dplyr Tutorial
Terui Lab

Justin Pomeranz

April 13, 2021

This tutorial is aimed at giving you an introduction to the popular `dplyr` package. `dplyr` provides a few simple verbs that allow you to quickly and easily select and manipulate your data, and create an interactive environment for data exploration. This tutorial is based on the introduction and two-table vignettes for the dplyr package:

```
vignette("dplyr")
vignette("two-table", "dplyr")
```

As well as the book "*R for Data Science* by Hadley Wickham & Garrett Grolemund (O'reilly). Copyright 2017. 978-1-491-91039-9." Available free online at http://r4ds.had.co.nz/

At the end, I will make a brief plug for the `tidyr` package, which is designed to make messy datasets into "tidy" data sets.

Both `dplyr` and `tidyr` are part of the suite of packages known as `tidyverse`. The packages within the tidyverse were designed with a similar data and programming philosophy, and work together fluidly. Hadley Wickham is the lead developer of the tidyverse, and his book above takes a tidyverse-centric view, and is a fantastic resource if you want to learn more.

Part 1: dplyr

Data manipulation is one of the primary tasks that scientists undertake. the `dplyr` package makes it easy to select the data you want, organize it in a useful way, and calculate useful new variables.

There is an extremely helpful Rstudio data wrangling cheat sheet available at https://github.com/rstudio/cheatsheets/raw/master/data-transformation.pdf. You can also download this cheat sheet directly through Rstudio by clicking *Help > Cheatsheets > Data transformation with dplyr*. I almost always have this cheat sheet open while I'm conducting any analyses. It takes a little effort to learn how to "read" the cheat sheet, but it's well worth the effort.

First, make sure that the `tidyverse` package is installed and loaded. This package contains `dplyr` which we will be working with today, as well as many other usefule packages that are all designed to work together such as `tidyr`, `ggplot2`, etc.

If you need to install `tidyverse` or `nycflights13`, run the first line in the chunk without the "#" sign: `install.packages("nycflights13")`

As a reminder, you only have to run the `install.packages(...)` line once. Running this line downloads the package onto the maching that you are currently using. However, you do need to run the `library(tidyverse)` line at the beginning of every session. This lines "loads" the package and makes the commands available to you.

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

For this tutorial, we will be using the `nycflights13` data. Most likely, you will need to install it before loading it into your session.

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

Data: nycflights13

In order to demonstrate the basic data manipulation verbs of dplyr, we will be using the `nycflights13` database. This data object contains three dataframes: `flights`, `airlines`, `airports`. The `flights` dataset contains all 336776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transportation Statistics, and is documented in `?nycflights13::flights`

It is worth noting that this data is already in the "tidydata" format. Unfortunately, going into this in detail is beyond the scope of this tutorial, but I recommend you look at the "tidydata" chapter in R for Data Science: https://r4ds.had.co.nz/tidy-data.html. For tools to turn a non-tidy data set into tidy data, look at the `tidyr` package within `tidyverse`. You can browse the vignettes available by running the following code: `browseVignettes("tidyr")`

Finally, see my plug for tidydata at the end of this document.

```r
# make sure the nycflights13 library is loaded
library(nycflights13)
#> Warning: package 'nycflights13' was built under R version 4.0.4
dim(flights)
#> [1] 336776     19
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
#> # ... with 2 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>
```

You may notice that the `head(flights)` output is different from many other data frames that you may have worked with before. It only shows the rows and columns that easily fit into your window. That's because the `flights` object is a tibble:

```r
class(flights)
#> [1] "tbl_df"     "tbl"        "data.frame"
```

Tibbles `"tbl_df"` act just as a data frame would, but they print only the information that can fit into your window. To see the full data frame in the Rstudio viewer use:

```
View(flights)
```

Or as a normal data frame use:

```
head(as.data.frame(flights))
```

To convert a data frame to a tibble, use `as_tibble()`.

As a general rule, I almost always use the `names()` function when I'm looking at a new data set. This helps me get an idea of the data structure, and if the variable names are meaningful, can help me understand what types of questions I can ask, and how I may be able to answer them.

```
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"
```

These variables should be relatively self-explanatory, but you can run `?flights` for more information.

Single table verbs

dplyr aims to provide a function for each basic verb of data manipulation:

- Pick observations based on their values: `filter()`
- pick observations based on their row number `slice()`
- Reorder the rows: `arrange()`
- Pick variables based on their names: `select()`
- Add new variables using functions on existing variables: `mutate()`
- Reduce many observations to a single summary: `summarise()`

All of these verbs can also be used with `group_by()`, which allows you to apply a verb group-by-group instead of on the whole data set, but more on this later.

All verbs work similarly and have similar syntaxes:

1. The first argument is a data frame.
2. Following arguments tell what to do with the data frame using un-quoted variable names.
3. Result is new data frame.

There are other verbs in the `dplyr` package, along with many useful "helper" functions. Browse the vignettes for dplyr: `browseVignettes(package = "dplyr")`, or check the Rstudio data wrangling cheat sheet https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

Filter rows with `filter()`

`filter()` allows you to select a subset of rows in a data frame. The first argument is the name of the data frame. The second and subsequent arguments are the expressions that filter the data frame:

For example, we can select all flights on July 4th with:

```r
filter(flights, month == 7, day == 4)
#> # A tibble: 737 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     7     4       11           2359        12      400            340
#> 2  2013     7     4       59           2359        60      501            350
#> 3  2013     7     4      454            500        -6      635            640
#> 4  2013     7     4      535            536        -1      802            806
#> # ... with 733 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>
```

This is equivalent to the standard subsetting notation in base R:

```r
flights[flights$month == 7 & flights$day == 4, ]
```

`filter()` works similarly to `subset()` but is slightly more flexible, where an `&` can be replaced with a `,`:

```r
# flights between March and September
# ','
filter(flights, month >= 3, month <= 9)
# '&'
filter(flights, month >= 3 & month <= 9)
```

You can also use other boolean operators, singly and in combination:

```r
# flights in January OR February
filter(flights, month == 1 | month == 2)
# flights in January OR February AND that flew > 1000 OR < 500 miles
filter(flights, month == 1 | month == 2 & distance > 1000 | distance < 500)
```

Sometimes you want to see all of the data that do *not* meet a criteria. For example, maybe you don't want to know what flights left in July. You could use a combination of arguments within `filter()` such as: `month < 7 | month > 7`, or you could use the logical negation command `!=`. This command can be read as "not equal to".

```r
# month less than "OR" greater than 7
not_july_1 <- filter(flights, month < 7 | month >7)
# month "not equal to" 7
not_july_2 <- filter(flights, month != 7)
# are these to objects the same?
identical(not_july_1, not_july_2)
#> [1] TRUE
# yes
```

Both lines of code above give the same result, but the second line is simpler to write, and you don't have to remember if you're supposed to use an "AND" (`&`) or an "OR" (`|`) command.

It is also often useful to see all of the values that occur within a variable column. For example, what are all of the airport origins in this data set?

Here, I will use the base R function `unique()` to see all of the values in the variable "origin". To select just that variable, I will use the `$` subsetting function.

```
unique(flights$origin)
#> [1] "EWR" "LGA" "JFK"
```

This data contains all of the flights leaving New York City, and the codes above refer to Newark, LaGuardia, and JFK, respectively.

exercises

1. Filter out all of the flights that departed in December.
2. What are all of the carriers in this data set? Choose one and filter out all of the flights for that carrier.
3. Filter out all of the flights that did *not* depart on the 13th day of any month.

Arrange rows with `arrange()`

`arrange()` has a similar syntax as `filter()` but instead of filtering or selecting rows, it reorders them. It takes a data frame, and column name(s) to order by as arguments. The data will be arranged by the first column name provided, with ties being broken by subsequent columns:

```
# arrange by sched_dep_time
arrange(flights, sched_dep_time)
#> # 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     7    27       NA            106        NA       NA            245
#> 2  2013     1     2      458            500        -2      703            650
#> 3  2013     1     3      458            500        -2      650            650
#> 4  2013     1     4      456            500        -4      631            650
#> # ... with 336,772 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>

# arrange by sched_dep_time, then dep_delay
arrange(flights, sched_dep_time, 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     7    27       NA            106        NA       NA            245
#> 2  2013     5     8      445            500       -15      620            640
#> 3  2013     5     5      446            500       -14      636            640
#> 4  2013     9     4      446            500       -14      618            648
#> # ... with 336,772 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 default order is from smallest to largest, numeric to character, and a to z. To reverse this order, use `desc()` to order a column in descending order:

```
arrange(flights, desc(sched_dep_time))
#> # 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     2353           2359        -6      425            445
#> 2  2013     1     1     2353           2359        -6      418            442
#> 3  2013     1     1     2356           2359        -3      425            437
#> 4  2013     1     2       42           2359        43      518            442
#> # ... with 336,772 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>
arrange(flights, desc(sched_dep_time), 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    10     2     2341           2359       -18      324            350
#> 2  2013     9    23     2342           2359       -17      331            350
#> 3  2013    10    22     2343           2359       -16      347            350
#> 4  2013     3     4     2343           2359       -16      418            438
#> # ... with 336,772 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>
```

`arrange()` is a straightforward wrapper around `order()` that requires less typing. The previous code is equivalent to:

```
flights[order(flights$sched_dep_time), ]
flights[order(flights$sched_dep_time, decreasing = TRUE, flights$dep_delay),]
```

exercises

1. Arrange the flights in order of flight distance.
2. Arrange the flights in descending order of month (12 –> 1), and increasing order of day (1 –> 31).

Select columns with `select()`

Data sets generally contain numerous columns, but oftentimes you are only interested in a few for a given analysis. `select()` allows you to focus on a useful subset of your data while dropping un-needed columns:

```
# Select columns by name
select(flights, month, day, carrier)
#> # A tibble: 336,776 x 3
#>   month   day carrier
#>   <int> <int> <chr>
#> 1     1     1 UA
#> 2     1     1 UA
#> 3     1     1 AA
#> 4     1     1 B6
#> # ... with 336,772 more rows

# Select all columns between day and arr_time (inclusive)
select(flights, day:arr_time)
#> # A tibble: 336,776 x 5
#>     day dep_time sched_dep_time dep_delay arr_time
#>   <int>    <int>          <int>     <dbl>    <int>
```

```
#> 1     1      517            515         2      830
#> 2     1      533            529         4      850
#> 3     1      542            540         2      923
#> 4     1      544            545        -1     1004
#> # ... with 336,772 more rows

# Select all columns except those from year to day (inclusive)
select(flights, -(year:day))
#> # A tibble: 336,776 x 16
#>   dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier
#>      <int>          <int>     <dbl>    <int>          <int>     <dbl> <chr>
#> 1      517            515         2      830            819        11 UA
#> 2      533            529         4      850            830        20 UA
#> 3      542            540         2      923            850        33 AA
#> 4      544            545        -1     1004           1022       -18 B6
#> # ... with 336,772 more rows, and 9 more variables: flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

This function works similarly to the `select` argument in `base::subset()`. The dplyr philosophy is to have small functions that do one thing well, so it's its own function.

There are a number of helper functions you can use within `select()`, like `starts_with()`, `ends_with()`, `matches()` and `contains()`. These let you quickly match larger blocks of variables that meet some criterion. See `?select` for more details.

The order you write the arguments in `select()` are retained in the output. This can be helpful if you want to quickly compare the values in two columns which do not normally appear next to each other.

```
select(flights, air_time, dep_time, arr_time)
#> # A tibble: 336,776 x 3
#>   air_time dep_time arr_time
#>      <dbl>    <int>    <int>
#> 1      227      517      830
#> 2      227      533      850
#> 3      160      542      923
#> 4      183      544     1004
#> # ... with 336,772 more rows
```

exercises

1. Select the hour and minute columns.
2. Select the dest, carrier, and arr_time variables, in that order.

Add new columns with `mutate()`

Data analysis often requires the creation of new variable columns based on values within your dataset. The `mutate()` function allows you to do this:

```
# make two new columns, "gain" and "speed"
d2 <- mutate(flights,
  gain = arr_delay - dep_delay)
```

```r
# select new columns in d2, and columns used to calculate them
select(d2, gain, arr_delay, dep_delay)
#> # A tibble: 336,776 x 3
#>    gain arr_delay dep_delay
#>   <dbl>     <dbl>     <dbl>
#> 1     9        11         2
#> 2    16        20         4
#> 3    31        33         2
#> 4   -17       -18        -1
#> # ... with 336,772 more rows
```

`dplyr::mutate()` works similarly to `base::transform()`. The key difference between `mutate()` and `transform()` is that mutate allows you to refer to columns that you've just created:

```r
d3 <- mutate(flights,
  # calculate "gain" column
  gain = arr_delay - dep_delay,
  # use "gain" column in next calculation
  gain_per_hour = gain / (air_time / 60)
)
select(d3, gain, gain_per_hour)
#> # A tibble: 336,776 x 2
#>    gain gain_per_hour
#>   <dbl>         <dbl>
#> 1     9          2.38
#> 2    16          4.23
#> 3    31         11.6
#> 4   -17         -5.57
#> # ... with 336,772 more rows
```

Whereas `transform` will throw an error:

```r
transform(flights,
  gain = arr_delay - dep_delay,
  gain_per_hour = gain / (air_time / 60)
)
#> Error in eval(substitute(list(...)), `_data`, parent.frame()) : object 'delay' not found
```

exercises

1. Calculate the average speed per flight, in units of miles per minute. Reminder, speed = distance / time, and in this data, air_time is in minutes.
2. Do the same as above, but this time report the speed in miles per hour.
3. Calculate the departure delay (i.e. `calc = dep_time - sched_dep_time`) and see if it is equal to the `dep_delay` column provided. Hint, to get a `TRUE/FASLE` value, use `calc_equal = calc == dep_delay`.

Summarise values with `summarise()`

The last verb is `summarise()`. This collapses a dataframe to a single value, based on a function:

```r
# mean departure delay
summarise(flights,
  delay = mean(dep_delay, na.rm = TRUE))
#> # A tibble: 1 x 1
#>   delay
#>   <dbl>
#> 1  12.6
# shortest flight distance
summarise(flights, min.dist = min(distance))
#> # A tibble: 1 x 1
#>   min.dist
#>      <dbl>
#> 1       17
```

Notice the argument `na.rm = TRUE` within the summarize function. If you have missing or `NA` values within your data, it will cause the summary functions to return `NA`.

Additionally, summarize is optimized to work with functions that return a single value. For example, `range()` returns the minimum and maximum value of a set of numbers:

```r
range(c(1, 2, 3, NA, 5, 6, 7), na.rm = TRUE)
#> [1] 1 7
```

When used in combination with `summarise()`, two values are returned, but they are not labeled

```r
summarise(flights, delay.range = range(dep_delay, na.rm = TRUE))
#> # A tibble: 2 x 1
#>   delay.range
#>         <dbl>
#> 1         -43
#> 2        1301
```

If this is the only thing you are calling, it may be easy to tell which is which, but if we have multiple summary arguments, it can be less obvious:

```r
summarise(flights,
          delay.range = range(dep_delay, na.rm = TRUE),
          mean.delay = mean(dep_delay, na.rm = TRUE))
#> # A tibble: 2 x 2
#>   delay.range mean.delay
#>         <dbl>      <dbl>
#> 1         -43       12.6
#> 2        1301       12.6
```

However, we can get around this by calling the `min()` and `max()` functions separately within summarise:

```r
summarise(flights,
          min.delay = min(dep_delay, na.rm = TRUE),
          max.delay = max(dep_delay, na.rm = TRUE),
          mean.delay = mean(dep_delay, na.rm = TRUE))
#> # A tibble: 1 x 3
#>   min.delay max.delay mean.delay
#>       <dbl>     <dbl>      <dbl>
#> 1       -43      1301       12.6
```

Exercises

1. What is the mean air_time in this data?
2. What is the minimum and maximum flight distance?
3. What is the mean and standard deviation of arr_delay?

Grouped operations

These verbs are useful on their own, but when used in conjuction with the `group_by()` function, the awesomeness of dplyr starts to shine through. It organizes a dataset into specified groups of rows. Verbs are then applied group-by-group within the dataset. Conveniently, this is accomplished using the exact same syntax as above.

Here, we will group the data by "carrier" and then summarize the mean departure delay:

```
carrier_group <- group_by(flights, carrier)
summarize(carrier_group,
          mean_delay = mean(dep_delay, na.rm = TRUE))
#> # A tibble: 16 x 2
#>   carrier mean_delay
#> * <chr>        <dbl>
#> 1 9E           16.7
#> 2 AA            8.59
#> 3 AS            5.80
#> 4 B6           13.0
#> # ... with 12 more rows
```

The output now contains 1 row per carrier.

Just to emphasize a previous point, including functions that return 2 values in this context can be difficult to interpret:

```
summarize(carrier_group,
          range_delay = range(dep_delay, na.rm = TRUE),
          mean_delay = mean(dep_delay, na.rm = TRUE))
#> # A tibble: 32 x 3
#> # Groups:   carrier [16]
#>   carrier range_delay mean_delay
#>   <chr>         <dbl>      <dbl>
#> 1 9E              -24      16.7
#> 2 9E              747      16.7
#> 3 AA              -24       8.59
#> 4 AA             1014       8.59
#> # ... with 28 more rows
```

To illustrate the power of `dplyr`, here is a more complex data summarization.

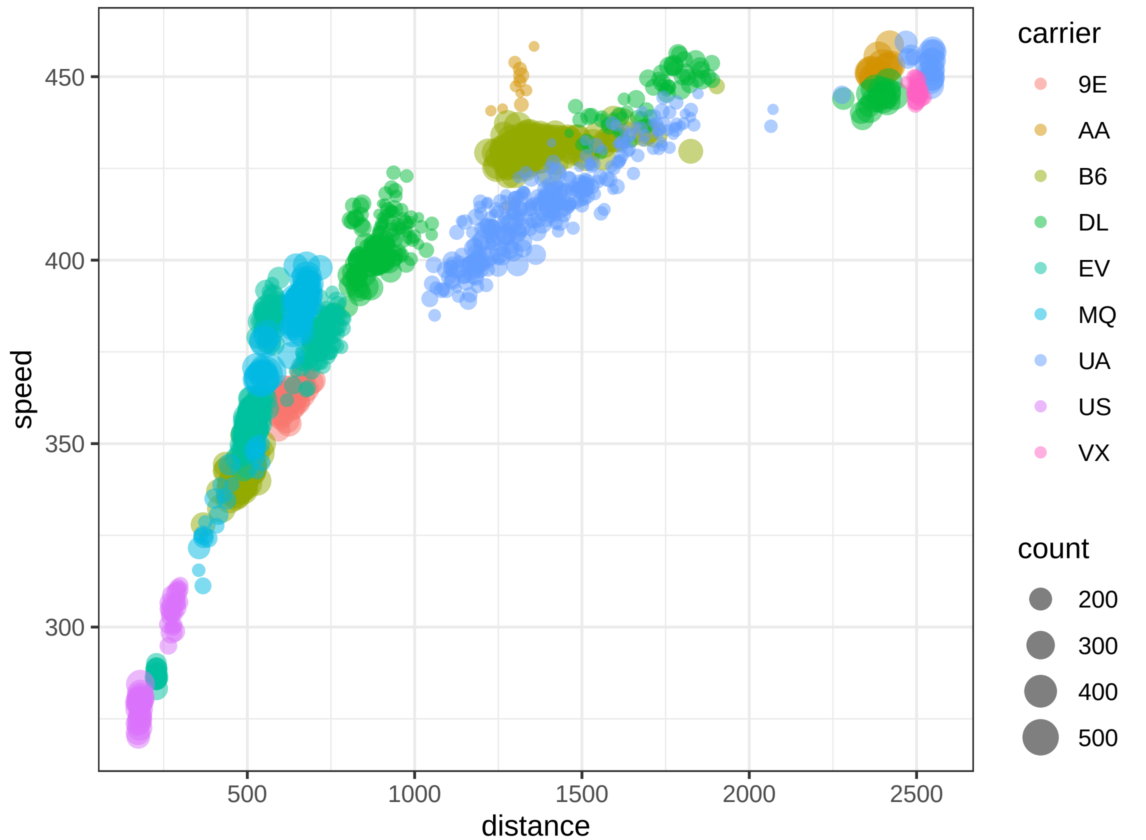
1. First, we **filter** out the flights that were canceled. In this data, canceled flights have `dep_time = NA`. The command we will use is: `!is.na(dep_time)`. You can read this as "all of the flights that *do not* have an `NA` value in the `dep_time` variable.
2. The entire data set is **grouped** into carrier and individual planes.
3. A **new column** for air speed is added (using `mutate()`).

4. It is then **summarized** by counting the number of flights (`count = n()`) and computing the average speed (`mean.speed = mean(speed, na.rm = TRUE)`) and arrival delay (`delay = mean(arr_dellay, na.rm = TRUE)`).
5. We will filter out results to make the display (below) nicer.
6. The results are then displayed using ggplot (after filtering to only include)

```
#filter out canceled flights
not_cancelled <- filter(flights, !is.na(dep_time))
by_carrier <- group_by(not_cancelled, carrier, tailnum)
air_speed <- mutate(by_carrier, speed = distance / air_time * 60)
speed_summary <- summarise(air_speed,
  count = n(),
  distance = mean(distance, na.rm = TRUE),
  speed = mean(speed, na.rm = TRUE))

speed_summary <- filter(speed_summary, count > 100)

ggplot(speed_summary, aes(distance, speed)) +
  geom_point(aes(color = carrier, size = count), alpha = 0.5) +
  theme_bw()
```



Not surprisingly, flights that travel a shorter distance do not reach their maximal cruising speed. The difference in speed between different carriers (colored dots) may be reflicitive of different plane models or operation procedures.

Chaining

The dplyr function calls don't have any side-effects (unlike some base functions), making it easy to explore your data in an interactive way. However, one disadvantage of this is it doesn't lead to very succint code, particularly if you want to perform many operations at once. You can do it step-by-step, saving a new object each time:

```
# calculate flights either whose arrival or departure were delayed > 30 minutes
a1 <- group_by(flights, year, month, day)
a2 <- select(a1, arr_delay, dep_delay)
a3 <- summarise(a2,
  arr = mean(arr_delay, na.rm = TRUE),
  dep = mean(dep_delay, na.rm = TRUE))
a4 <- filter(a3, arr > 30 | dep > 30)
a4
```

However this can lead to many problems. Giving objects appropriate names can be difficult (e.g. object names in ggplot example above). Or naming them chronologically (as above), it can be difficult to remember which object is which (was the summarized object a2 or a3?...). Especially if you want to do the same thing over and over on different data sets or subsets of observations.

Alternatively, if you don't want to save the intermediate results, you need to wrap the function calls inside each other:

```
# calculate flights either whose arrival or departure were delayed > 30 minutes
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
)
```

This is difficult to write and to read, and not intuitive because the order of the operations is from inside to out. The arguments are a long way away from the function. For example the arguments for the first `filter()` function above are the last thing within the full call (`arr > 30 | dep >30)`), approximately 8 lines lower down.

In order to get around this and write intuitive, easy to read code, we need to chain the function calls together. But before we do this, we need to introduce the pipe operator `%>%`

The pipe operator: `%>%`

The pipe operator is from the `magrittr` package, but is included in the `dplyr` package, so no need to load this library separately.

The pipe operator allows you to write this function: `f(x,y)` as `x %>% f(y)`.

I think that this is a bit confusing to think about, but when you see some examples the power and ease of use becomes obvious.

```r
# a silly example
x <- seq(10)
max(x)
#> [1] 10
# is the same as
x %>% max()
#> [1] 10

mean(x)
#> [1] 5.5
x %>% mean()
#> [1] 5.5
```

You can see that both syntaxes give the same result. Now, let's go back our example looking at arrival and departure delays > 30 minutes

```r
# original method, saving new object at each step
a1 <- group_by(flights, year, month, day)
a2 <- select(a1, arr_delay, dep_delay)
a3 <- summarise(a2,
  arr = mean(arr_delay, na.rm = TRUE),
  dep = mean(dep_delay, na.rm = TRUE))
a4 <- filter(a3, arr > 30 | dep > 30)

# now using the %>% operator
a5 <- 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)

identical(a4, a5)
#> TRUE
```

It is helpful to say "then" when you see a `%>%` operator. e.g. take object "flights", *then* group by year, month and day, *then* select variables arr_delay and dep_delay *then* …

The pipe operator in combination with `dplyr` allows you to quickly examine your data and explore interesting results. One of my favorite aspects of this is you can answer interesting questions almost immediately.

"I wonder what the smallest arrival delay on Christmas day was, and how far the plane travelled?"

```r
flights %>%
  filter(month == 12, day == 25) %>%
  select(dep_delay, distance) %>%
  arrange(dep_delay)
#> # A tibble: 719 x 2
#>   dep_delay distance
#>       <dbl>    <dbl>
#> 1       -23      762
#> 2       -16     1389
#> 3       -15     2402
```

```
#> 4        -15      1608
#> # ... with 715 more rows
# note that negative values indicate an early arrival
```

If you want to also know the destination:

```
flights %>%
  filter(month == 12, day == 25) %>%
  select(dep_delay, distance, dest) %>%
  arrange(dep_delay)
#> # A tibble: 719 x 3
#>   dep_delay distance dest
#>       <dbl>    <dbl> <chr>
#> 1       -23      762 ATL
#> 2       -16     1389 DFW
#> 3       -15     2402 SEA
#> 4       -15     1608 SJU
#> # ... with 715 more rows
```

What carrier has the longest flights on average?

```
flights %>%
  group_by(carrier) %>%
  summarise(mean.dist = mean(distance),
              count = n()) %>%
  arrange(desc(mean.dist))
#> # A tibble: 16 x 3
#>   carrier mean.dist count
#>   <chr>       <dbl> <int>
#> 1 HA          4983    342
#> 2 VX          2499.  5162
#> 3 AS          2402    714
#> 4 F9          1620    685
#> # ... with 12 more rows
```

Formatting with pipes

A couple of pointers.

- always enter a new line after a pipe `%>%`
- if you have many arguments within a function in a pipe, enter a new line after each comma
- if you are making a new variable with `mutate`, or summarizing variables with `summarize`, give them a meaningful name
- try and limit the number of pipes in a single call. 5-6 is OK, but 10 or more should be avoided. If you need that many, save an intermediate object and then pipe that.

bad:

```
# don't do these:
# no new line after each pipe
flights %>% group_by(year, month, day) %>% filter(carrier == "FL" |
carrier == "AA"| carrier == "UA") %>% 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)
# no new line after each summarize argument, and meaningless variable names
flights %>% summarize(x1 = mean(dep_delay, na.rm = TRUE), x2 =
min(dep_delay, na.rm = TRUE), x3 = max(dep_delay, na.rm = TRUE),
x4 = n())
```

better

```
# new line after each pipe
flights %>%
  group_by(year, month, day) %>%
  select(arr_delay, dep_delay) %>%
  summarise(arr = mean(arr_delay, na.rm = TRUE), # new line here
            dep = mean(dep_delay, na.rm = TRUE)) %>%
  filter(arr > 30 | dep > 30)

# new line after each summarize argument, and variable names
flights %>%
  summarize(mean_delay = mean(dep_delay, na.rm = TRUE),
            min_delay = min(dep_delay, na.rm = TRUE),
            max_delay = max(dep_delay, na.rm = TRUE),
            n_samples = n())
```

Pipe with other functions

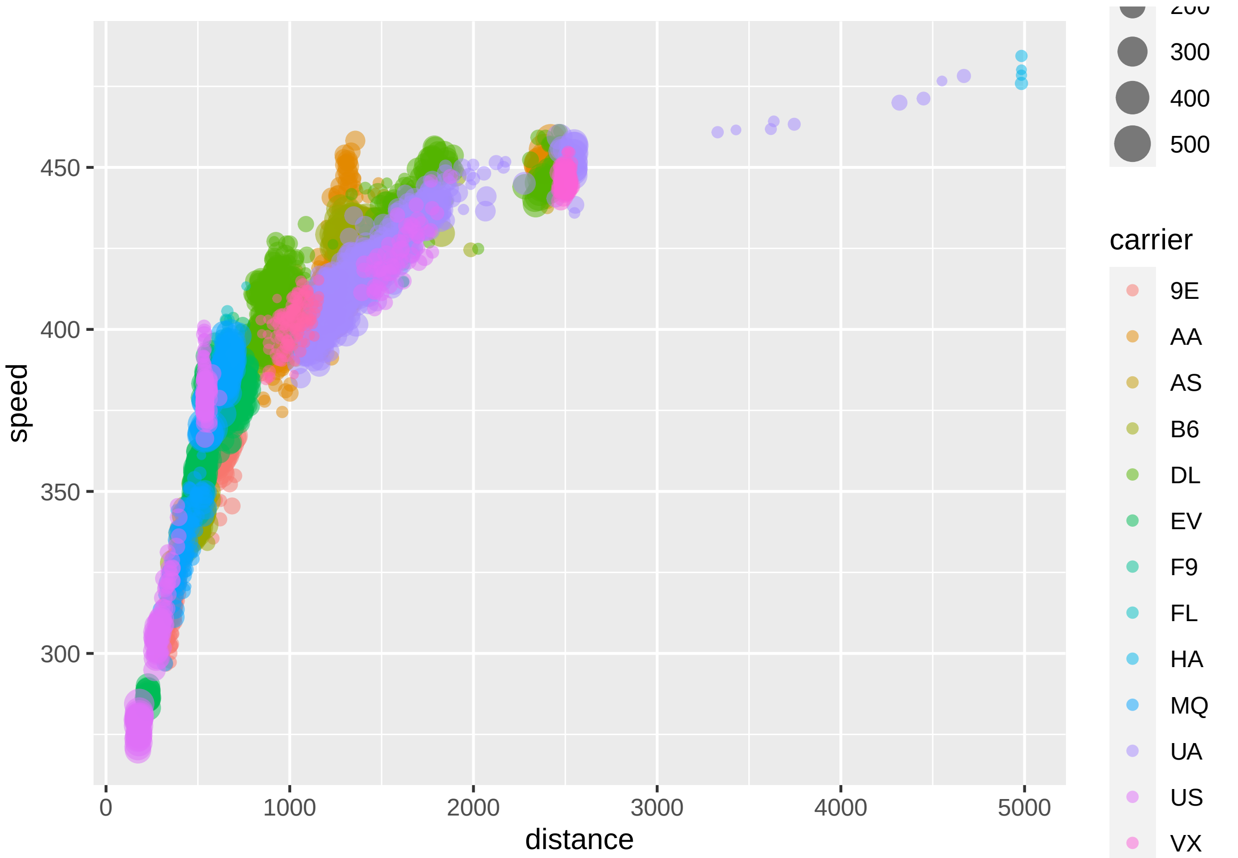
You can also pipe objects into other functions outside the dplyr package:

```
flights %>%
  filter(carrier == "HA") %>%
  dim()
#> [1] 342  19

# pipe results into write.csv()
flights %>%
  group_by(carrier, dest) %>%
  select(dest, carrier, dep_delay) %>%
  summarize(mean.delay = mean(dep_delay,na.rm = T),
            count = n()) %>%
  arrange(carrier, desc(mean.delay)) %>%
  write.csv("mean_delay_by_carrier_and_dest.csv", row.names = F)

# revisit our ggplot example above
# pipe results into ggplot()
flights %>%
  filter(!is.na(dep_time)) %>%
```

```
group_by(carrier, tailnum) %>%
mutate(speed = distance / air_time * 60) %>%
summarize(
  count = n(),
  distance = mean(distance, na.rm = T),
  speed = mean(speed, na.rm = T)) %>%
filter(count > 30) %>%
ggplot(aes(distance,speed)) +
geom_point(aes(color = carrier, size = count), alpha = 0.5)
```



Quick note, if you don't know where to find the .csv file you created in the example above, you can check your working directory with `getwd()`. This is also a good time to mention that if you're not already using projects in Rstudio, I highly recommend it. See Chapter 6 "Workflow: Projects" in *R for Data Science* for more information. http://r4ds.had.co.nz/workflow-projects.html/

Exercises

For all of the following exercises, try and write code to answer the question using step-by-step object creation, as well as in a single command by chaining functions together with the pipe `%>%` operator.

1) What day of the year did the longest departure delay occur on? How long was the delay?
2) What flight arrived the earliest on your birthday? Where was the flight coming from and going to?
3) What is the longest flight duration in this data set? What is the longest flight duration for each carrier?

4) What questions do you want to answer? Think about what you can ask with this data set, and then go step-by-step to answer the question.

Two table verbs

Oftentimes the data you need is not in only one data set. For example, the above results showing which destinations have the largest average departure delay may be interesting, but who has all the carrier and airport abbreviations memorized?

The above result would be more useful to lay persons if it had more specific information. For this we can use the two table verbs of `dplyr`.

The `airports` and `airlines` objects within the `nycflights13` data has some useful information.

```
head(airlines)
#> # A tibble: 6 x 2
#>   carrier name
#>   <chr>   <chr>
#> 1 9E      Endeavor Air Inc.
#> 2 AA      American Airlines Inc.
#> 3 AS      Alaska Airlines Inc.
#> 4 B6      JetBlue Airways
#> # ... with 2 more rows
head(airports)
#> # A tibble: 6 x 8
#>   faa   name                           lat   lon   alt    tz dst   tzone
#>   <chr> <chr>                        <dbl> <dbl> <dbl> <dbl> <chr> <chr>
#> 1 04G   Lansdowne Airport             41.1 -80.6  1044    -5 A     America/New_Y~
#> 2 06A   Moton Field Municipal Airp~  32.5 -85.7   264    -6 A     America/Chica~
#> 3 06C   Schaumburg Regional           42.0 -88.1   801    -6 A     America/Chica~
#> 4 06N   Randall Airport               41.4 -74.4   523    -5 A     America/New_Y~
#> # ... with 2 more rows
```

Exercise

1. What columns can you use to join all of these data objects? Hint: call the `names()` function to each of the three objects: `flights`, `airports`, and `airlines`.
2. **Advanced** use the `dplyr::intersect()` call to see which common variables occur in two data objects. `intersect( names(object1), names(object2))`

There are a number of joining commands within `dplyr`, primarily differing in how they treat missing data.

- Join rows from `b` matching to `a`, keeping all rows from `a`: `left_join(a, b, by = "x1")`
- Join rows from `a` matching to `b`, keeping all rows from `b`: `right_join(a, b, by = "x1")`
- Join data, only keeping rows in `a` and `b`: `inner_join(a, b, by = "x1")`
- Join data, keep all rows in both data sets: `full_join(a, b, by = "x1")`

The Rstudio data wrangling cheat sheet has very useful diagrams explaining this visually. Second page of this cheat sheet: https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

If both data objects have a column with the same name, only one argument in the `by = ...` is needed. However, you can join two objects with different column names by calling them specifically: `left_join(object1, object2, by = c("column_in_1" = "column_in_2"))`

All of these can be accomplished with using arguments within the `base::merge()` function, but as noted previously, one of the core features of dplyr is to have "small" functions that do one thing well.

The most commonly used joining verb is `left_join` because it retains all rows in the `a` data set, whether they have a match in `b` or not. In the following, we will combine the `flights` and `airlines` objects. They both have the variable `"carrier"`, so only one argument is needed.

```
flights2 <- left_join(flights, airlines, by = "carrier") %>%
  # change column name to be more descriptive
  rename(airline = name)
dim(flights) # 19 columns
#> [1] 336776     19
dim(flights2) # 20 columns
#> [1] 336776     20
```

We used the function `dplyr::rename(newname = oldname)` to change the `name` column to `airline`. This is so when we add the airport names there are not multiple columns with the same id.

The `airports` data set has a lot of information, but we're really only interested in the airport names. We can combine single table and two table verbs to only join the airport name with the airport code. This gets us the information that we need, without making the object too large and unwieldly.

```
flights3 <- airports %>%
  # select only columns of interest
  select(faa, name) %>%
  # join the two datasets
  left_join(flights2, ., by = c("dest" = "faa")) %>%
  rename(dest.name = name)
```

You may have noticed something tricky in that last section of code. What does the `.,` represent? So far, all of our piping operations have used the output of one function as the first argument in the next function. But in the above example, we want the output of the first function as the *second* argument in the second function. In order to do this we use the "." to let R know that that's where we want the output from the pipe to go. The reason we want it as the second argument is that `left_join()` keeps all of the *first* objects rows, and gets rid of any non-matched in the second dataset. An alternative to using the "." notation would have been to use `right_join()`:

```
flights3b <- airports %>%
  select(faa, name) %>%
  right_join(flights2, by = c("faa" = "dest")) %>%
  rename(dest.name = name)
```

Notice that the order or "faa" and "dest" has been switched. If you tried it with the original order of `"dest" = "faa"` it would have thrown an error because R would be looking for a column named "dest" in the first dataset which doesn't exist.

Also important to note that we need to be careful here, as the `right_join()` default keeps all of the `b` dataset and drops any non-matches in `a`.

All of this is to say that the order of data inputs affects the order of your code. The syntaxes are all so similar that it's very easy to write code that reads well and looks like it should work, but can throw unexpected errors. But back to the task at hand; finishing adding airport names to the `flights3` dataset.

```
flights4 <- airports %>%
  select(faa, name) %>%
  left_join(flights3, ., by = c("origin" = "faa")) %>%
  rename(origin.name = name)
```

Now we can re-run our example above and have more useful information. Make sure to change the variable names so you're working with our updated objects and variables.

```
flights4 %>%
  group_by(airline, dest.name) %>%
  select(dest.name, airline, dep_delay) %>%
  summarize(mean.delay = mean(dep_delay,na.rm = T),
            count = n()) %>%
  arrange(airline, desc(mean.delay))
#> # A tibble: 308 x 4
#> # Groups:   airline [16]
#>   airline                   dest.name                          mean.delay count
#>   <chr>                     <chr>                                   <dbl> <int>
#> 1 AirTran Airways Corporation Akron Canton Regional Airport         20.8    864
#> 2 AirTran Airways Corporation Hartsfield Jackson Atlanta Intl       18.4   2337
#> 3 AirTran Airways Corporation General Mitchell Intl                 -1.71    59
#> 4 Alaska Airlines Inc.        Seattle Tacoma Intl                    5.80   714
#> # ... with 304 more rows
```

Exercises

1. Look at the `weather` object. Now, combine the `weather` and `flights` object without calling any column names: `left_join(flights, weather)`. What happened? How many / which columns did it join by? What do you think would happen if you called fewer column? i.e. `left_join(flights, weather, by = c("year", "month", "day", "origin"))`. Hint: what are the number of rows in each?

2. Return a data frame that has the average arrival delay by destination airport, and include the airport's lat, lon, and tz. Hint: `flights` then `group_by()`, then `summarize()` (remember `na.rm = TRUE` !!!) then `join()` to `airports` then `select()` columns.

Plug for "tidy" data

"Tidy datasets are all alike, but every messy dataset is unique in its own way" -*Hadley Wickham*

You may have noticed that manipulating the `flights` dataset was extremely easy and intuitive using the `dplyr` package. One of the reasons for this, is that `dplyr` is optimized to work with "tidy" data, and the `flights` data is already in the tidy format.

So what is tidy data? For an in depth look, see the *Tidy Data* paper http://www.jstatsoft.org/v59/i10/paper/.

Briefly, a tidy dataset is where all columns are a single variable, each observation is its own row, and each value has it's own cell.

However, many datasets are messy, and decidedly not tidy. This may be due to how data was collected, the output readings of our instruments, the fact that's its easier to enter data in a certain way, etc.

If possible, it is easiest to enter all of your data in a tidy format, foregoing any need to manipulate our data in R or excel. However, if you're working with old datasets, or have to enter data in an un-tidy way, have no fear, as there's a package for that.

The `tidyr` package was designed to turn messy data into tidy data, with surprisingly few commands (see `?tidyr`, `browseVignettes("tidyr")` or https://tidyr.tidyverse.org/ for more). Unfortunately we don't have time to go into this now but there are a number of resources available for further reading and example datasets available online (see resources below).

One more quick plug for data formatting and manipulation in R, as opposed to Excel. One of the best utilities of R (or a similar language) is the ability to make your analysis completely reproducible. It may be easier now to just do your data formatting in excel, but it is fraught with dangers. It's easy to change an equation in a cell, or reference the wrong cell, or accidentally copy a cell's equation instead of the value etc., and working backwards to find the error can be a literal nightmare. Also, changing your data's format in excel can be dangerous. You may have done it right the first time, but there's no paper trail, no breadcrumbs to follow backwards to find your error. Likewise, if you return to your data much later, it is amazingly easy to forget how exactly you moved things around, or what values were calculated how.

However, doing all of this in R allows you to leave clear signposts for exactly what you did and the order you did it in. This also allows you to leave your raw data alone. I can't tell you how many times I've messed something up and had to start over with the raw data. But creating a data formatting script allows me to find where the mistake is, fix it, and then re-import the raw data, source the entire document `Ctr + Shift + S` and be on my merry way. This is also useful if you have numerous excel sheets that all need the same functions called on them (e.g. HOBO data logger output files).

This concludes the introduction to the dplyr package. Hopefully it has given you a taste of the power and ease of using dplyr with tidy data, and provided you with enough resources to continue on and learn more if you need it.

Resources

dplyr has a whole website! http://dplyr.tidyverse.org/

vignettes

dplyr: `browseVignettes("dplyr")` tidyr: `browseVignettes("tidyr")`

Tutorials

if you download the `learnr` package you can run tutorials right in Rstudio. Install the package, and then click on the "Tutorial" tab in the Environment window in Rstudio.

Rstudio cloud has a number of tutorials that you can browse here: https://rstudio.cloud/learn/primers/2

Books

"*R for Data Science* by Hadley Wickham & Garrett Grolemund (O'reilly). Copyright 2017. 978-1-491-91039-9." Covers dplyr, tidyr, as well as the full tidyverse. Also a great general introduction to R, data analysis, workflow, programming, etc. Available free online at http://r4ds.had.co.nz/

Rstudio cheat sheets

https://www.rstudio.com/resources/cheatsheets/ I pretty much always have the dplyr one open everytime I turn R on.

Generally good resources https://blog.rstudio.org/ Aggregates a number of blogs. I follow them on twitter and can't tell you how many times I've randomly come across an article that provides a simple solution to an issue that that I had deemed impossible.

Online courses

Many options available, some free, some paid.

When I started my PhD I paid for a few months subscription to DataCamp https://www.datacamp.com/ and found their courses to be very useful. They have a whole course on dplyr, although when I took it it was a bit dated; at the end they were talking about "introducing" some function soon which I had already been using for a while at the time, but still a good place to start. And they may have updated their material at this point. I did start my PhD a long time ago…

Twitter:

@Rbloggers #rstats

Google. Seriously, if you have a question about *ANYTHING* in R, just Google it

jfpomeranz@gmail.com twitter @NotAPomegranite