

Introduction to the Tidyverse

W. Evan Johnson, Ph.D.
Professor, Division of Infectious Disease
Director, Center for Data Science
Rutgers University – New Jersey Medical School

2024-05-23

Tidy format (murders data)

We say that a data table is in **tidy** format if each row represents one observation and columns represent the different variables available for each of these observations. For example, the following data is in tidy format:

```
data(murders)
```

```
head(murders)
```

##	state	abb	region	population	total
## 1	Alabama	AL	South	4779736	135
## 2	Alaska	AK	West	710231	19
## 3	Arizona	AZ	West	6392017	232
## 4	Arkansas	AR	South	2915918	93
## 5	California	CA	West	37253956	1257
## 6	Colorado	CO	West	5029196	65

Not tidy format (fertility)

The following dataset is organized, but not tidy. Why?

```
path <- system.file("extdata", package = "dslabs")
filename <- file.path(path, "fertility-two-countries-example.csv")
wide_data <- read_csv(filename)
```

```
## Rows: 2 Columns: 57
## -- Column specification -----
## Delimiter: ","
## chr (1): country
## dbl (56): 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970,
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this mes
select(wide_data, country, `1960`:`1962`) %>% as.data.frame()
```

```
##      country 1960 1961 1962
## 1      Germany 2.41 2.44 2.47
## 2  South Korea 6.16 5.99 5.79
```

Tidy format (fertility)

Here is how we would organize these data to be tidy:

```
data("gapminder")
tidy_data <- gapminder %>%
  filter(country %in% c("South Korea", "Germany") &
         !is.na(fertility)) %>%
  select(country, year, fertility)
head(tidy_data, 6)
```

```
##      country year fertility
## 1      Germany 1960      2.41
## 2 South Korea 1960      6.16
## 3      Germany 1961      2.44
## 4 South Korea 1961      5.99
## 5      Germany 1962      2.47
## 6 South Korea 1962      5.79
```

Tidy format

The same information is provided, but there are important differences in the format. For the **tidyverse** packages to be optimally used, data need to be reshaped into 'tidy' format. The advantage of working in tidy format allows the data analyst to focus on more important aspects of the analysis rather than the format of the data.

Tidy data wrangling

The **dplyr** package, which is part of the **tidyverse**, presents a basic grammar for wrangling tidy data:

- `mutate()`: add or modify existing columns
- `select()`: take a subset of the columns (variables)
- `filter()`: take a subset of the rows (observations)
- `arrange()`: sort the rows
- `summarize()`: aggregate data across rows

Note an important point: most dplyr functions (and most functions in the tidyverse) input a tibble and then output a modified tibble!

Mutate

The function **mutate** takes the data frame, the instructions for the new columns in next arguments, and returns a modified data frame. For example:

```
head(murders)
```

##		state	abb	region	population	total
## 1		Alabama	AL	South	4779736	135
## 2		Alaska	AK	West	710231	19
## 3		Arizona	AZ	West	6392017	232
## 4		Arkansas	AR	South	2915918	93
## 5		California	CA	West	37253956	1257
## 6		Colorado	CO	West	5029196	65

Mutate

To add murder rates, we mutate as follows:

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

##	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
## 4	Arkansas	AR	South	2915918	93	3.189390
## 5	California	CA	West	37253956	1257	3.374138
## 6	Colorado	CO	West	5029196	65	1.292453

Filter

Now suppose that we want to filter the data table to only show the entries for which the murder rate is lower than 0.71. We do this as follows:

```
filter(murdersRate, 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

Select

If we want to view just a few of our columns, we can use the following:

```
murdersRate <- mutate(murders,  
  rate = total / population * 100000  
)  
murdersRateSelect <- select(murdersRate, state, rate)  
filter(murdersRateSelect, rate <= 0.71)
```

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

Nesting functions

Instead of defining new objects along the way, we could do everything in one complex nested function:

```
filter(  
  select(  
    mutate(murders, rate = total / population * 100000),  
    state, rate  
  ),  
  rate <= 0.71  
)
```

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

This is fairly concise but a little confusing. Is there a better way?

In the previous example, we performed the following wrangling operations:

original data → mutate → select → filter

As with Unix, we can perform a series of operations in R by sending the results of one function to another using the **pipe operator**: `%>%`. As of R version 4.1.0, you can also use `|>`.

The pipe is a combination of characters that when used properly does two things: *It shortens and simplifies the code* and it makes the code intuitive to read.

Pipes

All the pipe does is provide **forward application** of an object to the first argument of a function. The pipe sends left side of the input to the function to the right of the pipe. For example, if we wanted to calculate

$$\log_2(\sqrt{16})$$

We could use:

```
16 %>% sqrt() %>% log2()
```

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

Since the pipe sends values to the first argument, we can define other arguments as follows:

```
16 %>% sqrt() %>% log(base = 2)
```

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

Pipes (murders)

Completing the prior tibble operation using pipes:

```
murders %>%  
  mutate(rate = total / population * 100000) %>%  
  select(state, rate) %>%  
  filter(rate <= 0.71)
```

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

Note that as you can see, the pipe operators (`%>%` or `|>`) are not specific to the tidyverse, in fact they come from the **magrittr** package (which is loaded by the tidyverse and dplyr libraries)

Arrange

We know about the **order** and **sort** functions, but for ordering entire tables, the **arrange** function is much more useful. For example, here we order the states murder rate:

```
murdersRate %>%  
  arrange(rate) %>%  
  head()
```

##	state	abb	region	population	total	rate
## 1	Vermont	VT	Northeast	625741	2	0.3196211
## 2	New Hampshire	NH	Northeast	1316470	5	0.3798036
## 3	Hawaii	HI	West	1360301	7	0.5145920
## 4	North Dakota	ND	North Central	672591	4	0.5947151
## 5	Iowa	IA	North Central	3046355	21	0.6893484
## 6	Idaho	ID	West	1567582	12	0.7655102

Arrange (descending order)

Note that the default behavior is to order in ascending order. The function **desc** transforms a vector so that it is in descending order. To sort the table in descending order, we can type:

```
murdersRate %>%  
  arrange(desc(rate)) %>%  
  head()
```

##	state	abb	region	population	total	rate
## 1	District of Columbia	DC	South	601723	99	16.452753
## 2	Louisiana	LA	South	4533372	351	7.742581
## 3	Missouri	MO	North Central	5988927	321	5.359892
## 4	Maryland	MD	South	5773552	293	5.074866
## 5	South Carolina	SC	South	4625364	207	4.475323
## 6	Delaware	DE	South	897934	38	4.231937

Nested sorting

If we are ordering by a column with ties, we can use a second (or third) column to break the tie. for example:

```
murdersRate %>%  
  arrange(region, rate) %>%  
  head()
```

##	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
## 6	New York	NY	Northeast	19378102	517	2.6679599

Summarize

The **summarize** function computes summary statistics in an intuitive way. The 'heights' dataset includes heights and sex reported by students in an in-class survey.

```
data(heights)
heights %>%
  filter(sex == "Female") %>%
  summarize(
    avg = mean(height),
    std_dev = sd(height)
  )
```

```
##           avg  std_dev
## 1 64.93942  3.760656
```

Group then summarize with 'group_by'

A common operation in data exploration is to first split data into groups and then compute summaries for each group. For example, we may want to compute the average and standard deviation for men's and women's heights separately. We can do the following

```
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
```

More on the tidyverse

In as you learn more about the tidyverse you will learn a few more tidyverse operations, including the **inner_join**, **left_join**, **pull**, **dot**, and **do** functions, and the **tidyr** package.

Session info

```
sessionInfo()
```

```
## R version 4.4.0 (2024-04-24)
## Platform: aarch64-apple-darwin20
## Running under: macOS Sonoma 14.2.1
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRlapack.dylib; LAPACK version 3.11.0
##
## locale:
##  [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## time zone: America/Denver
## tzcode source: internal
##
## attached base packages:
## [1] stats      graphics  grDevices  utils      datasets  methods    base
##
## other attached packages:
##  [1] dslabs_0.8.0    lubridate_1.9.3 forcats_1.0.0  stringr_1.5.1
##  [5] dplyr_1.1.4     purrr_1.0.2    readr_2.1.5    tidyr_1.3.1
##  [9] tibble_3.2.1    ggplot2_3.5.1  tidyverse_2.0.0
##
## loaded via a namespace (and not attached):
##  [1] bit_4.0.5      gtable_0.3.5    crayon_1.5.2    compiler_4.4.0
##  [5] tidyselect_1.2.1 parallel_4.4.0   scales_1.3.0    yaml_2.3.8
##  [9] fastmap_1.1.1  R6_2.5.1        generics_0.1.3  knitr_1.46
## [13] munsell_0.5.1  pillar_1.9.0    tzdb_0.4.0      rlang_1.1.3
## [17] utf8_1.2.4     stringi_1.8.3   xfun_0.43       bit64_4.0.5
## [21] timechange_0.3.0 cli_3.6.2        withr_3.0.0     magrittr_2.0.3
## [25] digest_0.6.35  grid_4.4.0      vroom_1.6.5     rstudioapi_0.16.0
## [29] hms_1.1.3      lifecycle_1.0.4 vctrs_0.6.5     evaluate_0.23
## [33] glue_1.7.0     fansi_1.0.6     colorspace_2.1-0 rmarkdown_2.26
## [37] tools_4.4.0    pkgconfig_2.0.3 htmltools_0.5.8.1
```