Lab 3

Stats 32: Introduction to R for Undergraduates

Harrison Li

April 9, 2024

Your solutions to this lab should be uploaded to Gradescope as a knitted .pdf file before 1:20 pm today. Note you do not need to fully complete the lab to receive credit, but please read the solutions when they are posted later today.

Please type text responses to each question in the space below. You may also need to write code, which must go in a code chunk.

Note: The content of this lab is borrowed heavily from Kenneth Tay's course materials in the Autumn 2019 iteration of this course.

Let's start by loading the `tidyverse` packages:

```r
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.4.4     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become error
```

Did you notice the warning messages? What's going on there?

It turns out that the `dplyr` package has a function named `filter()`, but the `stats` package, which is automatically loaded when you start an R session, also has a function named `filter()`! So, if I type the command `filter(dataset, ...)`, how does R know which `filter()` function to use?

R looks for the function `filter()` starting with the package that was loaded most recently, and going backwards in time. Since `dplyr` was the last package loaded, R will assume that we meant `dplyr`'s version of `filter()` and use that.

What if I meant the `filter()` in the `stats` package instead? Is there a way that I can reference it?

Yes! We can use "double colon" notation: `stats::filter()`. (The general syntax for this is `packageName::functionName()`.)

nycflights13

Today we'll be returning to the `nycflights13` package from Homework 1. Instead of looking at airports, though, we'll look at flights.

```r
library(nycflights13)
data(flights)
```

We can use the `?`, `str()` and `View()` functions to examine the dataset:

```r
?flights
str(flights)
View(flights)
```

We see this data frame contains ~336,000 flights that departed from New York City (all 3 major airports) in 2013.

`filter()`, logical operators, and comparison operators

Since we are here in Stanford, we may only be interested in flights from NYC to SFO. We can use the `filter()` verb to achieve this. Recall `filter()` selects particular **rows** of a tibble, usually based on logical conditions.

```r
filter(flights, dest == "SFO")
```

```
## # A tibble: 13,331 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      558            600        -2      923            937
##  2  2013     1     1      611            600        11      945            931
##  3  2013     1     1      655            700        -5     1037           1045
##  4  2013     1     1      729            730        -1     1049           1115
##  5  2013     1     1      734            737        -3     1047           1113
##  6  2013     1     1      745            745         0     1135           1125
##  7  2013     1     1      746            746         0     1119           1129
##  8  2013     1     1      803            800         3     1132           1144
##  9  2013     1     1      826            817         9     1145           1158
## 10  2013     1     1     1029           1030        -1     1427           1355
## # i 13,321 more rows
## # i 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 could alternatively write this code using the pipe operator `%>%`:

```r
flights %>%
  filter(dest == "SFO")
```

```
## # A tibble: 13,331 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      558            600        -2      923            937
##  2  2013     1     1      611            600        11      945            931
##  3  2013     1     1      655            700        -5     1037           1045
##  4  2013     1     1      729            730        -1     1049           1115
##  5  2013     1     1      734            737        -3     1047           1113
##  6  2013     1     1      745            745         0     1135           1125
##  7  2013     1     1      746            746         0     1119           1129
##  8  2013     1     1      803            800         3     1132           1144
##  9  2013     1     1      826            817         9     1145           1158
## 10  2013     1     1     1029           1030        -1     1427           1355
## # i 13,321 more rows
## # i 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>
```

With only one dplyr verb being used here, it's not really any cleaner to use the pipe operator. But when we need to chain lots of dplyr verbs (below), it will be cleaner and less error prone to do so.

Note that we used the comparison operator `==` to test whether `dest` was equal to `"SFO"`. **DO NOT USE `=`**. In programming, `=` usually means variable assignment (equivalent to `<-` in R). Other comparison operators are `<=`, `<`, `>=`, and `>`.

We can also use the following logical operators to chain together multiple comparisons:

- And (`&`)
- Or (`|`)
- Not (`!`)

For example, there are two other airports near Stanford, San Jose International Airport ("SJC") and Oakland International Airport ("OAK"). So if we want to analyze flights that people could take to get from NYC to Stanford, we should probably include these flights by filtering to all flights whose `dest` is "SFO", or whose `dest` is "SJC", or whose `dest` is "OAK".

```r
flights %>%
  filter(dest == "SFO" | dest == "SJC" | dest == "OAK")
```

```
## # A tibble: 13,972 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      558            600        -2      923            937
##  2  2013     1     1      611            600        11      945            931
##  3  2013     1     1      655            700        -5     1037           1045
##  4  2013     1     1      729            730        -1     1049           1115
##  5  2013     1     1      734            737        -3     1047           1113
##  6  2013     1     1      745            745         0     1135           1125
##  7  2013     1     1      746            746         0     1119           1129
##  8  2013     1     1      803            800         3     1132           1144
##  9  2013     1     1      826            817         9     1145           1158
## 10  2013     1     1     1029           1030        -1     1427           1355
## # i 13,962 more rows
## # i 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 command above filters the dataset and prints it out, but does not retain the output. To keep the extracted data for further analysis, we have to assign it to a variable:

```r
Stanford <- flights %>%
  filter(dest == "SFO" | dest == "SJC" | dest == "OAK")
```

We now have flights from NYC to SFO/SJC/OAK for the entire year.

For the following questions, it may be helpful to look at the help page for the `flights` tibble to figure out what all the columns in the tibble mean.

1. Filter the `Stanford` tibble further so that it only contains flights in months when school is in session (September - June).

2. Select all flights in the original `flights` tibble that originated from EWR and had an arrival delay of at least 10 minutes.

3. How many flights in the original `flights` tibble had a scheduled departure time before 6am?

select() & rename()

Sometimes our datasets will have hundreds or thousands of variables! Not all of them may be of interest to us. `select()` allows us to choose a subset of these variables to form a smaller dataset that may be easier to work with. This corresponds to selecting only certain **columns** from the tibble (contrast with `filter()`).

The most common usage of `select` is to select columns by name. If we just want the `year`, `month` and `day` columns in our `Stanford` tibble from the previous section, we can use the following code:

```
Stanford %>%
  select(year, month, day)
```

```
## # A tibble: 13,972 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
## # i 13,962 more rows
```

If the columns we want form a contiguous block, then we can use simpler syntax. To select columns from `year` to `arr_delay` (inclusive):

```
Stanford %>%
  select(year:arr_delay)
```

```
## # A tibble: 13,972 x 9
##     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      558            600        -2      923            937
##  2  2013     1     1      611            600        11      945            931
##  3  2013     1     1      655            700        -5     1037           1045
##  4  2013     1     1      729            730        -1     1049           1115
##  5  2013     1     1      734            737        -3     1047           1113
##  6  2013     1     1      745            745         0     1135           1125
##  7  2013     1     1      746            746         0     1119           1129
##  8  2013     1     1      803            800         3     1132           1144
##  9  2013     1     1      826            817         9     1145           1158
## 10  2013     1     1     1029           1030        -1     1427           1355
## # i 13,962 more rows
## # i 1 more variable: arr_delay <dbl>
```

In this dataset, the `year` column is superfluous, since all the values are all 2013. The code below drops the year column, keeping the rest:

```
Stanford %>%
  select(-year)
```

```
## # A tibble: 13,972 x 18
##    month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1     1     1      558            600        -2      923            937
##  2     1     1      611            600        11      945            931
##  3     1     1      655            700        -5     1037           1045
##  4     1     1      729            730        -1     1049           1115
##  5     1     1      734            737        -3     1047           1113
##  6     1     1      745            745         0     1135           1125
##  7     1     1      746            746         0     1119           1129
##  8     1     1      803            800         3     1132           1144
##  9     1     1      826            817         9     1145           1158
## 10     1     1     1029           1030        -1     1427           1355
## # i 13,962 more rows
## # i 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>
```

select() can also be used to rearrange the columns. If, for example, I wanted to have the first 3 columns be day, month, year instead of year, month, day:

```
Stanford %>%
  select(day, month, year, everything())
```

```
## # A tibble: 13,972 x 19
##      day month  year dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1     1     1  2013      558            600        -2      923            937
##  2     1     1  2013      611            600        11      945            931
##  3     1     1  2013      655            700        -5     1037           1045
##  4     1     1  2013      729            730        -1     1049           1115
##  5     1     1  2013      734            737        -3     1047           1113
##  6     1     1  2013      745            745         0     1135           1125
##  7     1     1  2013      746            746         0     1119           1129
##  8     1     1  2013      803            800         3     1132           1144
##  9     1     1  2013      826            817         9     1145           1158
## 10     1     1  2013     1029           1030        -1     1427           1355
## # i 13,962 more rows
## # i 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 everything() selection helper intelligently refers to all columns not already explicitly referenced.

To rename columns, note we can use the rename() function:

```
Stanford %>%
  rename(tail_num = tailnum)
```

```
## # A tibble: 13,972 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      558            600        -2      923            937
##  2  2013     1     1      611            600        11      945            931
##  3  2013     1     1      655            700        -5     1037           1045
##  4  2013     1     1      729            730        -1     1049           1115
##  5  2013     1     1      734            737        -3     1047           1113
##  6  2013     1     1      745            745         0     1135           1125
##  7  2013     1     1      746            746         0     1119           1129
##  8  2013     1     1      803            800         3     1132           1144
##  9  2013     1     1      826            817         9     1145           1158
## 10  2013     1     1     1029           1030        -1     1427           1355
## # i 13,962 more rows
## # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tail_num <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

4. Create a tibble containing just the origin and destination airports of all flights in the `flights` tibble that departed before 6 am. Hint: You will need to chain multiple dplyr verbs.

arrange()

Often we get datasets whose rows are not in order, or at least not in an order that is useful. The `arrange()` function allows us to reorder the rows according to an order we want.

The `Stanford` dataset looks like it is already ordered by actual departure time. But perhaps I'm most interested in the flights which had the longest departure delay. I could sort the dataset as follows:

```
Stanford %>%
  arrange(dep_delay)
```

```
## # A tibble: 13,972 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    11      710            730       -20     1039           1105
##  2  2013    11    16      712            730       -18     1025           1055
##  3  2013     9    11      712            730       -18      946           1045
##  4  2013    11    19      713            730       -17     1036           1055
##  5  2013     7    14     1151           1208       -17     1450           1515
##  6  2013    12    10      714            730       -16     1104           1110
##  7  2013     3    29     1050           1106       -16     1359           1431
##  8  2013     4    20     1420           1436       -16     1737           1755
##  9  2013     5    20      719            735       -16      951           1110
## 10  2013     1    23      545            600       -15      948            925
## # i 13,962 more rows
## # i 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>
```

Looks like the flights with the *shortest* delays are at the top instead! To re-order by descending order, use `desc()`:

```
Stanford %>%
  arrange(desc(dep_delay))
```

```
## # A tibble: 13,972 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    20     1139           1845      1014     1457           2210
##  2  2013     7     7     2123           1030       653       17           1345
##  3  2013     7     7     2059           1030       629      106           1350
##  4  2013     7     6      149           1600       589      456           1935
##  5  2013     7    10      133           1800       453      455           2130
##  6  2013     7    10     2342           1630       432      312           1959
##  7  2013     7     7     2204           1525       399      107           1823
##  8  2013     7     7     2306           1630       396      250           1959
##  9  2013     6    23     1833           1200       393       NA           1507
## 10  2013     7    10     2232           1609       383      138            1928
## # i 13,962 more rows
## # i 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()` also allows us to sort by more than one column, in that each additional column will be used to break ties in the values of the preceding ones. For example, `flights` seems to be sorted by year, month, day, and actual departure time. If I wanted to sort by year, month, day and scheduled departure time instead:

```
Stanford %>%
  arrange(year, month, day, sched_dep_time)
```

```
## # A tibble: 13,972 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      558            600        -2      923            937
##  2  2013     1     1      611            600        11      945            931
##  3  2013     1     1      655            700        -5     1037           1045
##  4  2013     1     1      729            730        -1     1049           1115
##  5  2013     1     1      734            737        -3     1047           1113
##  6  2013     1     1      745            745         0     1135           1125
##  7  2013     1     1      746            746         0     1119           1129
##  8  2013     1     1      803            800         3     1132           1144
##  9  2013     1     1      826            817         9     1145           1158
## 10  2013     1     1     1029           1030        -1     1427           1355
## # i 13,962 more rows
## # i 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>
```

5. Create a tibble containing all flights departing on April 9, 2013, ordered by the scheduled arrival time from earliest to latest.

mutate()

In this dataset we have both the time the plane spent in the air (`air_time`) and distance traveled (`distance`). From these two pieces of information, we can create a new column containing the average speed of the plane using `mutate()`.

`mutate()` adds new columns to the end of the dataset, so let's work with a smaller dataset for now so that we can easily see the values of our new column.

```
Stanford_small <- Stanford %>%
    select(month, carrier, origin, dest, air_time, distance) %>%
    mutate(speed = distance / air_time * 60)
Stanford_small
```

```
## # A tibble: 13,972 x 7
##    month carrier origin dest  air_time distance speed
##    <int> <chr>   <chr>  <chr>    <dbl>    <dbl> <dbl>
##  1     1 UA      EWR    SFO        361     2565  426.
##  2     1 UA      JFK    SFO        366     2586  424.
##  3     1 DL      JFK    SFO        362     2586  429.
##  4     1 VX      JFK    SFO        356     2586  436.
##  5     1 B6      JFK    SFO        350     2586  443.
##  6     1 AA      JFK    SFO        378     2586  410.
##  7     1 UA      EWR    SFO         373     2565  413.
##  8     1 UA      JFK    SFO         369     2586  420.
##  9     1 UA      EWR    SFO         357     2565  431.
## 10     1 AA      JFK    SFO         389     2586  399.
## # i 13,962 more rows
```

`mutate()` can be used to create several new variables at once. The later variables created can reference the other variables just created in the same `mutate()` call. For example, the following code is valid:

```
Stanford_small %>%
  mutate(speed_miles_per_min = air_time / distance,
         speed_miles_per_hour = speed_miles_per_min * 60)
```

```
## # A tibble: 13,972 x 9
##    month carrier origin dest  air_time distance speed speed_miles_per_min
##    <int> <chr>   <chr>  <chr>    <dbl>    <dbl> <dbl>               <dbl>
##  1     1 UA      EWR    SFO        361     2565  426.               0.141
##  2     1 UA      JFK    SFO        366     2586  424.               0.142
##  3     1 DL      JFK    SFO        362     2586  429.               0.140
##  4     1 VX      JFK    SFO        356     2586  436.               0.138
##  5     1 B6      JFK    SFO        350     2586  443.               0.135
##  6     1 AA      JFK    SFO        378     2586  410.               0.146
##  7     1 UA      EWR    SFO        373     2565  413.               0.145
##  8     1 UA      JFK    SFO        369     2586  420.               0.143
##  9     1 UA      EWR    SFO        357     2565  431.               0.139
## 10     1 AA      JFK    SFO        389     2586  399.               0.150
## # i 13,962 more rows
## # i 1 more variable: speed_miles_per_hour <dbl>
```

If you only want to keep the newly created variables, use `transmute()` instead of `mutate()`.

Writing/reading files

I now change my working directory and write out `Stanford` into a `.csv` file `Stanford.csv`. The code is commented out so that you can knit this lab (hopefully) without further modification.

```
# setwd("/Users/harrisonli/Documents/iCloud_Documents/Stanford/Teaching/Stats 32/")
# write_csv(Stanford, "Stanford.csv")
```

We can read it back in:

```
setwd("/Users/harrisonli/Documents/iCloud_Documents/Stanford/Teaching/Stats 32/")
Stanford_in <- read_csv("Stanford.csv")
```

```
## Rows: 13972 Columns: 19
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr   (4): carrier, tailnum, origin, dest
## dbl  (14): year, month, day, dep_time, sched_dep_time, dep_delay, arr_time, ...
## dttm  (1): time_hour
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

You should change the working directory above to get this to work on your machine.

The `%in%` operator

Recall that we used the following line of code to extract flights that landed in SFO, SJC or OAK:

```
Stanford <- flights %>% filter(dest == "SFO" | dest == "SJC" | dest == "OAK")
```

We can use the `%in%` operator to make the code more succinct:

```
flights %>%
  filter(dest %in% c("SFO", "SJC", "OAK"))
```

```
## # A tibble: 13,972 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      558            600        -2      923            937
##  2  2013     1     1      611            600        11      945            931
##  3  2013     1     1      655            700        -5     1037           1045
##  4  2013     1     1      729            730        -1     1049           1115
##  5  2013     1     1      734            737        -3     1047           1113
##  6  2013     1     1      745            745         0     1135           1125
##  7  2013     1     1      746            746         0     1119           1129
##  8  2013     1     1      803            800         3     1132           1144
##  9  2013     1     1      826            817         9     1145           1158
## 10  2013     1     1     1029           1030        -1     1427           1355
## # i 13,962 more rows
## # i 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 `%in%` operator is very useful to make your code more readable (and less work to type), especially we are checking if `dest` belongs to a long list of airports.

6. How would you select the flights whose destination was *not* one of the Bay Area airports (SFO, SJC, OAK)?