Introduction to Data Science
Transforming and Cleaning Data

Visualization is one of the most powerful tools we have for understanding data. Sadly, however, we rarely get the data in the form we need. So one of the core skills for anyone doing data science is reshaping the data to get it into a useable format. The `dplyr` package has a core set of functions to help you work with data. Honestly, this is the set of commands I use the most when working with R. These 5 key functions are really the workhorses of day-to-day data analysis.

In this lesson, we'll learn the five key functions of this library, all of them very useful for data science: `filter()`, `arrange()`, `mutate()`, `select()`, and `summarize()`.

To learn about these functions, we'll explore the `flights` dataset, which contains data on all 336, 776 domestic flights that departed from New York City in 2013. Just as a reminder, New York City has 3 main airports: Newark (EWR), John F. Kennedy (JFK), and La Guardia (LGA). The dataset contains the origin and destination of each flight, the date and time of the flight, the carrier of each flight, and how delayed the flight was. We'll learn a bit about airlines and flight times today!

To access this data, we need to load the `nycflights13` library. As always, our first step is to load the libraries:

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

You should see both libraries load into R. If you instead see:

```
Error in library(tidyverse) : there is no package called
'tidyverse'
```

(or the equivalent error for nycflights13), it means that you haven't installed the library on your machine. So you need to run:

```
install.packages("tidyverse")
library(tidyverse)
```

Which will install and then load it onto your machine. If you need to install nycflights13, you'd substitute that package for the tidyverse in the code above. Remember, we only need to install the library once, but we need to load it with the `library()` command each time we use it.

Let's begin by looking at the data:

```
>flights
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
```

```
 1  2013     1     1      517            515          2      830
 2  2013     1     1      533            529          4      850
 3  2013     1     1      542            540          2      923
 4  2013     1     1      544            545         -1     1004
 5  2013     1     1      554            600         -6      812
 6  2013     1     1      554            558         -4      740
 7  2013     1     1      555            600         -5      913
 8  2013     1     1      557            600         -3      709
 9  2013     1     1      557            600         -3      838
10  2013     1     1      558            600         -2      753
# ... with 336,766 more rows, and 12 more variables:
#   sched_arr_time <int>, 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>
```

Note that unlike the mpg dataset from last week, we cannot display all of the data at once because there are too Viemany rows (observations) and columns (variables) to fit onto one screen.

If you want to see all of the rows/columns in your dataset, you have to run the command:

```
View(Flights)
```

Which will open the dataset in the data browser in RStudio. In general, to see all of your data in the data browser, run `View(DATASET)`, where DATASET is the name of your dataset.

Notice that there are 4 types of variables in this dataset:

- <Int>: integers, whole numbers like 0, 1, 2, 3, …
- <Dbl>: doubles, which are real numbers: 2.1232, 3.2456, 5, etc.
- <Chr>: character vectors, which are text (string) variables. So, for example, the airline code is a text string (e.g., UA for United Airlines).
- <Dttm>: dates and time variables

There are a few other data types (such as logical and factor variables), but we'll cover those in future lectures. Throughout the term, we'll see various functions that help you work with these different types of variables.

The key functions in the dplyr library, which a key part of the Tidyverse, are:

- `Filter()`: select observations by their values
- `Arrange()`: reorder the rows
- `Select()`: pick variables by their name
- `Mutate()`: make new variables as functions of other variables
- `Summarise()`: collapse variables down to a single summary

Let's see how to use each of these functions, and how they help us to work with our data.

Filter: Select Observations by their Values

`Filter()` does just what its name implies: it lets you select (filter) certain observations based on their values. This is a very handy command, because we'll often want to work with a sub-set of our data.

For example, in our dataset, suppose I wanted to find all flights that left on December 1st. To do that, we would run:

```
>filter(flights,month==12,day==1)
# A tibble: 987 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013    12     1       13           2359        14      446
 2  2013    12     1       17           2359        18      443
 3  2013    12     1      453            500        -7      636
 4  2013    12     1      520            515         5      749
 5  2013    12     1      536            540        -4      845
 6  2013    12     1      540            550       -10     1005
 7  2013    12     1      541            545        -4      734
 8  2013    12     1      546            545         1      826
 9  2013    12     1      549            600       -11      648
10  2013    12     1      550            600       -10      825
# ... with 977 more rows, and 12 more variables: sched_arr_time
<int>,
#   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>
```

But note that R just returns this as a data set to our window. If I wanted to save this output for future analyses, I could assign it to an object in R. I do that by telling R:

```
dec1 <- filter(flights,month==12,day==1)
```

Let's break down this code. `<-` is the assignment operator, which is one of the core operators in R (remember that we first introduced this in our lecture on setting up R and RStudio). The code above says take all of the flights on December 1st and assign them to the object `dec1`. It is an object in R: note that on the right-hand side of RStudio, you now have an entry called dec1 under the "data" tab in the Environment window. This is R's way of saying that you can now refer to `dec1` in R code and perform operations on it! We'll see how to do this below.

In the example above, I referred to this new object as `dec1`, which is short for December 1st. You can name objects in R anything you would like, but it's best to name them something sensible that will be easy to remember.

When you use filter, you need to think about the observations you want to select. To do that, remember the comparison operators: >, >=, <, <=, !=, and ==. The first four should be familiar (greater than, greater than or equal to, less than, less than or equal to); the last two are not equal to (! is the not operator), and == (is equal to).

Note that in R, if you want to test for equality (i.e., is X equal to Y?), then you **must** use two equal signs (==), <u>not</u> one equal sign (=).

<u>*Aside: Double Equal Signs vs. Single Equals Signs*</u>

You might be wondering why we use ==, and not just =, when testing for equality. This section explains why. I've put it in italics since it's not vital information, but it is useful. Think of this as a bit of "bonus" content. If you're interested great. If not, skip to the section below.

A common mistake is to use =, rather than == (single vs. a double equal sign). What's the difference?

A double equal sign, ==, is used to test for equality. So in our code above, saying `month == 12` tells R: check each observation of the data, and if month is equal to 12 (the flight left in December), then include it in the resulting dataset, otherwise do not. The `day == 1` does the same, except for flights that left on the first of any month. The combination together then gets us the flights that left on December 1st. One small point to note is that this is exact equality. What does that mean? It's best illustrated with an example:

```
> x1 <- 0.5 - 0.4
> x2 <- 0.4 - 0.3
> x1 == x2
[1] FALSE
```

Wait, how is that false—that implies x1 *is not equal to* x2*? The answer is that when R (like all computer programs) does floating point operations, there is a tiny bit of accuracy lost, so* x1 *here is not exactly equal to* x2*. Remember, computers have a finite level of precision, since they can't store an infinite number of digits, and hence very minute differences creep into these sorts of operations.*

To see if two numbers are equal to each other, it is better to use `all.equal(number1, number2)`. If we do that here, `all.equal(x1,x2)`, we get TRUE (which is what we'd want). All.equal() is a function that accounts for this tiny bit of inaccuracy in how R stores numbers. == is better for the method I used above (in a function comparing a variable to a value).

Well then what does = do? It is a shorthand for assignment (it also gets used in some functions, as we'll see below). Typically, = is a shorthand for <-. So you could write: `jan1 = filter(flights,month==1,day==1)` *and that would be fine.*

Now back to our regular lecture…

We can make filter more powerful if we combine with "Or" and "And" operators. In R, `|` denotes "or" and `&` denotes and. So, for example, suppose we wanted to find all flights that left in November or December. We would then write:

```
> filter(flights, month== 11 | month == 12)
# A tibble: 55,403 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013    11     1        5           2359         6      352
 2  2013    11     1       35           2250       105      123
 3  2013    11     1      455            500        -5      641
 4  2013    11     1      539            545        -6      856
 5  2013    11     1      542            545        -3      831
 6  2013    11     1      549            600       -11      912
 7  2013    11     1      550            600       -10      705
 8  2013    11     1      554            600        -6      659
 9  2013    11     1      554            600        -6      826
10  2013    11     1      554            600        -6      749
# ... with 55,393 more rows, and 12 more variables:
#   sched_arr_time <int>, 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>
```

Note that we have to be careful with our code: we can't say `filter(flights, month == 11 |de 12).` We must tell R that both arguments (11 and 12) pass to month.

We could also find all flights in January that left before 6 AM:

```
filter(flights, month == 1 & dep_time < 600)
# A tibble: 651 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515         2      830
 2  2013     1     1      533            529         4      850
 3  2013     1     1      542            540         2      923
 4  2013     1     1      544            545        -1     1004
 5  2013     1     1      554            600        -6      812
 6  2013     1     1      554            558        -4      740
 7  2013     1     1      555            600        -5      913
```

```
 8  2013     1     1      557             600         -3      709
 9  2013     1     1      557             600         -3      838
10  2013     1     1      558             600         -2      753
# ... with 641 more rows, and 12 more variables:
#   sched_arr_time <int>, 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>
```

TEST YOURSELF: How would you find every flight delayed by more than 30 minutes? How would you find every flight operated by United Airlines? Here, the airline code for United Airlines is UA.

ANSWER:

```
filter(flights,dep_delay>30)
filter(flights,carrier=="UA")
```

Note, however, that you can't see carrier in the output on our screen, unfortunately. To see that, you can use `View(Flights)` (which will show you the entire dataset in a new window) or you can use select, as we explain below.

Note that because the airline carrier is a character (text) variable, I need to put its value in quotation makrs. So `filter(flights,carrier==UA)` will produce an error (try it!), because it is looking for an object UA (which doesn't exist). Instead, by putting it in quotation marks tells R that I want to find text values of carrier equal to UA.

By the way, if you're not familiar with various airline codes, you can see them by referencing the `airlines` dataset, which is part of the `nycflights13` library. If you display this dataset, you'll see that it brings up a list of all of the airline codes referenced in our main `flights` data. Most are intuitive, but a few are not.

Arrange: Reordering Your Dataset

`Arrange()` is similarly useful. Instead of selecting particular rows (as filter does), it reorders them for you. So, for example:

```
arrange(flights, year, month, day)
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515         2      830
 2  2013     1     1      533            529         4      850
 3  2013     1     1      542            540         2      923
 4  2013     1     1      544            545        -1     1004
 5  2013     1     1      554            600        -6      812
 6  2013     1     1      554            558        -4      740
```

```
 7  2013     1     1      555              600        -5      913
 8  2013     1     1      557              600        -3      709
 9  2013     1     1      557              600        -3      838
10  2013     1     1      558              600        -2      753
# ... with 336,766 more rows, and 12 more variables:
#   sched_arr_time <int>, 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>
```

So this sorts all of the flights by year, then month, then day. Since all of the flights left in 2013, we didn't really need to sort by that variable (we just did it for completeness). It then sorts the flights by month, and within each month, by date. So if you run `View(Flights)` to bring up the data browser and look at the data, you'll see that it lists the flights for January 1st, then those for January 2nd, and so forth.

Using `desc()` sorts things in descending order. If I wanted to find the most delayed flight, I would write:

```
> arrange(flights, desc(dep_delay))
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     9      641            900      1301     1242
 2  2013     6    15     1432           1935      1137     1607
 3  2013     1    10     1121           1635      1126     1239
 4  2013     9    20     1139           1845      1014     1457
 5  2013     7    22      845           1600      1005     1044
 6  2013     4    10     1100           1900       960     1342
 7  2013     3    17     2321            810       911      135
 8  2013     6    27      959           1900       899     1236
 9  2013     7    22     2257            759       898      121
10  2013    12     5      756           1700       896     1058
# ... with 336,766 more rows, and 12 more variables:
#   sched_arr_time <int>, 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>
```

So here we see that the most delayed flight was 1301 minutes late, or over 21 hours! Weep a moment for the poor souls of Hawaiian Airlines #51 from JFK to HNL on January 9th, who waited nearly a full day to leave. Note that there were 5 flights delayed by more than 1,000 minutes, or 16.67 hours.

TEST YOURSELF: How would you find the flight that was in the air the longest (using the `air_time` variable)? How would you find the flight that is in the air for the shortest amount of time?

ANSWER:

```
arrange(flights,desc(air_time))
arrange(flights,air_time)
```

Note that if you use the data viewer, you see that the shortest flight is only 20 minutes to Bradley Airport in Windsor Locks, CT, and the longest flight is just under 12 hours from Newark to Honolulu (this data must only include domestic flights).

Select: Choose Subsets of Variables

`Select()` is the parallel to filter. Filter gives us particular observations (rows), select gives us particular variables (columns). While this dataset only has a small number of variables, many datasets have hundreds, if not thousands, of variables. With that sort of data, it can make sense to focus in on the handful that are relevant to your particular analysis.

For example, suppose you wanted to just look at the days and times of the flights data. To get that, we would write:

```
select(flights,month,day)
# A tibble: 336,776 x 2
   month   day
   <int> <int>
 1     1     1
 2     1     1
 3     1     1
 4     1     1
 5     1     1
 6     1     1
 7     1     1
 8     1     1
 9     1     1
10     1     1
# ... with 336,766 more rows
```

You can also select everything but a particular variable. To do that, you use select(), except you say "-variable" where the minus sign (-) means you drop that variable. For example, to get everything but the departure time, you would tell R:

```
select(flights,-dep_time)
# A tibble: 336,776 x 18
    year month   day sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1            515         2      830            819
 2  2013     1     1            529         4      850            830
 3  2013     1     1            540         2      923            850
 4  2013     1     1            545        -1     1004           1022
 5  2013     1     1            600        -6      812            837
```

```
 6  2013     1     1             558        -4      740             728
 7  2013     1     1             600        -5      913             854
 8  2013     1     1             600        -3      709             723
 9  2013     1     1             600        -3      838             846
10  2013     1     1             600        -2      753             745
# ... with 336,766 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>
```

Often, you'll want to select a subset of similar variables, and there are a few commands to help with this:

- `starts_with("xyz")` : Select all variables that begin with xyz
- `ends_with("xyz")` : Selects all variables that end with xyz
- `contains("xyz")` : Selects all variables that contain xyz
- `num_range("x",1:3)` : Selects x1, x2, and x3. This is useful when you have a set of variables that end in numbers. For example, if I had election returns for 2000, 2002, and 2004 called `elect2000, elect2002,` and `elect2004`, I could call `num_range("elect", 2000:2004)` to select these variables.

So, for example, to find the variables that begin with "scheduled", which here is scheduled departure time and scheduled arrival time:

```
select(flights,starts_with("sch"))
# A tibble: 336,776 x 2
   sched_dep_time sched_arr_time
            <int>          <int>
 1            515            819
 2            529            830
 3            540            850
 4            545           1022
 5            600            837
 6            558            728
 7            600            854
 8            600            723
 9            600            846
10            600            745
# ... with 336,766 more rows
```

Select also allows you to reorder variables in your dataset. For example, if you want to move the air time variable to the front of the dataset, you can write:

```
select(flights,air_time,everything())
# A tibble: 336,776 x 19
   air_time  year month   day dep_time sched_dep_time dep_delay
      <dbl> <int> <int> <int>    <int>          <int>     <dbl>
 1      227  2013     1     1      517            515         2
```

```
 2      227  2013     1     1      533            529          4
 3      160  2013     1     1      542            540          2
 4      183  2013     1     1      544            545         -1
 5      116  2013     1     1      554            600         -6
 6      150  2013     1     1      554            558         -4
 7      158  2013     1     1      555            600         -5
 8       53  2013     1     1      557            600         -3
 9      140  2013     1     1      557            600         -3
10      138  2013     1     1      558            600         -2
# ... with 336,766 more rows, and 12 more variables: arr_time
<int>,
#   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

The key here is the `everything()` option. That tells R to keep all of the data, but just to move the variable before (`air_time`) to the front of the data.

So, for example, in one of our "Test Yourself" problems above, we wanted to see all flights operated by United Airlines. We can do this with filter(), but carrier doesn't automatically display on the screen (since it's later in the dataset). We can now combine a call to filter with a call to select to see carrier in our on-screen display:

```
> united <- filter(flights,carrier=="UA")
> select(united,carrier,everything())
# A tibble: 58,665 x 19
   carrier  year month   day dep_time sched_dep_time dep_delay arr_time
   <chr>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1 UA       2013     1     1      517            515         2      830
 2 UA       2013     1     1      533            529         4      850
 3 UA       2013     1     1      554            558        -4      740
 4 UA       2013     1     1      558            600        -2      924
 5 UA       2013     1     1      558            600        -2      923
 6 UA       2013     1     1      559            600        -1      854
 7 UA       2013     1     1      607            607         0      858
 8 UA       2013     1     1      611            600        11      945
 9 UA       2013     1     1      623            627        -4      933
10 UA       2013     1     1      628            630        -2     1016
# ... with 58,655 more rows, and 11 more variables:
#   sched_arr_time <int>, arr_delay <dbl>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

So note that here, we created a new object (united) that is all of the flights operated by United Airlines, and then used select to bring the carrier code (here, UA, since all flights were operated by United) to the front of the display.

Finally, select can also be used to rename variables, but this is not a good idea. If you use select, it drops all other variables that you are not renaming. Instead, use `rename`:

```
rename(flights,sched_dep=sched_dep_time)
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep dep_delay arr_time
   <int> <int> <int>    <int>     <int>     <dbl>    <int>
 1  2013     1     1      517       515         2      830
 2  2013     1     1      533       529         4      850
 3  2013     1     1      542       540         2      923
 4  2013     1     1      544       545        -1     1004
 5  2013     1     1      554       600        -6      812
 6  2013     1     1      554       558        -4      740
 7  2013     1     1      555       600        -5      913
 8  2013     1     1      557       600        -3      709
 9  2013     1     1      557       600        -3      838
10  2013     1     1      558       600        -2      753
# ... with 336,766 more rows, and 12 more variables:
#   sched_arr_time <int>, 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>
```

So here, that code changes `sched_dep_time` to `sched_dep`. Note here that we use = here, because we are using it as an assignment operator (rather than as a means of testing equality).

Rename is a command that's often used to make sure your data has sensible names.

TEST YOURSELF: How would you make a dataset without either actual or scheduled departure time?

ANSWER:

```
select(flights,-c(sched_dep_time,dep_time))
```

When you see `c(variable1, variable2)` in R, it's a shorthand of telling R that you want to apply the operator to both variable 1 and variable 2 (remember we introduced this `c()` syntax in our first lecture on R). And again, the minus sign (-) used with variables tells R we want to drop that variable. So here, this is telling R to drop both scheduled and actual departure time.

You could say:

```
select(flights, -sched_dep_time)
select(flights, -dep_time)
```

But that's not quite right: I wanted a dataset without either one. For that, the syntax above is preferable.

Mutate: Create Functions of Variables

`Mutate()` allows you to create variables that are functions of other variables. This is an extremely useful function, as this is a workhorse function in R. To keep it simple (and so we can see them with the default print options), let's create a small (sml) version of the dataset:

```
flights_sml <- select(flights,
+                       year:day,
+                       ends_with("delay"),
+                       distance,
+                       air_time)
> flights_sml
# A tibble: 336,776 x 7
    year month   day dep_delay arr_delay distance air_time
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>
 1  2013     1     1         2        11     1400      227
 2  2013     1     1         4        20     1416      227
 3  2013     1     1         2        33     1089      160
 4  2013     1     1        -1       -18     1576      183
 5  2013     1     1        -6       -25      762      116
 6  2013     1     1        -4        12      719      150
 7  2013     1     1        -5        19     1065      158
 8  2013     1     1        -3       -14      229       53
 9  2013     1     1        -3        -8      944      140
10  2013     1     1        -2         8      733      138
# ... with 336,766 more rows
```

So let's see mutate in action. For example, suppose I want to find the speed of a particular flight. The speed is just distance divided by time. So here to get the speed, I would write:

```
mutate(flights_sml,speed = distance/air_time*60)
# A tibble: 336,776 x 8
    year month   day dep_delay arr_delay distance air_time speed
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl>
 1  2013     1     1         2        11     1400      227  370.
 2  2013     1     1         4        20     1416      227  374.
 3  2013     1     1         2        33     1089      160  408.
 4  2013     1     1        -1       -18     1576      183  517.
 5  2013     1     1        -6       -25      762      116  394.
 6  2013     1     1        -4        12      719      150  288.
 7  2013     1     1        -5        19     1065      158  404.
 8  2013     1     1        -3       -14      229       53  259.
 9  2013     1     1        -3        -8      944      140  405.
10  2013     1     1        -2         8      733      138  319.
# ... with 336,766 more rows
```

So here, I now have a new variable, speed, that records the speed of each flight (distance divided by time; we multiply by 60 to put it in miles per hour, since `air_time` is measured in minutes).

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

```
transmute(flights_sml, speed = distance/air_time*60)
# A tibble: 336,776 x 1
   speed
   <dbl>
 1  370.
 2  374.
 3  408.
 4  517.
 5  394.
 6  288.
 7  404.
 8  259.
 9  405.
10  319.
# ... with 336,766 more rows
```

There is a nearly infinite number of variables you can create in most datasets, it just simply depends on what you want to do with your data. You can use arithmetic operators (+, - , /, *), logarithmic functions, and many others. A few particularly useful ones:

- Modular arithmetic: this is really useful with time variables. For example, departure time is reported as HHMM (or HMM for times before 10 AM). So we can use this to separate out the hours and minutes into separate variables:

```
transmute(flights,
  dep_time,
  hour = dep_time %/% 100,
  minute = dep_time %% 100
)
# A tibble: 336,776 x 3
   dep_time  hour minute
      <int> <dbl>  <dbl>
 1      517     5     17
 2      533     5     33
 3      542     5     42
 4      544     5     44
 5      554     5     54
 6      554     5     54
 7      555     5     55
 8      557     5     57
 9      557     5     57
10      558     5     58
# ... with 336,766 more rows
```

Why does that work? It follows from the logic of modular operations. `517 %/% 100` divides 517 by 100, and returns the quotient (so here, 5); this is integer division. Then `%%` gives the remainder, so `517 %% 100` means divide 517 by 100, and give the remainder (here, 17). So this is a very handy way of splitting up numbers when they're recorded in this function.

- You can also use `lead()` and `lag()` when you have time series datasets. `Lead()` gives you the next period's value, and `lag()` gives you the previous period's value. For example, suppose I had data on the temperature in Philadelphia every day. I could use those functions to calculate the daily average temperature change.

- You can also use logical operators (<, >, etc.), or ranking (largest/smallest values, etc.).

- You can also calculate ranks using functions like `min_rank()`. For example:

```
longest_delay <- mutate(flights_sml,
                        delay_rank = min_rank(arr_delay))
arrange(longest_delay,delay_rank)
# A tibble: 336,776 x 8
    year month   day dep_delay arr_delay distance air_time delay_rank
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>      <int>
 1  2013     5     7       -14       -86     2565      315          1
 2  2013     5    20       -16       -79     2586      316          2
 3  2013     5     2        -2       -75     2454      300          3
 4  2013     5     6        -4       -75     2422      289          3
 5  2013     5     4        -4       -74     2402      281          5
 6  2013     5     2        -3       -73     2565      314          6
 7  2013     5     6        -2       -71     2454      283          7
 8  2013     5     7        -1       -71     2565      309          7
 9  2013     5    13        -3       -71     2475      290          7
10  2013     1     4        -4       -70     2586      324         10
# ... with 336,766 more rows
```

So the first line of the code creates a new variable (`delay_rank`) that orders the delays (by arrival delay) from shortest to longest. We use `arrange()` to reorder the flights from shortest to longest delay. So, for example, the flight with the shortest delay arrived 86 minutes early (after leaving 14 minutes early; som+(dep_ehow I never manage to find these flights!). If you re-sort from longest to shortest, you'll see that the longest delay was 1272 minutes, or 21 hours and 12 minutes.

You can also use the functions `percent_rank()` and `cume_dist()` to do more complex operations with rankings. See the help file for more details.

Really, there are many different options. This is all about working with the data, since basically mutate can help you create most of the variables that you'll need.

TEST YOURSELF: Change departure time into a variable that records the number of minutes elapsed since midnight. So, for example, 12:40 AM would be 40, 2:15 AM would be 135, etc.

ANSWER:

```
transmute(flights,
  dep_time,
```

```
  hour = dep_time %/% 100,
  minute = dep_time %% 100,(
  elapsed_time = (hour*60) + minute)
```

Note that I built on the modular operations above, and then called a new variable to sum up the number of minutes since midnight. Why would you do this? It is a useful mechanism for calculating elapsed time, as you'll see in the homework below.

Summarise: Create Useful Summaries of Data

`Summarise()` [British spelling since the developers were from New Zealand] collapses a dataset or variable down to a single value. For example, if I want to find the average flight departure delay, I would tell R:

```
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
# A tibble: 1 x 1
  delay
  <dbl>
1  12.6
```

For now, don't worry about `na.rm = TRUE` (it removes missing data, as we'll see below). So we know that the average departure delay is 12.6 minutes. This is nice, but not terribly useful. We gain a lot more power when we add in `group_by()`, which groups your data by another variable. For example, we could group the data by date, and then calculate the average delay by day:

```
> 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 [?]
    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
 7  2013     1     7  5.42
 8  2013     1     8  2.55
 9  2013     1     9  2.28
10  2013     1    10  2.84
# ... with 355 more rows
```

So the average delay on January 1st was 11.5 minutes, 13.9 minutes on January 2nd, and so forth.

TEST YOURSELF: Which month has the longest average departure delay? What about the shortest?

ANSWER:
```
by_month <- group_by(flights, month)
summarise(by_month,mean(dep_delay, na.rm=TRUE))
```

We see that November has the shortest average delay: only 5.44 minutes, whereas the longest delays are in July, 21.7 minutes.

Using the Pipe

`Summarise()` becomes even more powerful when we use a tool called the pipe, which is best illustrated with an example. Suppose I wanted to see how distance relates to delays: are longer flights more likely to be delayed? To do that, I would need to proceed in a few steps:

1. Group flights by destination
2. Use the grouped data to calculate distance and average delay
3. Then plot using ggplot (remember our visualization discussion last time)

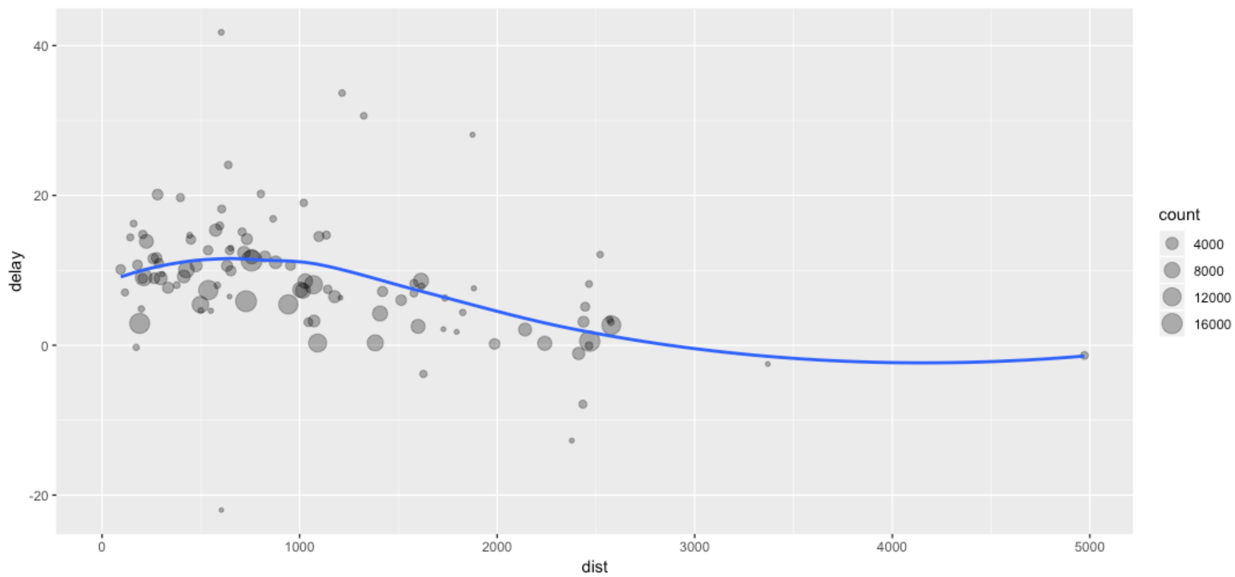
So let's see how we do this in code, step-by-step:

First, group flights by destination
```
by_dest <- group_by(flights, dest)
```

Second, use the grouped data to calculate distance and average delay
```
delay <- summarise(by_dest,
  count = n(),
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE)
)
```

Third, plot using ggplot
```
ggplot(data = delay, mapping = aes(x = dist, y = delay)) +
  geom_point(aes(size = count), alpha = 1/3) +
  geom_smooth(se = FALSE)
```

Which gives us the following:



Note that here, in my call to `summarise()`, I'm using the n() function, which returns a count. This is a very useful function with `summarise()` and `group_by()`. In the call to ggplot(), notice two things as well. First, we made the point size proportional to the number of trips there (so a bigger dot means there are more flights to a given location), and second, we used alpha to make the points transparent, so we can see dots that are close to/on top of one another.

So delays increase as distance travelled increased up to around 750 or 800 miles, and then go down. This makes sense: longer flights can make up more time in the air.

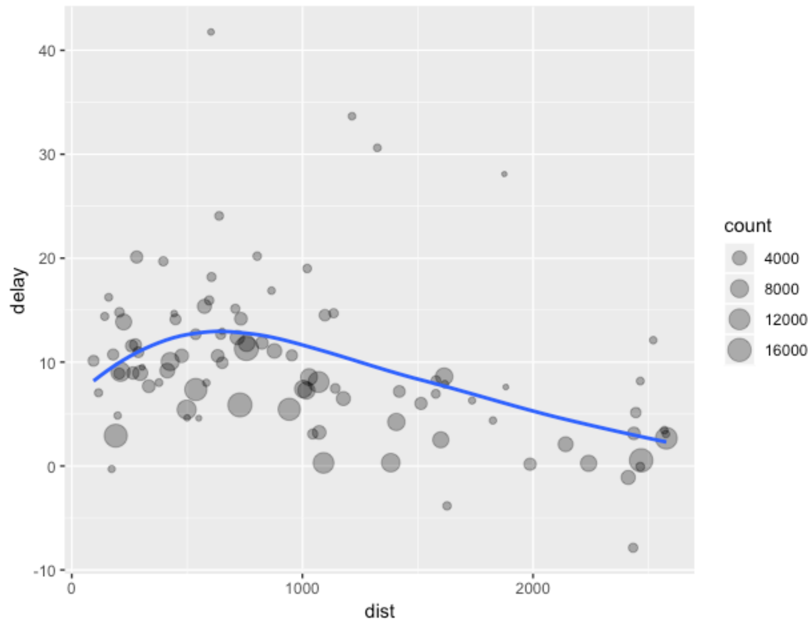
This graph is fine, but we can tidy it up a bit. In particular, note that there's one dot way to the left, which is 5,000 miles from NYC! What is that? Let's find out:

```
> filter(delay,dist>4500)
# A tibble: 1 x 4
  dest  count  dist delay
  <chr> <int> <dbl> <dbl>
1 HNL     707 4973. -1.37
```

It's Honolulu, HI. So notice that because it's so far from the rest of the data, it's skewing our answer a bit (basically, flights to Hawaii are almost twice as far as anywhere else, so it's not very informative about the rest of our data). Let's drop it so we can see the bulk of our data more clearly. Also, I'll remove flights that are not really regular routes: I'll remove airports with fewer than 20 flights in a calendar year (these would be seasonal or special occasion flights, not core routes). To do that, I'm going to add a call to `filter()` before I call `ggplot()`:

```
 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)
```

Let's look at the call to filter. It's doing two things here: (1) removing destinations with fewer than 20 flights, and (2) excluding flights to Honolulu. Here, `dest != "HNL"` says to give me flights not going to Honolulu. In general `!` in computer code means not, so `!=` means "not equal to." Now this gives us a plot where the general relationship is easier to see:



Now we can see the pattern more clearly: delays rise until a distance of around 800 miles or so, and then decline in distance, as longer flights can make up more time in the air.

The code above is perfectly acceptable R code. But there's an easier way to write it using something called 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")
```

`%>%` is called a pipe, and is typically read as "then" when reading code. Piping makes code much more readable. So here, this makes the code more legible, as it tells me to create an object delays that takes flights, then groups it by destination, then summarizes the average distance/delay by destination, and then filters out the noise. As you get longer blocks of code stringing together more operations, piping becomes more valuable.

TEST YOURSELF: Calculate the average arrival delay by carrier.

ANSWER:
```
flights %>%
```

```
  group_by(carrier) %>%
  summarise(
    delay = mean(arr_delay, na.rm=T))
```

From this, we see that some carriers are actually pretty good: Alaska (AS) and Hawaiian (HA) both arrive early by a few minutes (consistent with our argument above about longer flights making up time in the air). Others, like Frontier (F9) and AirTran (FL) arrival, on average, more than 20 minutes late.

Exploring Missing Data

Several times in the code above, we issued the option `na.rm = TRUE` (for example, when we calculated the averages (means) above). What does that do? To see, it's helpful to omit it (this is often a good trick for unpacking a function in R):

```
> flights %>%
+   group_by(year, month, day) %>%
+   summarise(mean = mean(dep_delay))
# A tibble: 365 x 4
# Groups:   year, month [?]
    year month   day  mean
   <int> <int> <int> <dbl>
 1  2013     1     1    NA
 2  2013     1     2    NA
 3  2013     1     3    NA
 4  2013     1     4    NA
 5  2013     1     5    NA
 6  2013     1     6    NA
 7  2013     1     7    NA
 8  2013     1     8    NA
 9  2013     1     9    NA
10  2013     1    10    NA
# ... with 355 more rows
```

So now it tells me that each value is NA, which is R's way of saying missing (NA = Not Available). Why does it do that? Because R, like most statistical programs, follows a basic rule: if there is missing data in the input, then the output is a missing value. Here, we have missing data in departure delay. What are these flights? To find missing data in R, you use the function `is.na()`. Let's use filter to tell us:

```
filter(flights, is.na(dep_delay))
# A tibble: 8,255 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1       NA           1630        NA       NA
 2  2013     1     1       NA           1935        NA       NA
 3  2013     1     1       NA           1500        NA       NA
```

```
 4  2013     1     1       NA           600        NA       NA
 5  2013     1     2       NA          1540        NA       NA
 6  2013     1     2       NA          1620        NA       NA
 7  2013     1     2       NA          1355        NA       NA
 8  2013     1     2       NA          1420        NA       NA
 9  2013     1     2       NA          1321        NA       NA
10  2013     1     2       NA          1545        NA       NA
# ... with 8,245 more rows, and 12 more variables:
#   sched_arr_time <int>, 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>
```

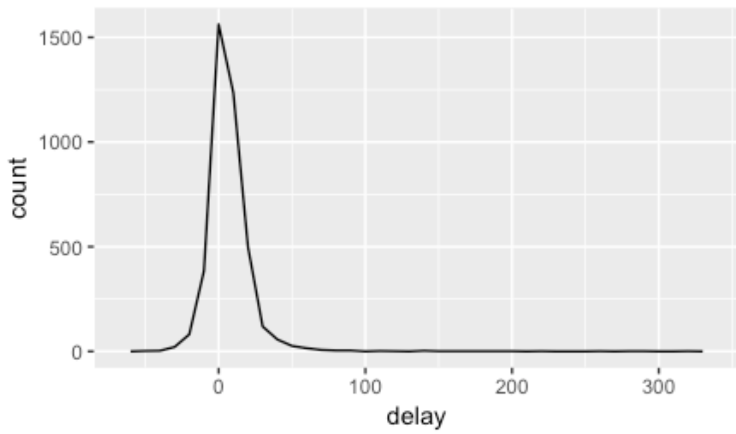
So that gives us the flights that having missing data on the `dep_delay` variable. What do we see? These are cancelled flights! They never depart or arrive, so they by definition have no departure delay! So how can we use these flights to calculate a departure delay? We can't, so if we try to include them, R tells us the answer is missing: it doesn't know how to find it. Instead, we have to tell R to remove these cases, and calculate the answer without them. This comes up frequently in R when calculating any sort of statistic (e.g., mean, median, standard deviation, and so forth).

This also brings up a good point about using `group_by()` and `summarise()`: you should keep track of how much data is underlying each observation. Sometimes, you'll get an extreme value, but that's because you have only a small amount of data. Let's see this with an example. Whissummch airplanes (identified by their tail number) are the most frequently delayed?

To make our lives simpler, let's begin by making a subset of our data that contains only flights that actually happened (i.e., non-cancelled flights, the kind you want when you're travelling). We'll reference this dataset several times in the examples below, since it makes our lives easier (think about what we now don't have to do!).

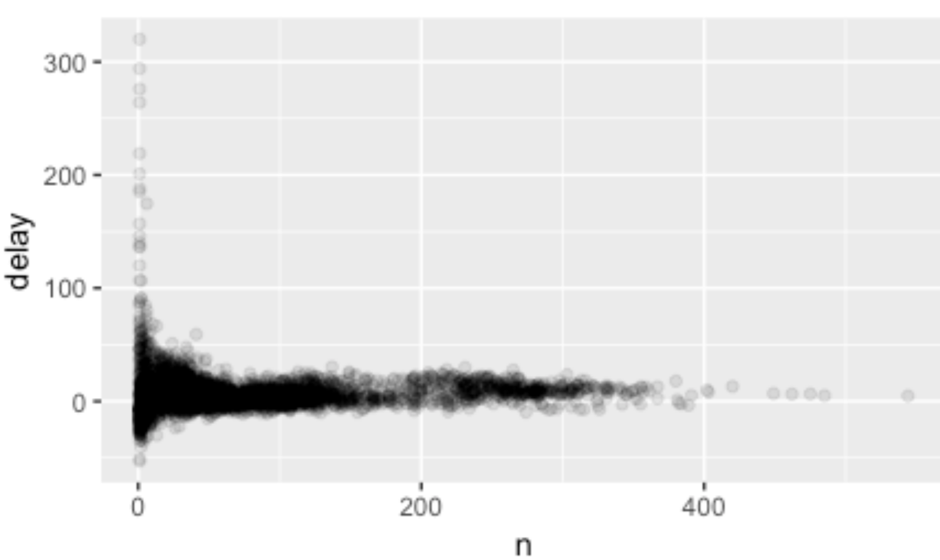
```
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarise(
    delay = mean(arr_delay)
  )

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

Which gives us:



So note that while there are a lot of planes that have very short delays, there is a long right tail, with a small number of planes having an average delay of 300 minutes, or 5 hours!
But this is a bit misleading: let's look at the number of flights by plane and plot delays against number of flights:

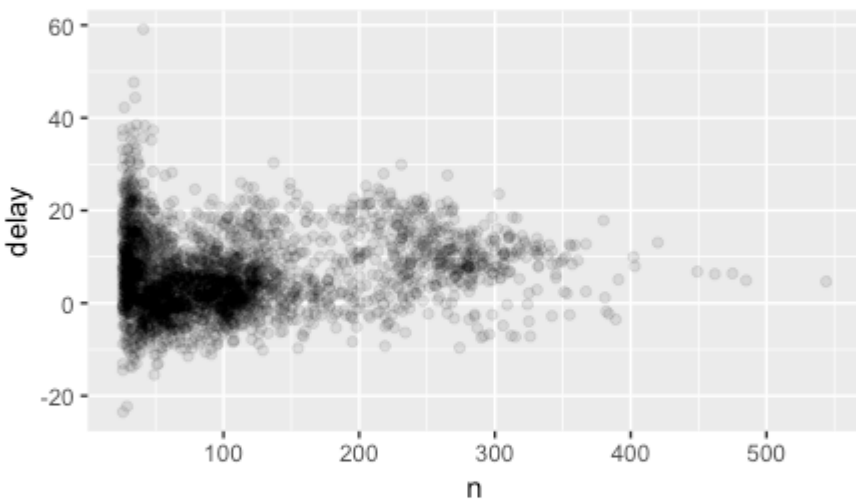
```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarise(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )
ggplot(data = delays, mapping = aes(x = n, y = delay)) +
  geom_point(alpha = 1/10)
```



So note that basically all of the delays over 100 minutes come from planes that fly very few flights. Not that as a plane flies more flights, the average delay shrinks more toward 0. This is true in almost all datasets: when you plot the mean (or another summary function) against sample size, you find that as sample size increases, variation decreases (you'll see why this happens formally in the second course in this series, DATA 201).

To filter out the noise in the plot above, it's useful to eliminate the planes that flew the smallest number of flights. To see that, we can call:

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



Which gets rid of some of the extreme values from the first plot. Note that in ggplot(), which was written before the pipe was introduced, you need to use a +, rather than the pipe. Chapter 5 of the textbook does more on this point, using a different dataset on batting averages. It's worth exploring on your own as a test of your understanding of this material.

Combining These Functions Together!

So far, we've largely worked with each function (filter(), arrange(), etc.) on their own. But we can also see how to combine these different functions to do some more advanced calculations. For example, suppose we wanted to find the percentage of cancelled flights by carrier. To do this, we'll proceed in several steps:

1. We'll create an indicator for whether a flight was cancelled using `ifelse()`, one of the most commonly used R functions)
2. We'll use `groupby()` and `summarize()` to count the number of flights and cancelled flights by carrier
3. We'll calculate the percentage of cancelled flights by carrier

But because of the pipe, the code is actually pretty straightforward:

```
flights %>%
  mutate(.,cancelled = ifelse((is.na(dep_delay) & is.na(arr_delay)),1,0)) %>%
  group_by(carrier) %>%
  summarise(100*mean(cancelled))
```

```
# A tibble: 16 x 2
   carrier `100 * mean(cancelled)`
   <chr>                     <dbl>
 1 9E                        5.66
 2 AA                        1.94
 3 AS                        0.280
 4 B6                        0.853
 5 DL                        0.725
 6 EV                        5.20
 7 F9                        0.438
 8 FL                        2.24
 9 HA                        0
10 MQ                        4.67
11 OO                        9.38
12 UA                        1.17
13 US                        3.23
14 VX                        0.601
15 WN                        1.56
16 YV                        9.32
```

Let's unpack the mutate statement and the use of `ifelse()`. Note that the first argument inside mutate is a dot (.): this is R shorthand, and just means take what was passed to it via the pipe. So I could equivalently write:

```
flights %>%
  mutate(flights,cancelled = ifelse(is.na(dep_delay) & is.na(arr_delay),1,0))
```

And we'd get the same answer. Here, it doesn't matter much, but as you use more complicated pipes, it's a valuable shorthand.

The other part of the first operation is the call to ifelse. Ifelse is one of the most powerful R expressions, and its one that I use perhaps more than any other. It operates as follows:

```
ifelse(logical expression, value if TRUE, value if FALSE)
```

So in the example above, if both arrival and departure delay are missing, it gives a value of 1 to the variable cancelled, and 0 otherwise. So this is a way of creating a new variable that operationalizes our definition of cancelled, and creates a binary (0/1) variable that tells us whether or not a given flight was cancelled. We'll come back to `ifelse()` in a future lecture, but it is an <u>extremely</u> powerful tool.

The next line of the code uses `group_by()`, as we saw above, and then we exploit a convenient trick in the final line. If you have a 0/1 variable (like cancelled), its mean (average) is just the proportion of cases where the variable equals 1 (I multiply by 100 to get a percentage rather than a proportion). So note that with just a few lines of code, we can start calculating much more useful summaries of different types of information.

Now that we understand the code, what does this mean substantively? We see that Hawaiian Airlines (HA) *never* cancelled a flight from New York in 2013 (even if one of their flights was delayed by 21 hours, it wasn't cancelled!), while SkyWest (OO) cancelled over 9% of its flights. Of the legacy carriers, Delta (DL) is the best (0.73% cancelled), and American (AA) is the worst (1.94% cancelled). Of course, this brings up the point we discussed above: we should we calculating the number of flights each carrier makes to ensure that we're not drawing too many conclusions from small data! Let's do that now.

TEST YOURSELF: How many flights does each carrier fly?

ANSWER:
```
flights %>%
   group_by(carrier) %>%
   summarise(
     count = n()
   )
```

And note what this tells us: SkyWest may be cancelled 9.4% of the time, but it only flew 32 total flights, so that's only 3 cancelled flights (so not a huge deal). To put it in perspective, even though Delta cancelled less than 1% of its flights, it still cancelled almost 350 flights! This also suggests that Endeavor Air (9E) is actually a pretty bad airline for cancellations, since they flew over 18,000 flights, but were delayed 5% of the time (Endeavor is a subsidiary of Delta, and flies the short-haul flights for them).

We can also combine aggregation with logical subsetting. This just means that we're going to look at part of a data that meets a given logical condition. For example, notice that delays include negative values, which indicate that the plane left early. I can also calculate the average delay conditional on the flight actually being delayed (i.e., the average delay conditioned on having a positive delay). To see this compare the following two values:

```
> 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
+   )
# A tibble: 365 x 5
# Groups:   year, month [?]
    year month   day avg_delay1 avg_delay2
   <int> <int> <int>      <dbl>      <dbl>
 1  2013     1     1      12.7        32.5
 2  2013     1     2      12.7        32.0
```

```
 3  2013     1     3      5.73        27.7
 4  2013     1     4     -1.93        28.3
 5  2013     1     5     -1.53        22.6
 6  2013     1     6      4.24        24.4
 7  2013     1     7     -4.95        27.8
 8  2013     1     8     -3.23        20.8
 9  2013     1     9     -0.264       25.6
10  2013     1    10     -5.90        27.3
# ... with 355 more rows
```

So note how much longer the delays are once we condition on having an actual delay. For example, on January 10th, across all non-cancelled flights, the average flight left almost 6 minutes early. But of flights that were delayed, the average delay was 27 minutes.

So far, we've just called the `mean()` and `n()` from within summarize. But as always, there are many different functions you can call (these are just perhaps the two most useful). Some others include the median, standard deviation, minimum/maximum, and so forth. The book has many examples, let's see just a few of them.

For example, let's find the first and last departure on each day, using the `first()` and `last()` functions:

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    first_dep = first(dep_time),
    last_dep = last(dep_time)
  )
# A tibble: 365 x 5
# Groups:   year, month [?]
    year month   day first_dep last_dep
   <int> <int> <int>     <int>    <int>
 1  2013     1     1       517     2356
 2  2013     1     2        42     2354
 3  2013     1     3        32     2349
 4  2013     1     4        25     2358
 5  2013     1     5        14     2357
 6  2013     1     6        16     2355
 7  2013     1     7        49     2359
 8  2013     1     8       454     2351
 9  2013     1     9         2     2252
10  2013     1    10         3     2320
# ... with 355 more rows
```

So notice that the first flight typically departs right after midnight or around 5 AM, with the last flight of the day right before midnight.

It's also frequently helpful to know the number of unique values of a given variable, which we can get with the `n_distinct()` command. For example, how many carriers fly to each destination? If we group the data by destination, and then count the number of unique carriers by destination, we'll have our answer:

```
> not_cancelled %>%
+   group_by(dest) %>%
+   summarise(carriers = n_distinct(carrier)) %>%
+   arrange(desc(carriers))
# A tibble: 104 x 2
   dest  carriers
   <chr>    <int>
 1 ATL          7
 2 BOS          7
 3 CLT          7
 4 ORD          7
 5 TPA          7
 6 AUS          6
 7 DCA          6
 8 DTW          6
 9 IAD          6
10 MSP          6
# ... with 94 more rows
```

So notice that Atlanta, Boston, Charlotte, Chicago O'Hare, and Tampa Bay—other airport hubs—are served by 7 carriers, and then a set of large cities (Austin, Washington, Detroit, etc.) are served by 6, and so forth.

TEST YOURSELF: In the complete flights dataset, how many destinations are included? How many planes?

ANSWER:

```
flights %>%
  summarise(n_distinct(dest),
            n_distinct(tailnum))
```

TEST YOURSELF: How many different airplanes (tail numbers, not models) does each carrier fly?

ANSWER:

```
flights %>%
  group_by(carrier) %>%
  summarise(n_distinct(tailnum))
```

You can see that the number of planes varies quite markedly by carrier. You could also plot this against destination, distance, etc. to explore further.

Counts come up a lot in R, so there's a general function for that: `count()`. For example, we can count the number of flights going to each destination (grouping by destination and then counting the number of flights):

```
> not_cancelled %>%
+   count(dest)
# A tibble: 104 x 2
   dest      n
   <chr> <int>
 1 ABQ     254
 2 ACK     264
 3 ALB     418
 4 ANC       8
 5 ATL   16837
 6 AUS    2411
 7 AVL     261
 8 BDL     412
 9 BGR     358
10 BHM     269
# ... with 94 more rows
```

So note that there are 16,837 not cancelled flights from New York to Atlanta, or more than 46 flights per day!

Note that you can sum and find the mean of logical expressions as well. This is very handy, because summing a logical expression gives you how many cases where something is true, and the mean gives you the proportion. We saw this trick above in our example of the new variable cancelled. Let's see another example by finding the percentage of flights each day that are delayed by more than 1 hour:

```
> not_cancelled %>%
+   group_by(year, month, day) %>%
+   summarise(hour_perc = 100*mean(arr_delay > 60))
# A tibble: 365 x 4
# Groups:   year, month [?]
    year month   day hour_perc
   <int> <int> <int>     <dbl>
 1  2013     1     1      7.22
 2  2013     1     2      8.51
 3  2013     1     3      5.67
 4  2013     1     4      3.96
 5  2013     1     5      3.49
 6  2013     1     6      4.70
 7  2013     1     7      3.33
 8  2013     1     8      2.13
 9  2013     1     9      2.02
10  2013     1    10      1.83
```

```
# ... with 355 more rows
```

Note that again, 100*proportion gives us the percentage (I find that easier to read, but your mileage may vary). So you can see that the percentage of flights delayed by more than 1 hour varies quite a lot, from just about 2% on January 10th, to 8.5% on the 2nd.

Throughout the above, we've frequently been grouping by year, month, and day. Note that R knows this grouping order, and we can roll up the data one layer at a time. For example, in that grouped data, I can find the number of flights per day, then per month, then per year in a straightforward way:

```
> 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))
> per_day
# 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
 7  2013     1     7     933
 8  2013     1     8     899
 9  2013     1     9     902
10  2013     1    10     932
# ... with 355 more rows
> per_month
# 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
 7  2013     7   29425
 8  2013     8   29327
 9  2013     9   27574
10  2013    10   28889
11  2013    11   27268
```

```
12  2013    12   28135
> per_year
# A tibble: 1 x 2
   year flights
  <int>   <int>
   1  2013  336776
```

So that's quite neat: it knows how I've got the data sorted, so I can summarize the data across levels in a very straightforward manner! But note that this only works because of that initial call to `group_by()`.

Note that you can ungroup data as well using the `ungroup()` function:

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

So as you would expect, ungroup() reverses group_by().

Finally, let's end with one more example that helps us think through how to combine all of the different tools in the `dplyr()` library. This will be helpful for your homework where I ask you to think through multi-step questions to find the answer (hint hint!).

Suppose we wanted to know which destinations have the smallest percentage of cancelled flights. After all, we want to avoid cancellations, so we'd want to know which airports are least likely to see flights cancelled.

How would we solve this problem? We'd proceed in a few steps:

1. We'll see if each flight was cancelled
2. We'd need to group flights by destination
3. For each destination, calculate the percentage of flights that were cancelled
4. Then arrange them so we can see which destinations have the fewest cancelled flights

So note that each one of those steps corresponds to a function in `dplyr()`!

Let's break it down:

Step #1: Tell whether each flight was cancelled

Since there isn't already a variable telling us whether or not a flight was cancelled, we will make a new variable using `mutate()` that will indicate whether a flight was cancelled or not. To do this, we can look at the variable `dep_time`. If a flight has no recorded departure time information, it is safe to say that it did not depart (in other words, it was cancelled). With this in mind, we can use the `is.na()` function with the `dep_time` variable to let us know whether each flight departed or was cancelled. With this new variable, `is.na(dep_time)` will give a 1 for each flight where there is an NA for the `dep_time` variable and a 0 for each flight there is a value for the `dep_time` variable. We can call this new variable `cancelled` so we remember what we are finding.

```
flights %>%
+   mutate(cancelled = is.na(dep_time))
```

#2: Group flights by airport

Next, we look back at the question and see that we want to know not only about the cancelled flights, but *which* destination has the smallest percentage of these cancellations. Usually, when you want to know something about a specific variable (like we do here with the destination airports), `group_by()` will be helpful analyze data by that certain variable. So, as shown in the code below, we will add to the earlier code `group_by(dest)`, indicating that we want to group by the `dest` variable for destination. This will allow us to learn about the flights going to each destination airport, instead of just the individual flights.

```
flights %>%
+   mutate(cancelled = is.na(dep_time)) %>%
+   group_by(., dest)
```

#3: For each destination, what percentage of flights were cancelled?

Typically, after `group_by()`, you will want to use `summarise()` to find out further information based on your new grouping. The question wants to know the percentage of cancelled flights. Using `summarise()`,we can calculate that based on the groupings set by `group_by()`, in this case destination. Therefore, to find the percentage, we can use `mean()` to get the average of flights that are cancelled for each airport.

Why does taking the mean work? Remember that our variable cancelled is either 0 (not cancelled) or 1 (cancelled). Since each value is either a 1 or a 0, the mean of that variable will be the proportion of flights with a 1, or the proportion of cancelled flights. To make this into a percentage, I simply multiply by 100. This is a neat trick that comes up a lot in these problems, so it's worth making a note of this so you can use it in future problems.

In my call to `summarise()` I create a new variable, called `pct.cancel`, that tells me the percentage of cancelled flights for each destination airport.

```
flights %>%
+   mutate(cancelled = is.na(dep_time)) %>%
+   group_by(., dest) %>%
```

```
+   summarise(pct.cancel = 100*mean(cancelled))
```

#4: Arrange so that we can see which destinations have the smallest percentage of flights

Finally, we want to see which destination has the *smallest* percentage. To see things in order of a certain variable, we will use `arrange()`. This will allow us to see the data by the grouping and in the order of the variable we define. In this case, we grouped by destination and will arrange in ascending order of `pct.cancel` to see which destination airport has the smallest percentage of cancelled flights.

Putting it all together, we get the following code:

```
flights %>%
+   mutate(cancelled = is.na(dep_time)) %>%
+   group_by(., dest) %>%
+   summarise(pct.cancel = 100*mean(cancelled)) %>%
+   arrange(pct.cancel)
```

If we run all of this code, we see the following output:

```
# A tibble: 105 x 2
   dest  pct.cancel
   <chr>      <dbl>
 1 ABQ        0
 2 ACK        0
 3 ANC        0
 4 EYW        0
 5 LEX        0
 6 SBN        0
 7 BUR        0.270
 8 HNL        0.283
 9 SJC        0.304
10 OAK        0.321
# … with 95 more rows
```

We can see that the first 6 destination airports have zero cancelled flights (or such small percentages that R rounds down to zero). All of these airports are pretty small airports, so it may be that they don't receive many flights, and therefore don't have many, if any, cancellations.

TEST YOURSELF: Above, we found the destinations with the fewest cancellations. Find the 5 destinations with the most cancellations.

ANSWER:

This is very similar to the problem we solved above, with just a slight twist at the end. Here, our steps would be:

1. We'll see if each flight was cancelled
2. We'd need to group flights by destination
3. For each destination, calculate the percentage of flights that were cancelled
4. Then arrange them so we can see which destinations have the most cancelled flights
5. Show just the 5 worst destinations for cancellations

Note that steps #1-3 are identical to the problem above, but steps #4 and #5 are slightly different. So for steps #1-3, we have:

```
flights %>%
+   mutate(cancelled = is.na(dep_time)) %>%
+   group_by(., dest) %>%
+   summarise(pct.cancel = 100*mean(cancelled))
```

#4: Arrange to show which flights are most likely to be cancelled

This is very similar to step #4 above, except that now we use the arrange(desc()) options to rank them from high to low:

```
flights %>%
+   mutate(cancelled = is.na(dep_time)) %>%
+   group_by(., dest) %>%
+   summarise(pct.cancel = 100*mean(cancelled)) %>%
+   arrange(desc(pct.cancel)
```

#5: Show just the 5 worst destinations

To do this, we can use the `head(n)` function, which just gives us the first n rows of the data. Here, because we want the first 5 rows, we write:

```
flights %>%
+   mutate(cancelled = is.na(dep_time)) %>%
+   group_by(., dest) %>%
+   summarise(pct.cancel = 100*mean(cancelled)) %>%
+   arrange(desc(pct.cancel))%>%
+   head(5)
```

Which gives us:

```
# A tibble: 5 x 2
  dest  pct.cancel
  <chr>      <dbl>
1 LGA       100
2 JAC        12
3 CHO        11.5
4 BHM         8.42
5 TYS         8.24
```

If you look at the data, you'll see that there was only 1 flight to LGA (LaGuardia) in our data: US Airways 1632 on July 27th, from Newark to LaGuardia, and it was cancelled, so that's a clear outlier. Otherwise, Jackson Hole (JAC) and Charlottesville (CHO) are the only destinations where more than 10% of the flights get cancelled.

TEST YOURSELF: Above, we saw that 6 destinations—ABQ, ACK, ANC, EYW, LEX, and SBN—have no cancelled flights in 2013. How many flights flew to those destinations?

ANSWER:

There are three steps to answering this question:

1. Filter down the data to just those 6 airports
2. Group by airport
3. Calculate the number of flights to each airport

So step #1 is a call to `filter()`, step #2 involves `group_by()`, and finally step #3 has us using `summarise()` and the `n()` function to get the count. So putting it all together, we have:

```
flights %>%
  filter(dest == "ABQ" | dest=="ACK" | dest == "ANC" | dest ==
"EYW" | dest == "LEX" | dest=="SBN") %>%
  group_by(., dest) %>%
  summarise(n_flights = n())
```

Which gives us:

```
# A tibble: 6 x 2
  dest  n_flights
  <chr>     <int>
1 ABQ         254
2 ACK         265
3 ANC           8
4 EYW          17
5 LEX           1
6 SBN          10
```

So interestingly, there are a reasonable number of flights to Albuquerque (ABQ) and Nantucket (ACK), but the others have 17 or fewer flights, so these are not any sort of regular destinations. We'd want to investigate more, but because Nantucket is only a seasonal destination, and Albuquerque has good weather (lots of sun, little rain/snow), that likely explains why these destinations have so few cancellations.

This is just a tiny taste of what `dplyr()` can do. Really, the sky is the limit, and it just takes a lot of practice, and trial-and-error, to learn it.