Intro to `R` - `dplyr`

a **Data Science Drop-in** Tutorial

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

Before We Begin

- Slides / Code available at:
  https://5harad.com/drop-in/tutorials

- Git repo at:
  https://github.com/5harad/datascience

- Get latest ...
  R (http://cran.r-project.org/)
  RStudio (http://www.rstudio.com)

- Install `dplyr` package and sample data

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

Before We Begin

- 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 336776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transporation Statistics, and is documented in **`?nycflights13`**.

verb

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

- Most of your data manipulation needs can be satisfied with 5 basic **`verb`**s

5 basic `verb`s

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

selecting rows - `filter()`

- select a subset of **rows**
- multiple conditions can be used
- conditions are combined with AND by default

```
filter(flights, month == 1, day == 1)
```

- use | to specify an OR operation

```
filter(flights, month == 1 | month == 2)
```

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

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))
```

selecting columns - `select()`

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

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

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

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

- starts_with(), ends_with(), matches() and contains().

selecting columns - `select()`

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

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

- 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))
```

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

```
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"

Bad example #1: step-by-step, saving intermediate results

```
a1 <- group_by(flights, year, month, day)
a2 <- select(a1, arr_delay, dep_delay)
a3 <- summarise(a2,
  arr = mean(arr_delay, na.rm = TRUE),
  dep = mean(dep_delay, na.rm = TRUE))
a4 <- filter(a3, arr > 30 | dep > 30)
```

> ends up with too many temporary variables

multiple (chained) operations

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

Bad example #2: wrapping function calls

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

> confusing, difficult to read

multiple (chained) operations

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

the pipe operator

%>%

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

use the result from this...

... as the first argument for this

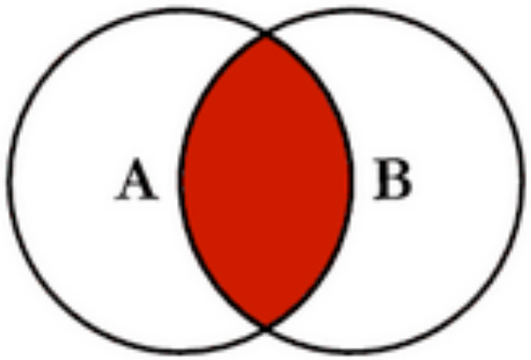
multiple (chained) operations

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

Good example: using the pipe operator (%>%)

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

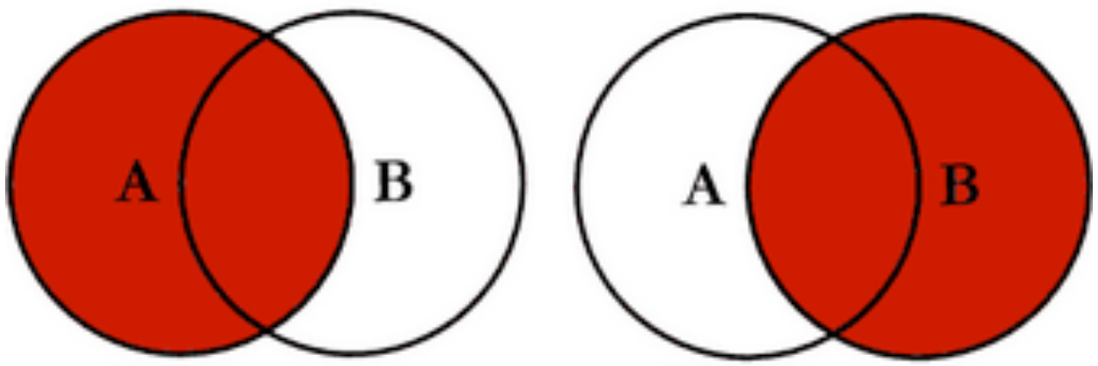
joins (inner)



- **inner_join(x, y)**
  return all rows from x where there are matching values in y, and all columns from x and y. If there are multiple matches between x and y, all combination of the matches are returned.

- **semi_join(x, y)**
  return all rows from x where there are matching values in y, keeping just columns from x.
  While an inner join will return one row of x for each matching row of y, a semi join will never duplicate rows of x.

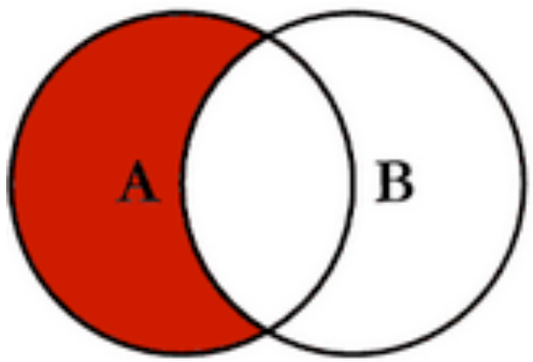
joins (left / right)



- **left_join(x, y)**
  return all rows from x, and all columns from x and y. If there are multiple matches between x and y, all combination of the matches are returned. For rows in x with no matches in y, the columns from y are marked **NA**.

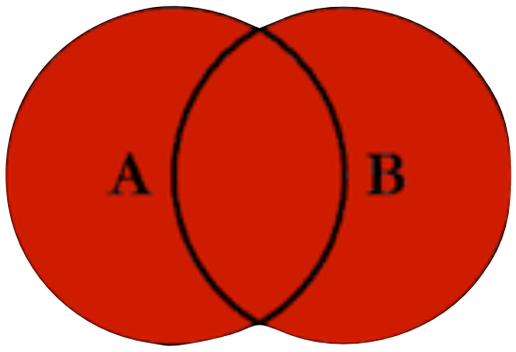
- Note that a **right** join can be easily achieved by reversing the order of x and y

joins (anti)



- **anti_join(x, y)**
  return all rows from x where there are no matching values in y, keeping just columns from x.

joins (outer)



- **`dplyr`** does not have an implementation of outer joins, yet

- in the meantime, the **`R`** function **`merge()`** does the job well enough

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