R Bootcamp Day 2

The tidyverse

Christopher T. Kenny
Data-Driven Social Science

Table of contents

purrr
Applied exercise

Today

Goals

- Familiarity with the core of the tidyverse for data manipulation
- Ability to use `tibble`, `dplyr`, and `purrr` over base R equivalents
- Practice with actual datasets in R
- Tomorrow: `ggplot2`

Sources

- Wickham, Çetinkaya-Rundel, and Grolemund, *R for Data Science*, 2e
  - [Ch. 3: Data transformation](https://r4ds.hadley.nz/data-transform.html) (especially §§3.1-3.5)
  - [Ch. 19: Joins](https://r4ds.hadley.nz/joins.html) (especially §§19.2-19.3)
  - [Ch. 26: Iteration](https://r4ds.hadley.nz/iteration.html) (especially §§26.2-26.3)
- Harvard IQSS, *Math Prefresher for Political Scientists*
  - [Ch. 9: Objects, Functions, Loops](https://iqss.github.io/prefresher/13_functions_obj_loops.html)
  - [Ch. 11: Joins and Merges](https://iqss.github.io/prefresher/15_project-dempeace.html)

The tidyverse

The tidyverse

- A collection of R packages for data science
- Packages designed around tidy data
  - each observation is a row and each column is a variable
- Core packages today: `tibble`, `dplyr`, `tidyr`, and `purrr`

```
library(tidyverse)
```

```
── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
✓ dplyr     1.2.1     ✓ readr     2.2.0
✓ forcats   1.0.1     ✓ stringr   1.6.0
✓ ggplot2   4.0.3     ✓ tibble    3.3.1
✓ lubridate 1.9.5     ✓ tidyr     1.3.2
✓ purrr     1.2.2
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package to force all conflicts to become errors
```

Tibble

- A tibble is a data frame with more predictable behavior
- Printing shows the first rows and the columns that fit
- Column types appear below the column names

```
counties
```

```
# A tibble: 3,144 × 7
  GEOID state   county         income income_moe  rent rent_moe
  <chr> <chr>   <chr>           <dbl>      <dbl> <dbl>    <dbl>
1 01001 Alabama Autauga County  69841       5512  1200       75
2 01003 Alabama Baldwin County  75019       2751  1211       42
3 01005 Alabama Barbour County  44290       2762   644       41
# ℹ 3,141 more rows
```

Tibble

Figure 1: Tibble the cat

Printing a tibble

```
counties
```

```
# A tibble: 3,144 × 7
  GEOID state   county          income income_moe  rent rent_moe
  <chr> <chr>   <chr>            <dbl>      <dbl> <dbl>    <dbl>
1 01001 Alabama Autauga County  69841       5512  1200       75
2 01003 Alabama Baldwin County  75019       2751  1211       42
3 01005 Alabama Barbour County  44290       2762   644       41
# ℹ 3,141 more rows
```

Tibble printing

Figure 2: Tibble's `print()`

Creating a tibble

- `tibble()` builds a data frame from named columns
- Columns have equal length or length one

```
sample_respondents ← tibble(
  id = c(101L, 102L, 103L),
  party = c('Democrat', 'Independent', 'Republican'),
  trust_government = c(3L, 5L, 2L)
)
```

```
tibble(id = seq_len(3L), enrolled = TRUE)
```

```
# A tibble: 3 × 2
     id enrolled
  <int> <lgl>
1     1 TRUE
2     2 TRUE
3     3 TRUE
```

Inspecting a tibble

Inspect before transforming data

- `glimpse()` displays each column and its type
- `names()` returns column names
- `dim()` returns rows and columns

```
counties |>
  select(state, county, income, rent) |>
  glimpse()
```

```
Rows: 3,144
Columns: 4
$ state  <chr> "Alabama", "Alabama", "Alabama", "Alabama", "Alabama", "Alabama…
$ county <chr> "Autauga County", "Baldwin County", "Barbour County", "Bibb Cou…
$ income <dbl> 69841, 75019, 44290, 51215, 61096, 36723, 44881, 55826, 49295, …
$ rent   <dbl> 1200, 1211, 644, 802, 743, 635, 722, 804, 850, 750, 855, 653, 7…
```

Extracting a column

`$` and `[[` return a column as a vector

```
class(counties$income)
class(counties[['income']])
```

```
[1] "numeric"
[1] "numeric"
```

Both approaches return a vector

`[` return returns a one-column tibble

```
class(counties['income'])
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

Data transformation

dplyr verbs

- Rows - `filter()`: subset by logical conditions
- Rows - `slice()`: subset by row positions
- Rows - `arrange()`: reorder by a variable
- Columns - `select()`: keep, drop, or reorder columns
- Columns - `mutate()`: create or update columns
- Columns - `rename()`: rename specific columns
- Groups - `group_by()`: identify groupings in data
- Groups - `summarize()`: reduce groups to summaries

filter()

`filter()` keeps rows that satisfy a logical condition

```
counties |>
  filter(income ≥ 75000)
```

```
# A tibble: 716 × 7
  GEOID state   county           income income_moe  rent rent_moe
  <chr> <chr>   <chr>             <dbl>      <dbl> <dbl>    <dbl>
1 01003 Alabama Baldwin County    75019       2751  1211       42
2 01051 Alabama Elmore County     75553       3115  1014       48
3 01083 Alabama Limestone County  83534       2861   908       49
# ℹ 713 more rows
```

Multiple conditions: Equivalent to AND

Commas require every condition to be `TRUE`

```
counties |>
  filter(state == 'New York', income ≥ 75000)
```

```
# A tibble: 19 × 7
  GEOID state    county          income income_moe  rent rent_moe
  <chr> <chr>    <chr>            <dbl>      <dbl> <dbl>    <dbl>
1 36001 New York Albany County    83149       1788  1252       22
2 36021 New York Columbia County  83619       3965  1199       45
3 36027 New York Dutchess County  97273       2529  1522       45
# ℹ 16 more rows
```

which is the same as:

```
counties |>
  filter(state == 'New York') |>
  filter(income ≥ 75000)
```

Missing values

```
scores ← tibble(
  id = seq_len(4L),
  score = c(88L, NA, 91L, 76L)
)
```

What happens with NAs?

```
scores |> filter(score > 90)
```

```
# A tibble: 1 × 2
     id score
  <int> <int>
1     3    91
```

arrange()

- `arrange()` orders rows by one or more columns
- Use `desc()` for descending order

```
counties |>
  arrange(desc(income))
```

```
# A tibble: 3,144 × 7
  GEOID state      county              income income_moe  rent rent_moe
  <chr> <chr>      <chr>                <dbl>      <dbl> <dbl>    <dbl>
1 51107 Virginia   Loudoun County      178707       3411  2317       39
2 06085 California Santa Clara County 159674       1962  2814       16
3 06081 California San Mateo County   156000       3019  2893       36
# ℹ 3,141 more rows
```

slice_*() functions

- `slice()` selects integer row positions
- `slice_head()` and `slice_tail()` keep the first or last rows
- `slice_min()` and `slice_max()` select rows by an ordering variable
- `slice_sample()` takes a random sample of rows

```
counties |>
  slice_max(income, n = 3L, with_ties = FALSE) |>
  select(state, county, income)
```

```
# A tibble: 3 × 3
  state      county              income
  <chr>      <chr>                <dbl>
1 Virginia   Loudoun County      178707
2 California Santa Clara County 159674
3 California San Mateo County   156000
```

distinct()

`distinct()` keeps unique values

```
counties |> distinct(state)
```

```
# A tibble: 51 × 1
  state
  <chr>
1 Alabama
2 Alaska
3 Arizona
# i 48 more rows
```

Use multiple columns for unique combinations

```
counties |> distinct(state, county) |> nrow()
```

```
[1] 3144
```

Stack rows: bind_rows()

`bind_rows()` stacks tibbles, matching columns by name

`.id` adds a source column

```
respondents ← bind_rows(
  list(
    before = tibble(id = c(101L, 102L), score = c(3L, 5L)),
    after = tibble(id = c(101L, 102L), score = c(4L, 5L))
  ),
  .id = 'wave'
)
```

Stack rows: bind_rows()

`bind_rows()` stacks tibbles, matching columns by name

`.id` adds a source column

```
respondents
```

```
# A tibble: 4 × 3
  wave      id score
  <chr>  <int> <int>
1 before   101     3
2 before   102     5
3 after    101     4
# ℹ 1 more row
```

select()

`select()` keeps, drops, or reorders columns:

```
counties |>
  select(state, county, income, rent)
```

```
# A tibble: 3,144 × 4
  state   county          income  rent
  <chr>   <chr>            <dbl> <dbl>
1 Alabama Autauga County  69841  1200
2 Alabama Baldwin County  75019  1211
3 Alabama Barbour County  44290   644
# ℹ 3,141 more rows
```

select(): ranges and exclusions

We can also select with ranges ...

```
counties |> select(state:income) |> names()
```

```
[1] "state"  "county" "income"
```

... and with exclusions:

```
counties |> select(-c(income_moe, rent_moe)) |> names()
```

```
[1] "GEOID"  "state"  "county" "income" "rent"
```

Selection helpers

We can select based on names using `starts_with()`, `ends_with()`, and `contains()`

```
counties |> select(starts_with('rent')) |> names()
```

```
[1] "rent"     "rent_moe"
```

Or with a helper function inside `where()`:

```
counties |> select(where(is.numeric)) |> names()
```

```
[1] "income"     "income_moe" "rent"       "rent_moe"
```

rename()

```
counties |>
  rename(
    median_renter_income = income,
    median_gross_rent = rent
  )
```

```
# A tibble: 3,144 × 7
  GEOID state  county median_renter_income income_moe median_gross_rent rent_moe
  <chr> <chr>  <chr>                 <dbl>      <dbl>             <dbl>    <dbl>
1 01001 Alaba… Autau…                69841       5512              1200       75
2 01003 Alaba… Baldw…                75019       2751              1211       42
3 01005 Alaba… Barbo…                44290       2762               644       41
# ℹ 3,141 more rows
```

relocate()

Move columns to the front:

```
counties |> relocate(income, rent) |> names()
```

```
[1] "income"     "rent"       "GEOID"      "state"      "county"
[6] "income_moe" "rent_moe"
```

Or to a specific location:

```
counties |> relocate(county, .after = state) |> names()
```

```
[1] "GEOID"      "state"      "county"     "income"     "income_moe"
[6] "rent"       "rent_moe"
```

mutate()

`mutate()` creates new columns or replaces old ones

```
counties |>
  mutate(
    income_thousands = income / 1000,
    rent_hundreds = rent / 100
  ) |>
  select(state, county, income_thousands, rent_hundreds)
```

```
# A tibble: 3,144 × 4
  state   county          income_thousands rent_hundreds
  <chr>   <chr>                      <dbl>         <dbl>
1 Alabama Autauga County              69.8          12
2 Alabama Baldwin County              75.0          12.1
3 Alabama Barbour County              44.3           6.44
# ℹ 3,141 more rows
```

Keeping only specific columns

`.keep = 'none'` keeps only the updated columns

```
counties |>
  mutate(
    income_thousands = income / 1000,
    rent_hundreds = rent / 100,
    .keep = 'none'
  )
```

```
# A tibble: 3,144 × 2
  income_thousands rent_hundreds
             <dbl>         <dbl>
1             69.8          12
2             75.0          12.1
3             44.3           6.44
# ℹ 3,141 more rows
```

Before 2025, this was `transmute()`

Conditional variables: if_else()

```
counties |>
  mutate(
    rent_level = if_else(
      rent ⩾ median(rent, na.rm = TRUE),
      'at or above median',
      'below median'
    )
  ) |>
  select(state, county, rent, rent_level)
```

```
# A tibble: 3,144 × 4
  state   county          rent rent_level
  <chr>   <chr>          <dbl> <chr>
1 Alabama Autauga County  1200 at or above median
2 Alabama Baldwin County  1211 at or above median
3 Alabama Barbour County   644 below median
# ℹ 3,141 more rows
```

Complex conditional variables: case_when()

```
counties |>
  mutate(
    income_group = case_when(
      is.na(income) ~ 'missing',
      income ⩾ 100000 ~ 'high',
      income ⩾ 75000 ~ 'middle',
      .default = 'low'
    )
  ) |>
  select(state, county, income, income_group)
```

```
# A tibble: 3,144 × 4
  state   county          income income_group
  <chr>   <chr>            <dbl> <chr>
1 Alabama Autauga County   69841 low
2 Alabama Baldwin County   75019 middle
3 Alabama Barbour County   44290 low
# ℹ 3,141 more rows
```

Exercise

- Keep counties in New York or Texas
- Create `income_thousands` from `income`
- Keep `state`, `county`, `income`, and `income_thousands`
- Order from highest to lowest income

Write this as one pipeline

A solution

```
counties |>
  filter(state %in% c('New York', 'Texas')) |>
  mutate(income_thousands = income / 1000) |>
  select(state, county, income, income_thousands) |>
  arrange(desc(income))
```

```
# A tibble: 316 × 4
  state    county          income income_thousands
  <chr>    <chr>            <dbl>            <dbl>
1 New York Nassau County   143408             143.
2 New York Suffolk County  128329             128.
3 New York Putnam County   127405             127.
# ℹ 313 more rows
```

Strings with stringr

Find text with stringr

- `str_detect()` returns `TRUE`/`FALSE` for each string
- `str_subset()` returns only the matching strings
- `str_count()` counts matches in each string

```
states ← c('New York', 'Texas', 'West Virginia', 'New Jersey')

str_detect(states, '^New ')
str_subset(states, 'New')
```

```
[1]  TRUE FALSE FALSE  TRUE
[1] "New York"   "New Jersey"
```

Transform text with stringr

- `str_to_lower()` and `str_to_upper()` standardize case
- `str_replace()` and `str_replace_all()` edit matches
- `str_c()` combines strings

Transform text with stringr

```
places ← c('New York', 'West Virginia')

tibble(
  original = places,
  lower = str_to_lower(places),
  slug = str_replace_all(
    str_to_lower(places),
    ' ',
    '_'
  ),
  label = str_c('State: ', places)
)
```

```
# A tibble: 2 × 4
  original      lower         slug          label
  <chr>         <chr>         <chr>         <chr>
1 New York      new york      new-york      State: New York
2 West Virginia west virginia west-virginia State: West Virginia
```

Pivots in tidyr

Long and wide data

Wide data

Each observation is a row

```
wide ← tribble(
  ~id, ~v1, ~v2, ~v3,
  'A', 100L, 120L, 105L,
  'B', 80L, 90L, 90L
)
```

Long data

Observations are split across rows

```
long ← tribble(
  ~id, ~statistic, ~value,
  'A', 'v1', 100L,
  'A', 'v2', 120L,
  'A', 'v3', 105L,
  'B', 'v1', 80L,
  'B', 'v2', 90L,
  'B', 'v3', 90L,
)
```

pivot_wider()

`pivot_wider()` shortens the data by adding columns

```
long |>
  pivot_wider(
    names_from = statistic,
    values_from = value
  )
```

```
# A tibble: 2 × 4
  id       v1    v2    v3
  <chr> <int> <int> <int>
1 A       100   120   105
2 B        80    90    90
```

pivot_longer()

`pivot_longer()` lengthens the data by turning columns into rows

```
wide |>
  pivot_longer(
    cols = v1:v3,
    names_to = 'statistic',
    values_to = 'value'
  )
```

```
# A tibble: 6 × 3
  id    statistic value
  <chr> <chr>     <int>
1 A     v1          100
2 A     v2          120
3 A     v3          105
# ℹ 3 more rows
```

Other weird data shapes

- Most can be **tidied** with `tidyr`
- Nested data? Some variables are lists
  - `unnest()`, `unnest_wider()`, `unnest_longer()`
- Data should be nested?
  - Are you sure?
    - `nest()`
- Handle `NA` values
  - `drop_na()` or `replace_na()`
  - Missing within groups: `fill()`
- Missing combinations (e.g., all weeks should have 7 days)
  - `expand()` for new data or `complete()` to supplement

Groups and summaries

summarize()

`summarize()` reduces rows to summary values

```
counties |>
  summarize(
    n = n(),
    mean_income = mean(income, na.rm = TRUE),
    median_rent = median(rent, na.rm = TRUE),
    .groups = 'drop'
  )
```

```
# A tibble: 1 × 3
      n mean_income median_rent
  <int>       <dbl>       <dbl>
1  3144      66073.         854
```

If there are no groups, `summarize()` treats all rows as one group

group_by()

`group_by()` marks a `tibble` as having groups

```
counties |>
  group_by(state) |>
  summarize(
    n = n(),
    mean_income = mean(income, na.rm = TRUE),
    median_rent = median(rent, na.rm = TRUE)
  )
```

```
# A tibble: 51 × 4
  state       n mean_income median_rent
  <chr>   <int>       <dbl>       <dbl>
1 Alabama    67      54196.         762
2 Alaska     30      79407.        1243
3 Arizona    15      62663.         983
# ℹ 48 more rows
```

Grouped mutation

`mutate()` will respect groups for vectorized operations

```
counties |>
  group_by(state) |>
  mutate(
    income_difference = income - mean(income, na.rm = TRUE)
  ) |>
  select(state, county, income, income_difference) |>
  ungroup()
```

```
# A tibble: 3,144 × 4
  state   county          income income_difference
  <chr>   <chr>            <dbl>             <dbl>
1 Alabama Autauga County   69841            15645.
2 Alabama Baldwin County   75019            20823.
3 Alabama Barbour County   44290            -9906.
# ℹ 3,141 more rows
```

count()

```
counties |> count(state)
```

```
# A tibble: 51 × 2
  state       n
  <chr>   <int>
1 Alabama    67
2 Alaska     30
3 Arizona    15
# i 48 more rows
```

`count(state)` groups by `state` and counts rows

Equivalent to:

```
counties |> group_by(state) |> summarize(n = n())
```

Multiple summaries: across()

```
counties |>
  summarize(
    across(
      c(income, rent),
      list(
        mean = \(x) mean(x, na.rm = TRUE),
        sd = \(x) sd(x, na.rm = TRUE)
      )
    )
  )
```

```
# A tibble: 1 × 4
  income_mean income_sd rent_mean rent_sd
        <dbl>     <dbl>     <dbl>   <dbl>
1      66073.    17387.      938.    302.
```

`across()` applies the same functions to several columns

Multiple summaries: across()

`across()` simplifies making summaries:

```
counties |>
  summarize(across(where(is.numeric), \(x) mean(x, na.rm = TRUE)))
```

```
# A tibble: 1 × 4
  income income_moe  rent rent_moe
   <dbl>      <dbl> <dbl>    <dbl>
1 66073.      5068.  938.     73.6
```

Exercise

Compare counties above and below the median income.

- Exclude counties with missing `income`
- Label counties as `at or above median` or `below median`
- For each group, report `n` and the mean `income` and `rent` using `across()`

Write this as one pipeline

A solution

```
sol ← counties |>
  filter(!is.na(income)) |>
  mutate(
    income_level = if_else(
      income ⩾ median(income),
      'at or above median',
      'below median'
    )
  ) |>
  group_by(income_level) |>
  summarize(
    n = n(),
    across(c(income, rent), \(x) mean(x, na.rm = TRUE)),
    .groups = 'drop'
  ) |>
  arrange(desc(income))
```

A solution

```
sol
```

```
# A tibble: 2 × 4
  income_level           n income  rent
  <chr>              <int>  <dbl> <dbl>
1 at or above median  1571 78668. 1090.
2 below median        1571 53479.  787.
```

Joins

Join tables by a fixed key

- A join combines columns from related tables
- A key identifies which rows should match

```
party ← tibble(
  id = c(101L, 102L, 103L),
  party = c('Democrat', 'Independent', 'Republican')
)

attitudes ← tibble(
  id = c(101L, 102L, 104L),
  trust_government = c(3L, 5L, 2L)
)
```

A key is (often) just a numeric ID

Check the key

A key should identify the intended unit of observation

```
attitudes |>
  count(id) |>
  filter(n > 1L)
```

```
# A tibble: 0 × 2
# ℹ 2 variables: id <int>, n <int>
```

An empty result means no duplicate keys were found

Left and right tables

left

id	party
101	Democrat
102	Independent
103	Republican
104	Democrat
106	Republican

right

id	trust
102	5
103	3
104	2
105	4
106	7

Types of joins

- `left_join()`: keep all rows in left and add columns from right
- `right_join()`: keep all rows in right and add columns from left
- `inner_join()` keeps rows matched in both tables
- `full_join()` keeps rows from either table
- `semi_join()` keeps left rows with a match
- `anti_join()` keeps left rows without a match

Which rows are saved for each join?

```
left |>
  *_join(right, by = 'id')
```

Joining real data

`counties` and `county_pop` use different names for the same key

```
counties_joined ← counties |>
  left_join(county_pop, join_by(GEOID == fips))

counties_joined |>
  select(GEOID, state, county, pop, region, division)
```

```
# A tibble: 3,144 × 6
  GEOID state   county           pop region division
  <chr> <chr>   <chr>          <dbl> <chr>  <chr>
1 01001 Alabama Autauga County  58805 South  East South Central
2 01003 Alabama Baldwin County 231767 South  East South Central
3 01005 Alabama Barbour County  25223 South  East South Central
# ℹ 3,141 more rows
```

Joining real data: multiple syntaxes

Current, recommended syntax:

```
counties_joined ← counties |>
  left_join(county_pop, join_by(GEOID == fips))
```

Older, but also valid syntax:

```
counties_joined ← counties |>
  left_join(county_pop, by = c('GEOID' = 'fips'))
```

Most replication data (and many of your colleagues) will use the older syntax!

New syntax allows for *fancier* joins, like `join_by(x > x)`...

Find unmatched keys

Check the rows that do not match:

```
counties |>
  select(GEOID, state, county) |>
  anti_join(county_pop, join_by(GEOID == fips)) |>
  select(GEOID, state, county)
```

```
# A tibble: 9 × 3
  GEOID state       county
  <chr> <chr>       <chr>
1 09110 Connecticut Capitol Planning Region
2 09120 Connecticut Greater Bridgeport Planning Region
3 09130 Connecticut Lower Connecticut River Valley Planning Region
# i 6 more rows
```

What's wrong with Connecticut?

Duplicate keys change row counts

A join can duplicate left rows when the right key is not unique

```
attitudes_duplicate ← tibble(
  id = c(101L, 101L, 102L),
  trust_government = c(3L, 4L, 5L)
)

party |>
  left_join(attitudes_duplicate, join_by(id))
```

```
# A tibble: 4 × 3
     id party       trust_government
  <int> <chr>                  <int>
1   101 Democrat                   3
2   101 Democrat                   4
3   102 Independent                5
# ℹ 1 more row
```

purrr

map()

`map(x, f)` is a new way to do yesterday's `lapply(x, f)`

Why `map()` then? Large family of *type-stable* `map_*()` functions!

And bits of friendlier syntax:

```
lapply(x, `[[`, "a")
map(x, "a")
```

Typed maps

Typed maps return a vector of a *known* type:

- `map_dbl()`: double vector
- `map_int()`: integer vector
- `map_chr()`: character vector
- `map_lgl()`: logical vector

```
counties |>
  select(income, rent) |>
  map_dbl(
    \(x) mean(x, na.rm = TRUE)
  )
```

```
    income       rent
66073.0945   938.1388
```

Mapping over data frames

Create a list of `tibbles` with `group_split()`:

```
counties_by_region ← counties_joined |>
  filter(!is.na(region)) |>
  group_by(region) |>
  group_split()
```

`map_dbl()` then returns one value per `tibble`:

```
map_dbl(
  counties_by_region,
  \(x) mean(x$income, na.rm = TRUE)
)
```

```
[1] 67678.39 78195.17 60859.68 72626.80
```

Combining results

First return one tibble for each region

```
summaries ← map(
  counties_by_region,
  function(data) {
    summarize(
      data,
      n = n(),
      mean_income = mean(income, na.rm = TRUE)
    )
  }
)
```

```
all(map_lgl(summaries, is_tibble))
length(summaries)
```

```
[1] TRUE
[1] 4
```

Stack compatible results

`list_rbind()` combines the tibbles by row

```
list_rbind(summaries)
```

```
# A tibble: 4 × 2
      n mean_income
  <int>       <dbl>
1  1055      67678.
2   209      78195.
3  1422      60860.
# ℹ 1 more row
```

walk() for side effects

`walk()` is like `map()`, but it is for actions rather than returned results

- `walk(x, f)`: one input
- `walk2(x, y, f)`: two equal-length inputs
- `iwalk(x, f)`: also passes the index or name
- `pwalk(list( ... ), f)`: many inputs

```
counties_by_region |>
  walk(\(data) print(unique(data$region)))
```

```
[1] "Midwest"
[1] "Northeast"
[1] "South"
[1] "West"
```

Progress bars in purrr

- Map functions accept `.progress`
- Use `TRUE` for a basic bar or a short name for a labeled bar
- Progress bars are most useful for long-running maps

```
map_dbl(
  counties_by_region,
  \(data) mean(data$income, na.rm = TRUE),
  .progress = 'mean by region'
)
```

Parallel maps with in_parallel()

- `in_parallel()` is an experimental `purrr` adverb
- `mirai::daemons(n)` starts parallel workers
- Without daemons, the map falls back to sequential processing
- Keep the function self-contained; namespace package calls with `::`

```
mirai::daemons(2L)

map_dbl(
  counties_by_region,
  purrr::in_parallel(
    \(data) mean(data$income, na.rm = TRUE)
  ),
  .progress = 'mean by region'
)

mirai::daemons(0)
```

Several inputs

`map2()` iterates over two (equal length) inputs in parallel

```
district_turnout ← list(c(0.62, 0.58, 0.64), c(0.71, 0.69, 0.74))
benchmarks ← c(0.60, 0.70)

map2_lgl(
  district_turnout,
  benchmarks,
  \(turnout, benchmark) mean(turnout) > benchmark
)
```

```
[1] TRUE TRUE
```

Applied exercise

Exercise

Using `counties_joined`, compare income by region and population level

- Label counties as above or below the median population
- Compute the number of counties and mean income by `region` and population level
- Order the summary from highest to lowest mean income

Write one pipeline

Step 1: label population

```
counties_labeled ← counties_joined |>
  mutate(
    pop_level = #  ...
  )
```

Use `if_else()` or `case_when()`

Step 2: summarize groups

```
result ← counties_labeled |>
  # group by region and population level
  # summarize n and mean_income
  # arrange by descending mean_income
```

How many rows should the result contain?

One possible solution

```
result ← counties_joined |>
  filter(!is.na(pop), !is.na(region)) |>
  mutate(
    pop_level = if_else(
      pop ⩾ median(pop),
      'at or above median',
      'below median'
    )
  ) |>
  group_by(region, pop_level) |>
  summarize(
    n = n(),
    mean_income = mean(income, na.rm = TRUE),
    .groups = 'drop'
  ) |>
  arrange(desc(mean_income))
```

Apply a function to each group

```
region_summary ← function(data) {
  data |>
    summarize(
      n = n(),
      mean_income = mean(income, na.rm = TRUE),
      median_rent = median(rent, na.rm = TRUE)
    )
}
```

Apply `region_summary()` to each region with `map()`

One possible solution

```
by_region ← counties_joined |>
  filter(!is.na(region)) |>
  group_by(region) |>
  group_split()

by_region |>
  map(region_summary) |>
  list_rbind()
```

```
# A tibble: 4 × 3
      n mean_income median_rent
  <int>       <dbl>       <dbl>
1  1055      67678.         804
2   209      78195.        1056
3  1422      60860.         845
# ℹ 1 more row
```

Other tidyverse packages

- `stringr` works with character strings
- `lubridate` parses and transforms dates and times
- `forcats` works with categorical variables (“factors”)
- `readr` imports rectangular data files

Summary

- Start with a tibble and inspect its rows, columns, and types
- Use one `dplyr` verb for each transformation
- Use pipes to connect transformations
- Group before computing group-level summaries
- Check keys before and after joins
- Use `map()` to apply a function to every list element