Data transformation {#transform}

Introduction

Visualisation is an important tool for insight generation, but it is rare that you get the data in exactly the right form you need. Often you'll need to create some new variables or summaries, or maybe you just want to rename the variables or reorder the observations in order to make the data a little easier to work with. You'll learn how to do all that (and more!) in this chapter, which will teach you how to transform your data using the dplyr package and a new dataset on flights departing New York City in 2013.

Prerequisites

In this chapter we're going to focus on how to use the dplyr package, another core member of the tidyverse. We'll illustrate the key ideas using data from the nycflights13 package, and use ggplot2 to help us understand the data.

```{r setup, message = FALSE}
library(nycflights13)
library(tidyverse)
```

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()`.

nycflights13

To explore the basic data manipulation verbs of dplyr, we'll use `nycflights13::flights`. This data frame contains all `r format(nrow(nycflights13::flights), big.mark = ",")` flights that departed from New York City in 2013. The data comes from the US [Bureau of Transportation Statistics] (http://www.transtats.bts.gov/DatabaseInfo.asp?DB_ID=120&Link=0), and is documented in `?flights`.

```{r}
flights
```

You might notice that this data frame prints a little differently from other data frames you might have used in the past: it only shows the first few rows and all the columns that fit on one screen. (To see the whole dataset, you can run `View(flights)` which will open the dataset in the RStudio viewer). It prints differently because it's a __tibble__. Tibbles are data frames, but slightly tweaked to work better in the tidyverse. For now, you don't need to worry about the differences; we'll come back to tibbles in more detail in [wrangle] (#wrangle-intro).

You might also have noticed the row of three (or four) letter abbreviations under the column names. These describe the type of each variable:

* `int` stands for integers.

* `dbl` stands for doubles, or real numbers.

* `chr` stands for character vectors, or strings.

* `dttm` stands for date-times (a date + a time).

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

* `lgl` stands for logical, vectors that contain only `TRUE` or `FALSE`.

* `fctr` stands for factors, which R uses to represent categorical variables
  with fixed possible values.

* `date` stands for dates.

dplyr basics

In this chapter you are going to learn the five key dplyr functions that allow you to solve the vast majority of your data manipulation challenges:

* Pick observations by their values (`filter()`).
* Reorder the rows (`arrange()`).
* Pick variables by their names (`select()`).
* Create new variables with functions of existing variables (`mutate()`).
* Collapse many values down to a single summary (`summarise()`).

These can all be used in conjunction with `group_by()` which changes the scope of each function from operating on the entire dataset to operating on it group-by-group. These six functions provide the verbs for a language of data manipulation.

All verbs work similarly:

1. The first argument is a data frame.

1. The subsequent arguments describe what to do with the data frame,
   using the variable names (without quotes).

1. The result is a new data frame.

Together these properties make it easy to chain together multiple simple steps to achieve a complex result. Let's dive in and see how these verbs work.

Filter rows with `filter()`

`filter()` allows you to subset observations 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:

```{r}
filter(flights, month == 1, day == 1)
```

When you run that line of code, dplyr executes the filtering operation and returns a new data frame. dplyr functions never modify their inputs, so if you want to save the result, you'll need to use the assignment operator, `<-`:

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

R either prints out the results, or saves them to a variable. If you want to do both, you can wrap the assignment in parentheses:

```{r}
(dec25 <- filter(flights, month == 12, day == 25))
```

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).

When you're starting out with R, the easiest mistake to make is to use `=` instead of `==` when testing for equality. When this happens you'll get an informative error:

```{r, error = TRUE}
filter(flights, month = 1)
```

There's another common problem you might encounter when using `==`: floating point numbers. These results might surprise you!

```{r}
sqrt(2) ^ 2 == 2
1/49 * 49 == 1
```

Computers use finite precision arithmetic (they obviously can't store an infinite number of digits!) so remember that every number you see is an approximation. Instead of relying on `==`, use `near()`:

```{r}
near(sqrt(2) ^ 2,  2)
near(1 / 49 * 49, 1)
```

Logical operators

Multiple arguments to `filter()` are combined with "and": every expression must be true in order for a row to be included in the output. For other types of combinations, you'll need to use Boolean operators yourself: `&` is "and", `|` is "or", and `!` is "not". Figure \@ref(fig:bool-ops) shows the complete set of Boolean operations.

```{r bool-ops, echo = FALSE, fig.cap = "Complete set of boolean operations. `x` is the left-hand circle, `y` is the right-hand circle, and the shaded region show which parts each operator selects."}
knitr::include_graphics("diagrams/transform-logical.png")
```

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

```{r, eval = FALSE}
filter(flights, month == 11 | month == 12)
```

The order of operations doesn't work like English. You can't write `filter(flights, month == 11 | 12)`, which you might literally translate into "finds all flights that departed in November or December". Instead it finds all months that equal `11 | 12`, an expression that evaluates to `TRUE`. In a numeric context (like here), `TRUE` becomes one, so this finds all flights in January, not November or December. This is quite confusing!

A useful short-hand for this problem 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:

```{r, eval = FALSE}
nov_dec <- filter(flights, month %in% c(11, 12))
```

Sometimes you can simplify complicated subsetting by remembering De Morgan's law: `!(x & y)` is the same as `!x | !y`, and `!(x | y)` is the same as `!x & !y`. For example, 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:

```{r, eval = FALSE}
filter(flights, !(arr_delay > 120 | dep_delay > 120))
filter(flights, arr_delay <= 120, dep_delay <= 120)
```

As well as `&` and `|`, R also has `&&` and `||`. Don't use them here! You'll learn when you should use them in [conditional execution].

Whenever you start using complicated, multipart expressions in `filter()`, consider making them explicit variables instead. That makes it much easier to check your work. You'll learn how to create new variables shortly.

Missing values

One important feature of R that can make comparison tricky are missing values, or `NA`s ("not availables"). `NA` represents an unknown value so missing values are "contagious": almost any operation involving an unknown value will also be unknown.

```{r}
NA > 5
10 == NA
NA + 10
NA / 2
```

The most confusing result is this one:

```{r}
NA == NA
```

It's easiest to understand why this is true with a bit more context:

```{r}
# Let x be Mary's age. We don't know how old she is.
x <- NA

# Let y be John's age. We don't know how old he is.
y <- NA

# Are John and Mary the same age?
x == y
# We don't know!
```

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

```{r}
is.na(x)
```

`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:

```{r}
df <- tibble(x = c(1, NA, 3))
filter(df, x > 1)
filter(df, is.na(x) | x > 1)
```

Exercises

1. Find all flights that

    1. Had an arrival delay of two or more hours
    1. Flew to Houston (`IAH` or `HOU`)
    1. Were operated by United, American, or Delta
    1. Departed in summer (July, August, and September)
    1. Arrived more than two hours late, but didn't leave late
    1. Were delayed by at least an hour, but made up over 30 minutes in flight
    1. Departed between midnight and 6am (inclusive)

1. Another useful dplyr filtering helper is `between()`. What does it do?
    Can you use it to simplify the code needed to answer the previous challenges?

1. How many flights have a missing `dep_time`? What other variables are
    missing? What might these rows represent?

1. Why is `NA ^ 0` not missing? Why is `NA | TRUE` not missing?
    Why is `FALSE & NA` not missing? Can you figure out the general
    rule? (`NA * 0` is a tricky counterexample!)

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 (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:

```{r}
arrange(flights, year, month, day)
```

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

```{r}
arrange(flights, desc(arr_delay))
```

Missing values are always sorted at the end:

```{r}
df <- tibble(x = c(5, 2, NA))
arrange(df, x)
arrange(df, desc(x))
```

Exercises

1. How could you use `arrange()` to sort all missing values to the

```
start?
    (Hint: use `is.na()`).

1.  Sort `flights` to find the most delayed flights. Find the flights
that
    left earliest.

1.  Sort `flights` to find the fastest flights.

1.  Which flights travelled the longest? Which travelled the shortest?

## Select columns with `select()` {#select}

It's not uncommon to get datasets with hundreds or even thousands of
variables. In this case, the first challenge is often narrowing in on
the variables you're actually interested in. `select()` allows you to
rapidly zoom in on a useful subset using operations based on the names
of the variables.

`select()` is not terribly useful with the flights data because we only
have 19 variables, but you can still get the general idea:

```{r}
# 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))
```

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.
You'll
   learn more about regular expressions in [strings].

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

See `?select` for more details.

`select()` can be used to rename variables, but it's rarely useful
because it drops all of the variables not explicitly mentioned.
Instead, use `rename()`, which is a variant of `select()` that keeps
all the variables that aren't explicitly mentioned:

```{r}
rename(flights, tail_num = tailnum)
```

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.

```{r}
select(flights, time_hour, air_time, everything())
```

Exercises

1. Brainstorm as many ways as possible to select `dep_time`,
`dep_delay`,
    `arr_time`, and `arr_delay` from `flights`.

1. What happens if you include the name of a variable multiple times
in
    a `select()` call?

1. What does the `one_of()` function do? Why might it be helpful in
conjunction
    with this vector?

    ```{r}
    vars <- c("year", "month", "day", "dep_delay", "arr_delay")
    ```

1. Does the result of running the following code surprise you? How do
the
    select helpers deal with case by default? How can you change that
default?

    ```{r, eval = FALSE}
    select(flights, contains("TIME"))
    ```

Add new variables with `mutate()`

Besides selecting sets of existing columns, it's often useful to add
new columns that are functions of existing columns. That's the job of
`mutate()`.

`mutate()` always adds new columns at the end of your dataset so we'll
start by creating a narrower dataset so we can see the new variables.
Remember that when you're in RStudio, the easiest way to see all the
columns is `View()`.

```{r}
flights_sml <- select(flights,
  year:day,
  ends_with("delay"),
  distance,
  air_time
```
```

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

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

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

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

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

Useful creation functions {#mutate-funs}

There are many functions for creating new variables that you can use with `mutate()`. The key property is that the function must be vectorised: it must take a vector of values as input, return a vector with the same number of values as output. There's no way to list every possible function that you might use, but here's a selection of functions that are frequently useful:

*   Arithmetic operators: `+`, `-`, `*`, `/`, `^`. These are all vectorised,
    using the so called "recycling rules". If one parameter is shorter than
    the other, it will be automatically extended to be the same length. This
    is most useful when one of the arguments is a single number: `air_time / 60`,
    `hours * 60 + minute`, etc.

    Arithmetic operators are also useful in conjunction with the aggregate
    functions you'll learn about later. For example, `x / sum(x)` calculates
    the proportion of a total, and `y - mean(y)` computes the difference from
    the mean.

*   Modular arithmetic: `%/%` (integer division) and `%%` (remainder), where
    `x == y * (x %/% y) + (x %% y)`. Modular arithmetic is a handy tool because
    it allows you to break integers up into pieces. For example, in the
    flights dataset, you can compute `hour` and `minute` from `dep_time` with:

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

*   Logs: `log()`, `log2()`, `log10()`. Logarithms are an incredibly useful
    transformation for dealing with data that ranges across multiple orders of
    magnitude. They also convert multiplicative relationships to additive, a
    feature we'll come back to in modelling.

    All else being equal, I recommend using `log2()` because it's easy to
    interpret: a difference of 1 on the log scale corresponds to doubling on
    the original scale and a difference of -1 corresponds to halving.

*   Offsets: `lead()` and `lag()` allow you to refer to leading or lagging
    values. This allows you to compute running differences (e.g. `x - lag(x)`)
    or find when values change (`x != lag(x)`). They are most useful in
    conjunction with `group_by()`, which you'll learn about shortly.

    ```{r}
    (x <- 1:10)
    lag(x)
    lead(x)
    ```

*   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. If you need rolling
    aggregates (i.e. a sum computed over a rolling window), try the RcppRoll
    package.

    ```{r}
    x
    cumsum(x)
```

    cummean(x)
    ```

*   Logical comparisons, `<`, `<=`, `>`, `>=`, `!=`, which you learned
about
    earlier. If you're doing a complex sequence of logical operations
it's
    often a good idea to store the interim values in new variables so
you can
    check that each step is working as expected.

*   Ranking: there are a number of ranking functions, but you should
    start with `min_rank()`. It does the most usual type of ranking
    (e.g. 1st, 2nd, 2nd, 4th). The default gives smallest values the
small
    ranks; use `desc(x)` to give the largest values the smallest ranks.

    ```{r}
    y <- c(1, 2, 2, NA, 3, 4)
    min_rank(y)
    min_rank(desc(y))
    ```

    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.

    ```{r}
    row_number(y)
    dense_rank(y)
    percent_rank(y)
    cume_dist(y)
    ```

Exercises

```{r, eval = FALSE, echo = FALSE}
flights <- flights %>% mutate(
  dep_time = hour * 60 + minute,
  arr_time = (arr_time %/% 100) * 60 + (arr_time %% 100),
  airtime2 = arr_time - dep_time,
  dep_sched = dep_time + dep_delay
)

ggplot(flights, aes(dep_sched)) + geom_histogram(binwidth = 60)
ggplot(flights, aes(dep_sched %% 60)) + geom_histogram(binwidth = 1)
ggplot(flights, aes(air_time - airtime2)) + geom_histogram()
```

1. Currently `dep_time` and `sched_dep_time` are convenient to look
at, but
    hard to compute with because they're not really continuous numbers.
    Convert them to a more convenient representation of number of
minutes
    since midnight.

1. Compare `air_time` with `arr_time - dep_time`. What do you expect
to see?
    What do you see? What do you need to do to fix it?

1. Compare `dep_time`, `sched_dep_time`, and `dep_delay`. How would
you
    expect those three numbers to be related?

1. Find the 10 most delayed flights using a ranking function. How do
you want
    to handle ties? Carefully read the documentation for `min_rank()`.

1. What does `1:3 + 1:10` return? Why?

1. What trigonometric functions does R provide?

Grouped summaries with `summarise()`

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

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

(We'll come back to what that `na.rm = TRUE` means very shortly.)

`summarise()` is not terribly useful unless we pair it 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". For
example, if we applied exactly the same code to a data frame grouped by
date, we get the average delay per date:

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

Together `group_by()` and `summarise()` provide one of the tools that
you'll use most commonly when working with dplyr: grouped summaries.
But before we go any further with this, we need to introduce a powerful
new idea: the pipe.

Combining multiple operations with the 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:

```{r, fig.width = 6}
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")

# It looks like delays increase with distance up to ~750 miles
# and then decrease. Maybe as flights get longer there's more
# ability to make up delays in the air?
ggplot(data = delay, mapping = aes(x = dist, y = delay)) +
  geom_point(aes(size = count), alpha = 1/3) +
  geom_smooth(se = FALSE)
```

There are three steps to prepare this data:

1. Group flights by destination.

1. Summarise to compute distance, average delay, and number of
flights.

1. Filter to remove noisy points and Honolulu airport, which is almost
    twice as far away as the next closest airport.

This code is a little frustrating to write because we have to give each intermediate data frame a name, even though we don't care about it. Naming things is hard, so this slows down our analysis.

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

```{r}
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")
```

This focuses on the transformations, not what's being transformed, which makes the code easier to read. 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".

Behind the scenes, `x %>% f(y)` turns into `f(x, y)`, and `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. We'll use piping frequently from now on because it considerably improves the readability of code, and we'll come back to it in more detail in [pipes].

Working with the pipe is one of the key criteria for belonging to the tidyverse. The only exception is ggplot2: it was written before the pipe was discovered. Unfortunately, the next iteration of ggplot2, ggvis, which does use the pipe, isn't quite ready for prime time yet.

Missing values

You may have wondered about the `na.rm` argument we used above. What happens if we don't set it?

```{r}
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
```

We get a lot of missing values! That's because aggregation functions obey the usual rule of missing values: if there's 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:

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

In this case, where missing values represent cancelled flights, we could also tackle the problem by first removing the cancelled flights. We'll save this dataset so we can reuse in the next few examples.

```{r}
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))

not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
```

Counts

Whenever you do any aggregation, it's always a good idea to include either a count (`n()`), or a count of non-missing values (`sum(!is.na(x))`). That way you can check that you're not drawing conclusions based on very small amounts of data. For example, let's look at the planes (identified by their tail number) that have the highest average delays:

```{r}
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarise(
    delay = mean(arr_delay)
  )
```

```
ggplot(data = delays, mapping = aes(x = delay)) +
  geom_freqpoly(binwidth = 10)
```

Wow, there are some planes that have an _average_ delay of 5 hours (300 minutes)!

The story is actually a little more nuanced. We can get more insight if we draw a scatterplot of number of flights vs. average delay:

```{r}
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarise(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )

ggplot(data = delays, mapping = aes(x = n, y = delay)) +
  geom_point(alpha = 1/10)
```

Not surprisingly, there is much greater variation in the average delay when there are few flights. The shape of this plot is very characteristic: whenever you plot a mean (or other summary) vs. group size, you'll see that the variation decreases as the sample size increases.

When looking at this sort of plot, it's often useful to filter out the groups with the smallest numbers of observations, so you can see more of the pattern and less of the extreme variation in the smallest groups. This is what the following code does, as well as showing you a handy pattern for integrating ggplot2 into dplyr flows. It's a bit painful that you have to switch from `%>%` to `+`, but once you get the hang of it, it's quite convenient.

```{r}
delays %>%
  filter(n > 25) %>%
  ggplot(mapping = aes(x = n, y = delay)) +
    geom_point(alpha = 1/10)
```

RStudio tip: a useful keyboard shortcut is Cmd/Ctrl + Shift + P. This resends the previously sent chunk from the editor to the console. This is very convenient when you're (e.g.) exploring the value of `n` in the example above. You send the whole block once with Cmd/Ctrl + Enter, then you modify the value of `n` and press Cmd/Ctrl + Shift + P to resend the complete block.

There's another common variation of this type of pattern. Let's look at how the average performance of batters in baseball is related to the number of times they're at bat. Here I use data from the __Lahman__ package to compute the batting average (number of hits / number of attempts) of every major league baseball player.

When I plot the skill of the batter (measured by the batting average, `ba`) against the number of opportunities to hit the ball (measured by at bat, `ab`), you see two patterns:

1. As above, the variation in our aggregate decreases as we get more
    data points.

2. There's a positive correlation between skill (`ba`) and opportunities to
    hit the ball (`ab`). This is because teams control who gets to play,
    and obviously they'll pick their best players.

```{r}
# Convert to a tibble so it prints nicely
batting <- as_tibble(Lahman::Batting)

batters <- batting %>%
  group_by(playerID) %>%
  summarise(
    ba = sum(H, na.rm = TRUE) / sum(AB, na.rm = TRUE),
    ab = sum(AB, na.rm = TRUE)
  )

batters %>%
  filter(ab > 100) %>%
  ggplot(mapping = aes(x = ab, y = ba)) +
    geom_point() +
    geom_smooth(se = FALSE)
```

This also has important implications for ranking. If you naively sort on `desc(ba)`, the people with the best batting averages are clearly lucky, not skilled:

```{r}
batters %>%
  arrange(desc(ba))
```

You can find a good explanation of this problem at <http://varianceexplained.org/r/empirical_bayes_baseball/> and <http://www.evanmiller.org/how-not-to-sort-by-average-rating.html>.

Useful summary functions {#summarise-funs}

Just using means, counts, and sum can get you a long way, but R provides many other useful summary functions:

```
*   Measures of location: we've used `mean(x)`, but `median(x)` is also
    useful. The mean is the sum divided by the length; the median is a
value
    where 50% of `x` is above it, and 50% is below it.

    It's sometimes useful to combine aggregation with logical
subsetting.
    We haven't talked about this sort of subsetting yet, but you'll
learn more
    about it in [subsetting].

    ```{r}
    not_cancelled %>%
      group_by(year, month, day) %>%
      summarise(
        avg_delay1 = mean(arr_delay),
        avg_delay2 = mean(arr_delay[arr_delay > 0]) # the average
positive delay
      )
    ```

*   Measures of spread: `sd(x)`, `IQR(x)`, `mad(x)`. The mean squared
deviation,
    or standard deviation or sd for short, is the standard measure of
spread.
    The interquartile range `IQR()` and median absolute deviation
`mad(x)`
    are robust equivalents that may be more useful if you have
outliers.

    ```{r}
    # Why is distance to some destinations more variable than to
others?
    not_cancelled %>%
      group_by(dest) %>%
      summarise(distance_sd = sd(distance)) %>%
      arrange(desc(distance_sd))
    ```

*   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%.

    ```{r}
    # When do the first and last flights leave each day?
    not_cancelled %>%
      group_by(year, month, day) %>%
      summarise(
        first = min(dep_time),
        last = max(dep_time)
      )
    ```

*   Measures of position: `first(x)`, `nth(x, 2)`, `last(x)`. These
work
    similarly to `x[1]`, `x[2]`, and `x[length(x)]` but let you set a
default
    value if that position does not exist (i.e. you're trying to get
the 3rd
    element from a group that only has two elements). For example, we
can
    find the first and last departure for each day:

    ```{r}
    not_cancelled %>%
      group_by(year, month, day) %>%
      summarise(
        first_dep = first(dep_time),
        last_dep = last(dep_time)
      )
    ```

    These functions are complementary to filtering on ranks. Filtering
gives
    you all variables, with each observation in a separate row:

    ```{r}
    not_cancelled %>%
      group_by(year, month, day) %>%
      mutate(r = min_rank(desc(dep_time))) %>%
      filter(r %in% range(r))
    ```

*   Counts: You've seen `n()`, which takes no arguments, and 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)`.

    ```{r}
    # Which destinations have the most carriers?
    not_cancelled %>%
      group_by(dest) %>%
      summarise(carriers = n_distinct(carrier)) %>%
      arrange(desc(carriers))
    ```

    Counts are so useful that dplyr provides a simple helper if all you
want is
    a count:

    ```{r}
    not_cancelled %>%
      count(dest)
```
```

```
    You can optionally provide a weight variable. For example, you could use
    this to "count" (sum) the total number of miles a plane flew:

    ```{r}
    not_cancelled %>%
      count(tailnum, wt = distance)
    ```

*   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.

    ```{r}
    # 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))

    # What proportion of flights are delayed by more than an hour?
    not_cancelled %>%
      group_by(year, month, day) %>%
      summarise(hour_perc = mean(arr_delay > 60))
    ```

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:

```{r}
daily <- group_by(flights, year, month, day)
(per_day   <- summarise(daily, flights = n()))
(per_month <- summarise(per_day, flights = sum(flights)))
(per_year  <- summarise(per_month, flights = sum(flights)))
```

Be careful when progressively rolling up summaries: it's OK for sums
and counts, but you need to think about weighting means and variances,
and it's not possible to do it exactly for rank-based statistics like
the median. In other words, the sum of groupwise sums is the overall
sum, but the median of groupwise medians is not the overall median.

Ungrouping

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

```{r}
daily %>%
  ungroup() %>%             # no longer grouped by date
  summarise(flights = n())  # all flights
```

Exercises

1. Brainstorm at least 5 different ways to assess the typical delay
    characteristics of a group of flights. Consider the following
scenarios:

    * A flight is 15 minutes early 50% of the time, and 15 minutes late
50% of
      the time.

    * A flight is always 10 minutes late.

    * A flight is 30 minutes early 50% of the time, and 30 minutes late
50% of
      the time.

    * 99% of the time a flight is on time. 1% of the time it's 2 hours
late.

    Which is more important: arrival delay or departure delay?

1. Come up with another approach that will give you the same output as
    `not_cancelled %>% count(dest)` and
    `not_cancelled %>% count(tailnum, wt = distance)` (without using
    `count()`).

1. Our definition of cancelled flights (`is.na(dep_delay) |
is.na(arr_delay)`
) is slightly suboptimal. Why? Which is the most important column?

1. Look at the number of cancelled flights per day. Is there a
pattern?
    Is the proportion of cancelled flights related to the average
delay?

1. Which carrier has the worst delays? Challenge: can you disentangle
the
    effects of bad airports vs. bad carriers? Why/why not? (Hint: think
about
    `flights %>% group_by(carrier, dest) %>% summarise(n())`)

1. What does the `sort` argument to `count()` do. When might you use
it?

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:

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

*   Find all groups bigger than a threshold:

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

*   Standardise to compute per group metrics:

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

A grouped filter is a grouped mutate followed by an ungrouped filter. I generally avoid them except for quick and dirty manipulations: otherwise it's hard to check that you've done the manipulation correctly.

Functions that work most naturally in grouped mutates and filters are known as window functions (vs. the summary functions used for summaries). You can learn more about useful window functions in the corresponding vignette: `vignette("window-functions")`.

Exercises

1. Refer back to the lists of useful mutate and filtering functions.
    Describe how each operation changes when you combine it with grouping.

1. Which plane (`tailnum`) has the worst on-time record?

1. What time of day should you fly if you want to avoid delays as much
    as possible?

1. For each destination, compute the total minutes of delay. For each,
    flight, compute the proportion of the total delay for its destination.

1. Delays are typically temporally correlated: even once the problem that
    caused the initial delay has been resolved, later flights are delayed
    to allow earlier flights to leave. Using `lag()` explore how the delay
    of a flight is related to the delay of the immediately preceding flight.

1. Look at each destination. Can you find flights that are suspiciously
    fast? (i.e. flights that represent a potential data entry error). Compute
    the air time a flight relative to the shortest flight to that destination.
    Which flights were most delayed in the air?

1. Find all destinations that are flown by at least two carriers. Use that
    information to rank the carriers.

1. For each plane, count the number of flights before the first delay
    of greater than 1 hour.