

# Functional programming in R

2024-07-18

# purrr: A functional programming toolkit for R



*Complete and consistent set of tools for working  
with functions and vectors*

# Problems we want to solve:

- 1 Making code clear
- 2 Making code safe
- 3 Working iteratively with lists and data frames

# Lists, vectors, and data.frames (or tibbles)

```
1 c(char = "hello", num = 1)
```

char	num
"hello"	"1"

# lists can contain any object

```
1 list(char = "hello", num = 1, fun = mean)
```

```
$char
```

```
[1] "hello"
```

```
$num
```

```
[1] 1
```

```
$fun
```

```
function (x, ...)  
UseMethod("mean")  
<bytecode: 0x14578fb70>  
<environment: namespace:base>
```

# Your Turn 1

```
1 measurements <- list(  
2   blood_glucose = rnorm(10, mean = 140, sd = 10),  
3   age = rnorm(5, mean = 40, sd = 5),  
4   heartrate = rnorm(20, mean = 80, sd = 15)  
5 )
```

**There are two ways to subset lists: dollar signs and brackets. Try to subset `blood_glucose` from `measurements` using these approaches.**

**Are they different? What about  
`measurements[["blood_glucose"]]`?**

# Your Turn 1

```
1 measurements["blood_glucose"]
```

\$blood\_glucose

```
[1] 127.9293 142.7743 150.8444 116.5430 144.2912 145.0606  
134.2526 134.5337  
[9] 134.3555 131.0996
```

```
1 measurements$blood_glucose
```

```
[1] 127.9293 142.7743 150.8444 116.5430 144.2912 145.0606  
134.2526 134.5337  
[9] 134.3555 131.0996
```

```
1 measurements[["blood_glucose"]]
```

```
[1] 127.9293 142.7743 150.8444 116.5430 144.2912 145.0606  
134.2526 134.5337  
[9] 134.3555 131.0996
```

# data frames are lists

```
1 x <- list(char = "hello", num = 1)  
2 as.data.frame(x)
```

```
char num  
1 hello 1
```

# data frames are lists

```
1 library(gapminder)  
2 head(gapminder$pop)
```

```
[1] 8425333 9240934 10267083 11537966 13079460 14880372
```

# data frames are lists

```
1 gapminder[1:6, "pop"]
```

```
# A tibble: 6 × 1
```

```
  pop  
  <int>  
1 8425333  
2 9240934  
3 10267083  
4 11537966  
5 13079460  
6 14880372
```

# data frames are lists

```
1 head(gapminder[["pop"]])
```

```
[1] 8425333 9240934 10267083 11537966 13079460 14880372
```

# programming with functions

*functions are objects, too*

```
1 f <- mean  
2 f
```

```
function (x, ...)  
UseMethod("mean")  
<bytecode: 0x14578fb70>  
<environment: namespace:base>
```

```
1 identical(mean, f)
```

```
[1] TRUE
```

# programming with functions

## *source code of a function*

```
1 mean
```

```
function (x, ...)  
UseMethod("mean")  
<bytecode: 0x14578fb70>  
<environment: namespace:base>
```

```
1 sd
```

```
function (x, na.rm = FALSE)  
sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),  
      na.rm = na.rm))  
<bytecode: 0x1272ca9b0>  
<environment: namespace:stats>
```

# dplyr::across()

use within `mutate()` or `summarize()` to apply function(s) to a selection of columns!

## EXAMPLE:

```
df %>%  
  group_by(species) %>%  
  summarise(  
    across(where(is.numeric), mean)  
)
```



species	mass_g	age_yr	range_sqmi
pika	163	2.4	0.46
marmot	1509	3.0	0.87
marmot	2417	5.6	0.62



# mutate(across())

```
1 mutate(  
2   <DATA>,  
3   across(c(<VARIABLES>), list(<NAMES> = <FUNCTIONS>))  
4 )
```

# mutate(across())

```
1 mutate(  
2   diamonds,  
3   across(c("carat", "depth"), mean)  
4 )
```

```
# A tibble: 53,940 × 10  
  carat cut      color clarity depth table price     x     y  
  <dbl> <ord>    <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl>  
1 0.798 Ideal     E      SI2      61.7    55    326  3.95  3.98  
2 0.798 Premium   E      SI1      61.7    61    326  3.89  3.84  
3 0.798 Good      E      VS1      61.7    65    327  4.05  4.07  
4 0.798 Premium   I      VS2      61.7    58    334  4.2   4.23  
5 0.798 Good      J      SI2      61.7    58    335  4.34  4.35  
6 0.798 Very      G...  J      VVS2     61.7    57    336  3.94  3.96  
7 0.798 Very      G...  I      VVS1     61.7    57    336  3.95  3.98  
8 0.798 Very      G...  H      SI1      61.7    55    337  4.07  4.11  
9 0.798 Fair      E      VS2      61.7    61    337  3.87  3.78  
10 0.798 Very     G...  H      VS1      61.7    61    338  4     4.05  
# ... with 53,930 more rows
```

# mutate(across())

```
1 mutate(  
2   diamonds,  
3   across(c("carat", "depth"), list(mean = mean, sd = sd  
4 ))
```

```
# A tibble: 53,940 × 14  
  carat cut      color clarity depth table price     x     y  
  <dbl> <ord>    <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>  
1 0.23 Ideal     E       SI2     61.5    55    326  3.95  3.98  
2 0.21 Premium   E       SI1     59.8    61    326  3.89  3.84  
3 0.23 Good      E       VS1     56.9    65    327  4.05  4.07  
4 0.29 Premium   I       VS2     62.4    58    334  4.2   4.23  
5 0.31 Good      J       SI2     63.3    58    335  4.34  4.35  
6 0.24 Very      G...  J       VVS2    62.8    57    336  3.94  3.96  
7 0.24 Very      G...  I       VVS1    62.3    57    336  3.95  3.98  
8 0.26 Very      G...  H       SI1     61.9    55    337  4.07  4.11  
9 0.22 Fair      E       VS2     65.1    61    337  3.87  3.78  
10 0.23 Very     G...  H       VS1     59.4   61    338   4    4.05  
# ... etc.
```

# mutate(across(where()))

```
1 mutate(  
2   gapminder,  
3   across(where(is.numeric), median)  
4 )
```

```
# A tibble: 1,704 × 6  
  country continent year lifeExp      pop gdpPercap  
  <fct>    <fct>   <dbl>    <dbl>    <dbl>    <dbl>  
1 Afghanistan Asia     1980.    60.7 7023596. 3532.  
2 Afghanistan Asia     1980.    60.7 7023596. 3532.  
3 Afghanistan Asia     1980.    60.7 7023596. 3532.  
4 Afghanistan Asia     1980.    60.7 7023596. 3532.  
5 Afghanistan Asia     1980.    60.7 7023596. 3532.  
6 Afghanistan Asia     1980.    60.7 7023596. 3532.  
7 Afghanistan Asia     1980.    60.7 7023596. 3532.  
8 Afghanistan Asia     1980.    60.7 7023596. 3532.  
9 Afghanistan Asia     1980.    60.7 7023596. 3532.  
10 Afghanistan Asia    1980.    60.7 7023596. 3532.  
# ... 1,694 more rows
```

# Review: `tidyselect`

**Workhorse for `dplyr::select()`,  
`dplyr::pull()`, and `tidyr::pivot_`  
functions**

`starts_with()`, `ends_with()`, `contains()`,  
`matches()`, etc.

# Review: tidyselect

```
1 # column names contain a word  
2 select(diabetes, ends_with("ht"))
```

# A tibble: 403 × 2

height weight

<dbl> <dbl>

1	62	121
2	64	218
3	61	256
4	67	119
5	68	183
6	71	190
7	69	191
8	59	170
9	69	166
10	63	202

# i 393 more rows

```
1 # regular expression  
2 select(diabetes, matches("\\d"))
```

# A tibble: 403 × 4

bp.1s bp.1d bp.2s bp.2d

<dbl> <dbl> <dbl> <dbl>

1	118	59	NA	NA
2	112	68	NA	NA
3	190	92	185	92
4	110	50	NA	NA
5	138	80	NA	NA
6	132	86	NA	NA
7	161	112	161	112
8	NA	NA	NA	NA
9	160	80	128	86
10	108	72	NA	NA

# i 393 more rows

# mutate(across()) & summarise()

```
1 gapminder |>
2   group_by(continent) |>
3   summarise(
4     across(
5       c("lifeExp", "gdpPercap"),
6       list(med = median, iqr = IQR)
7     ))
```

```
# A tibble: 5 × 5
  continent lifeExp_med lifeExp_iqr gdpPercap_med
  <fct>      <dbl>      <dbl>      <dbl>
1 Africa        47.8       12.0      1192.
2 Americas      67.0       13.3      5466.
3 Asia          61.8       18.1      2647.
4 Europe        72.2       5.88     12082.
5 Oceania       73.7       6.35     17983.
# i 1 more variable: gdpPercap_iqr <dbl>
```

# mutate(across()) & summarise()

## Control output names with .names argument

```
1 gapminder |>
2   group_by(continent) |>
3   summarise(
4     across(
5       c("lifeExp", "gdpPercap"),
6       list(med = median, iqr = IQR),
7       .names = "{.fn}_{.col}"
8     ))
```

```
# A tibble: 5 × 5
  continent med_lifeExp iqr_lifeExp med_gdpPercap
  <fct>        <dbl>      <dbl>          <dbl>
1 Africa         47.8       12.0          1192.
2 Americas       67.0       13.3          5466.
3 Asia           61.8       18.1          2647.
4 Europe         72.2       5.88          12082.
5 Oceania        73.7       6.35          17983.
# i 1 more variable: iqr_gdpPercap <dbl>
```

## Your Turn 2

Use `starts_with()` from `tidyselect()` to calculate the average `bp` columns in `diabetes`, grouped by `gender`. Give the new columns the name pattern `column_function`, e.g. `bp.1s_mean`.

hint: `{.fn}` will give you the function name, and `{.col}` will give you the column name

# Your Turn 2

```
1 diabetes |>
2   group_by(gender) |>
3   summarise(
4     across(
5       starts_with("bp"),
6       mean,
7       na.rm = TRUE,
8       .names = "{.col}_{.fn}"
9     ))
```

# A tibble: 2 × 5

	gender	bp.1s_1	bp.1d_1	bp.2s_1	bp.2d_1
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1	female	136.	82.5	153.	91.8
2	male	138.	84.5	151.	93.5

# vectorized functions don't work on lists

```
1 sum(rnorm(10))
```

```
[1] -7.198318
```

# vectorized functions don't work on lists

```
1 sum(list(x = rnorm(10), y = rnorm(10), z = rnorm(10)))
```

Error in sum(list(x = rnorm(10), y = rnorm(10), z = rnorm(10))):  
invalid 'type' (list) of argument

# **map(.x, .f)**

.x: a vector, list, or data frame

.f: a function

Returns a list



# Using `map()`

```
1 library(purrr)
2 measurements <- list(
3   blood_glucose = rnorm(10, mean = 140, sd = 10),
4   age = rnorm(5, mean = 40, sd = 5),
5   heartrate = rnorm(20, mean = 80, sd = 15)
6 )
7
8 map(measurements, mean)
```

```
$blood_glucose
```

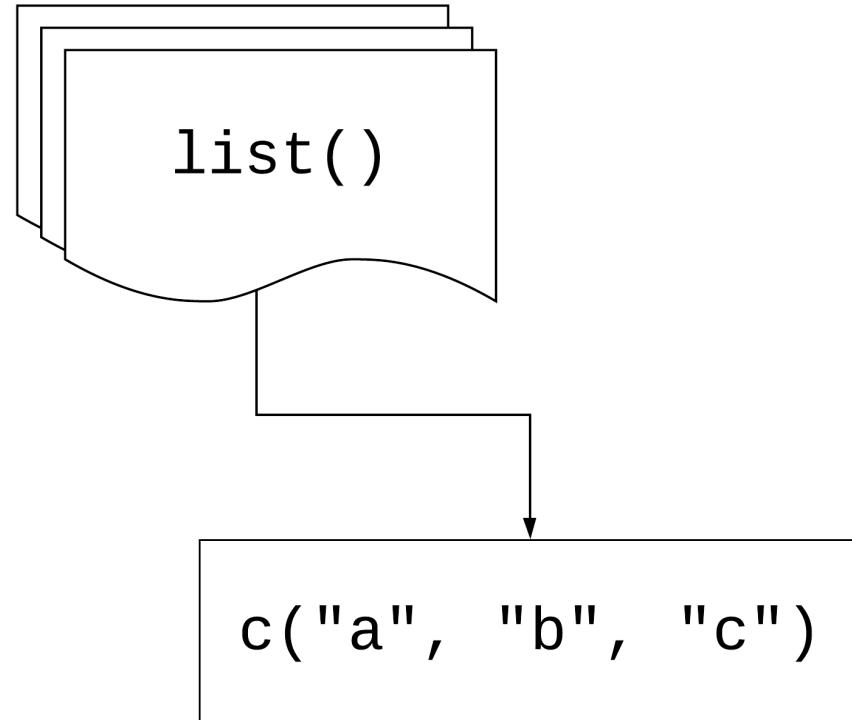
```
[1] 136.6299
```

```
$age
```

```
[1] 39.45875
```

```
$heartrate
```

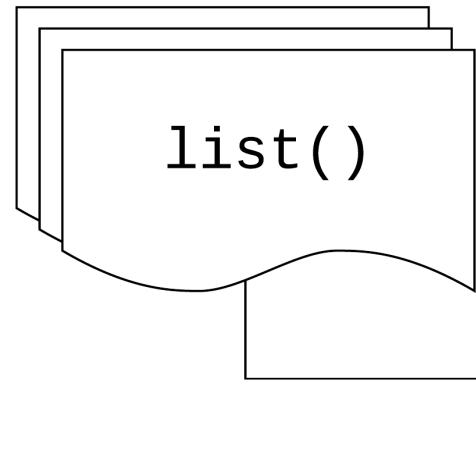
```
[1] 83.01965
```



```
c("a", "b", "c")
```

```
c("a", "b", "c")
```

```
map( list(), .f )
```



```
.f( c("a", "b", "c") )
```

```
.f( c("a", "b", "c") )
```

```
.f( c("a", "b", "c") )
```

```
map( c("a", "b", "c"), .f )
```

```
map( list(), .f )
```

```
map( data.frame(), .f )
```

## Your Turn 3

**Read the code in the first chunk and predict what will happen**

**Run the code in the first chunk. What does it return?**

```
1 list(  
2   blood_glucose = sum(measurements$blood_glucose),  
3   age = sum(measurements$age),  
4   heartrate = sum(measurements$heartrate)  
5 )
```

**Now, use `map()` to create the same output.**

# Your Turn 3

```
1 map(measurements, sum)
```

```
$blood_glucose
```

```
[1] 1366.299
```

```
$age
```

```
[1] 197.2938
```

```
$heartrate
```

```
[1] 1660.393
```

# using `map()` with data frames

```
1 library(dplyr)
2 gapminder |>
3   select(where(is.numeric)) |>
4   map(sd)
```

```
$year
[1] 17.26533
```

```
$lifeExp
[1] 12.91711
```

```
$pop
[1] 106157897
```

```
$gdpPercap
[1] 9857.455
```

## *Your Turn 4*

**Pass diabetes to `map()` and map using `class()`.  
What are these results telling you?**

## Your Turn 4

```
1 head(  
2   map(diabetes, class),  
3   3  
4 )
```

```
$id  
[1] "numeric"
```

```
$chol  
[1] "numeric"
```

```
$stab.glu  
[1] "numeric"
```

# Review: writing functions

```
1 library(readxl)
2 weird_data1 <- read_excel(
3   "data/weird_data1.xlsx",
4   col_names = c("id", "x", "y", "z"),
5   skip = 5
6 )
```

# Review: writing functions

```
1 library(readxl)
2 weird_data1 <- read_excel(
3   "data/weird_data1.xlsx",
4   col_names = c("id", "x", "y", "z"),
5   skip = 5
6 )
7
8 weird_data2 <- read_excel(
9   "data/weird_data1.xlsx",
10  col_names = c("id", "x", "y", "z"),
11  skip = 5
12 )
13
14 weird_data1 <- read_excel(
15   "data/weird_data3.xlsx",
16   col_names = c("id", "x", "y", "z"),
17   skip = 5
18 )
```

# Review: writing functions

```
1 library(readxl)
2 weird_data1 <- read_excel(
3   "data/weird_data1.xlsx",
4   sheet = 2,
5   col_names = c("id", "x", "y", "z"),
6   skip = 6
7 )
8
9 weird_data2 <- read_excel(
10  "data/weird_data1.xlsx",
11  col_names = c("id", "x", "y", "z"),
12  skip = 5
13 )
14
15 weird_data1 <- read_excel(
16  "data/weird_data3.xlsx",
17  col_names = c("id", "x", "y", "z"),
18  skip = 5
19 )
```

# Review: writing functions

```
1 library(readxl)
2 read_weird_excel <- function(path) {
3   read_excel(
4     path,
5     sheet = 2,
6     col_names = c("id", "x", "y", "z"),
7     skip = 6
8   )
9 }
10
11 weird_data1 <- read_weird_excel("data/weird_data1")
12 weird_data2 <- read_weird_excel("data/weird_data2")
13 weird_data3 <- read_weird_excel("data/weird_data3")
```

If you copy and paste  
your *code* three times,  
it's time to write a  
function

If you copy and paste  
your *function* three  
times, it's (*probably*) time  
to iterate

# Iterating with functions

```
1 files <- c(  
2   "data/weird_data1.xlsx",  
3   "data/weird_data2.xlsx",  
4   "data/weird_data3.xlsx"  
5 )  
6  
7 weird_data <- map(files, read_weird_excel) |>  
8 bind_rows()
```

## *Your Turn 5*

**Write a function that returns the mean and standard deviation of a numeric vector.**

**Give the function a name**

**Find the mean and SD of `x`**

**Map your function to `measurements`**

# Your Turn 5

```
1 mean_sd <- function(x) {  
2   x_mean <- mean(x)  
3   x_sd <- sd(x)  
4   tibble(mean = x_mean, sd = x_sd)  
5 }  
6  
7 map(measurements, mean_sd)
```

# Your Turn 5

```
$blood_glucose  
# A tibble: 1 × 2  
  mean      sd  
  <dbl> <dbl>  
1 137.   6.84
```

```
$age  
# A tibble: 1 × 2  
  mean      sd  
  <dbl> <dbl>  
1 39.5    3.84
```

```
$heartrate  
# A tibble: 1 × 2  
  mean      sd  
  <dbl> <dbl>
```

# Three ways to pass functions to `map()`

- 1 pass directly to `map()`
- 2 use an anonymous function
- 3 use a lambda (`\()` or `~`)

```
1 map(  
2   · x,  
3   mean,  
4   na.rm = TRUE  
5 )
```

```
1 map(  
2   .x,  
3   function(.x) mean(.x, na.rm = TRUE)  
4 )
```

```
1 map(  
2   ·x,  
3   \(.x) mean(.x, na.rm =  
4 )
```

```
1 map(  
2   ·x,  
3   ~ mean( ·x, na.rm = T  
4 )
```

```
1 map(  
2   gapminder,  
3   \(.x) length(unique(.x))  
4 )
```

```
$country  
[1] 142
```

```
$continent  
[1] 5
```

```
$year  
[1] 12
```

```
$lifeExp  
[1] 1626
```

```
$pop  
[1] 1704
```

# Returning types

map	returns
<code>map()</code>	list
<code>map_chr()</code>	character vector
<code>map_dbl()</code>	double vector (numeric)
<code>map_int()</code>	integer vector
<code>map_lgl()</code>	logical vector
<code>map_dfc()</code>	data frame (by column)
<code>map_dfr()</code>	data frame (by row)

# Iterating with functions: revisited

```
1 files <- c(  
2   "data/weird_data1.xlsx",  
3   "data/weird_data2.xlsx",  
4   "data/weird_data3.xlsx"  
5 )  
6  
7 weird_data <- map(files, read_weird_excel) |>  
8 bind_rows()
```

# Iterating with functions: revisited

```
1 files <- c("data/weird_data1.xlsx", "data/weird_data2.xlsx")
2 weird_data <- map_dfr(files, read_weird_excel)
```

# Returning types

```
1 map_int(gapminder, \(.x) length(unique(.x)))
```

country	continent	year	lifeExp	pop	gdpPerCap
142	5	12	1626	1704	1704

## *Your Turn 6*

**Do the same as #4 above but return a vector instead of a list.**

## Your Turn 6

```
1 map_chr(diabetes, class)
```

```
  id      chol    stab.glu      hdl      ratio      glyhb  
"numeric" "numeric" "numeric" "numeric" "numeric" "numeric"  
location      age      gender      height      weight      frame  
"character" "numeric" "character" "numeric" "numeric" "character"  
  bp.1s      bp.1d    bp.2s      bp.2d      waist      hip  
"numeric" "numeric" "numeric" "numeric" "numeric" "numeric"  
time.ppn  
"numeric"
```

## Your Turn 7

Check `diabetes` for any missing data.

Using the `\(.x)` `.f(.x)` shorthand, check each column for any missing values using `is.na()` and `any()`

Return a logical vector. Are any columns missing data? What happens if you don't include `any()`? Why?

Try counting the number of missing, returning an integer vector

## Your Turn 7

```
1 map_lgl(diabetes, \(.x) any(is.na(.x)))
```

id	chol	stab.glu	hdl	ratio	glyhb	location	age
FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	FALSE	FALSE
gender	height	weight	frame	bp.1s	bp.1d	bp.2s	bp.2d
FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
waist	hip	time.ppn					
TRUE	TRUE	TRUE					

## Your Turn 7

```
1 map_int(diabetes, \(.x) sum(is.na(.x)))
```

	id	chol	stab.glu	hdl	ratio	glyhb	location	age
0		1	0	1	1	13	0	0
gender	height	weight	frame	bp.1s	bp.1d	bp.2s	bp.2d	
0		5	1	12	5	5	262	262
waist	hip	time.ppn						
2		2	3					

# group\_map()

**Apply a function across a grouping variable and return a list of grouped tibbles**

```
1 library(broom)
2 diabetes |>
3   group_by(gender) |>
4   group_map(\(.x, ...)) tidy(lm(weight ~ height, data =
```

# group\_map()

```
[[1]]  
# A tibble: 2 × 5  
  term      estimate std.error statistic p.value  
  <chr>     <dbl>     <dbl>     <dbl>    <dbl>  
1 (Intercept) -73.8      59.2     -1.25  0.214  
2 height        3.90      0.928      4.20 0.0000383  
  
[[2]]  
# A tibble: 2 × 5  
  term      estimate std.error statistic p.value  
  <chr>     <dbl>     <dbl>     <dbl>    <dbl>  
1 (Intercept) -49.7      68.9     -0.722  0.471  
2 height        3.35      0.995      3.37 0.000945
```

# group\_modify()

**Apply a function across grouped tibbles and return grouped tibbles**

```
1 diabetes |>  
2   group_by(gender) |>  
3   group_modify(\(., ...)) tidy(lm(weight ~ height, data
```

```
# A tibble: 4 × 6  
# Groups:   gender [2]  
  gender term      estimate std.error statistic p.value  
  <chr>  <chr>     <dbl>     <dbl>     <dbl>     <dbl>  
1 female (Intercept) -73.8      59.2     -1.25    0.214  
2 female height       3.90      0.928      4.20    0.0000383  
3 male   (Intercept) -49.7      68.9     -0.722   0.471  
4 male   height        3.35      0.995      3.37    0.000945
```

## *Your Turn 8*

Fill in the `model_lm` function to model `chol` (the outcome) with `ratio` and pass the `.data` argument to `lm()`

Group `diabetes` by `location`

Use `group_modify()` with `model_lm`

# Your Turn 8

```
1 model_lm <- function(.data, ...) {  
2   mdl <- lm(chol ~ ratio, data = .data)  
3   # get model statistics  
4   glance(mdl)  
5 }  
6  
7 diabetes |>  
8   group_by(location) |>  
9   group_modify(model_lm)
```

# Your Turn 8

```
# A tibble: 2 × 13
# Groups:   location [2]
  location r.squared adj.r.squared sigma statistic p.value
  <chr>        <dbl>            <dbl>  <dbl>      <dbl>    <dbl>
1 Buckingh...     0.252           0.248   38.8      66.4  4.11e-14
2 Louisa         0.204           0.201   39.4      51.7  1.26e-11
# i 7 more variables: df <dbl>, logLik <dbl>, AIC <dbl>,
#   BIC <dbl>, deviance <dbl>, df.residual <int>, ...
```

# **map2(.x, .y, .f)**

.x, .y: a vector, list, or data frame

.f: a function that takes *two* arguments

Returns a list

map2(



,



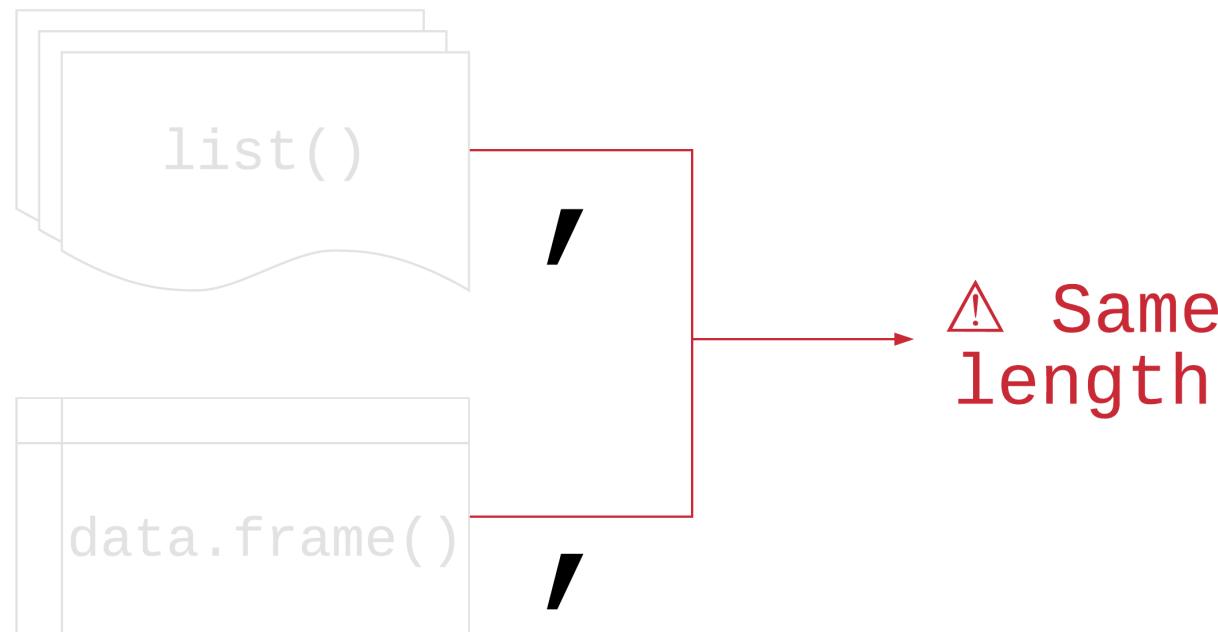
,

.f

)



# map2(



.f

)



# map2()

```
1 means <- c(-3, 4, 2, 2.3)
2 sds <- c(.3, 4, 2, 1)
3
4 map2_dbl(means, sds, rnorm, n = 1)
```

```
[1] -3.0587804  1.4037210 -0.2195345  3.1492742
```

## Your Turn 9

Split the gapminder dataset into a list by country using the `split()` function

Create a list of models using `map()`. For the first argument, pass `gapminder_countries`. For the second, use the `\()` notation to write a model with `lm()`. Use `lifeExp` on the left hand side of the formula and `year` on the second. Pass `.x` to the data argument.

Use `map2()` to take the models list and the data set list and map them to `predict()`. Since we're not adding new arguments, you don't need to use `\()`.

## Your Turn 9

```
1 gapminder_countries <- split(gapminder, gapminder$country)
2 models <- map(
3   gapminder_countries,
4   \(.x) lm(lifeExp ~ year, data = .x)
5 )
6 preds <- map2(models, gapminder_countries, predict)
7 head(preds, 3)
```

## Your Turn 9

\$Afghanistan

1	2	3	4	5	6	7	8
29.90729	31.28394	32.66058	34.03722	35.41387	36.79051	38.16716	39.54380
9	10	11	12				
40.92044	42.29709	43.67373	45.05037				

\$Albania

1	2	3	4	5	6	7	8
59.22913	60.90254	62.57596	64.24938	65.92279	67.59621	69.26962	70.94304
9	10	11	12				
72.61646	74.28987	75.96329	77.63671				

\$Algeria

1	2	3	4	5	6	7	8
43.37497	46.22137	49.06777	51.91417	54.76057	57.60697	60.45337	63.29976
9	10	11	12				
66.14616	68.99256	71.83896	74.68536				

input 1	input 2	returns
map()	map2()	list
map_chr()	map2_chr()	character vector
map dbl()	map2 dbl()	double vector (numeric)
map int()	map2 int()	integer vector
map lgl()	map2 lgl()	logical vector
map dfc()	map2 dfc()	data frame (by column)
map dfr()	map2 dfr()	data frame (by row)

# Other mapping functions

`pmap()` and friends: take n lists or data frame with argument names

`walk()` and friends: for side effects like plotting; returns input invisibly

`imap()` and friends: includes counter `i`

`map_if()`, `map_at()`: Apply only to certain elements

input 1	input 2	input n	returns
<code>map()</code>	<code>map2()</code>	<code>pmap()</code>	list
<code>map_chr()</code>	<code>map2_chr()</code>	<code>pmap_chr()</code>	character vector
<code>map_dbl()</code>	<code>map2_dbl()</code>	<code>pmap_dbl()</code>	double vector (numeric)
<code>map_int()</code>	<code>map2_int()</code>	<code>pmap_int()</code>	integer vector
<code>map_lgl()</code>	<code>map2_lgl()</code>	<code>pmap_lgl()</code>	logical vector
<code>map_dfc()</code>	<code>map2_dfc()</code>	<code>pmap_dfc()</code>	data frame (by column)
<code>map_dfr()</code>	<code>map2_dfr()</code>	<code>pmap_dfr()</code>	data frame (by row)
<code>walk()</code>	<code>walk2()</code>	<code>pwalk()</code>	input (side effects!)

# group\_walk()

Use **group\_walk()** for side effects across groups

```
1 # fs helps us work with files
2 library(fs)
3 temp <- "temporary_folder"
4 dir_create(temp)
5 gapminder |>
6   group_by(continent) |>
7   group_walk(
8     \(.x, .key) write_csv(
9       .x,
10      file = path(temp, paste0(.key$continent, ".xlsx"))
11    )
12  )
```

# group\_walk()

```
temporary_folder
└── Africa.xlsx
    ├── Americas.xlsx
    ├── Asia.xlsx
    ├── Europe.xlsx
    └── Oceania.xlsx
```

## Your turn 10

Create a new directory using the `fs` package. Call it “figures”.

Write a function to plot a line plot of a given variable in `gapminder` over time, faceted by continent. Then, save the plot (how do you save a `ggplot`?). For the file name, paste together the folder, name of the variable, and extension so it follows the pattern

`"folder/variable_name.png"`

Create a character vector that has the three variables we'll plot: “`lifeExp`”, “`pop`”, and “`gdpPercap`”.

Use `walk()` to save a plot for each of the variables

## Your turn 10

```
1 dir_create("figures")
2
3 ggsave_gapminder <- function(variable) {
4   p <- ggplot(
5     gapminder,
6     aes(x = year, y = {{ variable }}, color = country)
7   ) +
8     geom_line() +
9     scale_color_manual(values = country_colors) +
10    facet_wrap(~ continent) +
11    theme(legend.position = "none")
12
13   ggsave(
14     filename = paste0("figures/", variable, ".png"),
15     plot = p,
16     dpi = 320
17   )
18 }
```

## Your turn 10

```
1 vars <- c("lifeExp", "pop", "gdpPerCap")
2 walk(vars, ggsave_gapminder)
```

# Base R

base R	purrr
<code>lapply()</code>	<code>map()</code>
<code>vapply()</code>	<code>map_*</code> ( <code>)</code>
<code>sapply()</code>	<code>?</code>
<code>x[] &lt;- lapply()</code>	<code>map_dfc()</code>
<code>mapply()</code>	<code>map2()</code> , <code>pmap()</code>

# Benefits of purrr

- 1 Consistent
- 2 Type-safe

# Loops vs functional programming

```
1 x <- rnorm(10)
2 y <- map(x, mean)
```

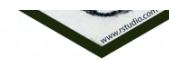
# Loops vs functional programming

```
1 x <- rnorm(10)
2 y <- vector("list", length(x))
3 for (i in seq_along(x)) {
4   y[[i]] <- mean(x[[i]])
5 }
```

Of course someone has to write loops. It doesn't have to be you.

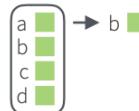
—Jenny Bryan

# Working with lists and nested data



## Work with Lists

### FILTER LISTS



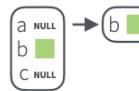
**pluck(.x, ..., .default=NULL)**  
Select an element by name or index, `pluck(x,"b")`, or its attribute with `attr_getter`.  
`pluck(x,"b",attr_getter("n"))`



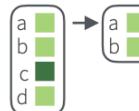
**keep(.x, .p, ...)** Select elements that pass a logical test. `keep(x, is.na)`



**discard(.x, .p, ...)** Select elements that do not pass a logical test. `discard(x, is.na)`

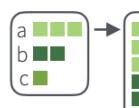


**compact(.x, .p = identity)**  
Drop empty elements.  
`compact(x)`

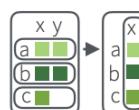


**head\_while(.x, .p, ...)**  
Return head elements until one does not pass. Also `tail_while`.  
`head_while(x, is.character)`

### RESHAPE LISTS



**flatten(.x)** Remove a level of indexes from a list. Also `flatten_chr`, `flatten_dbl`, `flatten_dfc`, `flatten_dfr`, `flatten_int`, `flatten_lgl`.  
`flatten(x)`



**transpose(.l, .names = NULL)** Transposes the index order in a multi-level list.  
`transpose(x)`

### SUMMARISE LISTS



**every(.x, .p, ...)** Do all elements pass a test?  
`every(x, is.character)`



**some(.x, .p, ...)** Do some elements pass a test?  
`some(x, is.character)`



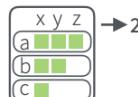
**has\_element(.x, .y)** Does a list contain an element?  
`has_element(x, "foo")`



**detect(.x, .f, ..., .right=FALSE, .p)** Find first element to pass.  
`detect(x, is.character)`

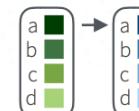


**detect\_index(.x, .f, ..., .right = FALSE, .p)** Find index of first element to pass.  
`detect_index(x, is.character)`

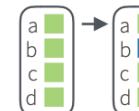


**vec\_depth(x)** Return depth (number of levels of indexes).  
`vec_depth(x)`

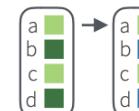
### TRANSFORM LISTS



**modify(.x, .f, ...)** Apply function to each element. Also `map`, `map_chr`, `map_dbl`, `map_dfc`, `map_dfr`, `map_int`, `map_lgl`.  
`modify(x, ~.+2)`



**modify\_at(.x, .at, .f, ...)** Apply function to elements by name or index. Also `map_at`.  
`modify_at(x, "b", ~.+2)`



**modify\_if(.x, .p, .f, ...)** Apply function to elements that pass a test. Also `map_if`.  
`modify_if(x, is.numeric, ~.+2)`

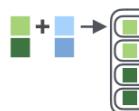


**modify\_depth(.x,.depth,.f,...)**  
Apply function to each element at a given level of a list.  
`modify_depth(x, 1, ~.+2)`

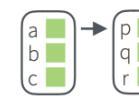
### WORK WITH LISTS



**array\_tree(array, margin = NULL)** Turn array into list. Also `array_branch`.  
`array_tree(x, margin = 3)`



**cross2(.x, .y, .filter = NULL)**  
All combinations of .x and .y. Also `cross`, `cross3`, `cross_df`.  
`cross2(1:3, 4:6)`



**set\_names(x, nm = x)** Set the names of a vector/list directly or with a function.  
`set_names(x, c("p", "q", "r"))`  
`set_names(x, tolower)`

# Working with lists and nested data

## Nested Data

A **nested data frame** stores individual tables within the cells of a larger, organizing table.

nested data frame	
Species	data
setosa	<tibble [50 x 4]>
versicolor	<tibble [50 x 4]>
virginica	<tibble [50 x 4]>

n\_iris

Use a nested data frame to:

- preserve relationships between observations and subsets of data
- manipulate many sub-tables at once with the **purrr** functions `map()`, `map2()`, or `pmap()`.

"cell" contents			
Sepal.L	Sepal.W	Petal.L	Petal.W
5.1	3.5	1.4	0.2
4.9	3.0	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5.0	3.6	1.4	0.2

n\_iris\$data[[1]]

Sepal.L	Sepal.W	Petal.L	Petal.W
7.0	3.2	4.7	1.4
6.4	3.2	4.5	1.5
6.9	3.1	4.9	1.5
5.5	2.3	4.0	1.3
6.5	2.8	4.6	1.5

n\_iris\$data[[2]]

Sepal.L	Sepal.W	Petal.L	Petal.W
6.3	3.3	6.0	2.5
5.8	2.7	5.1	1.9
7.1	3.0	5.9	2.1
6.3	2.9	5.6	1.8
6.5	3.0	5.8	2.2

n\_iris\$data[[3]]

## List Column Workflow

### 1 Make a list column

Species	S.L	S.W	P.L	P.W
setosa	5.1	3.5	1.4	0.2
setosa	4.9	3.0	1.4	0.2
setosa	4.7	3.2	1.3	0.2
setosa	4.6	3.1	1.5	0.2
versi	7.0	3.2	4.7	1.4
versi	6.4	3.2	4.5	1.5
versi	6.9	3.1	4.9	1.5
versi	5.5	2.3	4.0	1.3
virgini	6.3	3.3	6.0	2.5
virgini	5.8	2.7	5.1	1.9
virgini	7.1	3.0	5.9	2.1
virgini	6.3	2.9	5.6	1.8

```
n_iris <- iris %>%
  group_by(Species) %>%
  nest()
```

Nested data frames use a **list column**, a list that is stored as a column vector of a data frame. A typical **workflow** for list columns:

### 2 Work with list columns

S.L	S.W	P.L	P.W
7.0	3.2	4.7	1.4
6.4	3.2	4.5	1.5
6.9	3.1	4.9	1.5
5.5	2.3	4.0	1.3

```
mod_fun <- function(df)
  lm(Sepal.Length ~ ., data = df)
```

```
m_iris <- n_iris %>%
  mutate(model = map(data, mod_fun))
```

```
Call: lm(S.L ~ ., df)
Coefficients:
  (Int) S.W P.L P.W 
  2.3   0.6  0.2  0.2
```

```
Call: lm(S.L ~ ., df)
Coefficients:
  (Int) S.W P.L P.W 
  1.8   0.3  0.9 -0.6
```

```
Call: lm(S.L ~ ., df)
Coefficients:
  (Int) S.W P.L P.W 
  0.6   0.3  0.9 -0.1
```

```
b_fun <- function(mod)
  coefficients(mod)[[1]]
```

```
m_iris %>% transmute(Species,
  beta = map_dbl(model, b_fun))
```

```
1. MAKE A LIST COLUMN - You can create list columns with functions in the tibble and dplyr packages, as well as tidyR's nest()
```

`tibble::tribble(...)`

`tibble::tibble(...)`

`dplyr::mutate(.data, ...)` Also `transmute()`



# Adverbs: Modify function behavior

---

## Modify function behavior

**compose()** Compose multiple functions.

**lift()** Change the type of input a function takes. Also **lift\_dl**, **lift\_dv**, **lift\_Id**, **lift\_lv**, **lift\_vd**, **lift\_vl**.

**rerun()** Rerun expression n times.

**negate()** Negate a predicate function (a pipe friendly !)

**partial()** Create a version of a function that has some args preset to values.

**safely()** Modify func to return list of results and errors.

**quietly()** Modify function to return list of results, output, messages, warnings.

**possibly()** Modify function to return default value whenever an error occurs (instead of error).

# Resources

**Jenny Bryan's purrr tutorial:** A detailed introduction to purrr. Free online.

**R for Data Science, 2nd ed.:** A comprehensive but friendly introduction to the tidyverse. Free online.

**Posit Recipes:** Common code patterns in R (with some comparisons to SAS)

