COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

Data Science for Business Analytics

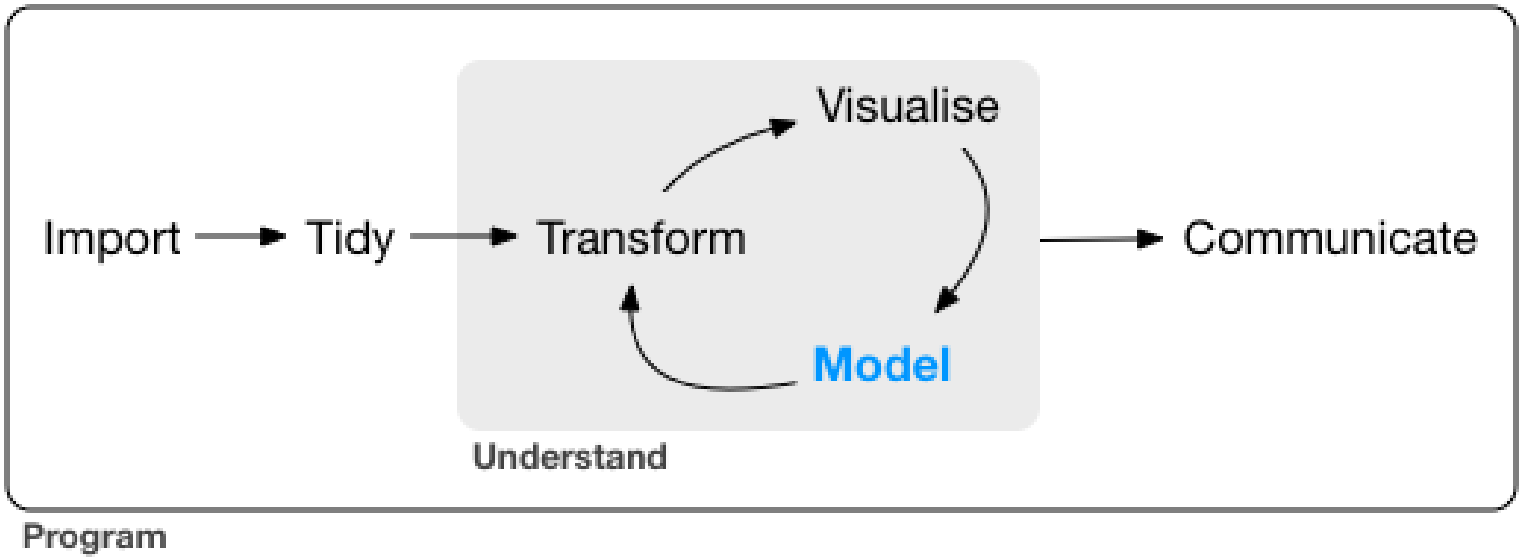
Lecture 7

Thibault Vatter

Department of Statistics, Columbia University

04/06/2020

Today



- First (last time):
  - how models work mechanistically (focus on linear models),
  - how to use models to find patterns in real data.
- Then (today):
  - how to use **many** simple models,
  - how to combine modeling and programming tools.
- As usual, material borrowed from R for data science.

Many models

- To work with large numbers of models, use:
  - The **broom** package to turn models into tidy data.
  - Many simple models to better understand complex datasets.
  - List-columns to store arbitrary data structures in a data frame.
- Note that this part
  - is harder than the others,
  - requires a deeper internalization of ideas (e.g., about modeling, data structures, and iteration).

Outline

Outline

Bookdown

DEMO!

- Additional resources:
  - The book on bookdown
  - Introducing bookdown (webinar)

Outline

Tidying models with broom

- `broom::glance(model)`
  - A row for each model.
  - Columns give a model summary (measure of model quality, complexity, or combination of both).
- `broom::tidy(model)`
  - A row for each coefficient in the model.
  - Columns give information about the estimate or its variability.
- `broom::augment(model, data)`
  - A row for each row in `data`.
  - Adds extra values like residuals, and influence statistics.

```
## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## Page 9: Plant Weight Data.
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2, 10, 20, labels = c("Ctl","Trt"))
weight <- c(ctl, trt)
lm_D9 <- lm(weight ~ group)
```

broom::glance() and broom::tidy()

- broom::glance(model)
  - A row for each model.
  - Columns give a model summary.

```
glance(lm_D9)
#> # A tibble: 1 x 11
#>   r.squared adj.r.squared sigma statistic p.value    df logLik   AIC
#>       <dbl>         <dbl> <dbl>     <dbl>   <dbl> <int>  <dbl> <dbl>
#> 1    0.0731        0.0216 0.696      1.42   0.249     2  -20.1  46.2
#> # ... with 3 more variables: BIC <dbl>, deviance <dbl>,
#> #   df.residual <int>
```

- broom::tidy(model)
  - A row for each coefficient in the model.
  - Columns give information about the estimate or its variability.

```
tidy(lm_D9)
#> # A tibble: 2 x 5
#>   term        estimate std.error statistic  p.value
#>   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
#> 1 (Intercept)    5.03      0.220     22.9  9.55e-15
#> 2 groupTrt      -0.371     0.311     -1.19 2.49e- 1
```

broom::augment()

- broom::augment(model, data)
  - A row for each row in data.
  - Adds extra values like residuals, and influence statistics.

```
augment(lm_D9) %>%
  print(n = 10)
#> # A tibble: 20 x 9
#>    weight group .fitted .se.fit .resid  .hat .sigma .cooksd
#>     <dbl> <fct>   <dbl>   <dbl>  <dbl> <dbl>  <dbl>   <dbl>
#>  1   4.17 Ctl      5.03   0.220 -0.862  0.10  0.682 0.0946
#>  2   5.58 Ctl      5.03   0.220  0.548  0.10  0.703 0.0382
#>  3   5.18 Ctl      5.03   0.220  0.148  0.1   0.716 0.00279
#>  4   6.11 Ctl      5.03   0.220  1.08   0.1   0.661 0.148
#>  5   4.5  Ctl      5.03   0.220 -0.532  0.1   0.704 0.0360
#>  6   4.61 Ctl      5.03   0.220 -0.422  0.1   0.708 0.0227
#>  7   5.17 Ctl      5.03   0.220  0.138  0.1   0.716 0.00242
#>  8   4.53 Ctl      5.03   0.220 -0.502  0.1   0.705 0.0321
#>  9   5.33 Ctl      5.03   0.220  0.298  0.1   0.713 0.0113
#> 10   5.14 Ctl      5.03   0.220  0.108  0.1   0.716 0.00148
#> # ... with 10 more rows, and 1 more variable: .std.resid <dbl>
```

Outline

Reducing code duplication

- Three main benefits:
  - Easier to see the intent of your code.
  - Easier to respond to changes in requirements.
  - Likely to have fewer bugs.
- Main tools for reducing duplication:
  - **Functions** identify repeated patterns of code and extract them out into independent pieces.
  - **Iteration** helps you when you need to do the same thing to multiple inputs.

For loops


```
df <- tibble(a = rnorm(10),
             b = rnorm(10),
             c = rnorm(10),
             d = rnorm(10))

c(median(df$a), median(df$b), median(df$c), median(df$d))
#> [1] -0.2458 -0.2873 -0.0567  0.1443

output <- vector("double", ncol(df))  # 1. output
for (i in seq_along(df)) {            # 2. sequence
  output[[i]] <- median(df[[i]])      # 3. body
}
output
#> [1] -0.2458 -0.2873 -0.0567  0.1443
```

- A side note:

```
y <- vector("double", 0)
seq_along(y)
#> integer(0)
1:length(y)
#> [1] 1 0
```

Modifying an existing object


```
df <- tibble(a = rnorm(10),
             b = rnorm(10),
             c = rnorm(10),
             d = rnorm(10))

rescale01 <- function(x) {
    rng <- range(x, na.rm = TRUE)
    (x - rng[1]) / (rng[2] - rng[1])
}

df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)

for (i in seq_along(df)) {
    df[[i]] <- rescale01(df[[i]])
}
```

For loops vs. functionals

- To compute the mean of every column:

```
output <- vector("double", length(df))
for (i in seq_along(df)) {
  output[[i]] <- mean(df[[i]])
}
```

- As a function:

```
col_mean <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- mean(df[[i]])
  }
  output
}
```

How about other quantities?


```
col_median <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- median(df[[i]])
  }
  output
}
col_sd <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- sd(df[[i]])
  }
  output
}
```

- What's "wrong" here?

What's "wrong" here?


```
f1 <- function(x) abs(x - mean(x)) ^ 1
f2 <- function(x) abs(x - mean(x)) ^ 2
f3 <- function(x) abs(x - mean(x)) ^ 3
```

What's "wrong" here?


```
f1 <- function(x) abs(x - mean(x)) ^ 1
f2 <- function(x) abs(x - mean(x)) ^ 2
f3 <- function(x) abs(x - mean(x)) ^ 3
```

- Without duplication:

```
f <- function(x, i) abs(x - mean(x)) ^ i
```

Back to column summaries

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

```
col_summary <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
    out[i] <- fun(df[[i]])
  }
  out
}
col_summary(df, median)
#> [1] 0.422 0.464 0.584 0.616
col_summary(df, mean)
#> [1] 0.434 0.484 0.572 0.624
```

Functional programming

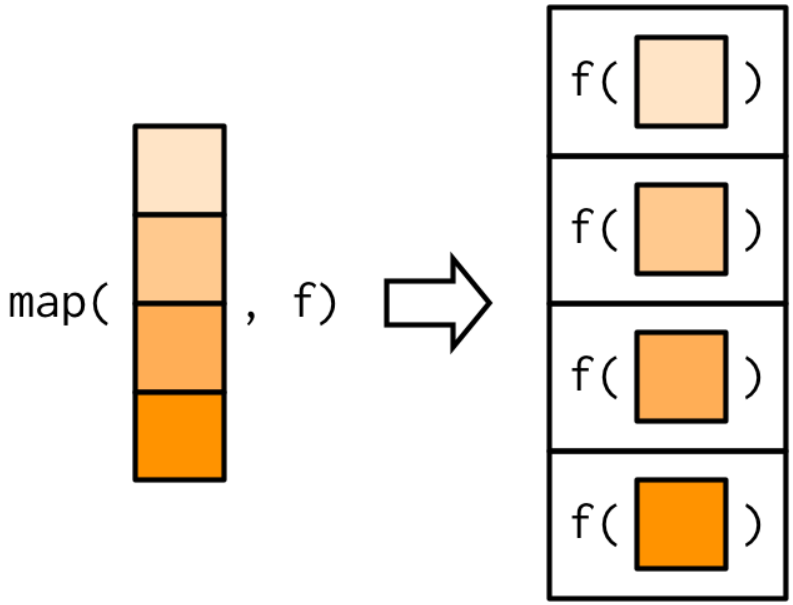
- Functional programming:
  - Uses **functions that return functions** as output.
  - Passes functions as arguments to others function.
  - Much more in the Advanced-R book chapter on FP
- The **purrr** package:
  - Functions eliminating the need for many common for loops.
  - Similar to the apply family in base R (`apply()`, `lapply()`, `tapply()`, etc), but more consistent and easier to learn.
- The goal is to break code into independent pieces:
  - Solve a problem for a single element of the list.
    - Once this is done, `purrr` generalizes to every element in the list.
  - Break a complex problem down into bite-sized pieces.
    - With `purrr`, small pieces are composed together with the pipe.

The map functions

- The basic map functions
  - `map()` makes a list.
  - `map_lgl()` makes a logical vector.
  - `map_int()` makes an integer vector.
  - `map_dbl()` makes a double vector.
  - `map_chr()` makes a character vector.
- Each function:
  - Takes a vector as input.
  - Applies a function to each piece.
  - Returns a new vector that's the same length (and has the same names) as the input.
- The return type is determined by the suffix.

Using map functions


```
map_dbl(df, mean)
#>     a     b     c     d
#> 0.434 0.484 0.572 0.624
map_dbl(df, median)
#>     a     b     c     d
#> 0.422 0.464 0.584 0.616
map_dbl(df, sd)
#>     a     b     c     d
#> 0.249 0.247 0.364 0.284
```



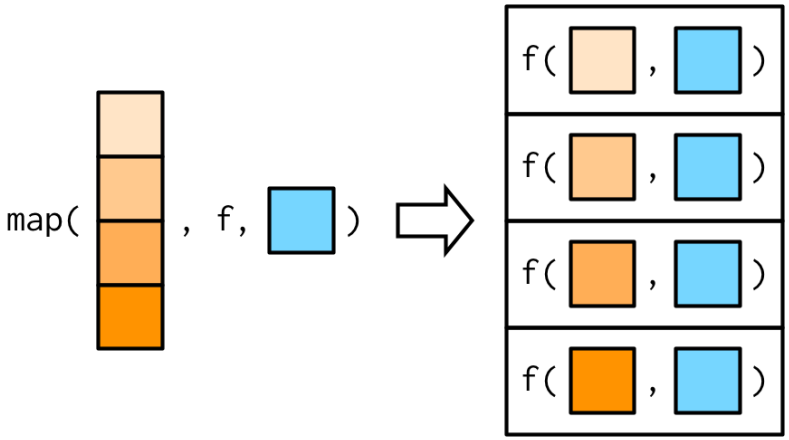
- Using the pipe:

```
df %>% map_dbl(mean)
#>     a     b     c     d
#> 0.434 0.484 0.572 0.624
df %>% map_dbl(median)
#>     a     b     c     d
#> 0.422 0.464 0.584 0.616
df %>% map_dbl(sd)
#>     a     b     c     d
#> 0.249 0.247 0.364 0.284
```

map_*() vs col_summary()

- All purrr functions are implemented in C (i.e., slightly faster).
- The second argument, `.f`, the function to apply, can be a formula, a character vector, or an integer vector.
- `map_*()` uses `...` ([dot dot dot]) to pass along additional arguments to `.f` each time it's called:

```
map_dbl(df, mean, trim = 0.5)
#>     a     b     c     d
#> 0.422 0.464 0.584 0.616
```



- The map functions preserve names:

```
z <- list(x = 1:3, y = 4:5)
map_int(z, length)
#> x y
#> 3 2
```

Shortcut 1

- Splits the `mtcars` dataset into three pieces and fits the same linear model to each piece:

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(function(df) lm(mpg ~ wt, data = df))
```

- Using a one-sided formula:

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(~lm(mpg ~ wt, data = .))
```

Shortcut 2

- Extract a summary statistic like the R^2:

```
models %>%
  map(summary) %>%
  map_dbl(~.$r.squared)
#>     4     6     8
#> 0.509 0.465 0.423
```

- Use a string:

```
models %>%
  map(summary) %>%
  map_dbl("r.squared")
#>     4     6     8
#> 0.509 0.465 0.423
```

Shortcut 3

- Use an integer:

```
x <- list(list(1, 2, 3), list(4, 5, 6), list(7, 8, 9))
x %>% map_dbl(2)
#> [1] 2 5 8
```

Dealing with failure using `safely()`

- `safely()` is an adverb: it takes a function (a verb) and returns a modified version.
- The modified function always returns a list with two elements:
  - `result` is the original result.
  - `error` is an error object.

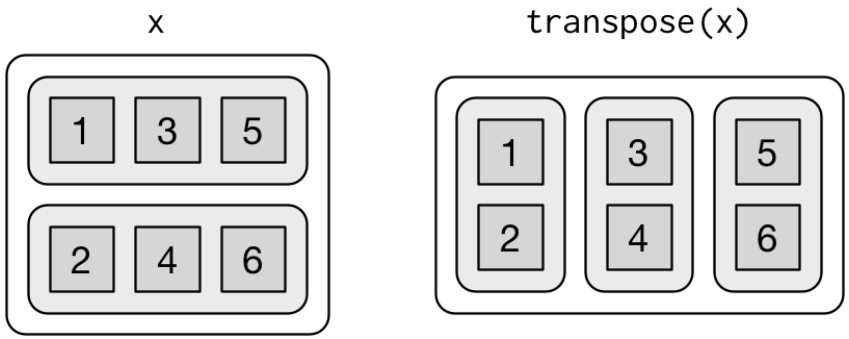
```
safe_log <- safely(log)
str(safe_log(10))
#> List of 2
#>  $ result: num 2.3
#>  $ error : NULL
str(safe_log("a"))
#> List of 2
#>  $ result: NULL
#>  $ error :List of 2
#>   ..$ message: chr "non-numeric argument to mathematical function"
#>   ..$ call   : language .Primitive("log")(x, base)
#>   ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

safely() and map()


```
x <- list(1, 10, "a")
y <- x %>% map(safely(log))
str(y)
#> List of 3
#>  $ :List of 2
#>   ..$ result: num 0
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: num 2.3
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: NULL
#>   ..$ error :List of 2
#>   .. ..$ message: chr "non-numeric argument to mathematical function"
#>   .. ..$ call   : language .Primitive("log")(x, base)
#>   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condit"..
```

transpose()


```
y <- y %>% transpose()
str(y)
#> List of 2
#>  $ result:List of 3
#>   ..$ : num 0
#>   ..$ : num 2.3
#>   ..$ : NULL
#>  $ error :List of 3
#>   ..$ : NULL
#>   ..$ : NULL
#>   ..$ :List of 2
#>   .. ..$ message: chr "non-numeric argument to mathematical function"
#>   .. ..$ call   : language .Primitive("log")(x, base)
#>   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condit"..
```



Typical use


```
is_ok <- y$error %>% map_lgl(is_null)
x[!is_ok]
#> [[1]]
#> [1] "a"
y$result[is_ok] %>% flatten_dbl()
#> [1] 0.0 2.3
```

Two other useful adverbs

- `possibly()`: "simpler" than `safely()` (uses a default value when there is an error)

```
list(1, 10, "a") %>% map_dbl(possibly(log, NA))
#> [1] 0.0 2.3  NA
```

- `quietly()`: captures printed output, messages, and warnings.

```
list(1, -1) %>% map(quietly(log)) %>% str()
#> List of 2
#>  $ :List of 4
#>   ..$ result  : num 0
#>   ..$ output  : chr ""
#>   ..$ warnings: chr(0)
#>   ..$ messages: chr(0)
#>  $ :List of 4
#>   ..$ result  : num NaN
#>   ..$ output  : chr ""
#>   ..$ warnings: chr "NaNs produced"
#>   ..$ messages: chr(0)
```

Mapping over multiple arguments

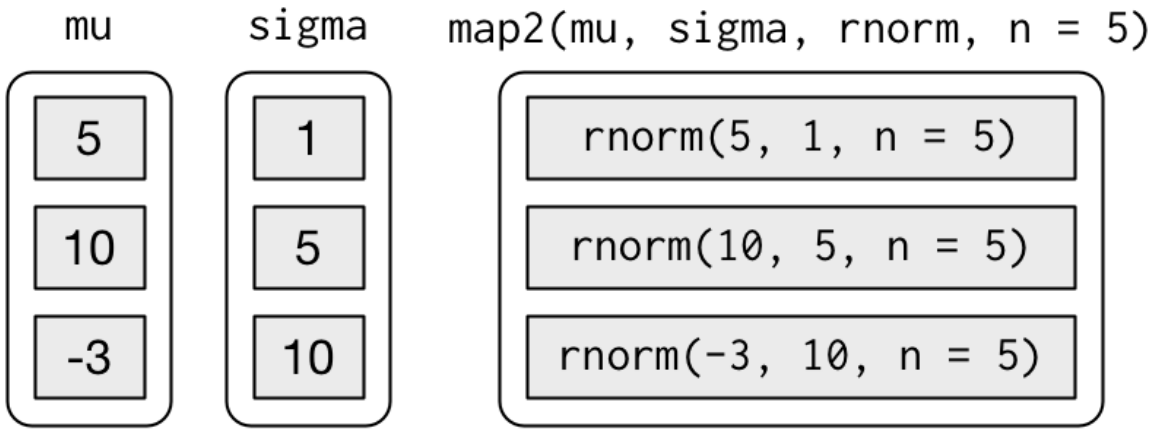

```
mu <- list(5, 10, -3)
mu %>% map(rnorm, n = 5) %>% str()
#> List of 3
#>  $ : num [1:5] 6.92 6.3 5.75 5.56 4.45
#>  $ : num [1:5] 11.11 7.39 9.84 10.43 9.62
#>  $ : num [1:5] -2.58 -1.94 -1.95 -3.04 -2.51
```

What if you also want to vary the standard deviation?

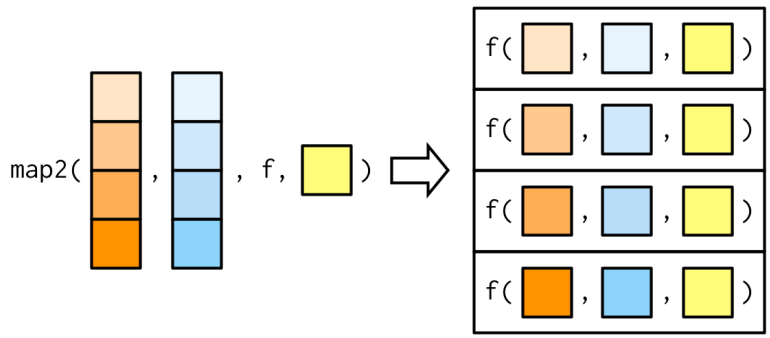
```
sigma <- list(1, 5, 10)
seq_along(mu) %>% map(~rnorm(5, mu[[.]], sigma[[.]])) %>% str()
#> List of 3
#>  $ : num [1:5] 6.67 4.65 5.95 6.32 4.7
#>  $ : num [1:5] 8.06 6.07 4.72 6.02 1.22
#>  $ : num [1:5] -9.905 -8.585 -8.367 -0.729 6.785
```

map2()


```
map2(mu, sigma, rnorm, n = 5) %>% str()
#> List of 3
#>  $ : num [1:5] 4.79 3.6 5.26 4.56 5.57
#>  $ : num [1:5] 20.63 12.12 1.58 11.25 15.36
#>  $ : num [1:5] 17.39 1.49 10.92 1.27 -1.92
```

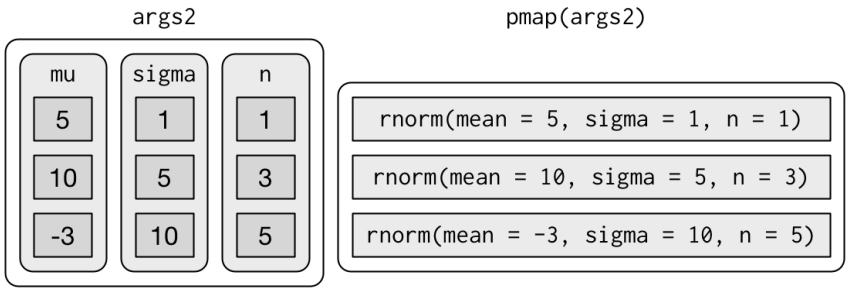


- Note that the arguments that
  - vary for each call come *before* the function,
  - are the same for every call come *after* (using ...).

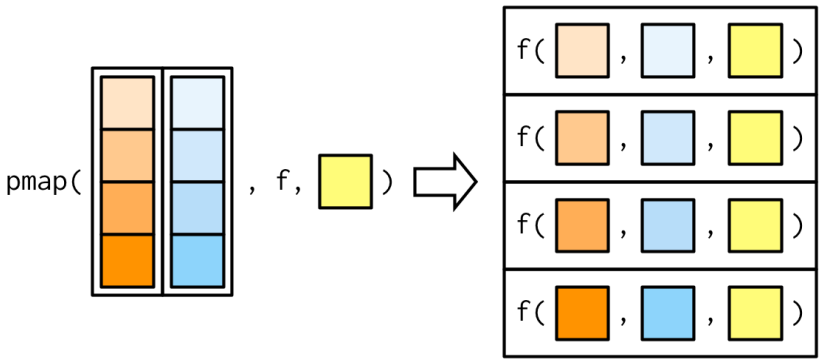


pmap()


```
list(mean = mu, sd = sigma, n = list(1, 3, 5)) %>% pmap(rnorm) %>% str()
#> List of 3
#>  $ : num 5.02
#>  $ : num [1:3] 13.02 8.69 7.36
#>  $ : num [1:5] -1.08 -14.46 5.46 -2.18 -16.05
```



- Also works with additional arguments:



Map variants

- 23 primary variants of `map()`:
  - `map()`, `map_dbl()`, `map_chr()`, `map_int()`, `map_lgl()`
  - 18 (!!) more to learn.
  - Five new ideas:
    - Output same type as input with `modify()`
    - Iterate over two inputs with `map2()`.
    - Iterate with an index using `imap()`
    - Return nothing with `walk()`.
    - Iterate over any number of inputs with `pmap()`.
- More details in the book!

	List	Atomic	Same type	Nothing
One argument	`map()`	`map_lgl()`, ...	`modify()`	`walk()`
Two arguments	`map2()`	`map2_lgl()`, ...	`modify2()`	`walk2()`
One argument + index	`imap()`	`imap_lgl()`, ...	`imodify()`	`iwalk()`
N arguments	`pmap()`	`pmap_lgl()`, ...	—	`pwalk()`

Outline

List-columns

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

```
tibble(x = list(1:3, 3:5),
       y = c("1, 2", "3, 4, 5"))
#> # A tibble: 2 x 2
#>   x         y
#>   <list>    <chr>
#> 1 <int [3]> 1, 2
#> 2 <int [3]> 3, 4, 5

tribble(~x, ~y,
        1:3, "1, 2",
        3:5, "3, 4, 5")
#> # A tibble: 2 x 2
#>   x         y
#>   <list>    <chr>
#> 1 <int [3]> 1, 2
#> 2 <int [3]> 3, 4, 5
```

An effective list-column pipeline

- Create the list-column:
  - With `nest()` to convert a grouped data frame into a nested data frame where you have list-column of data frames.
  - With `mutate()` and vectorised functions that return a list.
  - With `summarize()` and summary functions that return multiple results.
- Create other intermediate list-columns by transforming existing list columns with `map()`, `map2()` or `pmap()`.
- Simplify the list-column back down to a data frame or atomic vector.

Create with nesting


```
mtcars %>%
  group_by(cyl) %>%
  nest()
#> # A tibble: 3 x 2
#> # Groups:   cyl [3]
#>     cyl data
#>   <dbl> <list>
#> 1     6 <tibble [7 x 10]>
#> 2     4 <tibble [11 x 10]>
#> 3     8 <tibble [14 x 10]>
```

Create from multivalued summaries I

- What's wrong here?

```
mtcars %>%
  group_by(cyl) %>%
  summarize(q = quantile(mpg))
#> Error: Column `q` must be length 1 (a summary value), not 5
```

- Use list-columns:

```
mtcars %>%
  group_by(cyl) %>%
  summarize(q = list(quantile(mpg)))
#> # A tibble: 3 x 2
#>     cyl q
#>   <dbl> <list>
#> 1     4 <dbl [5]>
#> 2     6 <dbl [5]>
#> 3     8 <dbl [5]>
```

Create from multivalued summaries II


```
probs <- c(0.01, 0.25, 0.5, 0.75, 0.99)
mtcars %>%
  group_by(cyl) %>%
  summarize(p = list(probs),
            q = list(quantile(mpg, probs))) %>%
  unnest(cols = c(p, q)) %>%
  print(n = 7)
#> # A tibble: 15 x 3
#>     cyl     p     q
#>   <dbl> <dbl> <dbl>
#> 1     4  0.01  21.4
#> 2     4  0.25  22.8
#> 3     4  0.5   26
#> 4     4  0.75  30.4
#> 5     4  0.99  33.8
#> 6     6  0.01  17.8
#> 7     6  0.25  18.6
#> # ... with 8 more rows
```

Simplifying list-columns

- If you want a single value, use `mutate()` with `map_lgl()`, `map_int()`, `map_dbl()`, and `map_chr()` to create an atomic vector.
- If you want many values, use `unnest()` to convert list-columns back to regular columns, repeating the rows as many times as necessary.

List to vector

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

```
df <- tribble(~x, letters[1:5], 1:3, runif(5))

df %>%
  mutate(type = map_chr(x, typeof),
         length = map_int(x, length))
#> # A tibble: 3 x 3
#>   x         type      length
#>   <list>    <chr>      <int>
#> 1 <chr [5]> character      5
#> 2 <int [3]> integer        3
#> 3 <dbl [5]> double         5
```

Unnesting

- Columns with the same number of elements:

```
tibble(x = 1:2,
       y = list(1:4, 1)) %>%
  unnest(y)
#> # A tibble: 5 x 2
#>       x     y
#>   <int> <dbl>
#> 1     1     1
#> 2     1     2
#> 3     1     3
#> 4     1     4
#> 5     2     1
```

- Columns with different number of elements:

```
tribble(~x, ~y,         ~z,
        1, "a",          1:2,
        2, c("b", "c"), 3
        ) %>%
  unnest(c(y, z))
#> # A tibble: 4 x 3
#>       x y           z
#>   <dbl> <chr> <dbl>
#> 1     1 a           1
#> 2     1 a           2
#> 3     2 b           3
#> 4     2 c           3
```

Outline

gapminder

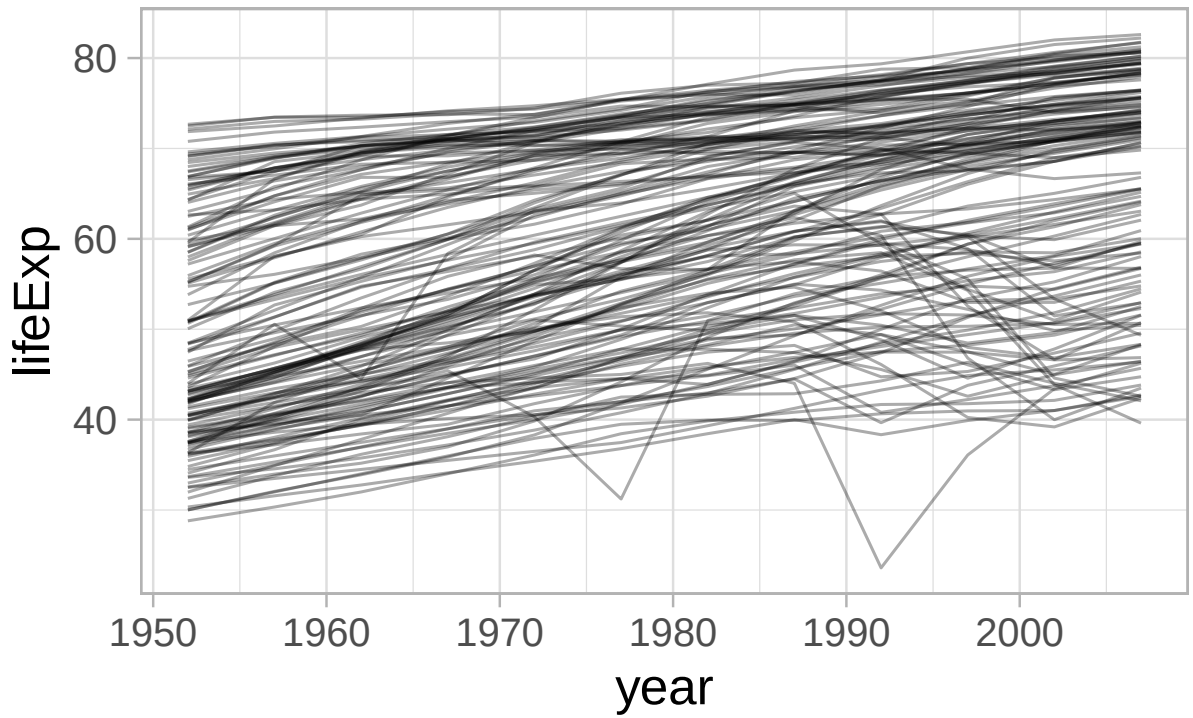
- Summarizes the progression of countries over time using variables like life expectancy and GDP.
- Popularized by Hans Rosling, a Swedish doctor and statistician, in a short video filmed in conjunction with the BBC

```
library(gapminder)
gapminder
#> # A tibble: 1,704 x 6
#>    country     continent  year lifeExp      pop gdpPercap
#>    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
#>  1 Afghanistan Asia       1952    28.8  8425333      779.
#>  2 Afghanistan Asia       1957    30.3  9240934      821.
#>  3 Afghanistan Asia       1962    32.0 10267083      853.
#>  4 Afghanistan Asia       1967    34.0 11537966      836.
#>  5 Afghanistan Asia       1972    36.1 13079460      740.
#>  6 Afghanistan Asia       1977    38.4 14880372      786.
#>  7 Afghanistan Asia       1982    39.9 12881816      978.
#>  8 Afghanistan Asia       1987    40.8 13867957      852.
#>  9 Afghanistan Asia       1992    41.7 16317921      649.
#> 10 Afghanistan Asia       1997    41.8 22227415      635.
#> # ... with 1,694 more rows
```

Focus on three variables

- How does life expectancy (`lifeExp`) change over time (`year`) for each country (`country`)?

```
gapminder %>%
  ggplot(aes(year, lifeExp, group = country)) +
    geom_line(alpha = 1/3)
```

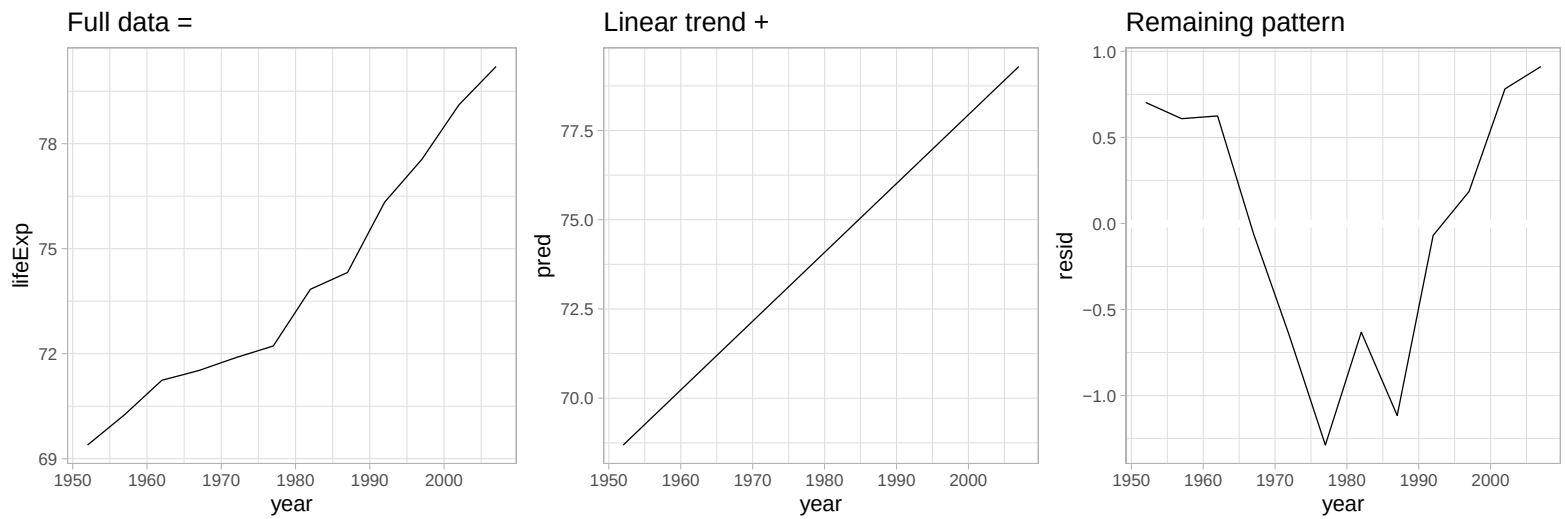


Model for a single country


```
nz <- filter(gapminder, country == "New Zealand")
nz %>% ggplot(aes(year, lifeExp)) + geom_line() + ggtitle("Full data = ")

nz_mod <- lm(lifeExp ~ year, data = nz)
nz %>% add_predictions(nz_mod) %>%
  ggplot(aes(year, pred)) + geom_line() + ggtitle("Linear trend + ")

nz %>% add_residuals(nz_mod) %>%
  ggplot(aes(year, resid)) +
    geom_hline(yintercept = 0, colour = "white", size = 3) +
    geom_line() + ggtitle("Remaining pattern")
```



Nested data


```
(by_country <- gapminder %>%
  group_by(country, continent) %>%
  nest())
#> # A tibble: 142 x 3
#> # Groups:   country, continent [710]
#>    country     continent data
#>    <fct>       <fct>     <list>
#>  1 Afghanistan Asia      <tibble [12 x 4]>
#>  2 Albania     Europe    <tibble [12 x 4]>
#>  3 Algeria     Africa    <tibble [12 x 4]>
#>  4 Angola      Africa    <tibble [12 x 4]>
#>  5 Argentina   Americas  <tibble [12 x 4]>
#>  6 Australia   Oceania   <tibble [12 x 4]>
#>  7 Austria     Europe    <tibble [12 x 4]>
#>  8 Bahrain     Asia      <tibble [12 x 4]>
#>  9 Bangladesh  Asia      <tibble [12 x 4]>
#> 10 Belgium     Europe    <tibble [12 x 4]>
#> # ... with 132 more rows
```

- In a grouped data frame, each row is an observation.
- In a nested data frame, each row is a group.

List-columns

- A model-fitting function applied to every country:

```
country_model <- function(df) lm(lifeExp ~ year, data = df)
models <- map(by_country$data, country_model)
```

- Or add an additional list-column:

```
(by_country <- by_country %>%
   mutate(model = map(data, country_model)))
#> # A tibble: 142 x 4
#> # Groups:   country, continent [710]
#>    country     continent data              model
#>    <fct>       <fct>     <list>            <list>
#>  1 Afghanistan Asia      <tibble [12 x 4]> <lm>
#>  2 Albania     Europe    <tibble [12 x 4]> <lm>
#>  3 Algeria     Africa    <tibble [12 x 4]> <lm>
#>  4 Angola      Africa    <tibble [12 x 4]> <lm>
#>  5 Argentina   Americas  <tibble [12 x 4]> <lm>
#>  6 Australia   Oceania   <tibble [12 x 4]> <lm>
#>  7 Austria     Europe    <tibble [12 x 4]> <lm>
#>  8 Bahrain     Asia      <tibble [12 x 4]> <lm>
#>  9 Bangladesh  Asia      <tibble [12 x 4]> <lm>
#> 10 Belgium     Europe    <tibble [12 x 4]> <lm>
#> # ... with 132 more rows
```

Why bother?

- Avoid leaving the list of models as a free-floating object.
- No need to manually keep them in sync when using e.g. `filter` or `arrange`.

```
by_country %>% filter(continent == "Europe")
#> # A tibble: 30 x 4
#> # Groups:   country, continent [710]
#>    country                  continent data              model
#>    <fct>                    <fct>     <list>            <list>
#>  1 Albania                  Europe    <tibble [12 x 4]> <lm>
#>  2 Austria                  Europe    <tibble [12 x 4]> <lm>
#>  3 Belgium                  Europe    <tibble [12 x 4]> <lm>
#>  4 Bosnia and Herzegovina Europe    <tibble [12 x 4]> <lm>
#>  5 Bulgaria                 Europe    <tibble [12 x 4]> <lm>
#>  6 Croatia                  Europe    <tibble [12 x 4]> <lm>
#>  7 Czech Republic           Europe    <tibble [12 x 4]> <lm>
#>  8 Denmark                  Europe    <tibble [12 x 4]> <lm>
#>  9 Finland                  Europe    <tibble [12 x 4]> <lm>
#> 10 France                   Europe    <tibble [12 x 4]> <lm>
#> # ... with 20 more rows
```

Adding residuals


```
(by_country <- by_country %>%
  mutate(resids = map2(data, model, add_residuals)))
#> # A tibble: 142 x 5
#> # Groups:   country, continent [710]
#>    country     continent data              model  resids
#>    <fct>       <fct>     <list>            <list> <list>
#>  1 Afghanistan Asia      <tibble [12 x 4]> <lm>   <tibble [12 x 5]>
#>  2 Albania     Europe    <tibble [12 x 4]> <lm>   <tibble [12 x 5]>
#>  3 Algeria     Africa    <tibble [12 x 4]> <lm>   <tibble [12 x 5]>
#>  4 Angola      Africa    <tibble [12 x 4]> <lm>   <tibble [12 x 5]>
#>  5 Argentina   Americas  <tibble [12 x 4]> <lm>   <tibble [12 x 5]>
#>  6 Australia   Oceania   <tibble [12 x 4]> <lm>   <tibble [12 x 5]>
#>  7 Austria     Europe    <tibble [12 x 4]> <lm>   <tibble [12 x 5]>
#>  8 Bahrain     Asia      <tibble [12 x 4]> <lm>   <tibble [12 x 5]>
#>  9 Bangladesh  Asia      <tibble [12 x 4]> <lm>   <tibble [12 x 5]>
#> 10 Belgium     Europe    <tibble [12 x 4]> <lm>   <tibble [12 x 5]>
#> # ... with 132 more rows
```

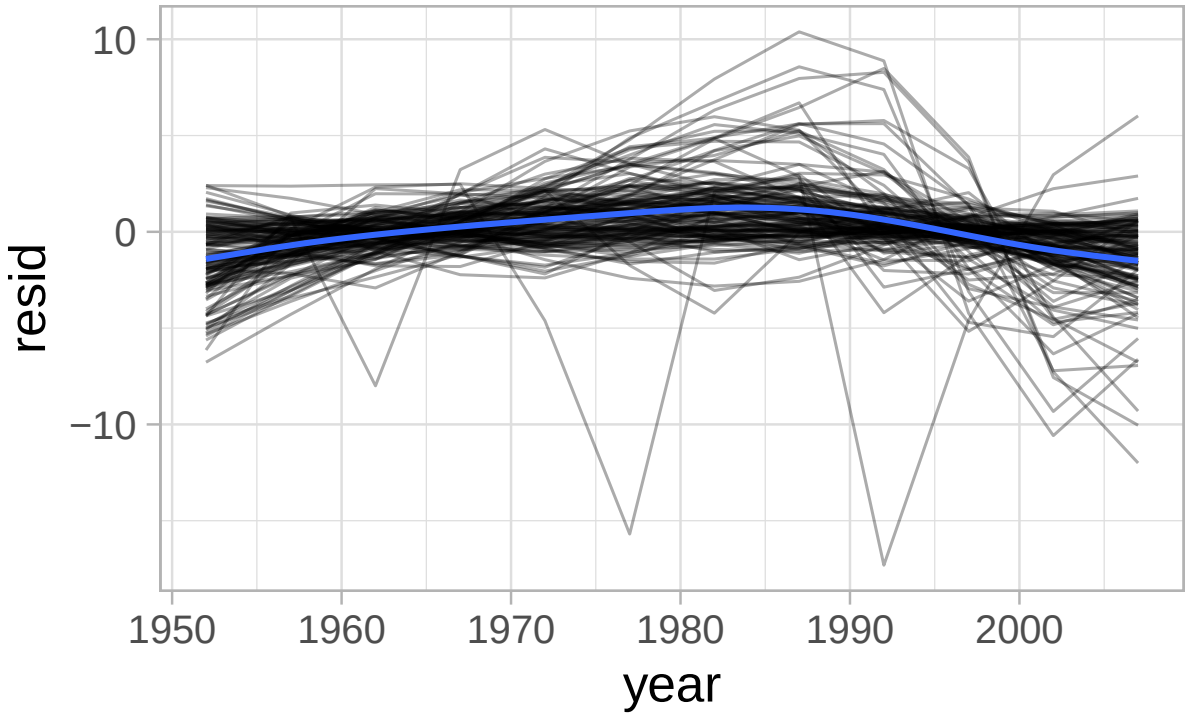
Unnesting


```
resids <- by_country %>%
  select(-c(data, model)) %>%
  unnest(resids)
resids
#> # A tibble: 1,704 x 7
#> # Groups:   country, continent [710]
#>    country     continent  year lifeExp      pop gdpPercap   resid
#>    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>   <dbl>
#>  1 Afghanistan Asia       1952    28.8  8425333      779. -1.11
#>  2 Afghanistan Asia       1957    30.3  9240934      821. -0.952
#>  3 Afghanistan Asia       1962    32.0 10267083      853. -0.664
#>  4 Afghanistan Asia       1967    34.0 11537966      836. -0.0172
#>  5 Afghanistan Asia       1972    36.1 13079460      740.  0.674
#>  6 Afghanistan Asia       1977    38.4 14880372      786.  1.65
#>  7 Afghanistan Asia       1982    39.9 12881816      978.  1.69
#>  8 Afghanistan Asia       1987    40.8 13867957      852.  1.28
#>  9 Afghanistan Asia       1992    41.7 16317921      649.  0.754
#> 10 Afghanistan Asia       1997    41.8 22227415      635. -0.534
#> # ... with 1,694 more rows
```

Visualizing the residuals

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

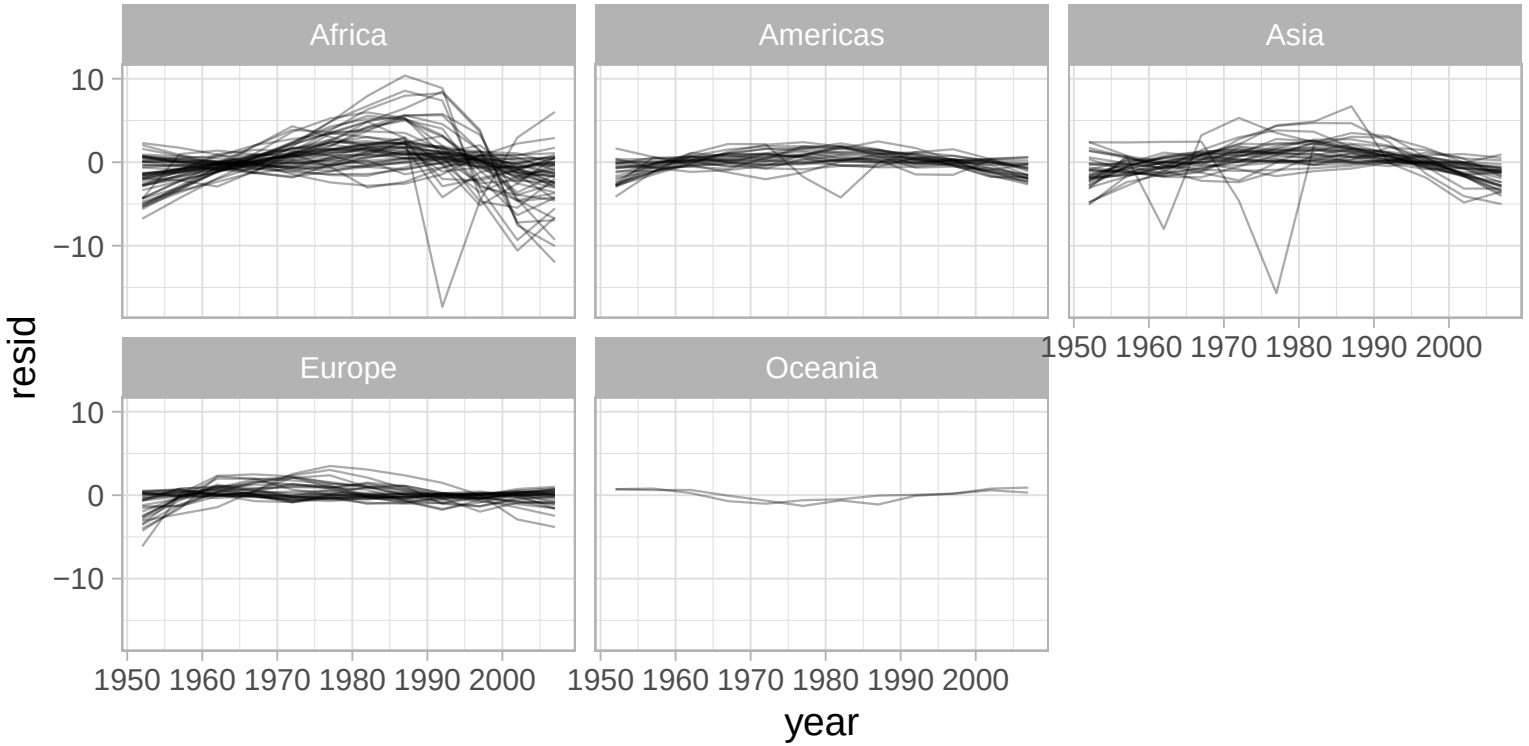
```
resids %>%
  ggplot(aes(year, resid)) +
    geom_line(aes(group = country), alpha = 1 / 3) +
    geom_smooth(se = FALSE)
```



Visualizing the residuals cont'd

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

```
resids %>%
  ggplot(aes(year, resid, group = country)) +
    geom_line(alpha = 1 / 3) +
  facet_wrap(~continent)
```



Model quality

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

```
library(broom)
glance(nz_mod)
#> # A tibble: 1 x 11
#>   r.squared adj.r.squared sigma statistic p.value    df logLik   AIC
#>       <dbl>         <dbl> <dbl>     <dbl>   <dbl> <int>  <dbl> <dbl>
#> 1     0.954         0.949 0.804      205. 5.41e-8     2  -13.3  32.6
#> # ... with 3 more variables: BIC <dbl>, deviance <dbl>,
#> #   df.residual <int>
by_country_glance <- by_country %>%
  mutate(glance = map(model, glance)) %>%
    select(-c(data, model)) %>%
  unnest(glance)
by_country_glance %>%  print(n = 3)
#> # A tibble: 142 x 14
#> # Groups:   country, continent [710]
#>   country continent resids r.squared adj.r.squared sigma statistic
#>   <fct>   <fct>     <list>     <dbl>         <dbl> <dbl>     <dbl>
#> 1 Afghan~ Asia      <tibb~     0.948         0.942  1.22      181.
#> 2 Albania Europe    <tibb~     0.911         0.902  1.98      102.
#> 3 Algeria Africa    <tibb~     0.985         0.984  1.32      662.
#> # ... with 139 more rows, and 7 more variables: p.value <dbl>,
#> #   df <int>, logLik <dbl>, AIC <dbl>, BIC <dbl>, deviance <dbl>,
#> #   df.residual <int>
```

Which models don't fit well?


```
by_country_glance %>%
  arrange(r.squared)
#> # A tibble: 142 x 14
#> # Groups:   country, continent [710]
#>    country continent resids r.squared adj.r.squared sigma statistic
#>    <fct>   <fct>     <list>     <dbl>         <dbl> <dbl>     <dbl>
#>  1 Rwanda  Africa    <tibb~    0.0172      -0.0811   6.56     0.175
#>  2 Botswa~ Africa    <tibb~    0.0340      -0.0626   6.11     0.352
#>  3 Zimbab~ Africa    <tibb~    0.0562      -0.0381   7.21     0.596
#>  4 Zambia  Africa    <tibb~    0.0598      -0.0342   4.53     0.636
#>  5 Swazil~ Africa    <tibb~    0.0682      -0.0250   6.64     0.732
#>  6 Lesotho Africa    <tibb~    0.0849      -0.00666  5.93     0.927
#>  7 Cote d~ Africa    <tibb~    0.283        0.212    3.93     3.95
#>  8 South ~ Africa    <tibb~    0.312        0.244    4.74     4.54
#>  9 Uganda  Africa    <tibb~    0.342        0.276    3.19     5.20
#> 10 Congo,~ Africa    <tibb~    0.348        0.283    2.43     5.34
#> # ... with 132 more rows, and 7 more variables: p.value <dbl>,
#> #   df <int>, logLik <dbl>, AIC <dbl>, BIC <dbl>, deviance <dbl>,
#> #   df.residual <int>
```

Visualize


```
bad_fit <- filter(by_country_glance, r.squared < 0.25)

gapminder %>%
  semi_join(bad_fit, by = "country") %>%
  ggplot(aes(year, lifeExp, colour = country)) +
  geom_line()
```