Intro to `R` - `dplyr`

a **Data Science Drop-in** Tutorial

by **Jongbin Jung**
(jongbin@stanford.edu)

Dependencies

- Install **`dplyr`** package and sample data

```
install.packages(c("dplyr", "nycflights13"))
```

- Load them to your workspace

```
library("dplyr")
library("nycflights13")
```

The **`nycflights13 data.frame (flights)`** contains all 336,776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transporation Statistics, and is documented in **`?nycflights13`**.

`nycflights13` data

- take look at the `flights data.frame`

```
  year month day dep_time dep_delay arr_time arr_delay carrier tailnum flight origin dest air_time distance hour minute
1 2013     1   1      517         2      830        11      UA  N14228   1545    EWR  IAH      227     1400    5     17
2 2013     1   1      533         4      850        20      UA  N24211   1714    LGA  IAH      227     1416    5     33
3 2013     1   1      542         2      923        33      AA  N619AA   1141    JFK  MIA      160     1089    5     42
   ...   ... ...      ...       ...      ...       ...     ...     ...    ...    ...  ...      ...      ...  ...    ...
```

- what questions could you ask with this data?
  - how many flights were there each day?
  - what's the mean departure delay for flights every month / day
  - what else?

verb

- A verb in **R** is a function that takes a **data.frame** as its first argument, for example, try

- The key concept of **dplyr**:
  most of your data manipulation needs can be satisfied with 5 basic **verbs**
  (4 verbs, depending on how you categorize them)

5 basic `verb`s

verb	action
`filter()`	select a subset of **rows** by conditions
`select()`	select a subset of **columns** from the data
`mutate()`	create a **new column** (usually based on existing columns)
`arrange()`	reorder (sort) **rows**
`summarise()`	aggregate values and reduce to single value

selecting rows - `filter()`

- select a subset of **rows**
- multiple conditions can be used
- use & to specify an AND operation

```
filter(flights, tailnum == "N14228" & arr_delay > 10)
```

- use | to specify an OR operation

```
filter(flights, tailnum == "N14228" | tailnum == "N24211")
```

- mix AND/OR operations (default behavior is AND)

```
filter(flights, tailnum == "N14228" | tailnum == "N24211", arr_delay > 10)
```

selecting rows - `slice()`

- similarly, select a subset of **rows** by position using `slice()`
- for example, to select the first 10 rows

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

- or to select the last 10 rows

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

- use `n()` inside a `dplyr` verb to use the *number of rows* in the data

selecting columns - `select()`

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

```
select(flights, c(carrier, tailnum))
```

- or specify the columns you don't want to select

```
select(flights, -c(year, month, day))
```

- also works without the `c()`

```
select(flights, carrier, tailnum)
select(flights, -year, -month, -day)
```

selecting columns - `select()`

- use helper functions such as `starts_with()`, `ends_with()`, `matches()` and `contains()`

```
select(flights, starts_with("dep"))
select(flights, matches("_"))
select(flights, contains("delay"))
```

- assign new column names with `select()`

```
select(flights, tail_num = tailnum)
```

- to keep the rest of the data, use `rename()`

```
rename(flights, tail_num = tailnum)
```

create columns - `mutate()`

- create new **columns**, usually as a function of old columns

```
mutate(flights, gain = arr_delay - dep_delay,
                speed = distance / air_time * 60)
```

- you can also refer to columns that you just created

```
mutate(flights, gain = arr_delay - dep_delay,
                gain_per_hour = gain / (air_time / 60))
```

create columns - `mutate()`

- if you just want to keep the new columns, use `transmute()` instead

```
transmute(flights, gain = arr_delay - dep_delay,
                   gain_per_hour = gain / (air_time / 60))
```

sorting rows - `arrange()`

- reorder (sort) the data by specified **rows**
- multiple conditions are arranged from left-to-right

```
arrange(flights, year, month, day)
```

- use `desc()` to arrange in descending order

```
arrange(flights, year, desc(month), day)
arrange(flights, year, month, desc(day))
arrange(flights, year, desc(month), desc(day))
```

summarise()

- aggregate/collapse data into single row

```
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
```

- more useful with grouped operations (see next)

group operations

- indicate a grouping variable with group_by()

```
flights_by_day <- group_by(flights, day)
```

- some verbs have specific behavior with groups

verb	group specific actions
`arrange()`	orders first by grouping variable
`slice()`	extract rows within each group
`summarise()`	aggregate values for each group, and reduce to single value

group `slice()`

- retrieve the first 2 rows of each day

```
slice(flights_by_day, 1:2)
```

group `summarise()`

- `summarise()` makes much more sense when used with grouped data
- retrieve (1) number of flights, (2) average distance, and (3) average arrival delay **<u>for each day</u>** (i.e., **<u>for flights grouped by days)</u>**

```
summarise(flights_by_day,
  count = n(),
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE))
```

multiple (chained) operations

"find days when the mean arrival delay
OR departure delay was greater than 30"

group data by date `(year, month, day)`

aggregate each group by mean arrival/departure delay

filter aggregated result `(mean arr_delay > 30 | mean dep_delay > 30)`

- **`dplyr`** verbs won't affect your original data (this is generally a good/safe thing, but potentially makes it difficult to do multiple operations on a single **`data.frame`**)
- there are two (acceptable) ways of doing this, and one is probably better than the other

multiple (chained) operations

"find days when the mean arrival delay
OR departure delay was greater than 30"

group data by date `(year, month, day)`

aggregate each group by mean arrival/departure delay

filter aggregated result `(mean arr_delay > 30 | mean dep_delay > 30)`

```
flights_by_date <- group_by(flights, year, month, day)
summary_by_date <- summarise(flights_by_date,
  arr = mean(arr_delay, na.rm = TRUE),
  dep = mean(dep_delay, na.rm = TRUE))
big_delay_days <- filter(summary_by_date, arr > 30 | dep > 30)
```

this isn't too bad. but it's not very nice.

multiple (chained) operations

"find days when the mean arrival delay
OR departure delay was greater than 30"

a better way to do this with **`dplyr`**

the pipe operator

%>%

```
some_function() %>% another_function()
```

use the result from this...

... as the first argument for this

multiple (chained) operations

"find days when the mean arrival delay
OR departure delay was greater than 30"

group data by date `(year, month, day)`

aggregate each group by mean arrival/departure delay

filter aggregated result `(mean arr_delay > 30 | mean dep_delay > 30)`

```
flights %>%
  group_by(year, month, day) %>%
  summarise(arr = mean(arr_delay, na.rm = TRUE),
            dep = mean(dep_delay, na.rm = TRUE)) %>%
  filter(arr > 30 | dep > 30)
```

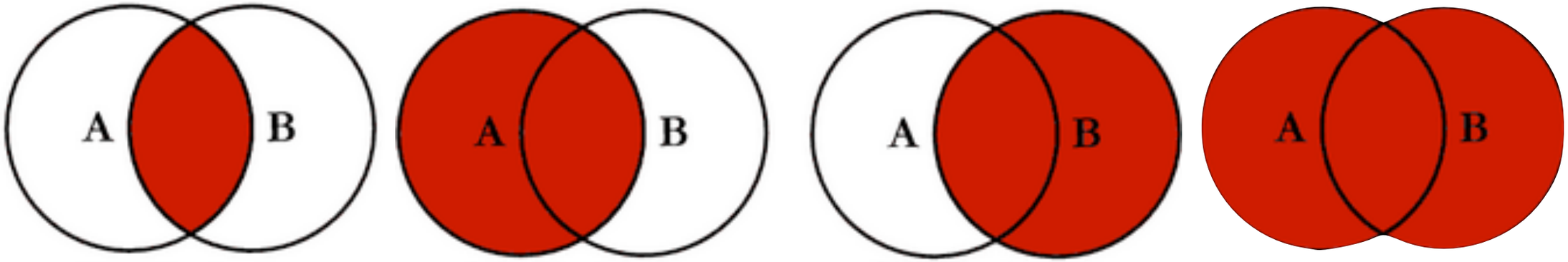
this is easier to read. no need to save intermediate results.

Today's **CHALLENGE**s

Find the average speed (`distance / air_time * 60`) by each carrier (ignore any `NA`s), and sort the data in descending order of average speed

Find the number of flights longer than 10 hours by each carrier in April

joins (merge)

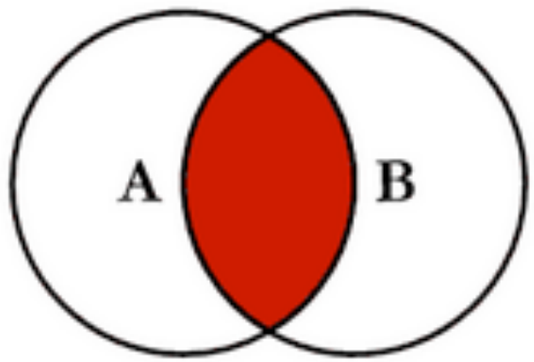


inner left right outer

- **merge(x, y, ...)**
  merge two data frames by common columns or row names,
  or do other versions of database *join* operations.

- Not really a **`dplyr`** function, but works well

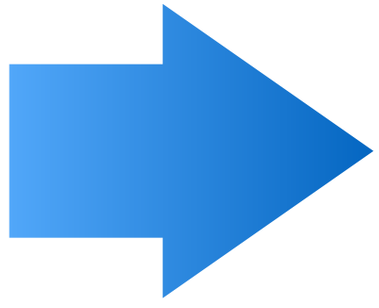
joins (inner)



- **merge(x, y,** all.x = FALSE, all.y = FALSE**)**
  default behavior is that only rows with data from both x and y are included in the output. this can also be specified with the arguments `all.x = FALSE` and `all.y = FALSE`

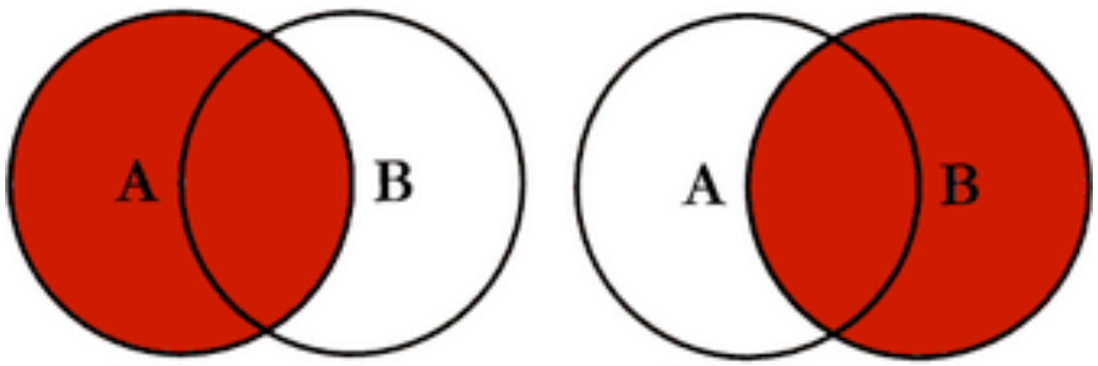
ID	sex
1	M
2	F
3	M
4	F

ID	age
2	20
3	18
6	23



ID	sex	age
2	F	20
3	M	18

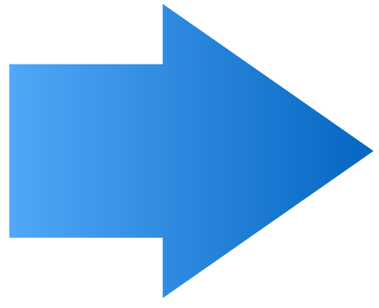
joins (left / right)



- **merge(x, y, all.x = TRUE,** all.y = FALSE**)**
  when `all.x = TRUE`, extra rows will be added to the output, one for each row in x that has no matching row in y. These rows will have NAs in those columns that are usually filled with values from y (right joins are achieved with `all.y = TRUE`)

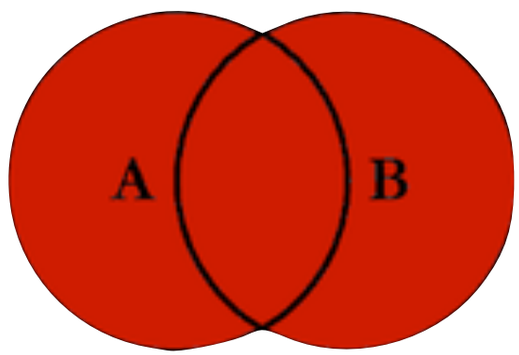
ID	sex
1	M
2	F
3	M
4	F

ID	age
2	20
3	18
6	23



ID	sex	age
1	M	*NA*
2	F	20
3	M	18
4	F	*NA*

joins (outer)



- **merge(x, y, all = TRUE)**
  `all = TRUE` is shorthand for `all.x = TRUE` and `all.y =TRUE`. If `all = TRUE` , then all the rows in both x and y are included in the output.

ID	sex
1	M
2	F
3	M
4	F

ID	age
2	20
3	18
6	23

ID	sex	age
1	M	*NA*
2	F	20
3	M	18
4	F	*NA*
6	*NA*	23