MTH 3270 Notes 3

4 Data Wrangling (4)

- ***Data wrangling*** refers to re-organizing, transforming, and re-formatting data to make it more suitable for statistical analysis.

4.1 Introduction: The "dplyr" Package

- The "dplyr" package contains several functions for **data wrangling**. Type:

```
help(package = dplyr)
```

  to see a list of the functions (and data sets) contained in "dplyr".

- Here are five important functions for data wrangling. These are the so-called ***verbs*** of "dplyr":

```
select()      Take a subset of the columns (i.e. variables).
filter()      Take a subset of the rows (i.e. observations).
mutate()      Add columns computed from existing ones.  Use
              transmute() instead if you only want to keep the
              newly computed columns.
arrange()     Sort the rows of a data set according to the values
              in one or more columns.
summarize()   Summarize a data frame or a grouped data frame (as
              created by group_by()), returning one row for each
              group.
```

- Each of these functions takes a *data frame* as its main argument (.data), and returns a *data frame*.

  This is important because it means **the output of any one of them can be used as the input of another**, and "chaining" together function calls in this manner is made simple by "dplyr"'s so-called ***pipe operator***:

```
%>%           # Pipe operator, for passing the returned value from one
              # function call as the main argument of another function
              # call, e.g. x %>% f() %>% g() is the same as g(f(x)).
```

- Another important function in `"dplyr"` is:

> *`rename()` Modify the names of columns in a data set.*

Data Set: `flights`

The `flights` data set is in the `"nycflights13"` package. It contains all 336,776 flights that departed New York City in 2013. The data set is from the U.S. Bureau of Transportation Statistics,

`https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236`

It contains 19 variables:

`year, month, day`	Date of departure
`dep_time, arr_time`	Actual departure and arrival times (format HHMM or HMM), local tz.
`sched_dep_time, sched_arr_time`	Scheduled departure and arrival times (format HHMM or HMM), local tz.
`dep_delay, arr_delay`	Departure and arrival delays, in minutes. Negative times represent early departures/arrivals.
`hour, minute`	Time of scheduled departure broken into hour and minutes.
`carrier`	Two letter carrier abbreviation. See `airlines()` to get name
`tailnum`	Plane tail number
`flight`	Flight number
`origin, dest`	Origin and destination. See airports() for additional metadata.
`air_time`	Amount of time spent in the air, in minutes
`distance`	Distance between airports, in miles
`time_hour`	Scheduled date and hour of the flight as a POSIXct date. Along with origin, can be used to join flights data to weather data.

For more information, see the help file (after installing the `"nycflights13"` package):

```
library(nycflights13)
? flights
```

4.2 Extracting Columns with `select()`

- We can extract **columns** (**variables**) from a data frame using `select()`.

For example, to extract the columns `year`, `month`, and `day` from the `flights` data set (from the `"nycflights13"` package), we can type:

```
library(nycflights13)
select(.data = flights, year, month, 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
```

(Recall that in Class Notes 1 we **selected** columns using the dollar sign `$`, single square brackets `[ ]`, or double square brackets `[[ ]]`.)

- None of `"dplyr"`'s *verb* functions changes the data frame passed to it. So in practice, we'd usually *save* the changes as a new data frame, for example (below) as `flights_ymd`:

```
flights_ymd <- select(.data = flights, year, month, day)
```

- Here's an example that uses the **pipe operator** `%>%` to accomplish the *same thing* as the command above:

```
flights_ymd <- flights %>% select(year, month, day)
```

(We'll discuss the **pipe operator** more later.)

- We can extract a *range* of columns using the colon operator ':'. For example:

```
select(.data = 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
```

- A minus sign '-' can be used to extract all columns *except* a specified set of them. For example:

```
select(.data = flights, -(year:day))
```

- Here are some "helper" functions that can be used with `select()`:
  - `starts_with("abc")` matches names that begin with `"abc"`.
  - `ends_with("xyz")` matches names that end with `"xyz"`.
  - `contains("ijk")` matches names that contain `"ijk"`.
  - `num_range("x", 1:3)` matches `x1`, `x2`, `x3`.

  For more information, see the help file for `select()`:

```
? select
```

- `select()` can also be used to *rearrange* columns. For this, the `everything()` function is useful for moving a few columns to the front:

```
select(.data = flights, time_hour, air_time, everything())
```

```
## # A tibble: 336,776 x 19
##    time_hour           air_time  year month   day dep_time sched_dep_time dep_delay
##    <dttm>                 <dbl> <int> <int> <int>    <int>          <int>     <dbl>
##  1 2013-01-01 05:00:00      227  2013     1     1      517            515         2
##  2 2013-01-01 05:00:00      227  2013     1     1      533            529         4
##  3 2013-01-01 05:00:00      160  2013     1     1      542            540         2
##  4 2013-01-01 05:00:00      183  2013     1     1      544            545        -1
##  5 2013-01-01 06:00:00      116  2013     1     1      554            600        -6
##  6 2013-01-01 05:00:00      150  2013     1     1      554            558        -4
##  7 2013-01-01 06:00:00      158  2013     1     1      555            600        -5
##  8 2013-01-01 06:00:00       53  2013     1     1      557            600        -3
##  9 2013-01-01 06:00:00      140  2013     1     1      557            600        -3
## 10 2013-01-01 06:00:00      138  2013     1     1      558            600        -2
## # ... with 336,766 more rows, and 11 more variables: arr_time <int>,
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, distance <dbl>, hour <dbl>, minute <dbl>
```

Section 4.2 Exercises

Exercise 1 The `flights` data set is contained in the `"nycflights13"` package. Load the package:

```
library(nycflights13)
```

You can look at the names of the variables in `nycflights13` by typing:

```
names(flights)
```

Guess what each of the following commands returns, then check your answers:

a) `select(.data = flights, year, day)`

b) `select(.data = flights, year:day)`

c) `select(.data = flights, -(year:day))`

Exercise 2 This exercise concerns the **"helper"** functions used with `select()`.

Look again at the names of the variables in `flights`:

```
names(flights)
```

Guess what each of the following commands returns, then check your answers:

a) `select(.data = flights, starts_with("sched"))`

b) `select(.data = flights, contains("arr"))`

c) `select(.data = flights, starts_with("dep_"), starts_with("arr_"))`

4.3 Tibbles

- In `"dplyr"` (and other packages in the so-called ***tidyverse*** suite of packages), a ***tibble*** *is* a **data frame**:

```
is.data.frame(flights)

## [1] TRUE
```

They differ from regular data frames in how they get printed to the console. Dimension and type information is printed, and only ten rows and as many columns as fit in the console appear.

To view the entire data set, use `View()`:

```
View(flights)
```

Tibbles belong to the `"tbl_df"` class of objects, which is a *special case* of the `"data frame"` class. Actually, they belong to *three* classes:

```
class(flights)

## [1] "tbl_df"     "tbl"        "data.frame"
```

When an object belongs to *more than one class*, each class is a *special case* of the classes that come after it in the list returned by `class()`.

For example, `"tbl_df"` is a *special case* of the `"tbl"` class, which in turn is a *special case* of the `"data.frame"` class.

Because `"tbl_df"` is a *special case* of `"data.frame"`, *any action that can be performed on data frames can also be performed on tibbles.*

4.4 Filtering Rows with `filter()`

- We can use `filter()` to extract from a data frame the rows that satisfy one or more conditions. For example, to obtain all the `flights` on January 1st, type:

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

## # A tibble: 842 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
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
```

```
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 832 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>
```

(Recall that in Class Notes 1 we **filtered** using single square brackets `[ ]` or `subset()`.)

- The logical operators '`&`', '`|`', and '`!`' ("and", "or", and "not") are useful when specifying the condition(s) to be met by extracted rows.

  Another useful operator is the "in" operator:

```
%in%          # Tests whether a value is in a set of values. Returns
              # TRUE or FALSE.
```

  It returns `TRUE` or `FALSE` depending on whether a given value is among a set of values:

```
7 %in% c(2, 4, 7, 9, 6)

## [1] TRUE
```

```
3 %in% c(2, 4, 7, 9, 6)

## [1] FALSE
```

  For example, to obtain the `flights` that departed in November *or* December, type:

```
# The condition is a "logical" vector:
filter(.data = flights, month == 11 | month == 12)
```

  Note that the specified condition (`month == 11 | month == 12`) is a `"logical"` vector.

  The operator `%in%` could be used to simplify the above command:

```
filter(.data = flights, month %in% c(11, 12))
```

- When we specify *more than one* condition in `filter()`, they're combined with '`&`'. For example, the following commands do the *same thing*:

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

```
filter(.data = flights, month == 1 & day == 1)
```

- We can remove rows for which one or more columns have `NA`s using `filter()` with `is.na()`. For example, consider this data frame:

```
x <- data.frame(x1 = c(2, 1, NA, 8, 7, 5, 4),
                x2 = c("a", NA, "c", "d", "c", "a", "d"),
                stringsAsFactors = FALSE)
x

##   x1   x2
## 1  2    a
## 2  1 <NA>
## 3 NA    c
## 4  8    d
## 5  7    c
## 6  5    a
## 7  4    d
```

(Note that R represents missing `"character"` values as `<NA>`. This allows for the possibility that `"NA"` might appear as a *value* in a `"character"` vector (e.g. the abbreviation for "Narcotics Anonymous").)

To remove rows for which `x1` is `NA`, type:

```
filter(x, !is.na(x1))

##   x1   x2
## 1  2    a
## 2  1 <NA>
## 3  8    d
## 4  7    c
## 5  5    a
## 6  4    d
```

To remove rows for which *either* of the variables `x1` or `x2` contains an `NA`, type:

```
filter(x, !is.na(x1), !is.na(x2))

##   x1 x2
## 1  2  a
## 2  8  d
## 3  7  c
## 4  5  a
## 5  4  d
```

Another way to remove rows for which *any* of the variables in a data frame contains an `NA` is using the `complete.cases()` function:

```
filter(x, complete.cases(x))

##   x1 x2
## 1  2  a
## 2  8  d
## 3  7  c
## 4  5  a
## 5  4  d
```

The `complete.cases()` function (which is in `base` R) returns a `"logical"` vector whose elements are `TRUE` if the corresponding row of `x` is "complete" (doesn't contain any `NA`s) and `FALSE` otherwise.

- `filter()` *only* returns rows for which the specified condition is `TRUE`. It *doesn't* return rows for which the condition is `NA`. For example:

```
filter(x, x1 < 5)

##   x1   x2
## 1  2    a
## 2  1 <NA>
## 3  4    d
```

To tell `filter()` to *also* return rows for which the condition is `NA`, type:

```
filter(x, is.na(x1) | x1 < 5)

##   x1   x2
## 1  2    a
## 2  1 <NA>
## 3 NA    c
## 4  4    d
```

Section 4.4 Exercises

Exercise 3 Report R commands that use `filter()` (and the logical operators '`&`', '`|`', and '`!`') to find all flights that:

a) Had an arrival delay of two or more hours.

b) Flew to Houston (`IAH` or `HOU`).

c) Were operated by United, American, or Delta.

d) Departed in summer (July, August, and September).

e) Departed between midnight and 6:00 AM (inclusive).

4.5 Arranging Rows with `arrange()`

- We can use `arrange()` sort the rows of a data frame according to the values in one or more columns.

  For example, to sort the rows of `flights` according to the departure delay, type:

```
arrange(.data = 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>
```

- To sort in **descending** order, use `desc()`, for example:

```
arrange(.data = flights, desc(dep_delay))
```

- If you specify more than one column, each additional column will be used to break ties in the previous columns. For example,

```
arrange(.data = flights, year, month, day)
```

```
## # 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
##  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
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
```

```
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
## #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Section 4.5 Exercises

Exercise 4 Report R commands that use `arrange()` to sort the `flights` data set to:

a) Find the least delayed `flights`.

b) Find the most delayed `flights`.

c) Find the `flights` that had the earliest departure times.

d) Find the `flights` that had the latest departure times.

e) Find the `flights` that traveled the shortest distance.

f) Find the `flights` that traveled the longest distance.

Exercise 5 Occasionally, we want to sort the rows of a data frame so that all of the `NA`s are at the top. Consider the following data frame:

```
x <- data.frame(x1 = c(2, 1, NA, 8, 7, 5, 4),
                x2 = c("a", NA, "c", "d", "c", "a", "d"),
                stringsAsFactors = FALSE)
x
```

```
##   x1   x2
## 1  2    a
## 2  1 <NA>
## 3 NA    c
## 4  8    d
## 5  7    c
## 6  5    a
## 7  4    d
```

Recall that `is.na()` returns a `"logical"` vector whose `TRUE`s indicate `NA`s, and `"logical"` values are treated as `0` and `1` in operations that expect a numerical value.

a) Guess what the following command does, then check your answer:

```
arrange(.data = x, is.na(x1))
```

b) Guess what the following command does, then check your answer:

```
arrange(.data = x, desc(is.na(x1)))
```

4.6 Creating New Variables (Columns) with `mutate()`

- We can use `mutate()` add or new columns that are computed from existing columns of a data frame.
- `mutate()` always adds the columns to the rightmost end of the data frame.
- For the examples, we'll us a smaller data frame:

```
flights_small <- select(.data = flights,
                        year:day,
                        ends_with("delay"),
                        distance,
                        air_time)
```

Here's an example:

```
mutate(.data = flights_small,
       gain = dep_delay - arr_delay,
       speed = distance / air_time * 60)
```

```
## # A tibble: 336,776 x 9
##     year month   day dep_delay arr_delay distance air_time  gain speed
##    <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
##  1  2013     1     1         2        11     1400      227    -9  370.
##  2  2013     1     1         4        20     1416      227   -16  374.
##  3  2013     1     1         2        33     1089      160   -31  408.
##  4  2013     1     1        -1       -18     1576      183    17  517.
##  5  2013     1     1        -6       -25      762      116    19  394.
##  6  2013     1     1        -4        12      719      150   -16  288.
##  7  2013     1     1        -5        19     1065      158   -24  404.
##  8  2013     1     1        -3       -14      229       53    11  259.
##  9  2013     1     1        -3        -8      944      140     5  405.
## 10  2013     1     1        -2         8      733      138   -10  319.
## # ... with 336,766 more rows
```

- If we *only* want to keep the newly computed variables, use `transmute()`:

```
transmute(.data = flights_small,
       gain = dep_delay - arr_delay,
       speed = distance / air_time * 60)
```

Section 4.6 Exercises

Exercise 6 Use `mutate()` or `transmute()`, with `flights`, to compute `arr_time - dep_time` and compare it with `air_time`. Why are they different?

Exercise 7 A newly computed variable (column) can be used *within* `mutate()`.

Execute the following command (using `flights_small` from above) and verify that the variable `gain_per_hour` gets computed:

```
mutate(.data = flights_small,
       gain = dep_delay - arr_delay,
       hours = air_time / 60,
       gain_per_hour = gain / hours)
```

4.7 Renaming Variables (Columns) with `rename()`

- We use `rename()` to rename variables (columns) in a data frame. For example (using `x` from above):

```
x

##   x1   x2
## 1  2    a
## 2  1 <NA>
## 3 NA    c
## 4  8    d
## 5  7    c
## 6  5    a
## 7  4    d

new_x <- rename(.data = x, new_x1 = x1, new_x2 = x2)
new_x

##   new_x1 new_x2
## 1      2      a
## 2      1   <NA>
## 3     NA      c
## 4      8      d
## 5      7      c
## 6      5      a
## 7      4      d
```

Section 4.7 Exercises

Exercise 8 Here's a small data frame:

```
z <- data.frame(z1 = c(5, 4, 3),
                z2 = c("a", "c", "b"),
                z3 = c(14, 22, 13))
```

Use `rename()` to change the names of the variables in `z` to `new_z1`, `new_z2`, and `new_z3`. Report your R command(s).

4.8 Summarizing Data with `summarize()`

- The function `summarize()` is used to **summarize** one or more variables (columns) of a data frame.

  It "collapses" the data frame into a *single row* containing summary statistics for the variables.

  For example, to compute the mean (average) departure and arrival delays, type:

```
summarize(.data = flights,
          mean_dep_delay = mean(dep_delay, na.rm = TRUE),
          mean_arr_delay = mean(arr_delay, na.rm = TRUE))

## # A tibble: 1 x 2
##   mean_dep_delay mean_arr_delay
##            <dbl>          <dbl>
## 1           12.6           6.90
```

  Above, we used the argument `na.rm` to remove the `NA`s before computing the means.

- Not all functions have the `na.rm` argument, so sometimes it's better to just remove the rows with `NA`s in the variables you're interested in:

```
not_cancelled <- filter(.data = flights, !is.na(dep_delay), !is.na(arr_delay))
```

```
summarize(.data = not_cancelled,
          mean_dep_delay = mean(dep_delay),
          mean_arr_delay = mean(arr_delay))

## # A tibble: 1 x 2
##   mean_dep_delay mean_arr_delay
##            <dbl>          <dbl>
## 1           12.6           6.90
```

- Here are some functions that compute **statistics** for use with `summarize()`.
  - For summarizing the **center** (typical value) of a variable:

```
mean()     # Mean (average)
median()   # Median (middle value, i.e. 50th percentile)
```

  - For summarizing the **spread** (variation) of a variable:

```
sd()       # Standard deviation (typical deviation away from the
           # mean)
IQR()      # Interquartile range (amount of spread between 25th
           # and 75th percentiles)
mad()      # Median absolute deviation (median of the deviations
           # away from the median of the data)
```

– For summarizing a **ranked value** (smallest, largest, 25th percentile, etc.) of a variable:

```
min()      # Minimum (smallest) value
max()      # Maximum (largest) value
quantile() # Percentile (also called quantile), e.g.
           # quantile(x, 0.25) returns the 25th percentile of
           # the data.
```

– For summarizing the **position** among the (unsorted) values of the variable (these are in the `"dplyr"` package):

```
first()    # First value, equivalent to x[1].
last()     # Last value, equivalent to x[length(x)]
nth()      # nth value, e.g. nth(x, 2) returns the 2nd value in x
```

– For **counting** values of the variable (this is in the `"dplyr"` package):

```
n()        # The number of values (i.e. number of observations),
           # equivalent to length(), but specifically for use in
           # summarize(), mutate(), and filter().
```

Section 4.8 Exercises

Exercise 9 Create the `not_cancelled` data frame:

```
not_cancelled <- filter(.data = flights, !is.na(dep_delay), !is.na(arr_delay))
```

Using `not_cancelled`, do the following.

a) Use `summarize()` with `median()` to find the median departure delay and the median arrival delay.

b) Use `summarize()` with `max()` to find the longest departure delay and the longest arrival delay.

c) Use `summarize()` with `min()` to find the shortest departure delay and the shortest arrival delay.

Exercise 10 Re-create the `not_cancelled` data frame from Exercise 9.

We might be interested in *how many* (non-cancelled) flights there were in total, *how many* arrivals were delayed by more than an hour, and what *proportion* of arrivals were delayed by more than an hour.

a) The function `n()` (from `"dplyr"`) is used in `summarize()` to *count* how many values (total) are in a variable. What does the following command do?

```
summarize(.data = not_cancelled,
          total_flights = n())
```

b) Recall that `"logical"` values are converted to `0` and `1` in computations. We can `sum()` `"logical"` values to *count* how many data values satisfy a condition. What does the following command do?

```
summarize(.data = not_cancelled,
          hour_arr_delay_total = sum(arr_delay > 60))
```

c) What does the following command do?

```
summarize(.data = not_cancelled,
          hour_arr_delay_proportion = sum(arr_delay > 60) / n())
```

4.9 Applying `summarize()` to Groups using `group_by()`

- We can summarize variables separately for each of two or more **groups** using `summarize()` and `group_by()` (from the `"dplyr"` package).

group_by()	*Group together rows of a data set according to values in one or more columns, for use in summarize(), etc.*

- For example, using the `flights` data (from the `"nycflights13"` package), to compute the average (mean) **delay** by `month`, type:

```
by_month <- group_by(.data = flights, month)
summarize(.data = by_month, delay = mean(dep_delay, na.rm = TRUE))

## # A tibble: 12 x 2
```

```
##    month delay
##    <int> <dbl>
##  1     1 10.0
##  2     2 10.8
##  3     3 13.2
##  4     4 13.9
##  5     5 13.0
##  6     6 20.8
##  7     7 21.7
##  8     8 12.6
##  9     9  6.72
## 10    10  6.24
## 11    11  5.44
## 12    12 16.6
```

This returns a data frame with one row for each `month`.

- `group_by()` returns a data frame that belongs to a class called `"grouped_df"` (a special case of the `"data.frame"` class):

```
class(by_month)

## [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

All of `"dplyr"`'s *verb* functions (`select()`, `filter()`, etc.) accept `"grouped_df"`s as their `.data` argument.

Passing a `"grouped_df"` to a *verb* function changes the scope the function from operating on the entire data set to operating on it **group-by-group**.

- We can **group** by *more than one* grouping variable. For example, here we group by `year`, `month`, and `day` to obtain the daily mean **delay** for each day in 2013:

```
by_day <- group_by(flights, year, month, day)
summarize(by_day, delay = mean(dep_delay, na.rm = TRUE))

## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day delay
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.5
##  2  2013     1     2 13.9
##  3  2013     1     3 11.0
##  4  2013     1     4  8.95
##  5  2013     1     5  5.73
##  6  2013     1     6  7.15
##  7  2013     1     7  5.42
##  8  2013     1     8  2.55
##  9  2013     1     9  2.28
```

```
## 10  2013     1    10  2.84
## # ... with 355 more rows
```

This returns a data frame with one row for each combination of `year`, `month`, and `day`.

Section 4.9 Exercises

Exercise 11 This problem concerns the `group_by()` and `summarize()` functions.

a) Explain in words what the following commands do (recall that `dest` is the *destination* of the `flight`):

```
by_dest <- group_by(.data = flights, dest)
delay <- summarize(.data = by_dest,
                   delay = mean(arr_delay, na.rm = TRUE))
```

b) `summarize()` can summarize *more than one variable* at a time. Explain in words what the following commands do:

```
by_dest <- group_by(.data = flights, dest)
delay <- summarize(.data = by_dest,
                       dist = mean(distance, na.rm = TRUE),
                       delay = mean(arr_delay, na.rm = TRUE))
```

Exercise 12 Here's a data frame `ExpData` containing responses to treatments in an experiment and ages and genders of the subjects who participated in the experiment:

```
resp <- c(23, 11, 14, 16, 19, 26, 24, 29, 31, 28, 34, 25)
trt <- c(rep("Ctrl", 4), rep("TrtA", 4), rep("TrtB", 4))
age <- c(33, 45, 30, 24, 22, 31, 39, 40, 29, 19, 27, 25)
gndr <- c("m", "m", "f", "f", "m", "f", "f", "m", "f", "m", "f", "m")
```

```
ExpData <- data.frame(TrtGrp = trt,
                      SubjectGender = gndr,
                      SubjectAge = age,
                      Response = resp, stringsAsFactors = FALSE)
```

a) Use `group_by()` and `summarize()` to compute the mean `Response` by `TrtGrp`. Report the three group mean responses.

b) Now use `group_by()` and `summarize()` to compute the mean `Response` *and* mean `Age` by `TrtGrp`. Report the three group mean responses *and* the three mean ages.

c) The function `n()` (from `"dplyr"`) is used (without any arguments) in `summarize()` to *count* observations. Explain in words what the following commands do:

```
by_trt <- group_by(.data = ExpData, TrtGrp)
summarize(.data = by_trt, Count = n())
```

Exercise 13 This problem concerns the `group_by()` and `summarize()` functions.

a) Use `group_by()`, `summarize()` with `n()`, and `arrange()` to determine which which `tailnum` (i.e. individual **airplane**) flew the most times.

b) Use `group_by()`, `summarize()` with `n()`, and `arrange()` to determine which which `dest` (i.e. which **destination**) was flown to the most times.

4.10 Acknowledgment

- The above notes (and several examples) on the `"dplyr"` package borrow heavily from the book:

  R for Data Science, by Wickham, H., Grolemund, G., O'Reilly, 2017.