

# Functional programming in R

2021-10-12

# **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 with lists and data frames

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

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

```
##     char      num
## "hello"      "1"
```

# lists can contain any object

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

```
## $char
## [1] "hello"
##
## $num
## [1] 1
##
## $fun
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x15400be78>
## <environment: namespace:base>
```

# Your Turn 1

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

**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

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

```
## $blood_glucose  
## [1] 127.9293 142.7743 150.8444 116.5430 144.2912  
## [6] 145.0606 134.2526 134.5337 134.3555 131.0996
```

```
measurements$blood_glucose
```

```
## [1] 127.9293 142.7743 150.8444 116.5430 144.2912  
## [6] 145.0606 134.2526 134.5337 134.3555 131.0996
```

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

```
## [1] 127.9293 142.7743 150.8444 116.5430 144.2912  
## [6] 145.0606 134.2526 134.5337 134.3555 131.0996
```

# data frames are lists

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

```
##      char num  
## 1 hello    1
```

# data frames are lists

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

```
## [1] 8425333 9240934 10267083 11537966 13079460  
## [6] 14880372
```

# data frames are lists

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

# data frames are lists

```
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

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

```
## [1] 8425333 9240934 10267083 11537966 13079460  
## [6] 14880372
```

# programming with functions

## View source code of a function

```
mean
```

```
## function (x, ...)  
## UseMethod("mean")  
## <bytecode: 0x15400be78>  
## <environment: namespace:base>
```

```
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: 0x124e56d50>  
## <environment: namespace:stats>
```

# programming with functions

**functions are objects and can be assigned to variables**

```
f <- mean
```

```
f
```

```
## function (x, ...)  
## UseMethod("mean")  
## <bytecode: 0x15400be78>  
## <environment: namespace:base>
```

```
identical(x = mean, y = f)
```

```
## [1] TRUE
```

# 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       |

@allison\_horst

Art by Allison Horst

# mutate(across())

```
mutate(  
  <DATA>,  
  across(c(<VARIABLES>), list(<NAMES> = <FUNCTIONS>))  
)
```

# mutate(across())

```
mutate(  
  diamonds,  
  across(c("carat", "depth"), list(sd = sd, mean = mean)))  
)
```

# mutate(across())

```
mutate(  
  diamonds,  
  across(c("carat", "depth"), list(sd = sd, mean = mean))  
)  
  
## # A tibble: 53,940 × 14  
##   carat cut     color clarity depth table price  
##   <dbl> <ord>    <ord> <ord>   <dbl> <dbl> <int>  
## 1 0.23 Ideal     E     SI2      61.5    55    326  
## 2 0.21 Premium   E     SI1      59.8    61    326  
## 3 0.23 Good      E     VS1      56.9    65    327  
## 4 0.29 Premium   I     VS2      62.4    58    334  
## 5 0.31 Good      J     SI2      63.3    58    335  
## 6 0.24 Very Go... J     VVS2     62.8    57    336  
## 7 0.24 Very Go... I     VVS1     62.3    57    336  
## 8 0.26 Very Go... H     SI1      61.9    55    337  
## 9 0.22 Fair      E     VS2      65.1    61    337  
## 10 0.23 Very Go... H    VS1      59.4    61    338  
## # ... with 53,930 more rows, and 7 more variables:  
## #   x <dbl>, y <dbl>, z <dbl>, carat_sd <dbl>,  
## #   carat_mean <dbl>, depth_sd <dbl>,  
## #   depth_mean <dbl>
```

# mutate(across(where()))

```
mutate(  
  gapminder,  
  across(where(is.numeric), list(mean = mean, median = median)))  
)
```

# mutate(across(where()))

```
mutate(  
  gapminder,  
  across(where(is.numeric), list(mean = mean, median = median)))  
)  
  
## # A tibble: 1,704 × 14  
##   country   continent year lifeExp      pop  
##   <fct>     <fct>    <int>   <dbl>    <int>  
## 1 Afghanistan Asia      1952    28.8  8425333  
## 2 Afghanistan Asia      1957    30.3  9240934  
## 3 Afghanistan Asia      1962    32.0  10267083  
## 4 Afghanistan Asia      1967    34.0  11537966  
## 5 Afghanistan Asia      1972    36.1  13079460  
## 6 Afghanistan Asia      1977    38.4  14880372  
## 7 Afghanistan Asia      1982    39.9  12881816  
## 8 Afghanistan Asia      1987    40.8  13867957  
## 9 Afghanistan Asia      1992    41.7  16317921  
## 10 Afghanistan Asia     1997    41.8  22227415  
## # ... with 1,694 more rows, and 9 more variables:  
## #   gdpPercap <dbl>, year_mean <dbl>,  
## #   year_median <dbl>, lifeExp_mean <dbl>,  
## #   lifeExp_median <dbl>, pop_mean <dbl>,  
## #   pop_median <dbl>, gdpPercap_mean <dbl>,  
## #   gdpPercap_median <dbl>
```

# Review: tidyselect

**'Behind the scenes' workhorse for dplyr::select(),  
dplyr::pull(), and tidyr::pivot\_ functions**

starts\_with(), ends\_with(), contains(), matches(), etc.

```
# column names contain a word  
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  
## # ... with 393 more rows
```

```
# regular expression  
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  
## # ... with 393 more rows
```

```
gapminder %>%
  group_by(continent) %>%
  summarise(across(c("lifeExp", "gdpPercap"), list(med = median,
    iqr = IQR)))
```

```
gapminder %>%
  group_by(continent) %>%
  summarise(across(c("lifeExp", "gdpPercap"), list(med = median,
    iqr = IQR)))
```

## # 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.        |
| ## # ... with 1 more variable: |           | gdpPercap_iqr | <dbl>       |               |

# mutate(across()) & summarise()

## Control output names with .names argument

```
gapminder %>%  
  group_by(continent) %>%  
  summarise(across(c("lifeExp", "gdpPercap"), list(med = median,  
    iqr = IQR) .names = "{.fn}_{.col}"))
```

```
## # 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.  
## # ... with 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. Name the new columns bp\_ + mean**

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

# Your Turn 2

```
diabetes %>%
  group_by(gender) %>%
  summarise(across(starts_with("bp"), list(mean = mean),
    na.rm = TRUE, .names = "{.col}_{.fn}"))
```

```
## # A tibble: 2 × 5
##   gender bp.1s_mean bp.1d_mean bp.2s_mean
##   <chr>     <dbl>      <dbl>      <dbl>
## 1 female     136.       82.5       153.
## 2 male       138.       84.5       151.
## # ... with 1 more variable: bp.2d_mean <dbl>
```

# vectorized functions don't work on lists

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

# vectorized functions don't work on lists

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

```
## [1] -3.831574
```

# vectorized functions don't work on lists

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

```
## [1] -3.831574
```

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

# vectorized functions don't work on lists

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

```
## [1] -3.831574
```

```
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
```

# `map(.x, .f)`

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

`.f`: a function

Returns a list



# Using map()

```
library(purrr)
x_list <- list(x = rnorm(10), y = rnorm(10), z = rnorm(10))

map(x_list, mean)
```

# Using map()

```
library(purrr)
x_list <- list(x = rnorm(10), y = rnorm(10), z = rnorm(10))
map(x_list, mean)
```

# Using map()

```
library(purrr)
x_list <- list(x = rnorm(10), y = rnorm(10), z = rnorm(10))

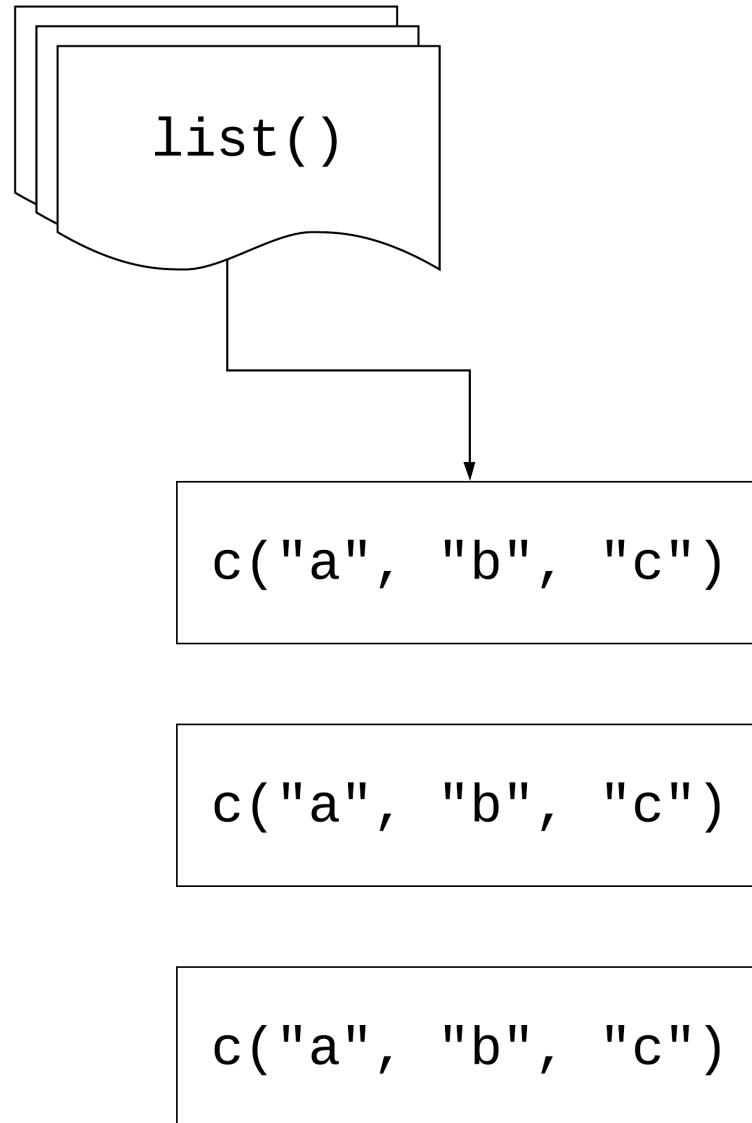
map(x_list, mean)
```

# Using map()

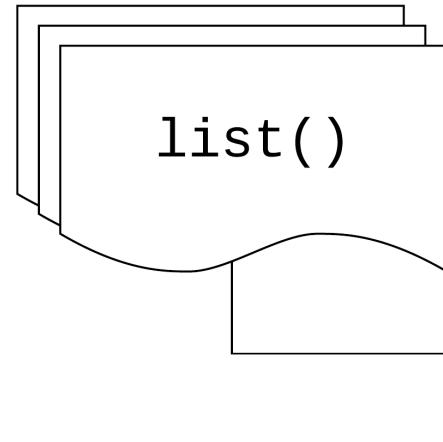
```
library(purrr)
x_list <- list(x = rnorm(10), y = rnorm(10), z = rnorm(10))

map(x_list, mean)
```

```
## $x
## [1] -0.6097971
##
## $y
## [1] -0.2788647
##
## $z
## [1] 0.6165922
```



`map(`



`, . 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?**

```
list(  
  blood_glucose = sum(measurements$blood_glucose),  
  age = sum(measurements$age),  
  heartrate = sum(measurements$heartrate)  
)
```

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

# Your Turn 3

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

```
## $blood_glucose  
## [1] 1361.684  
##  
## $age  
## [1] 193.8606  
##  
## $heartrate  
## [1] 1509.304
```

# using map() with data frames

```
library(dplyr)
gapminder %>%
  select(where(is.numeric)) %>%
  map(sd)
```

# using map() with data frames

```
library(dplyr)
gapminder %>%
  select(where(is.numeric)) %>%
  map(sd)
```

# using `map()` with data frames

```
library(dplyr)
gapminder %>%
  select(where(is.numeric)) %>%
  map(sd)
```

# using map() with data frames

```
library(dplyr)
gapminder %>%
  select(where(is.numeric)) %>%
  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

```
head(  
  map(diabetes, class),  
  3  
)
```

```
## $id  
## [1] "numeric"  
##  
## $chol  
## [1] "numeric"  
##  
## $stab.glu  
## [1] "numeric"
```

# Review: writing functions

```
x <- x^2  
x <- scale(x)  
x <- max(x)
```

# Review: writing functions

```
x <- x^2  
x <- scale(x)  
x <- max(x)
```

```
y <- x^2  
y <- scale(y)  
y <- max(y)
```

```
z <- z^2  
z <- scale(x)  
z <- max(z)
```

# Review: writing functions

```
x <- x^2  
x <- scale(x)  
x <- max(x)
```

```
y <- x^2  
y <- scale(y)  
y <- max(y)
```

```
z <- z^2  
z <- scale(x)  
z <- max(z)
```

# Review: writing functions

```
x <- x^3  
x <- scale(x)  
x <- max(x)
```

```
y <- x^2  
y <- scale(y)  
y <- max(y)
```

```
z <- z^2  
z <- scale(x)  
z <- max(z)
```

# Review: writing functions

```
.f <- function(x) {  
  x <- x^3  
  x <- scale(x)  
  
  max(x)  
}  
  
.f(x)  
.f(y)  
.f(z)
```

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

# 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

```
mean_sd <- function(x) {  
  x_mean <- mean(x)  
  x_sd <- sd(x)  
  tibble(mean = x_mean, sd = x_sd)  
}  
  
map(measurements, mean_sd)
```

# Your Turn 5

```
## $blood_glucose
## # A tibble: 1 × 2
##   mean     sd
##   <dbl> <dbl>
## 1 136.  9.96
##
## $age
## # A tibble: 1 × 2
##   mean     sd
##   <dbl> <dbl>
## 1 38.8  3.91
##
## $heartrate
## # A tibble: 1 × 2
##   mean     sd
##   <dbl> <dbl>
## 1 75.5 13.8
```

# Three ways to pass functions to map()

- 1 pass directly to map()
- 2 use an anonymous function
- 3 use ~

```
map(  
  .x,  
  mean,  
  na.rm = TRUE  
)
```

```
map(  
  .x,  
  function(.x) {  
    mean(.x,  
      na.rm = TRUE)  
  }  
)
```

```
map(  
  .x,  
  ~mean(.x,  
  na.rm = TRUE)  
)
```

```
map(gapminder, ~length(unique(.x)))
```

```
map(gapminder, ~length(unique(.x)))
```

```
## $country
## [1] 142
##
## $continent
## [1] 5
##
## $year
## [1] 12
##
## $lifeExp
## [1] 1626
##
## $pop
## [1] 1704
##
## $gdpPercap
## [1] 1704
```

# Returning types

| map       | returns                 |
|-----------|-------------------------|
| map()     | list                    |
| map_chr() | character vector        |
| map_dbl() | double vector (numeric) |
| map_int() | integer vector          |
| map_lgl() | logical vector          |
| map_dfc() | data frame (by column)  |
| map_dfr() | data frame (by row)     |

# Returning types

```
map_int(gapminder, ~length(unique(.x)))
```

# Returning types

```
map_int(gapminder, ~length(unique(.x)))
```

```
## #> #> #> #>
```

|    | country | continent | year | lifeExp |
|----|---------|-----------|------|---------|
| ## | 142     | 5         | 12   | 1626    |
| ## | pop     | gdpPercap |      |         |
| ## | 1704    | 1704      |      |         |

# Your Turn 6

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

# Your Turn 6

```
map_chr(diabetes, class)

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

# Your Turn 7

**Check diabetes for any missing data.**

**Using the `~.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

```
map_lgl(diabetes, ~any(is.na(.x)))  
  
##      id      chol stab.glu      hdl      ratio  
## FALSE TRUE FALSE TRUE TRUE  
## glyhb location      age gender height  
## TRUE FALSE FALSE FALSE TRUE  
## weight frame bp.1s bp.1d bp.2s  
## TRUE TRUE TRUE TRUE TRUE  
## bp.2d waist hip time.ppn  
## TRUE TRUE TRUE TRUE
```

# Your Turn 7

```
map_int(diabetes, ~sum(is.na(.x)))  
  
##      id      chol stab.glu      hdl      ratio  
##      0       1       0       1       1  
##  glyhb location      age   gender      height  
##      13      0       0       0       5  
## weight     frame    bp.1s    bp.1d    bp.2s  
##      1      12       5       5      262  
##  bp.2d     waist     hip time.ppn  
##      262      2       2       3
```

# group\_map()

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

```
diabetes %>%  
  group_by(gender) %>%  
  group_map(~ broom::tidy(lm(weight ~ height, data = .x)))
```

# group\_map()

```
diabetes %>%
  group_by(gender) %>%
  group_map(~ broom::tidy(lm(weight ~ height, data = .x)))
```

```
## [[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  2.14e-1
## 2 height       3.90      0.928      4.20  3.83e-5
## 
```

```
## [[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 4.71e-1
## 2 height       3.35      0.995      3.37  9.45e-4
```

# group\_modify()

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

```
diabetes %>%  
  group_by(gender) %>%  
  group_modify(~ broom::tidy(lm(weight ~ height, data = .x)))
```

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

# 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

```
model_lm <- function(.data, ...) {  
  mdl <- lm(chol ~ ratio, data = .data)  
  # get model statistics  
  broom::glance(mdl)  
}  
  
diabetes %>%  
  group_by(location) %>%  
  group_modify(model_lm)
```

# Your Turn 8

```
## # A tibble: 2 × 13
## # Groups:   location [2]
##   location    r.squared adj.r.squared sigma
##   <chr>        <dbl>          <dbl> <dbl>
## 1 Buckingham  0.252          0.248 38.8
## 2 Louisa      0.204          0.201 39.4
## # ... with 9 more variables: statistic <dbl>,
## #   p.value <dbl>, df <dbl>, logLik <dbl>,
## #   AIC <dbl>, BIC <dbl>, deviance <dbl>,
## #   df.residual <int>, nobs <int>
```

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

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

`.f`: a function

Returns a list

map2(



,

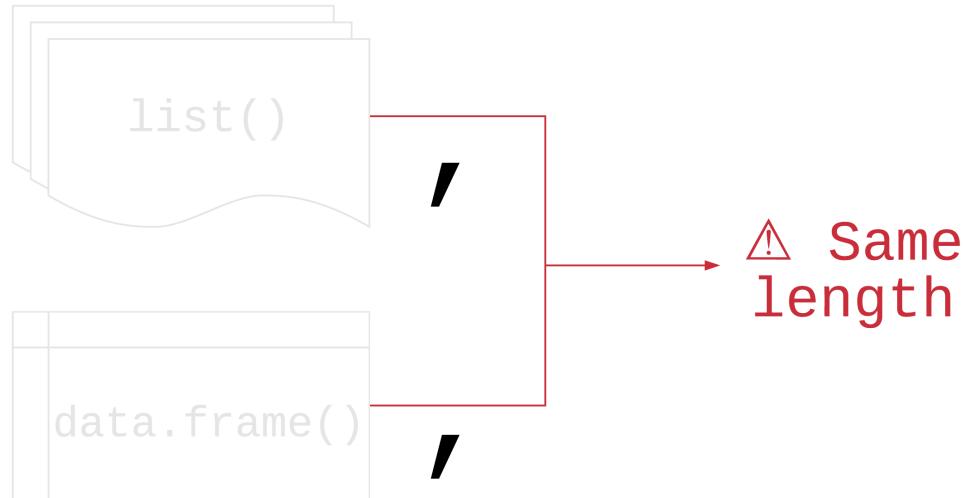


,

.f

)

# map2(



.f  
)

map2(



,



,

~ .f( .x, .y )

)

# map2()

```
means <- c(-3, 4, 2, 2.3)
sds <- c(.3, 4, 2, 1)

map2_dbl(means, sds, rnorm, n = 1)
```

# map2()

```
means <- c(-3, 4, 2, 2.3)
sds <- c(.3, 4, 2, 1)

map2_db1(means, sds, rnorm, n = 1)
```

# map2()

```
means <- c(-3, 4, 2, 2.3)
sds <- c(.3, 4, 2, 1)

map2_dbl(means, sds, rnorm, n = 1)
```

```
## [1] -2.997932  2.178125  1.266952  2.948287
```

# 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 ~.f() 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 ~.f().**

# Your Turn 9

```
gapminder_countries <- split(gapminder, gapminder$country)
models <- map(gapminder_countries, ~ lm(lifeExp ~ year, data = .x))
preds <- map2(models, gapminder_countries, predict)
head(preds, 3)
```

# Your Turn 9

```
gapminder_countries <- split(gapminder, gapminder$country)
models <- map(gapminder_countries, ~ lm(lifeExp ~ year, data = .x))
preds <- map2(models, gapminder_countries, predict)
head(preds, 3)
```

# Your Turn 9

```
gapminder_countries <- split(gapminder, gapminder$country)
models <- map(gapminder_countries, ~ lm(lifeExp ~ year, data = .x))
preds <- map2(models, gapminder_countries, predict)
head(preds, 3)
```

# Your Turn 9

```
## $Afghanistan
##      1      2      3      4      5
## 29.90729 31.28394 32.66058 34.03722 35.41387
##      6
## 36.79051
##
## $Albania
##      1      2      3      4      5
## 59.22913 60.90254 62.57596 64.24938 65.92279
##      6
## 67.59621
##
## $Algeria
##      1      2      3      4      5
## 43.37497 46.22137 49.06777 51.91417 54.76057
##      6
## 57.60697
```

| <b>input 1</b> | <b>input 2</b> | <b>returns</b>          |
|----------------|----------------|-------------------------|
| 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

# 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

# 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

# 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

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

# group\_walk()

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

```
temp <- "temporary_folder"  
fs::dir_create(temp)  
gapminder %>%  
  group_by(continent) %>%  
  group_walk(  
    ~ write_csv(  
      .x,  
      file = file.path(temp,  paste0(.y$continent,  ".csv"))  
    )  
  )
```

# group\_walk()

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

```
## temporary_folder  
## └── Africa.csv  
## └── Americas.csv  
## └── Asia.csv  
## └── Europe.csv  
## └── Oceania.csv
```

# 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

```
fs::dir_create("figures")

ggsave_gapminder <- function(variable) {
  # we're using `aes_string()` so we don't need the
  # curly-curly syntax
  p <- ggplot(
    gapminder,
    aes_string(x = "year", y = variable, color = "country")
  ) +
    geom_line() +
    scale_color_manual(values = country_colors) +
    facet_wrap(vars(continent)) +
    theme(legend.position = "none")

  ggsave(
    filename = paste0("figures/", variable, ".png"),
    plot = p,
    dpi = 320
  )
}
```

# Your turn 10

```
vars <- c("lifeExp", "pop", "gdpPercap")
walk(vars, ggsave_gapminder)
```

# Base R

| base R          | purrr          |
|-----------------|----------------|
| lapply()        | map()          |
| vapply()        | map_*          |
| sapply()        | ?              |
| x[] <- lapply() | map_dfc()      |
| mapply()        | map2(), pmap() |

# Benefits of purrr

- 1 Consistent
- 2 Type-safe
- 3  $\sim f(.x)$

# Loops vs functional programming

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

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

# Loops vs functional programming

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

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

# Loops vs functional programming

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

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

# Loops vs functional programming

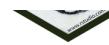
```
x <- rnorm(10)
y <- map(x, mean)
```

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

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

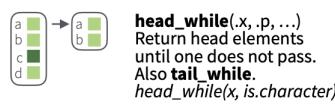
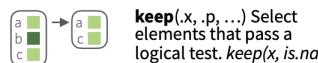
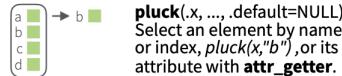
**—Jenny Bryan**

# Working with lists and nested data

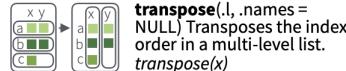


## Work with Lists

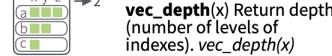
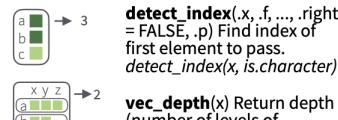
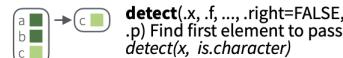
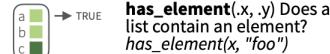
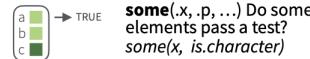
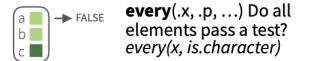
### FILTER LISTS



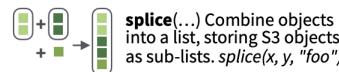
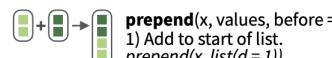
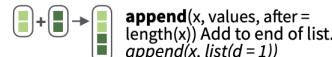
### RESHAPE LISTS



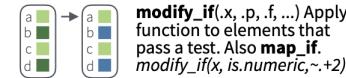
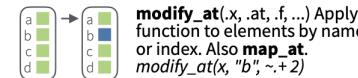
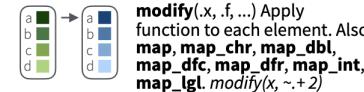
### SUMMARISE LISTS



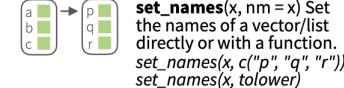
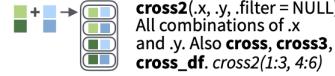
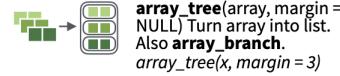
### JOIN (TO) LISTS



### TRANSFORM LISTS



### WORK WITH LISTS



# 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            | <ibble [50 x 4]> |
| versicolor        | <ibble [50 x 4]> |
| virginica         | <ibble [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 |

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

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

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

`tibble::tribble(...)`

`tibble::tibble(...)`

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

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

| Species | data              | model                                       |
|---------|-------------------|---|
| setosa  | <tibble [50 x 4]> | Call: lm(S.L ~ ., df)                       |
| versi   | <tibble [50 x 4]> | Coefs: (int) S.W P.L P.W<br>2.3 0.6 0.2 0.2 |
| virgini | <tibble [50 x 4]> | Call: lm(S.L ~ ., df)                       |

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

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



### 3 Simplify the list column

| Species | beta |
|---------|------|
| setos   | 2.35 |
| versi   | 1.89 |
| virgini | 0.69 |

# 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\_ld**, **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).

# Learn more!

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

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

**RStudio Primers**: Free interactive courses in the Tidyverse