The Tidyverse

Data Frame Manipulation in R

Yen-Yi Ho

Section 1

Introduction

What Is the Tidyverse?

- The **tidyverse** is a collection of R packages designed to work together around a common philosophy of data: `ggplot2`, `dplyr`, `tidyr`, `readr`, `purrr`, `tibble`, and more
- Loaded all at once with:

```
library(tidyverse)
```

- Today's focus: **manipulating data frames** with `dplyr`, working with **tibbles**, and applying functions with `purrr`
- Running example: the `murders` data frame (US gun murders by state) and the `heights` data frame, both from the `dslabs` package

Tidy Data

- A dataset is in **tidy format** if:
  - each **row** represents one observation
  - each **column** represents one variable
- The `murders` data frame is already tidy: one row per state, columns for state, region, population, total murders

```
head(murders, 3)
```

```
#>     state abb region population total
#> 1 Alabama  AL  South    4779736   135
#> 2  Alaska  AK   West     710231    19
#> 3 Arizona  AZ   West    6392017   232
```

- Tidyverse functions are built to expect (and produce) tidy data — this is what makes them compose so well

Section 2

Manipulating Data Frames

Adding Columns: `mutate()`

- `mutate()` adds a new column (or changes an existing one)
- Notice: we refer to `total` and `population` directly — no `$` needed

```
murders <- mutate(murders, rate = total / population * 100000)
head(murders, 3)
```

```
#>     state abb region population total     rate
#> 1 Alabama  AL  South    4779736   135 2.824424
#> 2  Alaska  AK   West     710231    19 2.675186
#> 3 Arizona  AZ   West    6392017   232 3.629527
```

- `dplyr` functions "know" they're working with `murders`, so they look for columns *inside* the data frame first

Subsetting Rows: `filter()`

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

```
filter(murders, rate <= 0.71)
```

```
#>           state abb        region population total      rate
#> 1        Hawaii  HI          West    1360301     7 0.5145920
#> 2          Iowa  IA North Central    3046355    21 0.6893484
#> 3 New Hampshire  NH     Northeast    1316470     5 0.3798036
#> 4  North Dakota  ND North Central     672591     4 0.5947151
#> 5       Vermont  VT     Northeast     625741     2 0.3196211
```

Subsetting Columns: `select()`

- `select()` keeps only the columns you name

```
new_data_frame <- select(murders, state, region, rate)
filter(new_data_frame, rate <= 0.71)
```

```
#>           state        region      rate
#> 1        Hawaii          West 0.5145920
#> 2          Iowa North Central 0.6893484
#> 3 New Hampshire     Northeast 0.3798036
#> 4  North Dakota North Central 0.5947151
#> 5       Vermont     Northeast 0.3196211
```

- Helper functions for column selection: `starts_with()`, `ends_with()`, `contains()`, `where()`, `everything()`

Subsetting Columns: `select()`

```
murders |> select(starts_with("p"))   |> colnames()
```

```
#> [1] "population"
```

```
murders |> select(where(is.numeric))   |> colnames()
```

```
#> [1] "population" "total"      "rate"
```

```
murders |> select(state, everything())  |> colnames()
```

```
#> [1] "state"      "abb"        "region"     "population" "total"
#> [6] "rate"
```

Transforming Variables with `mutate()`

- `mutate()` can also transform existing columns

```
mutate(murders, population = log10(population))
```

- Apply a function to several columns at once with `across()`

```
mutate(murders, across(c(population, total), log10))
mutate(murders, across(where(is.numeric), log10))
mutate(murders, across(where(is.character), tolower))
```

Section 3

The Pipe

The Pipe: |> and %>%

- The pipe takes the result on its **left** and feeds it as the **first argument** to the function on its **right**
- Base R native pipe |> (used throughout) and the `magrittr` pipe %>% (from the tidyverse) behave the same way in simple cases

```
16 |> sqrt()
```

```
#> [1] 4
```

```
16 |> sqrt() |> log2()
```

```
#> [1] 2
```

Chaining `dplyr` Verbs with the Pipe

- Instead of nesting or creating intermediate objects, chain steps:

```
murders |>
  select(state, region, rate) |>
  filter(rate <= 0.71)
```

```
#>           state        region      rate
#> 1        Hawaii          West 0.5145920
#> 2          Iowa North Central 0.6893484
#> 3 New Hampshire     Northeast 0.3798036
#> 4  North Dakota North Central 0.5947151
#> 5       Vermont     Northeast 0.3196211
```

- Reads naturally left to right / top to bottom: "take `murders`, then `select`, then `filter`"

Section 4

Summarizing Data

The summarize() Function

- summarize() collapses a data frame into one row of summary statistics

```
s <- heights |>
  filter(sex == "Female") |>
  summarize(average = mean(height),
            standard_deviation = sd(height))
s
```

```
#>    average standard_deviation
#> 1 64.93942           3.760656
```

- Access values with $ or pull():

```
s$average
s$standard_deviation
```

A Practical Example: US Murder Rate

```
us_murder_rate <- murders |>
  summarize(rate = sum(total) / sum(population) * 100000)
us_murder_rate
```

```
#>       rate
#> 1 3.034555
```

- Note this is **not** the average of the state rates — it's the rate for the country as a whole (sum of totals over sum of populations)

Multiple Summaries: `reframe()`

- `summarize()` expects each expression to return **one** value per group
- Use `reframe()` when a computation returns **multiple** values, e.g. quantiles

```
heights |> reframe(quantiles = quantile(height, c(0.5, 0, 1)))
```

```
#>   quantiles
#> 1  68.50000
#> 2  50.00000
#> 3  82.67717
```

- A custom function that returns a small data frame can also be passed to `summarize()` directly

Group Then Summarize: `group_by()`

- `group_by()` splits the data into groups; `summarize()` then computes statistics **within each group**

```
heights |>
  group_by(sex) |>
  summarize(average = mean(height),
            standard_deviation = sd(height))
```

```
#> # A tibble: 2 x 3
#>   sex    average standard_deviation
#>   <fct>    <dbl>              <dbl>
#> 1 Female    64.9               3.76
#> 2 Male      69.3               3.61
```

Grouped Summary: Murder Rate by Region

```
murders |>
  group_by(region) |>
  summarize(rate = sum(total) / sum(population) * 100000)
```

```
#> # A tibble: 4 x 2
#>   region         rate
#>   <fct>         <dbl>
#> 1 Northeast      2.66
#> 2 South          3.63
#> 3 North Central  2.73
#> 4 West           2.66
```

Extracting a Vector: `pull()`

- `summarize()` (and friends) always return a data frame — even with one cell
- `pull()` extracts a column as a plain vector, keeping the pipe going

```
us_murder_rate <- murders |>
  summarize(rate = sum(total) / sum(population) * 100000) |>
  pull(rate)
us_murder_rate
```

```
#> [1] 3.034555
```

```
class(us_murder_rate)
```

```
#> [1] "numeric"
```

Section 5

Sorting

Sorting with arrange()

- arrange() orders rows by one or more columns (ascending by default)

```
murders |> arrange(population) |> head()
```

- Use desc() for descending order

```
murders |> select(state, region, rate) |>
        arrange(desc(rate)) |> head(5)
```

```
#>                  state         region      rate
#> 1 District of Columbia          South 16.452753
#> 2             Louisiana          South  7.742581
#> 3              Missouri North Central  5.359892
#> 4              Maryland          South  5.074866
#> 5        South Carolina          South  4.475323
```

Nested Sorting

- Extra columns are used to break ties in the earlier sorting columns

```
murders |>
  arrange(region, rate) |>
  head(5)
```

```
#>           state abb    region population total      rate
#> 1       Vermont  VT Northeast     625741     2 0.3196211
#> 2 New Hampshire  NH Northeast    1316470     5 0.3798036
#> 3         Maine  ME Northeast    1328361    11 0.8280881
#> 4  Rhode Island  RI Northeast    1052567    16 1.5200933
#> 5 Massachusetts  MA Northeast    6547629   118 1.8021791
```

The Top *n*: `slice_max()` / `slice_min()`

- Directly extract the rows with the largest (or smallest) values of a variable — no need to `arrange()` first

```
murders |>select(state, region, rate) |> slice_max(rate, n = 5)
```

```
#>                  state        region      rate
#> 1 District of Columbia         South 16.452753
#> 2             Louisiana         South  7.742581
#> 3              Missouri North Central  5.359892
#> 4              Maryland         South  5.074866
#> 5        South Carolina         South  4.475323
```

- `slice_min(rate, n = 5)` would give the five **lowest** rates

Section 6

Tibbles

What Is a Tibble?

- `dplyr` functions like `group_by()` and `summarize()` return **tibbles**, not base data frames

```
murders |> group_by(region) |> class()
```

```
#> [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

- A tibble is a modern reimagining of the data frame; every tibble *is* a data frame, but with extra features and stricter behavior

Tibbles vs. Data Frames: Display

- Tibbles print more nicely: only the rows/columns that fit the screen, and each column's type is shown

```
as_tibble(murders)
```

```
#> # A tibble: 51 x 6
#>    state                abb   region    population total  rate
#>    <chr>                <chr> <fct>          <dbl> <dbl> <dbl>
#>  1 Alabama              AL    South        4779736   135  2.82
#>  2 Alaska               AK    West          710231    19  2.68
#>  3 Arizona              AZ    West         6392017   232  3.63
#>  4 Arkansas             AR    South        2915918    93  3.19
#>  5 California           CA    West        37253956  1257  3.37
#>  6 Colorado             CO    West         5029196    65  1.29
#>  7 Connecticut          CT    Northeast    3574097    97  2.71
#>  8 Delaware             DE    South         897934    38  4.23
#>  9 District of Columbia DC    South         601723    99 16.5
#> 10 Florida              FL    South       19687653   669  3.40
#> # i 41 more rows
```

Tibbles vs. Data Frames: Subsetting

- Subsetting a data frame with `[`, one column, returns a **vector**
- Subsetting a tibble the same way always returns another **tibble**

```
class(murders[, 4])
```

```
#> [1] "numeric"
```

```
class(as_tibble(murders)[, 4])
```

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

- Tibbles also **warn** you when you access a column that does not exist; data frames silently return `NULL`

Tibbles vs. Data Frames: Other Differences

- Tibble columns can hold **complex objects**: lists, vectors, even functions — useful for advanced workflows
- Tibbles can be **grouped**, storing grouping information that later `dplyr` verbs use automatically (as seen with `group_by()`)

Creating Tibbles

```
grades <- tibble(names = c("John", "Juan", "Jean", "Yao"),
                 exam_1 = c(95, 80, 90, 85),
                 exam_2 = c(90, 85, 85, 90))
grades
```

```
#> # A tibble: 4 x 3
#>   names exam_1 exam_2
#>   <chr>  <dbl>  <dbl>
#> 1 John      95     90
#> 2 Juan      80     85
#> 3 Jean      90     85
#> 4 Yao       85     90
```

- Convert an existing data frame with `as_tibble()`

Section 7

The Placeholder and purrr

The Placeholder: _ and .

- Pipe sends its left-hand result to the **first** argument by default
- To send it somewhere else, use a placeholder:
  - native pipe |> uses _ (with named argument)
  - `magrittr` pipe %>% uses .

```
2 |> log(8, base = _)
```

```
#> [1] 3
```

```
2 %>% log(8, base = .)
```

- Equivalent to `log(8, base = 2)`

The purrr Package: `map()`

- `sapply()` (base R) is convenient but its output type is unpredictable
- `purrr::map()` always returns a **list**; `map_dbl()`, `map_chr()`, etc. guarantee a specific vector type

```
compute_s_n <- function(n) sum(1:n)
n <- 1:25

s_n <- map_dbl(n, compute_s_n)
s_n[1:10]
```

```
#>  [1]  1  3  6 10 15 21 28 36 45 55
```

purrr: map_df()

- map_df() requires the function to return a data frame, and row-binds all the results into one tibble

```
compute_s_n <- function(n) tibble(n = n, sum = sum(1:n))
s_n <- map_df(n, compute_s_n)
s_n |> head()
```

```
#> # A tibble: 6 x 2
#>       n   sum
#>   <int> <int>
#> 1     1     1
#> 2     2     3
#> 3     3     6
#> 4     4    10
#> 5     5    15
#> 6     6    21
```

Section 8

Tidyverse Conditionals

case_when()

- A vectorized, readable alternative to nested `ifelse()` statements
- Evaluates conditions **in order**; the first `TRUE` match wins

```
x <- c(-2, -1, 0, 1, 2)
case_when(x < 0 ~ "Negative",
          x > 0 ~ "Positive",
          TRUE  ~ "Zero")
```

```
#> [1] "Negative" "Negative" "Zero"     "Positive" "Positive"
```

case_when(): Custom Regions Example

```
murders |>
  mutate(group = case_when(
    abb %in% c("ME", "NH", "VT", "MA", "RI", "CT") ~ "New England",
    abb %in% c("WA", "OR", "CA") ~ "West Coast",
    region == "South" ~ "South",
    TRUE ~ "Other")) |>
  group_by(group) |>
  summarize(rate = sum(total) / sum(population) * 10^5)
```

```
#> # A tibble: 4 x 2
#>   group        rate
#>   <chr>       <dbl>
#> 1 New England  1.72
#> 2 Other        2.71
#> 3 South        3.63
#> 4 West Coast   2.90
```

between()

- Shorthand for checking whether a value falls in a closed interval

```
between(x, a, b)
# equivalent to:
x >= a & x <= b
```

```
between(murders$rate, 2, 3) |> sum()
```

```
#> [1] 12
```

Section 9

Wrap-Up

Summary

- **Tidy data**: one row per observation, one column per variable
- `dplyr` **verbs**: `mutate()`, `filter()`, `select()`, `arrange()`, `summarize()`, `group_by()`, `slice_max()` / `slice_min()`
- **The pipe** (|> / %>%) chains verbs into readable pipelines
- **Tibbles**: safer, friendlier data frames used throughout the tidyverse
- `purrr::map()` **family**: type-safe alternatives to `sapply()`
- `case_when()` / `between()`: clean vectorized conditionals

Practice

- Try the chapter exercises using `murders`, `heights`, and other built-in datasets (`co2`, `ChickWeight`, `BOD`, `NHANES`)
- Rewrite a base-R script you know using a `|>` pipeline of `dplyr` verbs
- Reference: Rafael Irizarry, *Introduction to Data Science*, Chapter 4 "The Tidyverse"
  https://rafalab.dfci.harvard.edu/dsbook-part-1/R/tidyverse.html