Getting familiar with `dplyr`

Rei Sanchez-Arias

Motivation

Common tasks

- Often you will need to create some *new variables* or *summaries*, or maybe you just want to *rename* the variables or *reorder* the observations to make the data a little easier to work with.
- We will focus on how to use the `dplyr` package, another core member of the `tidyverse`.

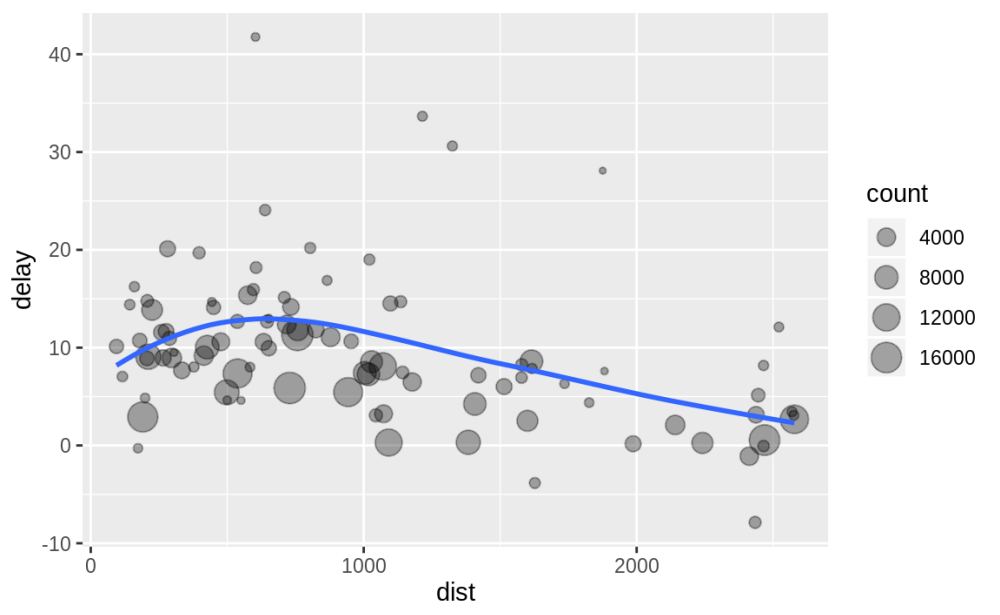
Prerequisites

- Install the `nycflights13` and `tidyverse` packages

```
library(tidyverse)
library(nycflights13)
```

Data from `nycflights13`

- This dataset contains flights departing New York City (NYC) in 2013. It contains all 336,776 flights that departed from NYC in 2013.
- The data comes from the US Bureau of Transportation Statistics, and is documented in `?flights`



Introducing `dplyr`

`dplyr` basics

1. Pick observations by their values: `filter()`
2. Reorder the rows: `arrange()`
3. Pick variables by their names: `select()`
4. Create new variables with functions of existing variables: `mutate()`
5. Collapse many values down to a single summary: `summarise()`
6. Operate on a group-by-group basis: `group_by()`

Filtering *rows*

`filter()` allows you to subset observations based on their values.

For example, we can select all flights on January 1st with:

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

Below are the first 7 rows of this filtering action:

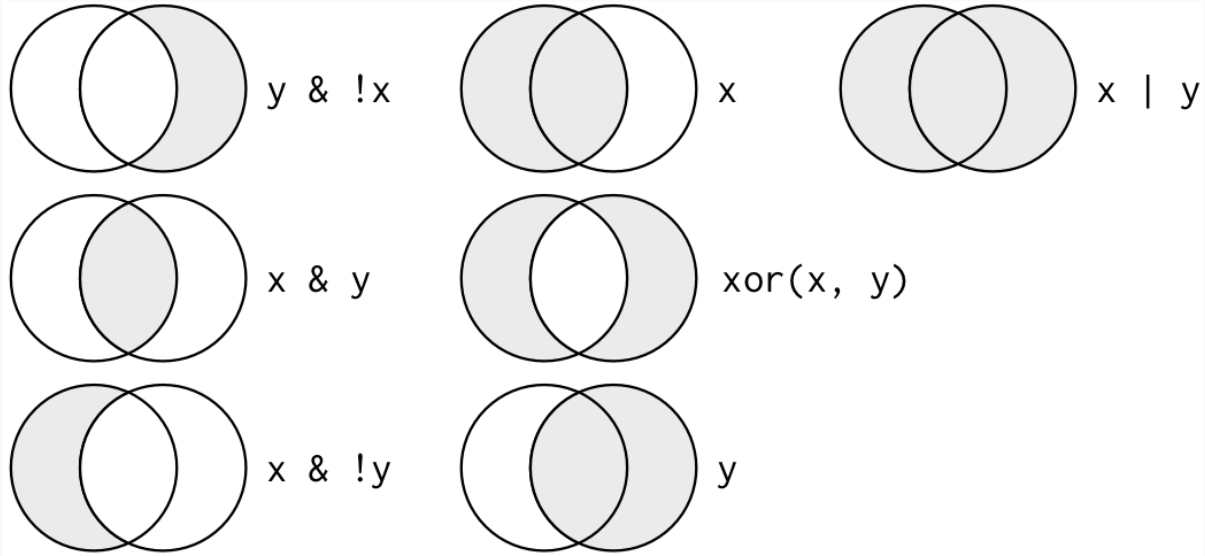
```
# A tibble: 7 x 8
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     1     1      517            515         2      830            819
2  2013     1     1      533            529         4      850            830
3  2013     1     1      542            540         2      923            850
4  2013     1     1      544            545        -1     1004           1022
5  2013     1     1      554            600        -6      812            837
6  2013     1     1      554            558        -4      740            728
7  2013     1     1      555            600        -5      913            854
```

`dplyr` functions never modify their inputs, so if you want to save the result, you will need to use the assignment operator, `<-`

Comparisons

To use filtering effectively, you have to know how to select the observations that you want using the comparison operators. R provides the standard suite: `>`, `>=`, `<`, `<=`, `!=` (not equal), and `==` (equal).

Logical operators



Flights in November OR December

The following code finds all flights that departed in November or December:

```
filter(flights, month == 11 | month == 12)
```

A useful short-hand is `x %in% y`. This will select every row where `x` is one of the values in `y`. We could use it to rewrite the code above:

```
nov_dec <- filter(flights, month %in% c(11, 12))
```

Arrange rows with `arrange()`

`arrange()` works similarly to `filter()` except that instead of selecting rows, it **changes their order**.

It takes a data frame and a set of column names to order by.

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

Use `desc()` to re-order by a column in descending order:

```
arrange(flights, desc(arr_delay))
```

Below are the first 5 rows of this filtering action:

```
# A tibble: 5 x 8
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     1     9      641            900      1301     1242           1530
2  2013     6    15     1432           1935      1137     1607           2120
3  2013     1    10     1121           1635      1126     1239           1810
4  2013     9    20     1139           1845      1014     1457           2210
5  2013     7    22      845           1600      1005     1044           1815
```

Select columns with `select()`

`select()` allows you to rapidly zoom in on a useful subset using the names of the **variables**.

```
# Select columns by name
select(flights, year, month, day)
```

There are a number of helper functions you can use within `select()`:

`starts_with("abc")`: matches names that begin with "abc".

`ends_with("xyz")`: matches names that end with "xyz".

`contains("ijk")`: matches names that contain "ijk".

Add new variables with `mutate()`

Add new columns that are functions of existing columns. That is the job of `mutate()`.

`mutate()` always adds new columns at the end of your dataset

```
# create a smaller dataset with less columns
flights_sml <- select(flights,
  year:day,
  ends_with("delay"),
  distance,
  air_time
)
```

Example

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

Note that you can refer to columns that you have just created:

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

Below are the first 4 rows of this filtering action:

```
# A tibble: 4 x 6
   year month   day  gain hours gain_per_hour
  <int> <int> <int> <dbl> <dbl>         <dbl>
1  2013     1     1     9  3.78          2.38
2  2013     1     1    16  3.78          4.23
3  2013     1     1    31  2.67         11.6
4  2013     1     1   -17  3.05         -5.57
```

Grouped summaries with `summarise()`

Another key verb is `summarise()`. It *collapses* a data frame to a single row:

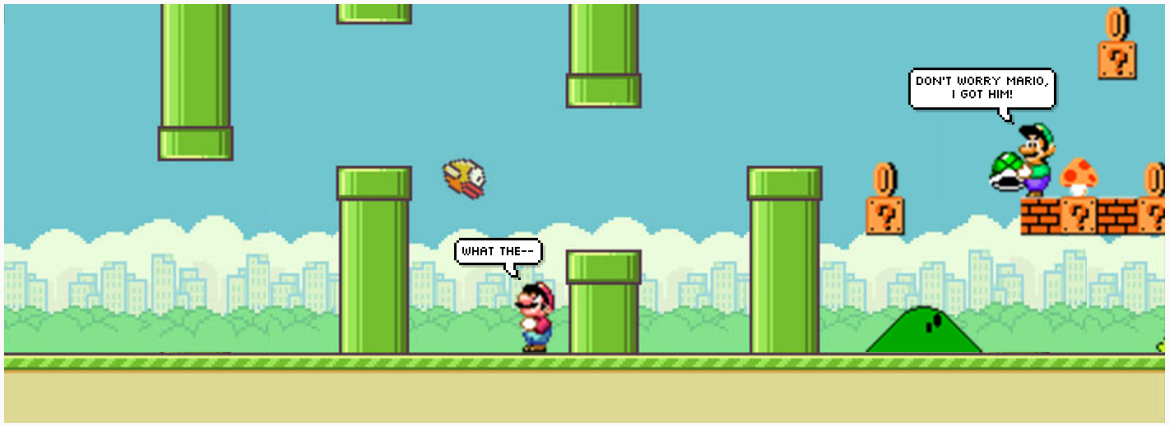
```
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
```

```
# A tibble: 1 x 1
  delay
  <dbl>
1  12.6
```

The `summarise()` function is useful when we pair it with `group_by()`.

This way, the analysis can be done for individual groups.

The Pipe



The pipe %>%

Sends the output of the LHS function to the first argument of the RHS function.

```
sum(1:8) %>%
  sqrt() %>%
  log()
```

```
[1] 1.791759
```

is equivalent to

```
log(sqrt(sum(1:8)))
```

```
[1] 1.791759
```

The Pipe

Imagine that we want to explore the relationship between the distance and average delay for each location.

There are three steps to prepare this data:

1. Group flights by destination.
2. Summarise to compute distance, average delay, and number of flights.
3. Filter to remove noisy points and Honolulu airport, which is almost twice as far away as the next closest airport.

Power of the pipe `%>%` operator

```
delays <- flights %>%
  group_by(dest) %>%
  summarise(
    count = n(),
    dist = mean(distance, na.rm = TRUE),
    delay = mean(arr_delay, na.rm = TRUE)
  ) %>%
  filter(count > 20, dest != "HNL")
```

> You can read it as a series of imperative statements: group, then summarise, then filter. As suggested by this reading, a good way to pronounce `%>%` when reading code is "then".

The `n()` function is implemented specifically for each data source and can be used from within `summarise()`, `mutate()` and `filter()`. It returns the number of observations in the current group.

> If you use RStudio, you can type the pipe with Ctrl + Shift + M if you have a PC or Cmd + Shift + M if you have a Mac.

Transformations

why `na.rm = TRUE` ?

Aggregation functions obey the usual rule of missing values: if there is any missing value in the input, the output will be a missing value! Fortunately, all aggregation functions have an `na.rm` argument which removes the missing values prior to computation

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay, na.rm = TRUE))
```

Below are the last 5 rows generated by this set of instructions:

```
# A tibble: 5 x 4
# Groups:   year, month [1]
   year month   day  mean
  <int> <int> <int> <dbl>
1  2013    12    27 10.9
2  2013    12    28  7.98
3  2013    12    29 22.3
4  2013    12    30 10.7
5  2013    12    31  7.00
```