Data Transformation

Matthew McDonald

Setup

Take careful note of the conflicts message that's printed when you load the tidyverse.

It tells you that dplyr overwrites some functions in base R.

If you want to use the base version of these functions after loading dplyr, you'll need to use their full names: `stats::filter()` and `stats::lag()`.

```
1 library(nycflights13)
2 library(tidyverse)
```

nycflights

This data frame contains all 336,776 flights that departed from New York City in 2013.

The data comes from the US Bureau of Transportation Statistics, and is documented in `?flights`.

```
1 flights
```

```
# A tibble: 336,776 × 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
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# ℹ 336,766 more rows
# ℹ 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>
```

Tibbles

Tibbles are a special type of data frame designed by the tidyverse team to avoid some common data.frame gotchas.

Data Types in the nycflights data are:

- `int` stands for integer.
- `dbl` stands for double, a vector of real numbers.
- `chr` stands for character, a vector of strings.
- `dttm` stands for date-time (a date + a time).

There are three other common types that aren't used here but you'll encounter later in the book:

- `lgl` stands for logical, a vector that contains only `TRUE` or `FALSE`.
- `fctr` stands for factor, which R uses to represent categorical variables with fixed possible values.
- `date` stands for date.

dplyr functions

All dplyr verbs work the same way:

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).
3. The result is a new data frame.

dplyr function groups

- Functions that operate on **rows**: `filter()` subsets rows based on the values of the columns and `arrange()` changes the order of the rows.
- Functions that operate on **columns**: `mutate()` creates new columns, `select()` columns, `rename()` changes their names, and `relocate()` changes their positions.
- Functions that operate on **groups**: `group_by()` divides data up into groups for analysis, and `summarise()` reduces each group to a single row.
- Functions that operate on **tables**, like the join functions and the set operations.

Row Operators

filter()

`filter()` allows you to choose rows based on their values.

The first argument is the name of the data frame.

The second and subsequent arguments are the expressions that filter the data frame.

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

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

```
# A tibble: 842 × 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
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# ℹ 832 more rows
# ℹ 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>
```

filter() Assignment

dplyr functions never modify their inputs, so if you want to save the result, you'll need to use the assignment operator, <–:

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

Comparison Operators

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).

It also provides `%in%`: `filter(df, x %in% c(a, b, c))` will return all rows where `x` is `a`, `b`, or `c`.

When you're starting out with R, the easiest mistake to make is to use `=` instead of `==` when testing for equality. `filter()` will let you know when this happens

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

```
Error in `filter()`:
! We detected a named input.
ℹ This usually means that you've used `=` instead of `==`.
ℹ Did you mean `month == 1`?
```

arrange()

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

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

```
# A tibble: 336,776 × 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
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# ℹ 336,766 more rows
# ℹ 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>
```

You can use `desc()` to re-order by a column in descending order:

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

Column Operators

`mutate()`, `select()`, `rename()`, and `relocate()` affect the columns (the variables) without changing the rows (the observations).

`mutate()` creates new variables that are functions of the existing variables; `select()`, `rename()`, and `relocate()` changes which variables are present, their names, and their positions.

mutate()

The job of `mutate()` is to add new columns that are functions of existing column.

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

```
# A tibble: 336,776 × 21
    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
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# ℹ 336,766 more rows
# ℹ 13 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>, gain <dbl>, speed <dbl>
```

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)

# Select all columns between year and day (inclusive)
select(flights, year:day)

# Select all columns except those from year to day (inclusive)
select(flights, -(year:day))
```

select() Helper Functions

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”.
- `num_range("x", 1:3)`: matches `x1`, `x2` and `x3`.

rename()

If you just want to keep all the existing variables and just want to rename a few, you can use `rename()` instead of `select()`:

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

relocate()

You can move variables around with `relocate`. By default it moves variables to the front:

```
relocate(flights, time_hour, air_time)
```

Like with `mutate()`, you can use the `.before` and `.after` arguments to choose where to place them:

```
relocate(flights, year:dep_time, .after = time_hour)
relocate(flights, starts_with("arr"), .before = dep_time)
```

Group Operators

The real power of dplyr comes when you add grouping into the mix.

The two key functions are `group_by()` and `summarise()`, but as you'll learn `group_by()` affects many other dplyr verbs in interesting ways.

group_by()

Use `group_by()` to divide your dataset into groups meaningful for your analysis

`group_by()` doesn't change the data but, if you look closely, you'll notice that it's now "grouped by" month.

The reason to group your data is because it changes the operation of subsequent verbs.

```
1 by_month <- group_by(flights, month)
2 by_month
```

```
# A tibble: 336,776 × 19
# Groups:   month [12]
    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
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# ℹ 336,766 more rows
# ℹ 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>
```

summarise()

The most important operation that you might apply to grouped data is a summary. It collapses each group to a single row.

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

```
# A tibble: 12 × 2
   month delay
   <int> <dbl>
 1     1 10.0
 2     2 10.8
 3     3 13.2
 4     4 13.9
 5     5 13.0
 6     6 20.8
 7     7 21.7
 8     8 12.6
 9     9  6.72
10    10  6.24
11    11  5.44
12    12 16.6
```

Counts Using summarise()

You can create any number of summaries in a single call to `summarise()`.

A very useful summary is `n()`, which returns the number of rows in each group:

```
1 summarise(by_month, delay = mean(dep_delay, na.rm = TRUE), n = n())
```

```
# A tibble: 12 × 3
   month delay     n
   <int> <dbl> <int>
 1     1 10.0  27004
 2     2 10.8  24951
 3     3 13.2  28834
 4     4 13.9  28330
 5     5 13.0  28796
 6     6 20.8  28243
 7     7 21.7  29425
 8     8 12.6  29327
 9     9  6.72 27574
10    10  6.24 28889
11    11  5.44 27268
12    12 16.6  28135
```

Combining Multiple Operations With The Pipe

```
flights %>%
  filter(!is.na(dep_delay)) %>%
  group_by(month) %>%
  summarise(delay = mean(dep_delay), n = n())
```

When you see `%>%` in code, a good way to "pronounce" it in your head is as "then".

That way you can read this code as a series of imperative statements: take the flights dataset, then filter it to remove rows with missing `dep_delay`, then group it by month, then summarise it with the average `dep_delay` and the number of observations.

Behind the scenes: - `x %>% f(y)` turns into `f(x, y)` - `x %>% f(y) %>% g(z)` turns into `g(f(x, y), z)` and so on.

You can use the pipe to rewrite multiple operations in a way that you can read left-to-right, top-to-bottom.