Intro to dplyr

Jongbin Jung

Wrangling Data with `dplyr`

Selecting rows (1/5)

Selecting columns (2/5)

Creating new columns (3/5)

Sorting (4/5)

Aggregating (5/5)

Multiple (Chained) Operations

Exercise Solution

Dependencies

- Latest version ($\geq$ `3.1.2`) of `R` (*free* from `https://www.r-project.org/`)
- Latest version of `Rstudio` (also *free* from `https://www.rstudio.com/`)
- `tidyverse` package

```
# tidyverse package now includes
install.packages("tidyverse")
library("tidyverse")
```

- Slides available here: `https://github.com/5harad/css/raw/master/crash-course/r/2-dataframes/dplyr.pdf`
- Many more resources available!

Wrangling Data with `dplyr`

Introduction to dplyr

- dplyr is a package that provides a convenient framework (along with a handful of useful functions) for wrangling data (frames)
- dplyr is a part of the tidyverse, so if you load tidyverse, dplyr is also loaded
- You can also, but don“t have to, install and load the dplyr as a standalone package like you would any other R package

```
# Install, if you haven"t already.
# Only need to do this once on a single machine.
install.packages("dplyr")
# load package into workspace
library("dplyr")
```

Data Frames: Introduction

- Data frames are the primary representation of data in `R`
- You can think of a data frame as a two-dimensional *table* of data
- It helps your sanity to always think of data frames as a table where

```
Each column represents a variable/feature
Each row represents an observation/instance
```

- For example: student score data

ID	name	age	score
1	A	16	85
2	B	17	89
3	C	15	81

Data Frames: First Impression

- We'll primarily use the `diamonds` data that's included with `tidyverse`
- Take a look by typing `diamonds` in the `R` console
- `diamonds` is a dataset containing the prices and other attributes of almost 54,000 diamonds. Included variables are:
  - price, carat, cut, color, clarity, dimensions (x, y, z, depth, table)
- See documentation for more details

```
?diamonds
```

Verbs

- A *verb* in the world of `dplyr` is a function that
  - takes a data frame as its first argument, and
  - returns another data frame as a result
- For example, the `head()` function can be considered a verb

```
head(diamonds, n = 3)
```

- Note that the result of the `head()` function is another data frame (in this case, with 3 rows)
- The **core idea of `dplyr`** is that most of your data manipulation needs can be satisfied with 5 basic verbs (or 4, depending on how you categorize them)

Five basic verbs

- The five basic verbs of `dplyr` and associated actions are presented below

verb	action
`filter()`	select a subset of *rows* by specified conditions
`select()`	select a subset of *columns*
`mutate()`	create a *new column* (often from existing columns)
`arrange()`	reorder (sort) *rows* by values of specified *column*(s)
`summarize()`	aggregate values and reduce to single value

- Some verbs have additional options or convenient wrappers

Selecting rows (1/5)

Selecting Rows: `filter()`

- Select a subset of *rows*
- Multiple conditions can be used
- Use `&` to specify `AND` conditions
- Use `|` to specify `OR` conditions
- `AND(&)/OR(|)` operations can be used together (where default behavior for multiple conditions is `AND`)

```
filter(diamonds, cut == "Ideal" & carat > 3)
filter(diamonds, carat <= .2 |  carat >= 3)
filter(diamonds,
       carat <= .2 |  carat >= 3, color == "D")
```

- Use `%in%` to match on a collection of values

```
filter(diamonds, cut %in% c("Fair", "Ideal"))
```

Selecting Rows: `slice()`

- To select rows by numerical index (position), use `slice()`
- For example, to select the first 10 rows

```
slice(diamonds, 1:10)
```

- or to select the last 10 rows

```
slice(diamonds, (n() - 9):n())
```

- Use `n()` inside a `dplyr` verb to to indicate the *number of rows* of the data frame

Selecting columns (2/5)

Selecting Columns: `select()`

- Select a subset of *columns*
- Either specify the columns that you want to select

```
select(diamonds, cut, price)
```

- Or specify the columns you wish to drop

```
select(diamonds, -x, -y, -z)
```

Selecting Columns: `select()` (cont'd)

- `dplyr` provides useful helper functions you can use to `select()` columns that match specific criteria such as
  - `starts_with(x)`: names that start with `x`
  - `ends_with(x)`: names that end with `x`
  - `contains(x)`: names that contain `x`
  - `matches(x)`: names that match the (regular expression) `x`
- See the documentation for more details

```
?dplyr::select
```

- While you can assign new column names with `select()` the convenience function `rename()` lets you rename columns while retaining the rest of the data frame

```
rename(diamonds, cut_type = cut)
```

Creating new columns (3/5)

Create New Columns: `mutate()`

- Create new columns, usually as a function of existing columns
- You can refer to new columns you just created, inside the same `mutate()` function

```
mutate(diamonds,
       price_per_carat = price / carat,
       volume = x * y * z,
       # Use the volume column we just created
       # to create yet another price_per_volume column
       price_per_volume = price / volume
       )
```

- Use `transmute()` to create a new data frame *just from* the new column(s)

```
transmute(diamonds, carat,
          price_per_carat = price / carat)
```

Sorting (4/5)

Sorting Rows by Column Value: `arrange()`

- Reorder the rows of a data frame by the specified column"s value
- Multiple conditions are arranged from left to right
- Use `desc()` to arrange in descending order

```
arrange(diamonds, carat, price)
arrange(diamonds, carat, desc(price))
arrange(diamonds, desc(carat), desc(price))
```

Aggregating (5/5)

Aggregate Data: `summarize()`

- Aggregate/collapse the data into a single row
- Think of as applying a function to columns

```
summarize(diamonds, avg_price = mean(price))
```

```
## # A tibble: 1 × 1
##   avg_price
##       <dbl>
## 1      3933
```

Recap:

- The five basic verbs:

verb	action
`filter()`	select a subset of *rows* by specified conditions
`select()`	select a subset of *columns*
`mutate()`	create a *new column* (often from existing columns)
`arrange()`	reorder (sort) *rows* by values of specified *column*(s)
`summarize()`	aggregate values and reduce to single value

- But what about . . .
  - Average price of diamonds for each cut type?
  - Largest (carat) diamond in each color category?
  - . . .

Bad example:

- A natural, but tedious way to compute:
- Average price of diamonds for each cut type?
  - use `filter` to create five different data frames, one for each `cut` type
  - use `summarize` to compute the mean price for each data frame
- Largest (carat) diamond in each color category?
  - use `filter` to create seven different data frames, one for each `color` category
  - use `arrange` to sort in descending order of `carat` for each data frame
  - use `slice` to get the first row from each of the `arranged` data frames
- The pattern:
  - `group` by some categorical value
  - do some operations, but to each category of the `group`

Split-Apply-Combine

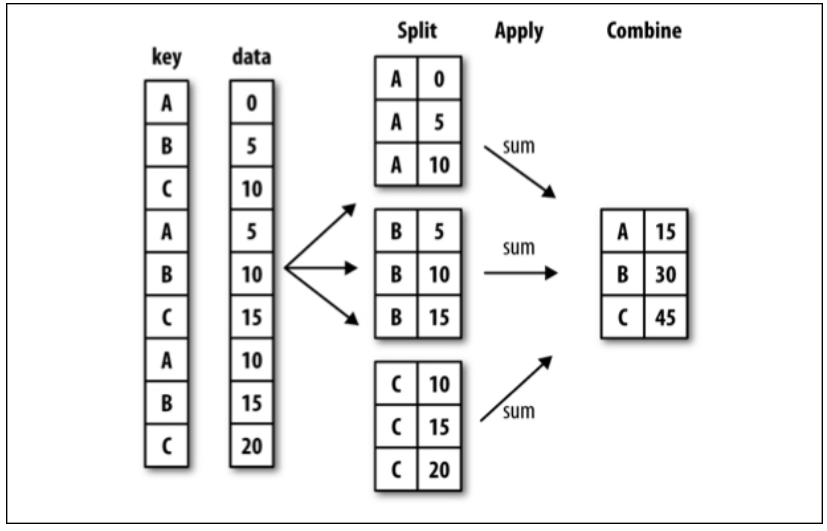

Figure 1: Illustration of SAC

Grouped Operations

- If a data frame is *grouped*, operations are applied to each group separately, and the results are combined back to a single data frame
- The `group_by()` verb lets you specify the *grouping* variables (e.g., `cut`, `color`)

```
diamonds_by_cut <- group_by(diamonds, cut)
```

- When the data frame is `group_by`'d, all verbs/functions will operate as if each category of the grouping variable is its own data frame, e.g.,

verb	group specific action
`arrange()`	sort rows within each group
`slice()`	extract rows within each group
`summarize()`	aggregate values group-wise
`n()`	count the number of rows in each group

Grouped `slice()`

- Retrieve one diamond from each cut

```
slice(diamonds_by_cut, 1)
```

```
## Source: local data frame [5 x 10]
## Groups: cut [5]
##
##   carat       cut color clarity depth table price
##   <dbl>     <ord> <ord>   <ord> <dbl> <dbl> <int>
## 1  0.22      Fair     E     VS2  65.1    61   337
## 2  0.23      Good     E     VS1  56.9    65   327
## 3  0.24 Very Good     J    VVS2  62.8    57   336
## 4  0.21   Premium     E     SI1  59.8    61   326
## 5  0.23     Ideal     E     SI2  61.5    55   326
## # ... with 3 more variables: x <dbl>, y <dbl>,
## #   z <dbl>
```

Grouped `summarize()`

- Retrieve (1) number of diamonds and (2) average price by cut type

```
summarize(diamonds_by_cut,
          count = n(),
          avg_price = mean(price))
```

```
## # A tibble: 5 × 3
##         cut count avg_price
##       <ord> <int>     <dbl>
## 1      Fair  1610      4359
## 2      Good  4906      3929
## 3 Very Good 12082      3982
## 4   Premium 13791      4584
## 5     Ideal 21551      3458
```

Multiple (Chained) Operations

Multiple (Chained) Operations

- Proportion of different `colors` in each `cut` category
- We can achieve the desired result with a few operations
  1. **`group_by`** `cut`
  2. **`mutate`** to create a column with total diamonds for each `cut`
  3. re-**`group_by`** `cut` and `color`
  4. Use the new column from above to `summarize` the proportion of each `color` within each `cut`
- Note that `dplyr` verbs do **not** modify the original data frame (i.e., they don't have side effects)
  - This is generally a good thing, since it guarantees the integrity of your data
  - But it makes multiple operations on a data frame difficult

Multiple Operations: The OK Way

- One way to perform multiple operations is to save intermediate data frames as new data frames
- This method delivers desired results, but makes your workspace quite messy (i.e., you"ll end up with a workspace full of intermediate results)

```
by_cut <- group_by(diamonds, cut)
count_cuts <- mutate(by_cut, N = n())
by_cut_and_color <- group_by(count_cuts, cut, color)
proportions <- summarize(by_cut_and_color,
                         prop = mean(n()/N))
```

- This method might be prefered if you need the intermediate results in the future
- If not, there is a better way to chain multiple operations with `dplyr`

The Pipe Operator %>%

- The pipe operator, takes the output from its left-hand side, and uses it as the first argument to whatever is on its right-hand side
- For example:

```
by_cut <- group_by(diamonds, cut)
count_cuts <- mutate(by_cut, N = n())
```

is equivalent to

```
count_cuts <- diamonds %>%
  group_by(cut) %>%
  mutate(N = n())
```

(except in the second case, the by_cut data frame is never created)

The Pipe Operator %>%

- Using the pipe, we can complete the entire task without saving any intermediate data frames

```
proportions <- diamonds %>%
  group_by(cut) %>%
  mutate(N = n()) %>%
  group_by(cut, color) %>%
  summarize(prop = mean(n()/N))
```

- No need to save intermediate results
- Easier to read (i.e., you can follow the operations step-by-step without too much mental accounting)

dplyr: Exercise

- Find the most expensive diamond for each `cut`.
- How many 1 carat diamonds are "Premium" `cut`, and what are the min/median/max prices?
- What is the average price for diamonds grouped by 0.1 `carats`?

Exercise Solution

WARNING

- Solutions to the exercise are presented in the next slide
- Try the exercise before proceeding!

Solutions

- Find the most expensive diamond for each cut.

```
diamonds %>%
  group_by(cut) %>%
  arrange(desc(price)) %>%
  slice(1)
```

```
## Source: local data frame [5 x 10]
## Groups: cut [5]
##
##   carat       cut color clarity depth table price
##   <dbl>     <ord> <ord>   <ord> <dbl> <dbl> <int>
## 1  2.01      Fair     G     SI1  70.6    64 18574
## 2  2.80      Good     G     SI2  63.8    58 18788
## 3  2.00 Very Good     G     SI1  63.5    56 18818
## 4  2.29   Premium     I     VS2  60.8    60 18823
## 5  1.51     Ideal     G      IF  61.7    55 18806
## # ... with 3 more variables: x <dbl>, y <dbl>,
```

Solutions

- What is the average price for diamonds grouped by 0.1 carats?

```
diamonds %>%
  mutate(carat_bin = round(carat, digits = 1)) %>%
  group_by(carat_bin) %>%
  summarize(avg_price = mean(price))
```

```
## # A tibble: 38 × 2
##   carat_bin avg_price
##       <dbl>     <dbl>
## 1       0.2     506.1
## 2       0.3     703.2
## 3       0.4     922.7
## 4       0.5    1589.7
## 5       0.6    1871.7
## # ... with 33 more rows
```

Reference

- A great "cheat sheet" for wrangling data with `dplyr` and `tidyr` is available for free at `https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf`
- Introductory text book by the creator of `tidyverse`: `http://r4ds.had.co.nz/`