Data Wrangling I

Dav King

1-20-2022

Main Ideas

- Organizing our data according to a consistent set of "tidy" principles makes data easy to work with and leverages the ways `R` is effective.
- Often we need to wrangle our data in order to extract meaning (including creating new variables, calculating summary statistics, subsetting data, etc).
- Using only **seven key verbs** we can accomplish a wide variety of data wrangling tasks.

Coming Up

- Homework #01 assigned today.
- Lab 3 on Monday.

"Happy families are all alike; every unhappy family is unhappy in its own way" - Leo Tolstoy

Lecture Notes and Exercises

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

Tidy Principles

Tidy data has three related characteristics

1. Each variable forms a column.
2. Each observation forms a row.
3. Each value has its own cell.

Let's look at some examples!

Data Wrangling

Often we need to wrangle our data to extract meaning. This includes calculating new variables, summary statistics, grouping by variables, renaming, reordering, selecting subsets of data, filtering by various conditions, etc.

We can accomplish a great deal of wrangling by learning just **seven key verbs**. Each of these functions takes a data frame as input and returns a data frame as output.

- filter
- arrange
- select
- slice
- mutate
- summarize
- group_by

To demonstrate data wrangling we will use a dataset of characteristics of all flights departing from New York City (JFK, LGA, EWR) in 2013. If the library command does not work, you may need to install the package first using the commented line of code (note you only need to do this once, **only do it if the library command does not work**).

```
#install.packages(nycflights13)
library(nycflights13)
```

We first explore the data a bit. Examine the documentation as well.

```
glimpse(flights)
```

```
## Rows: 336,776
## Columns: 19
## $ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2~
## $ month          <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
## $ day            <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
## $ dep_time       <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558, ~
## $ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600, ~
## $ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -1~
## $ arr_time       <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849,~
## $ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851,~
## $ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -1~
## $ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", "~
## $ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, 4~
## $ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N394~
## $ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA",~
## $ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD",~
## $ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, 1~
## $ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733, ~
## $ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, 6~
## $ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0, 59, 0~
## $ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 0~
```

```
names(flights)
```

```
##  [1] "year"           "month"          "day"            "dep_time"
##  [5] "sched_dep_time" "dep_delay"      "arr_time"       "sched_arr_time"
##  [9] "arr_delay"      "carrier"        "flight"         "tailnum"
## [13] "origin"         "dest"           "air_time"       "distance"
## [17] "hour"           "minute"         "time_hour"
```

```
head(flights)
```

```
## # A tibble: 6 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1  2013     1     1      517            515         2      830            819
## 2  2013     1     1      533            529         4      850            830
## 3  2013     1     1      542            540         2      923            850
## 4  2013     1     1      544            545        -1     1004           1022
## 5  2013     1     1      554            600        -6      812            837
## 6  2013     1     1      554            558        -4      740            728
## # ... with 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>
```

The `head()` function returns "A tibble: 6 x 19" and then the first six rows of the `flights` data. A **tibble** is a tweaked, opinionated version of the `R`data frame.

There are a few differences a `tidyverse` tibble and an `R` data frame. Two of the main ones are described below.

First, it provides more information than a data frame. When you print a tibble, it will show the first ten rows and all of the columns that fit on the screen, along with the type of each column. Try this with the `flights` data. You can modify the number of rows and columns shown using the `print()` function.

Question: Can you print the first three rows and all columns of the `flights` data? Check the documentation!

It's possible to print just 3 rows, but the number of columns you want to print is dictated by the width of your console - eventually, you run out of space and it refuses to keep going.

```
print(flights, n = 3, max_extra_cols = 19)
```

```
## # A tibble: 336,776 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1  2013     1     1      517            515         2      830            819
## 2  2013     1     1      533            529         4      850            830
## 3  2013     1     1      542            540         2      923            850
## # ... with 336,773 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>
```

Second, tibbles are somewhat more strict than data frames when it comes to subsetting data.

select()

The `select()` function picks off one or more columns by name.

Let's say we want a dataset that only contains the variables `dep_delay` and `arr_delay`.

```
select(flights, dep_delay, arr_delay)
```

```
## # A tibble: 336,776 x 2
##    dep_delay arr_delay
##        <dbl>     <dbl>
##  1         2        11
##  2         4        20
##  3         2        33
##  4        -1       -18
##  5        -6       -25
##  6        -4        12
##  7        -5        19
##  8        -3       -14
##  9        -3        -8
## 10        -2         8
## # ... with 336,766 more rows
```

We can also use `select()` to exclude variables. Let's exclude `dep_delay` but keep all other variables.

```
select(flights, -dep_delay)
```

```
## # A tibble: 336,776 x 18
##     year month   day dep_time sched_dep_time arr_time sched_arr_time arr_delay
##    <int> <int> <int>    <int>          <int>    <int>          <int>     <dbl>
##  1  2013     1     1      517            515      830            819        11
##  2  2013     1     1      533            529      850            830        20
##  3  2013     1     1      542            540      923            850        33
##  4  2013     1     1      544            545     1004           1022       -18
##  5  2013     1     1      554            600      812            837       -25
##  6  2013     1     1      554            558      740            728        12
##  7  2013     1     1      555            600      913            854        19
##  8  2013     1     1      557            600      709            723       -14
##  9  2013     1     1      557            600      838            846        -8
## 10  2013     1     1      558            600      753            745         8
## # ... with 336,766 more rows, and 10 more variables: carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
## #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

We can also use `select()` to select a range of variables. Here, we select the first three variables representing the departure day.

```
select(flights, year:day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
```

```
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

arrange()

The `arrange()` function orders rows (observations) in specific ways.

Let's arrange the data by descending departure delays, with large departure delays on top.

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

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     9      641            900      1301     1242           1530
##  2  2013     6    15     1432           1935      1137     1607           2120
##  3  2013     1    10     1121           1635      1126     1239           1810
##  4  2013     9    20     1139           1845      1014     1457           2210
##  5  2013     7    22      845           1600      1005     1044           1815
##  6  2013     4    10     1100           1900       960     1342           2211
##  7  2013     3    17     2321            810       911      135           1020
##  8  2013     6    27      959           1900       899     1236           2226
##  9  2013     7    22     2257            759       898      121           1026
## 10  2013    12     5      756           1700       896     1058           2020
## # ... 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>
```

Or with low departure delays on top.

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

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013    12     7     2040           2123       -43       40           2352
##  2  2013     2     3     2022           2055       -33     2240           2338
##  3  2013    11    10     1408           1440       -32     1549           1559
##  4  2013     1    11     1900           1930       -30     2233           2243
##  5  2013     1    29     1703           1730       -27     1947           1957
##  6  2013     8     9      729            755       -26     1002            955
##  7  2013    10    23     1907           1932       -25     2143           2143
##  8  2013     3    30     2030           2055       -25     2213           2250
```

```
##  9  2013     3     2     1431           1455       -24     1601           1631
## 10  2013     5     5      934            958       -24     1225           1309
## # ... 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>
```

What if we only want to examine the `dep_delay` and `arr_delay` columns? We can combine `arrange()` and `select()`!

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

```
## # A tibble: 336,776 x 2
##    dep_delay arr_delay
##        <dbl>     <dbl>
##  1      1301      1272
##  2      1137      1127
##  3      1126      1109
##  4      1014      1007
##  5      1005       989
##  6       960       931
##  7       911       915
##  8       899       850
##  9       898       895
## 10       896       878
## # ... with 336,766 more rows
```

It is not easy to understand what is going on in the code chunk above.

- we have to read from inside out and right to left
- not clear which argument goes with which function
- doesn't focus on the functions

The pipe is a technique for passing information from one process to another.

```
flights %>%
  arrange(desc(dep_delay)) %>%
  select(dep_delay, arr_delay)
```

```
## # A tibble: 336,776 x 2
##    dep_delay arr_delay
##        <dbl>     <dbl>
##  1      1301      1272
##  2      1137      1127
##  3      1126      1109
##  4      1014      1007
##  5      1005       989
##  6       960       931
##  7       911       915
##  8       899       850
##  9       898       895
## 10       896       878
## # ... with 336,766 more rows
```

When reading code "in English", say "and then" whenever you see a pipe.

Question: How would you read the code chunk above in English? What is it accomplishing?

You look within the `flights` data frame, and then you arrange it according to `dep_delay` in descending order, and then you select only the columns `dep_delay` and `arr_delay` for viewing.

`slice()`

Slice selects rows based on their position.

Here we slice off the first 5 rows of the `flights` data.

```
flights %>%
  slice(1:5)
```

```
## # A tibble: 5 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1  2013     1     1      517            515         2      830            819
## 2  2013     1     1      533            529         4      850            830
## 3  2013     1     1      542            540         2      923            850
## 4  2013     1     1      544            545        -1     1004           1022
## 5  2013     1     1      554            600        -6      812            837
## # ... with 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>
```

We can also slice the last two rows.

```
flights %>%
  slice((n()-1):n())
```

```
## # A tibble: 2 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1  2013     9    30       NA           1159        NA       NA           1344
## 2  2013     9    30       NA            840        NA       NA           1020
## # ... with 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>
```

Question: What is the code chunk below accomplishing? Guess before running the code.

Drawing from the data frame `flights`, the code chunk will arrange the tibble by `dep_delay` in descending order, and then slice off only the first 5 rows for viewing.

```
flights %>%
  #arrange(desc(dep_delay)) %>%
  slice(1:5)
```

```
## # A tibble: 5 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
```

```
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1  2013     1     1      517            515         2      830            819
## 2  2013     1     1      533            529         4      850            830
## 3  2013     1     1      542            540         2      923            850
## 4  2013     1     1      544            545        -1     1004           1022
## 5  2013     1     1      554            600        -6      812            837
## # ... with 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>
```

To add comments to code, use the pound sign. This is helpful for debugging as well - you can temporarily disable a line.

Question: What will happen if you comment out the line containing `arrange()` in the code chunk above? Try it.

It will only slice off the first five rows of `flights` for viewing, without sorting them by `dep_delay` at all.

`filter()`

`filter()` selects rows satisfying certain conditions.

We can use a single condition. Here we select all rows where the destination airport is RDU.

```
flights %>%
  filter(dest == "RDU")
```

```
## # A tibble: 8,163 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      800            810       -10      949            955
##  2  2013     1     1      832            840        -8     1006           1030
##  3  2013     1     1      851            851         0     1032           1036
##  4  2013     1     1      917            920        -3     1052           1108
##  5  2013     1     1     1024           1030        -6     1204           1215
##  6  2013     1     1     1127           1129        -2     1303           1309
##  7  2013     1     1     1157           1205        -8     1342           1345
##  8  2013     1     1     1240           1235         5     1415           1415
##  9  2013     1     1     1317           1325        -8     1454           1505
## 10  2013     1     1     1449           1450        -1     1651           1640
## # ... with 8,153 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>
```

We can use more than one condition. Here we select all rows where the destination airport is RDU and the arrival delay is less than 0.

```
flights %>%
  filter(dest == "RDU", arr_delay < 0)
```

```
## # A tibble: 4,232 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
```

```
##  1  2013     1     1      800            810       -10      949            955
##  2  2013     1     1      832            840        -8     1006           1030
##  3  2013     1     1      851            851         0     1032           1036
##  4  2013     1     1      917            920        -3     1052           1108
##  5  2013     1     1     1024           1030        -6     1204           1215
##  6  2013     1     1     1127           1129        -2     1303           1309
##  7  2013     1     1     1157           1205        -8     1342           1345
##  8  2013     1     1     1317           1325        -8     1454           1505
##  9  2013     1     1     1505           1510        -5     1654           1655
## 10  2013     1     1     1800           1800         0     1945           1951
## # ... with 4,222 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>
```

More complex conditions are possible!

Question: In plain English, what is the code below accomplishing?

Drawing from the `flights` data frame, it filters the data to only conditions where the destination is RDU or GSO within which the arrival delay or the departure delay are less than zero.

```
flights %>%
  filter(dest %in% c("RDU", "GSO"),
         arr_delay < 0 | dep_delay < 0)
```

```
## # A tibble: 6,203 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      800            810       -10      949            955
##  2  2013     1     1      832            840        -8     1006           1030
##  3  2013     1     1      851            851         0     1032           1036
##  4  2013     1     1      917            920        -3     1052           1108
##  5  2013     1     1     1024           1030        -6     1204           1215
##  6  2013     1     1     1127           1129        -2     1303           1309
##  7  2013     1     1     1157           1205        -8     1342           1345
##  8  2013     1     1     1317           1325        -8     1454           1505
##  9  2013     1     1     1449           1450        -1     1651           1640
## 10  2013     1     1     1505           1510        -5     1654           1655
## # ... with 6,193 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>
```

The table of logical operators below will be helpful as you work with filtering.

operator	definition
`<`	is less than?
`<=`	is less than or equal to?
`>`	is greater than?
`>=`	is greater than or equal to?
`==`	is exactly equal to?
`!=`	is not equal to?
`x & y`	is x AND y?
`x | y`	is x OR y?

operator	definition
`is.na(x)`	is x NA?
`!is.na(x)`	is x not NA?
`x %in% y`	is x in y?
`!(x %in% y)`	is x not in y?
`!x`	is not x?

The final operator only makes sense if `x` is logical (TRUE / FALSE).

`mutate()`

`mutate()` creates a new variable.

In the code chunk below, `air_time` is converted to hours, and a new variable `mph` is created, corresponding to the miles per hour of the flight.

```
flights %>%
  mutate(hours = air_time / 60,
         mph = distance / hours) %>%
  select(air_time, distance, hours, mph)
```

```
## # A tibble: 336,776 x 4
##    air_time distance hours   mph
##       <dbl>    <dbl> <dbl> <dbl>
##  1      227     1400 3.78   370.
##  2      227     1416 3.78   374.
##  3      160     1089 2.67   408.
##  4      183     1576 3.05   517.
##  5      116      762 1.93   394.
##  6      150      719 2.5    288.
##  7      158     1065 2.63   404.
##  8       53      229 0.883  259.
##  9      140      944 2.33   405.
## 10      138      733 2.3    319.
## # ... with 336,766 more rows
```

Using `<=` in the mutate creates a new variable `on_time` that is TRUE if the flight is on time and FALSE if it is not.

```
flights %>%
  mutate(on_time = arr_delay <= 0) %>%
  select(arr_delay, on_time)
```

```
## # A tibble: 336,776 x 2
##    arr_delay on_time
##        <dbl> <lgl>
##  1        11 FALSE
##  2        20 FALSE
##  3        33 FALSE
##  4       -18 TRUE
##  5       -25 TRUE
```

```
##  6        12 FALSE
##  7        19 FALSE
##  8       -14 TRUE
##  9        -8 TRUE
## 10         8 FALSE
## # ... with 336,766 more rows
```

Question: What do you think will happen if you take the mean of the `on_time` variable?

It will output an error message, because `on_time` is a logical variable and taking the mean is therefore "mean-ingless", so to speak.

`summarize()`

`summarize` calculates summary statistics. It collapses rows into summary statistics and removes columns irrelevant to the calculation.

Be sure to name your columns!

```
flights %>%
  summarize(mean_dep_delay = mean(dep_delay, na.rm = TRUE))
```

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

Question: The code chunk above should return an NA. What is going wrong? Try to fix it to find the mean departure delay.

There are values in `dep_delay` that are NA, and it cannot use those in a summary because they are not numbers.

```
flights %>%
  summarize(prop_on_time = mean(arr_delay <= 0, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   prop_on_time
##          <dbl>
## 1        0.594
```

`group_by()`

`group_by()` is used for grouped operations. It's very powerful when paired with summarize to calculate summary statistics by group.

Here we find the proportion of flights that are on time for each month of the year.

```
flights %>%
  group_by(month) %>%
  summarize(prop_on_time = mean(arr_delay <= 0, na.rm = TRUE))
```

```
## # A tibble: 12 x 2
##    month prop_on_time
##    <int>        <dbl>
##  1     1        0.578
##  2     2        0.572
##  3     3        0.609
##  4     4        0.546
##  5     5        0.638
##  6     6        0.539
##  7     7        0.530
##  8     8        0.596
##  9     9        0.747
## 10    10        0.657
## 11    11        0.643
## 12    12        0.467
```

We can calculate more than one summary statistic in `summarize()`. In addition to the proportion on time for each month, let's find the maximum delay, median delay, and the count of flights in each month.

Here `n()` calculates the current group size.

```
flights %>%
  group_by(month) %>%
  summarize(prop_on_time = mean(arr_delay <= 0, na.rm = TRUE),
            max_delay = max(arr_delay, na.rm = TRUE),
            median_delay = median(arr_delay, na.rm = TRUE),
            count = n())
```

```
## # A tibble: 12 x 5
##    month prop_on_time max_delay median_delay count
##    <int>        <dbl>     <dbl>        <dbl> <int>
##  1     1        0.578      1272           -3 27004
##  2     2        0.572       834           -3 24951
##  3     3        0.609       915           -6 28834
##  4     4        0.546       931           -2 28330
##  5     5        0.638       875           -8 28796
##  6     6        0.539      1127           -2 28243
##  7     7        0.530       989           -2 29425
##  8     8        0.596       490           -5 29327
##  9     9        0.747      1007          -12 27574
## 10    10        0.657       688           -7 28889
## 11    11        0.643       796           -6 27268
## 12    12        0.467       878            2 28135
```

Finally, let's see what the proportion on time is for EWR, JFK, and LGA.

```
flights %>%
  group_by(origin) %>%
  summarize(on_time = mean(dep_delay <= 0, na.rm = TRUE))
```

```
## # A tibble: 3 x 2
##   origin on_time
##   <chr>    <dbl>
```

```
## 1 EWR      0.552
## 2 JFK      0.616
## 3 LGA      0.668
```

`count()`

`count` counts the unique values of one or more variables. It creates frequency tables.

```
flights %>%
  count(origin)
```

```
## # A tibble: 3 x 2
##   origin      n
##   <chr>   <int>
## 1 EWR    120835
## 2 JFK    111279
## 3 LGA    104662
```

Question: What is the code chunk below doing?

It counts the number of unique values for each origin, and then creates a variable `prop` that calculates what proportion of flights came from each of those origin airports (and then, obviously, prints the output).

```
flights %>%
  count(origin) %>%
  mutate(prop = n / sum(n))
```

```
## # A tibble: 3 x 3
##   origin      n  prop
##   <chr>   <int> <dbl>
## 1 EWR    120835 0.359
## 2 JFK    111279 0.330
## 3 LGA    104662 0.311
```

Practice

(1) Create a new dataset that only contains flights that do not have a missing departure time. Include the columns `year`, `month`, `day`, `dep_time`, `dep_delay`, and `dep_delay_hours` (the departure delay in hours). Note you may need to use `mutate()` to make one or more of these variables.

```
flights_with_departure <- flights %>%
  filter(!is.na(dep_time)) %>%
  mutate(dep_delay_hours = dep_delay / 60) %>%
  select(year, month, day, dep_time, dep_delay, dep_delay_hours)
print(flights_with_departure)
```

```
## # A tibble: 328,521 x 6
##     year month   day dep_time dep_delay dep_delay_hours
##    <int> <int> <int>    <int>     <dbl>           <dbl>
##  1  2013     1     1      517         2          0.0333
##  2  2013     1     1      533         4          0.0667
```

```
##  3  2013     1     1      542         2          0.0333
##  4  2013     1     1      544        -1         -0.0167
##  5  2013     1     1      554        -6         -0.1
##  6  2013     1     1      554        -4         -0.0667
##  7  2013     1     1      555        -5         -0.0833
##  8  2013     1     1      557        -3         -0.05
##  9  2013     1     1      557        -3         -0.05
## 10  2013     1     1      558        -2         -0.0333
## # ... with 328,511 more rows
```

(2) For each airplane (uniquely identified by `tailnum`), use a `group_by()` paired with `summarize()` to find the sample size, mean, and standard deviation of flight distances. Then, pick off the top 5 and bottom 5 airplanes in terms of mean distance traveled per flight.

```
flights %>%
  group_by(tailnum) %>%
  summarize(sample_size = n(),
            mean_distance = mean(distance),
            sd_distance = sd(distance)) %>%
  arrange(mean_distance) %>%
  slice(1:5,(n()-5):n())
```

```
## # A tibble: 11 x 4
##    tailnum sample_size mean_distance sd_distance
##    <chr>         <int>         <dbl>       <dbl>
##  1 N955UW          225          173.        32.9
##  2 N948UW          232          174.        32.7
##  3 N959UW          213          174.        34.3
##  4 N956UW          222          174.        31.4
##  5 N945UW          285          176.        31.2
##  6 N389HA           32         4983          0
##  7 N390HA           20         4983          0
##  8 N391HA           21         4983          0
##  9 N392HA           13         4983          0
## 10 N393HA           10         4983          0
## 11 N395HA            7         4983          0
```

Additional Resources

- https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf
- https://style.tidyverse.org/