof)
```

```

cancion      autor      pelicula
"character"  "character" "character"

```

```
map_int(pelis, ~ sum(is.na(.x)))
```

cancion	autor	pelicula
0	0	1

- Las listas son estructuras de datos muy versátiles
- Un dataframe/tibble es una lista
- purrr sirve para iterar
- map es el caballito de batalla del paquete purrr
- Se puede especificar el tipo de salida con las variantes map_*
- Para iterar a lo largo de dos listas se usa map2
- Hay varias formas de pasar una función como argumento de map

- Las listas son estructuras de datos muy versátiles
- Un dataframe/tibble es una lista
- purrr sirve para iterar
- map es el caballito de batalla del paquete purrr
- Se puede especificar el tipo de salida con las variantes map_*
- Para iterar a lo largo de dos listas se usa map2
- Hay varias formas de pasar una función como argumento de map

- Las listas son estructuras de datos muy versátiles
- Un dataframe/tibble es una lista
- purrr sirve para iterar
- map es el caballito de batalla del paquete purrr
- Se puede especificar el tipo de salida con las variantes map_*
- Para iterar a lo largo de dos listas se usa map2
- Hay varias formas de pasar una función como argumento de map

- Las listas son estructuras de datos muy versátiles
- Un dataframe/tibble es una lista
- purrr sirve para iterar
- map es el caballito de batalla del paquete purrr
- Se puede especificar el tipo de salida con las variantes map_*
- Para iterar a lo largo de dos listas se usa map2
- Hay varias formas de pasar una función como argumento de map

- Las listas son estructuras de datos muy versátiles
- Un dataframe/tibble es una lista
- purrr sirve para iterar
- map es el caballito de batalla del paquete purrr
- Se puede especificar el tipo de salida con las variantes map_*
- Para iterar a lo largo de dos listas se usa map2
- Hay varias formas de pasar una función como argumento de map


```
animalitos <- list(animales = c("perro", "gato", "elefante", "oveja"),
  acciones = c("ladrar", "maullar", "barritar", "balar"),
  nombres = list(c("Pongo", "Marley", "Golfo"),
    c("Pelusa", "Tom"),
    c("Tantor", "Dumbo"),
    c("Oscar")))

as_tibble(animalitos)
```

```
# A tibble: 4 x 3
  animales acciones nombres
  <chr>      <chr>      <list>
1 perro     ladrar      <chr [3]>
2 gato      maullar     <chr [2]>
3 elefante  barritar   <chr [2]>
4 oveja     balar       <chr [1]>
```




```
mtcars %>%  
  nest(datos = !c(gear,carb))
```

```
# A tibble: 11 x 3
```

```
  gear carb datos
```

```
  <dbl> <dbl> <list>
```

```
1     4     4 <tibble [4 x 9]>
```

```
2     4     1 <tibble [4 x 9]>
```

```
3     3     1 <tibble [3 x 9]>
```

```
4     3     2 <tibble [4 x 9]>
```

```
5     3     4 <tibble [5 x 9]>
```

```
6     4     2 <tibble [4 x 9]>
```

```
7     3     3 <tibble [3 x 9]>
```

```
8     5     2 <tibble [2 x 9]>
```

```
9     5     4 <tibble [1 x 9]>
```

```
10    5     6 <tibble [1 x 9]>
```

```
# ... with 1 more row
```

III. Como resultado de una operación

Queremos unir estas dos tablas reemplazando los códigos del casting por los personajes.

```
peleas <- tibble::tribble(
  ~pelea, ~horario, ~casting,
  1, "20:30", "gsf901,fez195,yfm179",
  2, "20:50", "thf028,yfm179",
  3, "19:40", "jfa348,fez195,gky651,wpvx281",
  4, "21:00", "thf028,fez195")
```



```
peleas %>%
  mutate(casting_split = strsplit(casting, split = ",")) %>%
  select(-horario,-casting) %>%
  unnest(casting_split) %>%
  left_join(luchadores, by = c("casting_split" = "codigo"))
```

```
# A tibble: 11 x 3
```

```

pelea casting_split nombre
<dbl> <chr>          <chr>
1      1 gsf901        Vicente Viloni
2      1 fez195        Fulgencio Mejía
3      1 yfm179        Steve Murphy
4      2 thf028        Hip Hop Man
5      2 yfm179        Steve Murphy
6      3 jfa348        Mc Floyd
7      3 fez195        Fulgencio Mejía
8      3 gky651        Rulo Verde
9      3 wpx281        La Masa
10     4 thf028        Hip Hop Man
# ... with 1 more row

```

```
peleas %>%
  mutate(casting_split = strsplit(casting, split = ",")) %>%
  select(-horario,-casting)
```

```
# A tibble: 4 x 2
  pelea casting_split
<dbl> <list>
1     1 <chr [3]>
2     2 <chr [2]>
3     3 <chr [4]>
4     4 <chr [2]>
```

mutate + purrr

¿Y ahora qué?

Con las columnas lista se abre ante nosotros un universo de posibilidades y **purrr** es la herramienta ideal para explorarlo.

mutate + purrr

¿Y ahora qué?

Con las columnas lista se abre ante nosotros un universo de posibilidades y purrr es la herramienta ideal para explorarlo.

mutate + purrr

Aplicar map a un tibble realiza operaciones por columna. Con mutate + map hacemos magia por filas.

1. Identificar el mes

Queremos obtener el número de mes a partir de la abreviatura

```
datos <- tibble::tribble(
  ~id, ~dia, ~mes, ~año,
  1, 15, "Sep", 2019,
  2, 6, "oct", 2021,
  3, 3, "Ene", 2020,
  4, 31, "dic", 2019)
```


Idea

- 1 Armar un vector con las abreviaturas de los meses
- 2 Usar la función `which` junto con `map`

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

```
which(meses="Sep")
```

[1] 9

```
which(meses=c("Sep","Ene"))
```

[1] 9

```
datos %>%
  mutate(mes_n = map(mes, ~ which(meses==.x)))
```

```
# A tibble: 4 x 5
```

```

      id      dia mes      año mes_n
<dbl> <dbl> <chr> <dbl> <list>
1      1      15 Sep      2019 <int [1]>
2      2       6 oct      2021 <int [0]>
3      3       3 Ene      2020 <int [1]>
4      4      31 dic      2019 <int [0]>

```

```
datos %>%
  mutate(mes_n = map(mes, ~ which(toupper(meses)=toupper(.x))))
```

```
# A tibble: 4 x 5
```

	id	dia	mes	año	mes_n
	<dbl>	<dbl>	<chr>	<dbl>	<list>
1	1	15	Sep	2019	<int [1]>
2	2	6	oct	2021	<int [1]>
3	3	3	Ene	2020	<int [1]>
4	4	31	dic	2019	<int [1]>

- 1 Determinar la primera y última fecha de cada grupo
- 2 Generar una secuencia de fechas (seq.Date) para cada grupo y construir una tabla con todas las fechas
- 3 Unir esta tabla con la original

```
fechas_todas <-  
  datos %>%  
  group_by(empresa, producto) %>%  
  summarise(fecha_inicial = min(fecha),  
            fecha_final = max(fecha)) %>%  
  mutate(fechas = map2(fecha_inicial,  
                       fecha_final,  
                       ~ seq.Date(.x,.y,by="1 day"))) %>%  
  select(-fecha_inicial,-fecha_final) %>%  
  ungroup()
```

```
datos %>%
  group_by(empresa, producto) %>%
  summarise(fecha_inicial = min(fecha),
            fecha_final = max(fecha))
```

```
# A tibble: 3 x 4
```

```
# Groups: empresa [2]
```

	empresa	producto	fecha_inicial	fecha_final
	<chr>	<chr>	<date>	<date>
1	A	A1	2018-06-02	2018-07-13
2	A	A2	2018-05-01	2018-05-04
3	B	B1	2018-07-01	2018-07-11


```
datos %>%
  group_by(empresa, producto) %>%
  summarise(fecha_inicial = min(fecha),
            fecha_final = max(fecha)) %>%
  mutate(fechas = map2(fecha_inicial,
                       fecha_final,
                       ~ seq.Date(.x,.y,by="1 day"))) %>%
  ungroup()
```

```
# A tibble: 3 x 5
```

	empresa	producto	fecha_inicial	fecha_final	fechas
	<chr>	<chr>	<date>	<date>	<list>
1	A	A1	2018-06-02	2018-07-13	<date [42]>
2	A	A2	2018-05-01	2018-05-04	<date [4]>
3	B	B1	2018-07-01	2018-07-11	<date [11]>

```
fechas_todas %>%
  unnest(fechas) %>%
  left_join(datos, by = c("empresa", "producto", "fechas" = "fecha"))
```

```
# A tibble: 57 x 4
```

	empresa	producto	fechas	evento
	<chr>	<chr>	<date>	<dbl>
1	A	A1	2018-06-02	112
2	A	A1	2018-06-03	NA
3	A	A1	2018-06-04	NA
4	A	A1	2018-06-05	NA
5	A	A1	2018-06-06	141
6	A	A1	2018-06-07	NA
7	A	A1	2018-06-08	NA
8	A	A1	2018-06-09	NA
9	A	A1	2018-06-10	NA
10	A	A1	2018-06-11	NA

```
# ... with 47 more rows
```

3. Abrir varios archivos a la vez

En el directorio de trabajo hay varios archivos que debemos abrir y leer.

```
list.files(pattern="archivo")
```

```
[1] "archivo 1.csv" "archivo 2.csv" "archivo 3.csv"
```

Idea:

- 1 Listar los archivos y construir un tibble
- 2 Leer cada archivo con `read.csv`
- 3 *Desanidar*

```
list.files(pattern="archivo") %>%
  tibble(archivos = .) %>%
  mutate(contenido = map(archivos, read.csv)) %>%
  unnest(contenido)
```

```
list.files(pattern="archivo") %>%  
  tibble(archivos = .) %>%  
  mutate(contenido = map(archivos, read.csv))
```

```
# A tibble: 3 x 2  
  archivos      contenido  
  <chr>         <list>  
1 archivo_1.csv <df[,5] [6 x 5]>  
2 archivo_2.csv <df[,5] [6 x 5]>  
3 archivo_3.csv <df[,5] [6 x 5]>
```

```
list.files(pattern="archivo") %>%
  tibble(archivos = .) %>%
  mutate(contenido = map(archivos, read.csv)) %>%
  unnest(contenido)
```

```
# A tibble: 18 x 6
```

	archivos	name	cat	fecha	v1	v2
	<chr>	<fct>	<fct>	<fct>	<dbl>	<int>
1	archivo_1.csv	ocgux74	B	2021-05-24	8.3	19
2	archivo_1.csv	lecjd73	A	2021-03-16	2.1	60
3	archivo_1.csv	hzprt72	D	2021-04-30	5.8	24
4	archivo_1.csv	epwoz07	A	2021-02-15	8.4	65
5	archivo_1.csv	zfdas14	D	2020-12-21	5	46
6	archivo_1.csv	ywqiu85	D	2021-06-06	0.4	99
7	archivo_2.csv	shjqu73	D	2021-01-26	5.6	94
8	archivo_2.csv	oncxv34	A	2021-03-20	2.1	23
9	archivo_2.csv	yzqml54	D	2021-03-05	2.3	15
10	archivo_2.csv	ohsaq71	C	2021-07-13	4.4	31

```
# ... with 8 more rows
```


- 1 Definir una función que devuelva ambas cantidades
- 2 Aplicarla a cada frase

- 1 Definir una función que devuelva ambas cantidades
- 2 Aplicarla a cada frase


```
analizar_frase <- function(cancion){
  preposiciones <- c("a", "ante", "bajo", "cabe", "con",
                    "contra", "de", "desde", "durante",
                    "en", "entre", "hacia", "hasta", "mediante",
                    "para", "por", "según", "sin", "so", "sobre",
                    "tras", "versus", "vía")

  palabras <- strsplit(cancion, " ") %>% unlist

  cant_palabras <- length(palabras)

  cant_preposiciones <- sum(palabras %in% preposiciones)

  return(list(cant_palabras = cant_palabras,
             cant_preposiciones = cant_preposiciones))
}
```

```
datos %>%
  mutate(resultado = map(frase, analizar_frase)) %>%
  unnest_wider(resultado)
```

```
mutate(resultado = map(frase, analizar frase))
```

```
# A tibble: 7 x 4
```

banda	cancion	frase	resultado
<chr>	<chr>	<chr>	<list>
1 Los Wach~	Este es el pa~	El que no hace palma~	<named lis~
2 La Base	Sabor sabrosón	Según la moraleja, e~	<named lis~
3 Damas Gr~	Me va a extra~	ATR perro cumbia caj~	<named lis~
4 Altos Cu~	No voy a llor~	Andy, fijate que vol~	<named lis~
5 Los Pibe~	Llegamos los ~	Llegamos los pibes c~	<named lis~
6 La Liga	Se re pudrió	El que no hace palma~	<named lis~
7 Los Palm~	La cola	A la una, a la dos, ~	<named lis~

	banda	cancion	frase	cant_palabras	cant_preposicio~
	<chr>	<chr>	<chr>	<int>	<int>
1	Los Wa~	Este es ~	El que n~	8	0
2	La Base	Sabor sa~	Según la~	12	0
3	Damas ~	Me va a ~	ATR perr~	6	0
4	Altos ~	No voy a~	Andy, fi~	8	0
5	Los Pi~	Llegamos~	Llegamos~	10	1
6	La Liga	Se re pu~	El que n~	9	1
7	Los Pa~	La cola	A la una~	15	2

- 1 Combinamos los datos con cada uno de los posibles órdenes del polinomio
- 2 `group by + nest`
- 3 Aplicar una función que cree el gráfico utilizando `map`

```
plots <- crossing(orden = 1:6, datos) %>%
  nest(datos = !orden) %>%
  ungroup() %>%
  mutate(plot = map2(datos, orden,
    function(.x, .y) {
      ggplot(.x, aes(x = x, y = y)) +
        geom_point() +
        stat_smooth(
          method = "lm", se = FALSE,
          formula = y ~ poly(x,
            degree=.y,
            raw = TRUE),
          colour = "maroon1") +
        theme_minimal()
    })
  )
```

```
crossing(poly = 1:6, datos)
```

```
# A tibble: 60 x 3
```

```
poly      x      y
<int> <dbl> <dbl>
```

1 1 157 272

2 1 211 184

3 1 222 158

4 1 230 147

5 1 296 127

6 1 327 158

7 1 414 252

8 1 419 252

9 1 451 249

10 1 587 413

```
# ... with 50 more rows
```



```
6      6 <tibble [10 x 2]>
```

```
plots <- crossing(orden = 1:6, datos) %>%
  nest(datos = !orden) %>%
  ungroup() %>%
  mutate(plot = map2(datos, orden,
    function(.x, .y) {
      ggplot(.x, aes(x = x, y = y)) +
        geom_point() +
        stat_smooth(
          method = "lm", se = FALSE,
          formula = y ~ poly(x,
            degree = .y,
            raw = TRUE),
          colour = "maroon1") +
        theme_minimal()
    })
  )
```

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

orden datos

plot

```
<int> <list>
```

```
<list>
```

```
1 1 <tibble [10 x 2]> <gg>
```

```
2      2 <tibble [10 x 2]> <gg>
```

```
3      3 <tibble [10 x 2]> <gg>
```

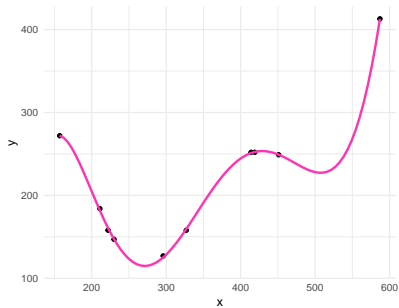
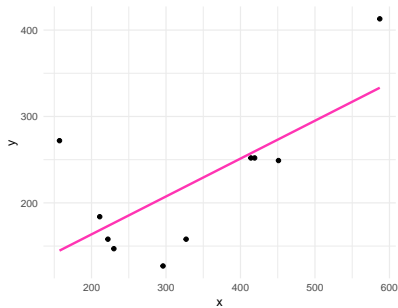
```
4      4 <tibble [10 x 2]> <gg>
```

```
5      5 <tibble [10 x 2]> <gg>
```

```
6      6 <tibble [10 x 2]> <gg>
```



```
library(patchwork)  
plots$plot[[1]] + plots$plot[[6]]
```



6. K-fold cross validation

```
K ← 3
data ← mtcars %>%
  mutate(fold = rep(1:K,length.out=nrow(.))) %>%
  arrange(fold) %>%
  group_by(fold) %>%
  nest() %>%
  mutate(dummy = 1)
```

```
train_test ← data %>%
  inner_join(data, by="dummy") %>%
  select(-dummy) %>%
  filter(fold.y ≠ fold.x) %>%
  group_by(fold.x) %>%
  summarise(test = list(first(data.x)),
            train = list(bind_rows(data.y)))
```

```
train_test %>%
  mutate(modelo = map(train, ~ lm(mpg ~ wt,data=.x)),
         pred = map2(modelo,test, ~ predict(.x,.y)),
         real = map(test,"mpg"))
```

data

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

```
# Groups: fold [3]
```

fold data

dummy

```
<int> <list>
```

<dbl>

```
1      1 <tibble [11 x 11]>      1
```

```
2      2 <tibble [11 x 11]>      1
```

```
3      3 <tibble [10 x 11]>      1
```

```
3 <tibble [10 x 11]>
```



```
data %>%  
  inner_join(data, by="dummy") %>%  
  select(-dummy) %>%  
  filter(fold.y != fold.x) %>%  
  group_by(fold.x)
```

```
# A tibble: 6 x 4
```

```
# Groups:   fold.x [3]
```

	fold.x	data.x		fold.y	data.y
	<int>	<list>		<int>	<list>
1	1	<tibble [11 x 11]>	2	<tibble [11 x 11]>	
2	1	<tibble [11 x 11]>	3	<tibble [10 x 11]>	
3	2	<tibble [11 x 11]>	1	<tibble [11 x 11]>	
4	2	<tibble [11 x 11]>	3	<tibble [10 x 11]>	
5	3	<tibble [10 x 11]>	1	<tibble [11 x 11]>	
6	3	<tibble [10 x 11]>	2	<tibble [11 x 11]>	

train_test

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

fold.x test

train

```
<int> <list>
```

```
<list>
```

```
1      1 <tibble [11 x 11]> <tibble [21 x 11]>
```

```
2      2 <tibble [11 x 11]> <tibble [21 x 11]>
```

```
3      3 <tibble [10 x 11]> <tibble [22 x 11]>
```

```
train_test %>%
  mutate(modelo = map(train, ~ lm(mpg ~ wt,data=.x)))
```

```
# A tibble: 3 x 4
```

	fold.x	test	train	modelo
	<int>	<list>	<list>	<list>
1	1	<tibble [11 x 11]>	<tibble [21 x 11]>	<lm>
2	2	<tibble [11 x 11]>	<tibble [21 x 11]>	<lm>
3	3	<tibble [10 x 11]>	<tibble [22 x 11]>	<lm>

```
train_test %>%
  mutate(modelo = map(train, ~ lm(mpg ~ wt,data=.x)),
         pred = map2(modelo,test, ~ predict(.x,.y)))
```

```
# A tibble: 3 x 5
```

	fold.x	test		train		modelo	pred
	<int>	<list>		<list>		<list>	<list>
1	1	<tibble [11 x 11~		<tibble [21 x 1~		<lm>	<dbl [11~
2	2	<tibble [11 x 11~		<tibble [21 x 1~		<lm>	<dbl [11~
3	3	<tibble [10 x 11~		<tibble [22 x 1~		<lm>	<dbl [10~

```
train_test %>%
  mutate(modelo = map(train, ~ lm(mpg ~ wt,data=.x)),
         pred = map2(modelo,test, ~ predict(.x,.y)),
         real = map(test,"mpg"))
```

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

	fold.x	test		train		modelo	pred		real
	<int>	<list>		<list>		<list>	<list>		<list>
1	1	<tibble [11 ~		<tibble [21 ~		<lm>	<dbl [1~		<dbl [~
2	2	<tibble [11 ~		<tibble [21 ~		<lm>	<dbl [1~		<dbl [~
3	3	<tibble [10 ~		<tibble [22 ~		<lm>	<dbl [1~		<dbl [~

Más de dos argumentos: pmap

Cuando queremos vectorizar operaciones sobre una lista (o vector) usamos `map_*`; para iterar sobre dos listas, `map2_*`; para más de dos listas se usa `pmap_*`

Función pmap

```
pmap(.l, .f, ...)
```

- `.l` es una lista de listas sobre la que aplicaremos la función,
- `.f` es la función en cuestión; si la definimos en forma anónima, sus argumentos son `.. 1`, `.. 2`, `.. 3`, etc.
- `...` son argumentos adicionales de `.f`

Más de dos argumentos: pmap

Quando queremos vectorizar operaciones sobre una lista (o vector) usamos `map_*`; para iterar sobre dos listas, `map2_*`; para más de dos listas se usa `pmap_*`

Función pmap

```
pmap(.l, .f, ...)
```

- `l` es una lista de listas sobre la que aplicaremos la función,
- `f` es la función en cuestión; si la definimos en forma anónima, sus argumentos son `.. 1`, `.. 2`, `.. 3`, etc.
- `...` son argumentos adicionales de `f`

- `l` es una lista de listas sobre la que aplicaremos la función,
- `f` es la función en cuestión; si la definimos en forma anónima, sus argumentos son `.. 1`, `.. 2`, `.. 3`, etc.
- `...` son argumentos adicionales de `f`

possibly

Ocasionalmente hay funciones que pueden fallar y es necesario manejar los errores

```
c(list.files(pattern="archivo"), "archivo_4.csv") %>%
  tibble(archivos = .) %>%
  mutate(contenido = map(archivos, read.csv)) %>%
  unnest(contenido)
```

```
Error : Problem with 'mutate()' input 'contenido'.
x cannot open the connection
i Input 'contenido' is 'map(archivos, read.csv)'.
```


Para salvar los errores existe en `purrr` la función `possibly`. Recibe como primer argumento una función y devuelve una nueva función que tiene una salida específica en caso de que falle (parámetro `otherwise`).

```
possibly_read.csv <- possibly(read.csv,otherwise = data.frame())
```

```
c(list.files(pattern="archivo"), "archivo_4.csv") %>%
  tibble(archivos = .) %>%
  mutate(contenido = map(archivos, possibly read.csv))
```

```
# A tibble: 4 x 2
```

```
archivos      contenido
<chr>        <list>
```

```
1 archivo_1.csv <df[,5] [6 x 5]>
```

```
2 archivo 2.csv <df[,5] [6 x 5]>
```

```
3 archivo_3.csv <df[,5] [6 x 5]>
```

```
4 archivo 4.csv <df[,0] [0 x 0]>
```


- Vale la pena amigarse con las listas
- El paquete `purrr` sirve para vectorizar operaciones
- El paquete `purrr` sirve para iterar
- En un `tibble` podemos meter cualquier cosa
- `dplyr` + `tidyr` + `purrr` = lo que vos quieras
- Superioridad estética y moral de soluciones con un enfoque funcional

Referencias

- Bryan, Jenny. (2019) *Purrr tutorial*. [Link](#).
- Barterm, Rebecca. (2019). *Learn to purrr*. [Link](#).
- Wickham, Hadley (2019). *Advanced R – 2nd Edition*. Capítulo 9. CRC Press. [Link](#).
- Baumer B., Kaplan D. y Horton N. (2021). *Modern Data Science with R – 2nd Edition*. Capítulo 7. CRC Press. [Link](#).

