Data Transformation with dplyr

Outline

Today, we'll cover the following:

- *dplyr*, filter and arrange rows
- Select columns and add new variables
- Grouped summaries, mutates and filters

Part I: dplyr, filter and arrange rows

dplyr

For this lesson, we'll use the dplyr package, another core member of the tidyverse.

We'll use data from `nycflights13` package.

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

filter()

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

We can select all flights on Jan 1st with:

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

We can save the result using assignment operator <-:

```
jan1 <- filter(flights, month == 1, day == 1)
```

Comparisons

R provides the standard suite: `>`, `>=`, `<`, `<=`, `!=` (not equal), and `==`(equal).

You should use `==` when testing for equality, otherwise you'll get an informative error:

```
filter(flights, month = 1)
#> Error: `month` (`month = 1`) must not be named, do you need `==`?
```

Logical operators

Boolean operators: `&` is “and”, `|` is “or”, and `!` is “not”.
The following code finds all flights that departed in November or December:

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

`x %in% y` 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))
```

```
oct_nov_dec <- filter(flights, between(month, 10, 12))
```

Logical operators

De Morgan's law: `!(x & y)` is the same as `!x | !y`, and `!(x | y)` is the same as `!x & !y`.

If you wanted to find flights that weren't delayed (on arrival or departure) by more than two hours, you could use either of the following two filters:

```
filter(flights, !(arr_delay > 120 | dep_delay > 120))
filter(flights, arr_delay <= 120, dep_delay <= 120)
```

Missing values

`NA` represents an unknown value so missing values are “contagious”: almost any operation involving an unknown value will also be unknown.

If you want to determine if a value is missing, use `is.na()`:

```
# Let x be Mary's age. We don't know how old she is.
```

```
x <- NA
```

```
is.na(x)
```

```
#> [1] TRUE
```

Missing values

`filter()` only includes rows where the condition is `TRUE`; it excludes both `FALSE` and `NA` values. If you want to preserve missing values, ask for them explicitly:

```
df <- tibble(x = c(1, NA, 3))
filter(df, is.na(x) | x > 1)
#> # A tibble: 2 x 1
#>       x
#>   <dbl>
#> 1    NA
#> 2     3
```

arrange()

`arrange()` takes a data frame and a set of column names (or more complicated expressions) to order by.

If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

arrange()

```
arrange(flights, year, month, day)
#> # A tibble: 336,776 x 19
#>    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
#> # … with 336,770 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

arrange()

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

```
arrange(flights, desc(dep_delay))
```

Find the top 5 rows.

```
flights_ordered <- arrange(flights, desc(dep_delay))
```

```
flights_ordered %>% head(5)
```

Missing values are always sorted at the end:

```
df <- tibble(x = c(5, 2, NA))
arrange(df, x)
```

```
#> # A tibble: 3 x 1
#>       x
#>   <dbl>
#> 1     2
#> 2     5
#> 3    NA
```

Part Ⅱ:
select() & mutate()

select()

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

```
# Select columns by name
select(flights, year, month, day)
#> # A tibble: 336,776 x 3
#>    year month   day
#>   <int> <int> <int>
#> 1  2013     1     1
#> 2  2013     1     1
#> 3  2013     1     1
#> 4  2013     1     1
#> 5  2013     1     1
#> 6  2013     1     1
#> # … with 336,770 more rows
```

```
# Select all columns between year and day
(inclusive)
select(flights, year:day)
#> # A tibble: 336,776 x 3
#>    year month   day
#>   <int> <int> <int>
#> 1  2013     1     1
#> 2  2013     1     1
#> 3  2013     1     1
#> 4  2013     1     1
#> 5  2013     1     1
#> 6  2013     1     1
#> # … with 336,770 more rows
```

select()

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”.
- `matches("(.)\\1")`: selects variables that match a regular expression. This one matches any variables that contain repeated characters.
- `num_range("x", 1:3)`: matches x1, x2 and x3.

See `?select` for more details.

select()

`select()` can be used to rename variables, but it drops all of the variables not explicitly mentioned.

Instead, use `rename()`, that keeps all the variables that aren't explicitly mentioned:

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

select()

Another option is to use `select()` in conjunction with the `everything()` helper. This is useful if you have a handful of variables you'd like to move to the start of the data frame.

```
select(flights, time_hour, air_time, everything())
#> # A tibble: 336,776 x 19
#>   time_hour           air_time  year month   day dep_time sched_dep_time
#>   <dttm>                 <dbl> <int> <int> <int>    <int>          <int>
#> 1 2013-01-01 05:00:00      227  2013     1     1      517            515
#> 2 2013-01-01 05:00:00      227  2013     1     1      533            529
#> 3 2013-01-01 05:00:00      160  2013     1     1      542            540
```

(partial of results)

mutate()

`mutate()` can add new columns that are functions of existing columns.

```
flights_sml <- select(flights,
  year:day,
  ends_with("delay"),
  distance,
  air_time
)
mutate(flights_sml,
  gain = dep_delay - arr_delay,
  speed = distance / air_time * 60
)
```

```
#> # A tibble: 336,776 x 9
#>    year month   day dep_delay arr_delay distance
air_time  gain speed
#>   <int> <int> <int>     <dbl>     <dbl>    <dbl>
<dbl> <dbl> <dbl>
#> 1  2013     1     1         2        11     1400
227    -9  370.
#> 2  2013     1     1         4        20     1416
227   -16  374.
#> 3  2013     1     1         2        33     1089
160   -31  408.
#> 4  2013     1     1        -1       -18     1576
183    17  517.
#> 5  2013     1     1        -6       -25      762
116    19  394.
#> 6  2013     1     1        -4        12      719
150   -16  288.
#> # … with 336,770 more rows
```

mutate()

Note that you can refer to columns that you've just created:

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

If you only want to keep the new variables, use `transmute()`:

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

Creation functions

- Arithmetic operators: `+`, `-`, `*`, `/`, `^`.
- Modular arithmetic: `%/%` (integer division) and `%%` (remainder).

  In the flights dataset, you can compute `hour` and `minute` from `dep_time` with:

```
transmute(flights, dep_time, hour = dep_time %/% 100,
 minute = dep_time %% 100)
```

Creation functions

- Logs: `log()`, `log2()`, `log10()`.
- Offsets: `lead()` and `lag()` allow you to refer to leading or lagging values.
- Cumulative and rolling aggregates: R provides functions for running sums, products, mins and maxes: `cumsum()`, `cumprod()`, `cummin()`, `cummax()`; and dplyr provides `cummean()` for cumulative means.
- Logical comparisons, `<`, `<=`, `>`, `>=`, `!=`, and `==`.

Creation Functions

- Ranking: `min_rank()` gives smallest values the small ranks; use `desc(x)` to give the largest values the smallest ranks.

```
y <- c(1, 2, 2, NA, 3, 4)
min_rank(y)    #> [1]  1  2  2 NA  4  5
min_rank(desc(y))    #> [1]  5  3  3 NA  2  1
```

- If `min_rank()` doesn't do what you need, look at the variants `row_number()`, `dense_rank()`, `percent_rank()`, `cume_dist()`, `ntile()`. See their help pages for more details.

Part Ⅲ: Grouped summaries, mutates and filters

summarise()

`summarise()` 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
```

`summarise()` is always paired with `group_by()`.

This changes the unit of analysis from the complete dataset to individual groups. Then, when you use the dplyr verbs on a grouped data frame they'll be automatically applied "by group".

summarise()

For example, if we applied exactly the same code to a dataframe grouped by date, we get the average delay per date:

```
by_day <- group_by(flights, year, month, day)
summarise(by_day, delay = mean(dep_delay, na.rm = TRUE))
```

```
#> # A tibble: 365 x 4
#> # Groups:   year, month [12]
#>    year month   day delay
#>   <int> <int> <int> <dbl>
#> 1  2013     1     1 11.5
#> 2  2013     1     2 13.9
#> 3  2013     1     3 11.0
#> 4  2013     1     4  8.95
#> 5  2013     1     5  5.73
#> 6  2013     1     6  7.15
#> # … with 359 more rows
```

pipe%>%

Imagine that we want to explore the relationship between the distance and average delay for each location. Using what you know about dplyr, you might write code like this:

```
by_dest <- group_by(flights, dest)

delay <- summarise(by_dest, count = n(), dist = mean(distance, na.rm = TRUE), delay =
mean(arr_delay, na.rm = TRUE))

delay <- filter(delay, count > 20, dest != "HNL")

ggplot(data = delay, mapping = aes(x = dist, y = delay)) +

  geom_point(aes(size = count), alpha = 1/3) +

  geom_smooth(se = FALSE)
```

pipe%>%

This code is a little frustrating to write because we have to give each intermediate data frame a name, and this slows down our analysis.

There's another way to tackle the same problem with the pipe, `%>%`:

```
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.

Missing values

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))
```

If we don't set it, we'll get a lot of missing values!

Useful summary functions

- Measures of location: `mean(x)`, `median(x)`.
- Measures of spread: `sd(x)`, `IQR(x)`, `mad(x)`.
- Measures of rank: `min(x)`, `quantile(x, 0.25)`, `max(x)`. Quantiles are a generalisation of the median. For example, `quantile(x, 0.25)` will find a value of x that is greater than 25% of the values, and less than the remaining 75%.
- Measures of position: `first(x)`, `nth(x, 2)`, `last(x)`.
- Counts: `n()` returns the size of the current group. To count the number of non-missing values, use `sum(!is.na(x))`. To count the number of distinct (unique) values, use `n_distinct(x)`.

Useful summary functions

- Counts and proportions of logical values: `sum(x > 10)`, `mean(y == 0)`.

  When used with numeric functions, `TRUE` is converted to 1 and `FALSE` to 0. This makes `sum()` and `mean()` very useful: `sum(x)` gives the number of `TRUE`s in `x`, and `mean(x)` gives the proportion.

```
# How many flights left before 5am? (these usually indicate delayed flights from the previous day)

not_cancelled %>%

  group_by(year, month, day) %>%

  summarise(n_early = sum(dep_time < 500))
```

Grouping by multiple variables

When you group by multiple variables, each summary peels off one level of the grouping. That makes it easy to progressively roll up a dataset:

```
daily <- group_by(flights, year, month, day)
(per_day  <- summarise(daily, flights = n()))
#> # A tibble: 365 x 4
#> # Groups:   year, month [12]
#>    year month   day flights
#>   <int> <int> <int>   <int>
#> 1  2013     1     1     842
#> 2  2013     1     2     943
#> 3  2013     1     3     914
#> 4  2013     1     4     915
#> 5  2013     1     5     720
#> 6  2013     1     6     832
#> # … with 359 more rows
```

```
(per_month <- summarise(per_day, flights = sum(flights)))
#> # A tibble: 12 x 3
#> # Groups:   year [1]
#>    year month flights
#>   <int> <int>   <int>
#> 1  2013     1   27004
#> 2  2013     2   24951
#> 3  2013     3   28834
#> 4  2013     4   28330
#> 5  2013     5   28796
#> 6  2013     6   28243
#> # … with 6 more rows
```

```
(per_year  <- summarise(per_month, flights = sum(flights)))
#> # A tibble: 1 x 2
#>    year flights
#>   <int>   <int>
#> 1  2013  336776
```

Ungrouping

If you need to remove grouping, and return to operations on ungrouped data, use `ungroup()`.

```
daily %>%
  ungroup() %>%               # no longer grouped by date
  summarise(flights = n())   # all flights
#> # A tibble: 1 x 1
#>   flights
#>     <int>
#> 1  336776
```

Grouped mutates (and filters)

Grouping is most useful in conjunction with `summarise()`, but you can also do convenient operations with `mutate()` and `filter()`:

- Find the worst members of each group:

```
flights_sml %>%
  group_by(year, month, day) %>%
  filter(rank(desc(arr_delay)) < 10)
```

Grouped mutates (and filters)

- Find all groups bigger than a threshold:

```
popular_dests <- flights %>%
  group_by(dest) %>%
  filter(n() > 365)
```

Grouped mutates (and filters)

- Standardise to compute per group metrics:

```
popular_dests %>%
  filter(arr_delay > 0) %>%
  mutate(prop_delay = arr_delay / sum(arr_delay)) %>%
  select(year:day, dest, arr_delay, prop_delay)
```