Lesson 09 - Data management and aggregation using `dplyr`

Robin Donatello

Last Updated 09-16-2019

Introduction

When working with data you must:

1. Figure out what you want to do.
2. Precisely describe what you want in the form of a computer program.
3. Execute the code.

The dplyr package makes each of these steps as fast and easy as possible by:

1. Elucidating the most common data manipulation operations, so that your options are helpfully constrained when thinking about how to tackle a problem.
2. Providing simple functions that correspond to the most common data manipulation verbs, so that you can easily translate your thoughts into code.
3. Using efficient data storage back ends, so that you spend as little time waiting for the computer as possible.

Student Learning Outcomes

After completing this lesson students will be able to

- Explain the difference between a `data.table` and a `tibble`.
- Build and execute a chain of command to accomplish a data management task
- Create new variables using `mutate`.
- Subset the data using `filter`.
- Create summary statistics using `summarize`
- Sort the data using `arrange`
- Learn how to use code chunk options to disable warning messages.

Preparation

Prior to this lesson students should

- Download the [09_dplyr_notes.Rmd] R markdown file and save into your Math130 folder.
- Ensure that the `dplyr` and `nyflights13` data sets are installed by running the first code chunk.

```
library(dplyr); library(nycflights13)
data(flights)
```

Exploring airline flight data with dplyr.

The `nycflights13` package contains several data sets that can be used to help understand what causes delays. We will be using the `flights` data set which contains information about all flights that departed from NYC (e.g. EWR, JFK and LGA) in 2013.

Tibbles

The `flights` data set, and any data set created with `dplyr`, has a specific data type called a `tibble`. These are not as furry and prolific as their cousins the `tribbles`. `tibbles` behaves for all intents and purposes as a `data.frame`, just gets displayed differently. For example, the `flights` data set contains data on 19 characteristics (variables) from 336,776 flights. There's no way I would want to print out a data set that large. But I'm gonna. . . .

```
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>
```

The output has been trimmed to something more reasonable for our viewing pleasure. This may not seem such a big deal because R Studio already provides some level of truncation for our viewing pleasure.

Basic verbs

The `dplyr` package contains new data manipulation functions, also called verbs. We will look at the following four:

- `filter()`: Returns a subset of the rows.
- `select()`: Returns a subset of the columns.
- `mutate()`: Adds columns from existing data.
- `summarise()`: Reduces each group to a single row by calculating aggregate measures.
- `group_by()`: Groups a data set on a factor variable, such that all functions peformed are then done on each level of the factor.

Filter

`filter()` allows you to select a subset of the rows of a data frame. The first argument is the name of the data frame, and the second and subsequent are filtering expressions evaluated in the context of that data frame. For example, we can select all flights on January 1st with

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

```
## # A tibble: 842 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 832 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>
```

`filter()` works similarly to `subset()` except that you can give it any number of filtering conditions which are joined together with &. You can use other Boolean operators explicitly. Here we select flights in January or February.

```r
filter(flights, month == 1 | month == 2)
```

```
## # A tibble: 51,955 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 51,945 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>
```

Select

Often you work with large data sets with many columns where only a few are actually of interest to you. `select()` allows you to rapidly zoom in on a useful subset using operations that usually only work on numeric variable positions.

```r
select(flights, month, day, year)
```

```
## # A tibble: 336,776 x 3
##    month   day  year
##    <int> <int> <int>
##   1     1     1  2013
##   2     1     1  2013
##   3     1     1  2013
##   4     1     1  2013
##   5     1     1  2013
##   6     1     1  2013
##   7     1     1  2013
##   8     1     1  2013
```

```
## 9      1      1  2013
## 10     1      1  2013
## # ... with 336,766 more rows
```

You can use a colon (:) to select all columns physically located between two variables.

```r
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
```

To exclude specific columns you use the minus sign (-)

```r
select(flights, -carrier)
```

```
## # A tibble: 336,776 x 18
##     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 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>
```

This also works to exclude all columns EXCEPT the ones between two variables.

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

```
## # A tibble: 336,776 x 16
##    dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay
##       <int>          <int>     <dbl>    <int>          <int>     <dbl>
##  1      517            515         2      830            819        11
##  2      533            529         4      850            830        20
##  3      542            540         2      923            850        33
##  4      544            545        -1     1004           1022       -18
##  5      554            600        -6      812            837       -25
##  6      554            558        -4      740            728        12
```

```
## 7      555            600        -5      913            854        19
## 8      557            600        -3      709            723       -14
## 9      557            600        -3      838            846        -8
## 10     558            600        -2      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>
```

Mutate

As well as selecting from the set of existing columns, it's often useful to add new columns that are functions of existing columns. This is the job of mutate()!

Here we create two variables: gain (as arrival delay minus departure delay) and speed (as distance divided by time, converted to hours).

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

```
## # A tibble: 336,776 x 4
##     gain distance air_time speed
##    <dbl>    <dbl>    <dbl> <dbl>
##  1     9     1400      227  370.
##  2    16     1416      227  374.
##  3    31     1089      160  408.
##  4   -17     1576      183  517.
##  5   -19      762      116  394.
##  6    16      719      150  288.
##  7    24     1065      158  404.
##  8   -11      229       53  259.
##  9    -5      944      140  405.
## 10    10      733      138  319.
## # ... with 336,766 more rows
```

One key advantage of mutate is that you can refer to the columns you just created. Mutate flights to create two variables, `gain = arr_delay - dep_delay` and `gain_per_hour = gain / (air_time / 60)`.

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

```
## # A tibble: 336,776 x 21
##     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 14 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>, gain <dbl>, gain_per_hour <dbl>
```

Summarize

The last verb is `summarise()`, which collapses a data frame to a single row. It's not very useful yet. We can create a new variable called delay that is the average departure delay on the entire flights data set.

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

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

Grouped Operations

The above verbs are useful, but they become really powerful when you combine them with the idea of "group by", repeating the operation individually on groups of observations within the dataset. In dplyr, you use the `group_by()` function to describe how to break a dataset down into groups of rows. You can then use the resulting object in exactly the same functions as above; they'll automatically work "by group" when the input is a grouped.

Let's demonstrate how some of these functions work after grouping the flights data set by month. First we'll create a new data set that is grouped by month.

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

- If we want to sort the data, `arrange()` orders first by grouping variables, then by the sorting variable.

```
how_long <- arrange(by_month, distance)
select(how_long, month, distance)
```

```
## # A tibble: 336,776 x 2
## # Groups:   month [12]
##    month distance
##    <int>    <dbl>
##  1     7       17
##  2     1       80
##  3     1       80
##  4     1       80
##  5     1       80
##  6     1       80
##  7     1       80
##  8     1       80
##  9     1       80
## 10     1       80
## # ... with 336,766 more rows
```

- The `summarise()` verb allows you to calculate summary statistics for each group. This is probably the most common function that is used in conjunction with `group_by`. For example, the average distance flown per month.

```
summarise(by_month, avg_airtime = mean(distance, na.rm=TRUE))
```

```
## # A tibble: 12 x 2
##    month avg_airtime
##    <int>       <dbl>
```

```
## 1      1        1007.
## 2      2        1001.
## 3      3        1012.
## 4      4        1039.
## 5      5        1041.
## 6      6        1057.
## 7      7        1059.
## 8      8        1062.
## 9      9        1041.
## 10    10        1039.
## 11    11        1050.
## 12    12        1065.
```

Or simply the total number of flights per month.

```
summarize(by_month, count=n())
```

```
## # A tibble: 12 x 2
##    month count
##    <int> <int>
##  1     1 27004
##  2     2 24951
##  3     3 28834
##  4     4 28330
##  5     5 28796
##  6     6 28243
##  7     7 29425
##  8     8 29327
##  9     9 27574
## 10    10 28889
## 11    11 27268
## 12    12 28135
```

Chaining Operations

Consider the following group of operations that take the data set `flights`, and produce a final data set (`a4`) that contains only the flights where the daily average delay is greater than a half hour.

```
a1 <- group_by(flights, year, month, day)
a2 <- select(a1, arr_delay, dep_delay)
```

```
## Adding missing grouping variables: `year`, `month`, `day`
```

```
a3 <- summarise(a2,
                arr = mean(arr_delay, na.rm = TRUE),
                dep = mean(dep_delay, na.rm = TRUE))
a4 <- filter(a3, arr > 30 | dep > 30)
head(a4)
```

```
## # A tibble: 6 x 5
## # Groups:   year, month [3]
##    year month   day   arr   dep
##   <int> <int> <int> <dbl> <dbl>
## 1  2013     1    16  34.2  24.6
## 2  2013     1    31  32.6  28.7
## 3  2013     2    11  36.3  39.1
```

```
## 4  2013     2    27  31.3  37.8
## 5  2013     3     8  85.9  83.5
## 6  2013     3    18  41.3  30.1
```

It does the trick, but what if you don't want to save all the intermediate results (`a1 - a3`)? Well these verbs are `function`, so they can be wrapped inside other functions to create a nesting type structure.

```
filter(
  summarise(
    select(
      group_by(flights, year, month, day),
      arr_delay, dep_delay
    ),
    arr = mean(arr_delay, na.rm = TRUE),
    dep = mean(dep_delay, na.rm = TRUE)
  ),
  arr > 30 | dep > 30
)
```

Woah, that is HARD to read! This is difficult to read because the order of the operations is from inside to out, and the arguments are a long way away from the function. To get around this problem, dplyr provides the `%>%` operator. `x %>% f(y)` turns into `f(x, y)` so you can use it to rewrite multiple operations so you can read from left-to-right, top-to-bottom:

```
flights %>%
  group_by(year, month, day) %>%
  select(arr_delay, dep_delay) %>%
  summarise(
    arr = mean(arr_delay, na.rm = TRUE),
    dep = mean(dep_delay, na.rm = TRUE)
  ) %>%
  filter(arr > 30 | dep > 30)
```

```
## Adding missing grouping variables: `year`, `month`, `day`
```

```
## # A tibble: 49 x 5
## # Groups:   year, month [11]
##     year month   day   arr   dep
##    <int> <int> <int> <dbl> <dbl>
##  1  2013     1    16  34.2  24.6
##  2  2013     1    31  32.6  28.7
##  3  2013     2    11  36.3  39.1
##  4  2013     2    27  31.3  37.8
##  5  2013     3     8  85.9  83.5
##  6  2013     3    18  41.3  30.1
##  7  2013     4    10  38.4  33.0
##  8  2013     4    12  36.0  34.8
##  9  2013     4    18  36.0  34.9
## 10  2013     4    19  47.9  46.1
## # ... with 39 more rows
```

Another way you can read this is by thinking "and then" when you see the `%>%` operator. So the above code takes the data set flights
.. and then groups by day
.. and then selects the delay variables
.. and then calculates the means
.. and then filters on a delay over half hour.

The same 4 steps that resulted in the `a4` data set, but without all the intermediate data saved! This can be **very important** when dealing with Big Data. `R` stores all data in memory, so if your little computer only has 2G of RAM and you're working with a data set that is 500M in size, your computers memory will be used up fast. `a1` takes 500M, `a2` another 500M, by now your computer is getting slow. Make another copy at `a3` and it gets worse, `a4` now likely won't even be able to be created because you'll be out of memory.

Additional Resources

The ability to manipulate, arrange and clean up data is an extremely important skill to have. It is advised that you review at least one other tutorial method for using dplyr. Remember, it is all about practice. The more you use it the easier it will become!

- R Studio's Data Wrangling Cheat Sheet
- Data Camp has a dplyr lesson
- Dplyr vignette
- Hands-on dplyr tutorial for faster data manipulation in R You Tube video by Data School
- UBC STAT 545 (this class is awesome!) http://stat545.com