Data Manipulation in R with dplyr

Davood Astaraky

- Introduction to dplyr and tbls
  - Load the dplyr and hflights package
  - Convert data.frame to table
  - Changing labels of hflights
  - The five verbs and their meaning
- Select and mutate
  - Choosing is not loosing! The select verb
  - Helper functions for variable selection
  - Comparison to basic R
  - Mutating is creating
  - Add multiple variables using mutate
- Filter and arrange
  - Logical operators
  - Combining tests using boolean operators
  - Blend together what you’ve learned!
  - Arranging your data
  - Reverse the order of arranging
- Summarise and the pipe operator
  - The syntax of summarise
  - Aggregate functions
  - dplyr aggregate functions
  - Overview of pipe operator syntax
  - Drive or fly?
  - Advanced piping
- Group_by and working with databases
  - Unite and conquer using group_by
  - Combine group_by with mutate
  - Advanced group_by
  - dplyr deals with different types
  - dplyr and mySQL databases
- Reference

Loading Libraries

```r
library(dplyr)
library(tidyr)
library(knitr)
library(printr)
```

Introduction to dplyr and tbls
Load the dplyr and hflights package

dplyr is an R package, a collection of functions and data sets that enhance the R language. Here will use dplyr to analyze a data set of airline flight data, containing flights that departed from Houston. This data is stored in a package called hflights. Below we load the hflights package. Now, a variable called hflights is available, a data.frame representing the data set.

```r
library(hflights)
str(hflights)
```

```
## 'data.frame': 227496 obs. of 21 variables:
## $ Year     : int 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011 ...
## $ Month    : int 1 1 1 1 1 1 1 1 1 1 ...
## $ DayofMonth: int 1 2 3 4 5 6 7 8 9 10 ...
## $ DayOfWeek: int 6 7 1 2 3 4 5 6 7 1 ...
## $ DepTime  : int 1400 1401 1352 1403 1405 1359 1359 1355 1443 1443 ...
## $ ArrTime  : int 1500 1501 1502 1513 1507 1503 1509 1454 1554 1554 ...
## $ UniqueCarrier: chr "AA" "AA" "AA" "AA" ...
## $ FlightNum: int 428 428 428 428 428 428 428 428 428 428 ...
## $ TailNum  : chr "N576AA" "N557AA" "N541AA" "N403AA" ...
## $ ActualElapsedTime: int 60 60 70 70 62 64 70 59 71 70 ...
## $ AirTime  : int 40 45 48 39 44 45 43 40 41 45 ...
## $ DepDelay : int 0 1 -8 3 5 -1 -5 43 43 ...
## $ Origin   : chr "IAH" "IAH" "IAH" "IAH" ...
## $ Dest     : chr "DFW" "DFW" "DFW" "DFW" ...
## $ Distance : int 224 224 224 224 224 224 224 224 224 224 ...
## $ TaxiIn   : int 7 6 5 9 6 12 7 8 6 ...
## $ TaxiOut  : int 13 9 17 22 9 13 15 12 22 19 ...
## $ Cancelled: int 0 0 0 0 0 0 0 0 0 0 ...
## $ CancellationCode: chr "" "" "" "" ...
## $ Diverted : int 0 0 0 0 0 0 0 0 0 0 ...
```

Convert data.frame to table

A tbl is just a special kind of data.frame. They make your data easier to look at, but also easier to work with. On top of this, a tbl is straightforwardly derived from a data.frame structure using tbl_df().

The tbl format changes how R displays your data, but it does not change the data's underlying data structure. A tbl inherits the original class of its input, in this case, a data.frame. This means
that you can still manipulate the tbl as if it were a data.frame; you can do anything with the `hflights` tbl that you could do with the `hflights` data.frame.

```r
# Convert the hflights data.frame into a hflights tbl
hflights <- tbl_df(hflights)
glimpse(hflights)
```

```r
## Observations: 227496
## Variables:
## $ Month             (int) 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
## $ DayofMonth        (int) 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 1,
## $ DayOfWeek         (int) 6, 7, 1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5,
## $ DepTime           (int) 1400, 1401, 1352, 1403, 1405, 1359, 1359,
## $ ArrTime           (int) 1500, 1501, 1502, 1503, 1504, 1505, 1505,
## $ UniqueCarrier     (chr) "AA", "AA", "AA", "AA", "AA", "AA", "AA",
## $ FlightNum         (int) 428, 428, 428, 428, 428, 428, 428, 428,
## $ TailNum           (chr) "N576AA", "N557AA", "N541AA", "N403AA",
## $ ActualElapsedTime (int) 60, 60, 70, 70, 62, 64, 70, 59, 71, 70,
## $ AirTime           (int) 40, 45, 48, 39, 45, 43, 40, 41, 45, 42,
## $ ArrDelay          (int) -10, -9, -8, 3, -3, -7, -1, -16, 44, 43,
## $ DepDelay          (int) 0, 1, -8, 3, 5, -1, -5, 43, 43, 29,
## $ Origin            (chr) "IAH", "IAH", "IAH", "IAH", "IAH", "IAH",
## $ Dest              (chr) "DFW", "DFW", "DFW", "DFW", "DFW", "DFW",
## $ Distance          (int) 224, 224, 224, 224, 224, 224, 224,
## $ TaxiIn            (int) 7, 6, 5, 9, 9, 6, 12, 7, 8, 6, 8, 4, 6,
## $ TaxiOut           (int) 13, 9, 17, 22, 9, 13, 15, 12, 22, 19, 20,
## $ Cancelled         (int) 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
## $ CancellationCode  (chr) ",", ",", ",", ",", ",", ",", ",", ",", 
## $ Diverted          (int) 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

**Changing labels of hflights**

You can “clean” `hflights` the same way you would clean a data.frame. A bit of cleaning would be a good idea since the `UniqueCarrier` variable of `hflights` uses a confusing code system.

You can create a lookup table with a named vector. When you subset the lookup table with a character string (like the character strings in `UniqueCarrier`), R will return the values of the lookup table that correspond to the names in the character string.

```r
# Build the lookup table
lut <- c("AA" = "American", "AS" = "Alaska", "B6" = "JetBlue", "CO" = "Continen"
# Use lut to translate the UniqueCarrier column of hflights

```r
hflights$UniqueCarrier <- lut[hflights$UniqueCarrier]
```

# Inspect the resulting raw values of your variables

```r
glimpse(hflights)
```

---

The five verbs and their meaning

The dplyr package contains five key data manipulation functions, also called verbs:

- **select()**, which returns a subset of the columns,
- **filter()**, that is able to return a subset of the rows,
- **arrange()**, that reorders the rows according to single or multiple variables,
- **mutate()**, used to add columns from existing data,
- **summarise()**, which reduces each group to a single row by calculating aggregate measures.
Below we explore each one in details.

**Select and mutate**

**Choosing is not loosing! The select verb**

To answer the simple question whether flight delays tend to shrink or grow during a flight, we can safely discard a lot of the variables of each flight. To select only the ones that matter, we can use `select()`. Its syntax is plain and simple:

```
select(data, Var1, Var2, ...)
```

The first argument being the `tbl` you want to select variables from and the `VarX` arguments the variables you want to retain. You can also use the `:` and `-` operators inside of select, similar to indexing a data.frame with hard brackets. `select()` lets you apply them to names as well as integer indexes. The `-` operator allows you to select everything except a column or a range of columns.

Below we select only four columns of the hflights dataset.

```
hflights_subset <- select(hflights, ActualElapsedTime, AirTime, ArrDelay, DepDelay)
kable(head(hflights_subset), align = 'c')
```

<table>
<thead>
<tr>
<th>ActualElapsedTime</th>
<th>AirTime</th>
<th>ArrDelay</th>
<th>DepDelay</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>40</td>
<td>-10</td>
<td>0</td>
</tr>
<tr>
<td>60</td>
<td>45</td>
<td>-9</td>
<td>1</td>
</tr>
<tr>
<td>70</td>
<td>48</td>
<td>-8</td>
<td>-8</td>
</tr>
<tr>
<td>70</td>
<td>39</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>62</td>
<td>44</td>
<td>-3</td>
<td>5</td>
</tr>
<tr>
<td>64</td>
<td>45</td>
<td>-7</td>
<td>-1</td>
</tr>
</tbody>
</table>

Below we demonstrate the most concise way to select: columns `Year` up to and including `DayOfWeek`, columns `ArrDelay` up to and including `Diverted`.

```
hflights_subset2 <- select (hflights, Year:DayOfWeek, ArrDelay:Diverted)
```

**Helper functions for variable selection**

dplyr comes with a set of helper functions that can help you select variables. These functions find
groups of variables to select, based on their names.

dplyr provides 6 helper functions, each of which only works when used inside `select()`:

- `starts_with("X")`: every name that starts with "X",
- `ends_with("X")`: every name that ends with "X",
- `contains("X")`: every name that contains "X",
- `matches("X")`: every name that matches "X", which can be a regular expression,
- `num_range("x", 1:5)`: the variables named x01, x02, x03, x04 and x05,
- `one_of(x)`: every name that appears in x, which should be a character vector.

Watch out: Surround character strings with quotes when you pass them to a helper function, but do not surround variable names with quotes if you are not passing them to a helper function.

Below are some example of helper functions:

```r
select (hflights, matches("ArrDelay"), matches("DepDelay"))

select(hflights, one_of(c("UniqueCarrier", "FlightNum", "TailNum", "Cancelled"))

select(hflights, ends_with("Time"), ends_with("Delay"))
```

**Comparison to basic R**

To see the added value of the dplyr package, it is useful to compare its syntax with basic R. Up to now, you have only considered functionality that is also available without the use of dplyr. However, the elegance and ease-of-use of dplyr should be clear from following short set of comparisons.

```r
ex1r <- hflights[c("TaxiIn","TaxiOut","Distance")]
ex1d <- select(hflights, TaxiIn, TaxiOut, Distance)

ex2r <- hflights[c("Year","Month","DayOfWeek","DepTime","ArrTime")]
ex2d <- select(hflights, Year:ArrTime, -DayofMonth)

ex3r <- hflights[c("TailNum","TaxiIn","TaxiOut")]
ex3d <- select(hflights, starts_with("ta"))
```

**Mutating is creating**

`mutate()` is the second of five data manipulation functions you will get familiar with in this course. In contrast to `select()`, which retains a subset of all variables, `mutate()` creates new columns which are added to a copy of the dataset.
Let's briefly recap the syntax:

\[
\text{mutate}(\text{data}, \text{Mutant1} = \text{expr}(\text{Var0, Var1, ...}))
\]

Here, \text{data} is the tbl you want to use to create new columns. The second argument is an expression that assigns the result of any R function using already existing variables \text{Var0, Var1, ...} to a new variable \text{Mutant1}.

Below are a couple of examples demonstrating the use of mutate function:

```
# Add the new variable ActualGroundTime to a copy of hflights and save the result as g1.
g1 <- mutate(hflights, ActualGroundTime = ActualElapsedTime - AirTime)

# Add the new variable GroundTime to a copy of g1 and save the result as g2.
g2 <- mutate(g1, GroundTime = TaxiIn + TaxiOut)

# Add the new variable AverageSpeed to a copy of g2 and save the result as g3.
g3 <- mutate(g2, AverageSpeed = Distance / AirTime * 60)
```

Add multiple variables using mutate

So far we've added variables to hflights one at a time, but you can also use \text{mutate()} to add multiple variables at once. To create more than one variable, place a comma between each variable that you define inside \text{mutate()}. Below we demonstrate how it can be done:

```
m1 <- mutate(hflights, loss = ArrDelay - DepDelay, loss_percent = (ArrDelay - DepDelay) / DepDelay)
m2 <- mutate(hflights, loss = ArrDelay - DepDelay, loss_percent = loss / DepDelay)
m3 <- mutate(hflights, TotalTaxi = TaxiIn + TaxiOut,
              ActualGroundTime = ActualElapsedTime - AirTime,
              Diff = TotalTaxi - ActualGroundTime)
```

Filter and arrange

Logical operators

R comes with a set of logical operators that you can use to extract rows with \text{filter()}. These operators are

- \text{\texttt{x < y}}, TRUE if \text{x} is less than \text{y}
- \( x <= y \), TRUE if \( x \) is less than or equal to \( y \)
- \( x == y \), TRUE if \( x \) equals \( y \)
- \( x != y \), TRUE if \( x \) does not equal \( y \)
- \( x >= y \), TRUE if \( x \) is greater than or equal to \( y \)
- \( x > y \), TRUE if \( x \) is greater than \( y \)
- \( x \; \%in\% \{a, b, c\} \), TRUE if \( x \) is in the vector \{a, b, c\}

Examples:

```r
# All flights that traveled 3000 miles or more.
f1 <- filter(hflights, Distance >= 3000)

# All flights flown by one of JetBlue, Southwest, or Delta airlines
f2 <- filter(hflights, UniqueCarrier %in% c("JetBlue", "Southwest", "Delta"))

# All flights where taxiing took longer than flying
f3 <- filter(hflights, TaxiIn + TaxiOut > AirTime)
```

Combining tests using boolean operators

R also comes with a set of boolean operators that you can use to combine multiple logical tests into a single test. These include \& \text{,} | \text{,} and \!, respectively the and, or and not operators.

You can thus use R’s \& operator to combine logical tests in filter(), but that is not necessary. If you supply filter() with multiple tests separated by commas, it will return just the rows that satisfy each test (as if the tests were joined by an \& operator).

Finally, filter() makes it very easy to screen out rows that contain NA’s, R’s symbol for missing information. You can identify an NA with the is.na() function.

Examples:

```r
# all flights that departed before 5am or arrived after 10pm.
f1 <- filter(hflights, DepTime < 500 | ArrTime > 2200)

# all flights that departed late but arrived ahead of schedule
f2 <- filter(hflights, DepDelay > 0, ArrDelay < 0)

# all cancelled weekend flights
f3 <- filter(hflights, DayOfWeek %in% c(6,7), Cancelled == 1)

# all flights that were cancelled after being delayed
f4 <- filter(hflights, Cancelled == 1, DepDelay > 0)
```
Blend together what you've learned!

Let's generate a new database from the `hflights` database that contains some useful information on flights that had JFK airport as their destination. We will need `select()`, `mutate()`, as well as `filter()`.

```r
# Select the flights that had JFK as their destination
c1 <- filter(hflights, Dest == "JFK")

# Combine the Year, Month and DayofMonth variables to create a Date column
c2 <- mutate(c1, Date = paste(Year, Month, DayofMonth, sep = "-"))

# Retain only a subset of columns to provide an overview
C3 <- select(c2, Date, DepTime, ArrTime, TailNum)
kable(head(c3), align = 'c')
```

<table>
<thead>
<tr>
<th>Date</th>
<th>DepTime</th>
<th>ArrTime</th>
<th>TailNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-1-1</td>
<td>654</td>
<td>1124</td>
<td>N324JB</td>
</tr>
<tr>
<td>2011-1-1</td>
<td>1639</td>
<td>2110</td>
<td>N324JB</td>
</tr>
<tr>
<td>2011-1-2</td>
<td>703</td>
<td>1113</td>
<td>N324JB</td>
</tr>
<tr>
<td>2011-1-2</td>
<td>1604</td>
<td>2040</td>
<td>N324JB</td>
</tr>
<tr>
<td>2011-1-3</td>
<td>659</td>
<td>1100</td>
<td>N229JB</td>
</tr>
<tr>
<td>2011-1-3</td>
<td>1801</td>
<td>2200</td>
<td>N206JB</td>
</tr>
</tbody>
</table>

Another example: How many weekend flights flew a distance of more than 1000 miles but had a total taxiing time below 15 minutes?

```r
filter(hflights, DayOfWeek %in% c(6,7), Distance > 1000, TaxiIn + TaxiOut < 15)
```

Arranging your data

The syntax of `arrange()` is the following:

```r
arrange(data, Var0, Var1, ...)
```

Here, `data` is again the tbl you’re working with and `Var0, Var1, ...` are the variables according to which you arrange. When `Var0` does not provide closure on the order, `Var1` and possibly additional variables will serve as tie breakers to decide the arrangement.
`arrange()` can be used to rearrange rows according to any type of data. If you pass `arrange()` a character variable, for example, R will rearrange the rows in alphabetical order according to values of the variable. If you pass a factor variable, R will rearrange the rows according to the order of the levels in your factor (running `levels()` on the variable reveals this order).

Examples:

```r
dtc <- filter(hflights, Cancelled == 1, !is.na(DepDelay))

# Arrange dtc by departure delays
a1 <- arrange(dtc, DepDelay)

# Arrange dtc so that cancellation reasons are grouped
a2 <- arrange(dtc, CancellationCode)

# Arrange according to carrier and departure delays
a3 <- arrange(hflights, UniqueCarrier, DepDelay)
```

Reverse the order of arranging

By default, `arrange()` arranges the rows from smallest to largest. Rows with the smallest value of the variable will appear at the top of the data set. You can reverse this behavior with the `desc()` function. `arrange()` will reorder the rows from largest to smallest values of a variable if you wrap the variable name in `desc()` before passing it to `arrange()`.

Examples:

```r
# Arrange according to carrier and decreasing departure delays
a1 <- arrange(hflights, UniqueCarrier, desc(DepDelay))

# Arrange flights by total delay (normal order).
# a2 <- arrange(hflights, DepDelay + ArrDelay)

# Keep flights leaving to DFW before 8am and arrange according to decreasing AirTime
a3 <- arrange(filter(hflights,Dest=="DFW" & DepTime < 800),desc(AirTime))
```

Summarise and the pipe operator

The syntax of summarise
**summarise()**, the last of the 5 verbs, follows the same syntax as **mutate()**, but the resulting dataset consists of a single row instead of an entire new column in the case of **mutate()**. Below, a typical **summarise()** function is repeated to show the syntax, without going into detail on all arguments:

```r
summarise(data, sumvar = sum(A),
           avgvar = avg(B))
```

In contrast to the four other data manipulation functions, **summarise()** does not return a copy of the dataset it is summarizing; instead, it builds a new dataset that contains only the summarizing statistics.

Examples:

```r
# Determine the shortest and longest distance flown and save statistics to min_dist and max_dist resp.
s1 <- summarise(hflights,
                min_dist = min(Distance),
                max_dist = max(Distance))

# Determine the longest distance for diverted flights, save statistic to max_div. Use a one-liner!
s2 <- summarise(filter(hflights, Diverted==1),
                max_div = max(Distance))
```

**Aggregate functions**

You can use any function you like in **summarise()**, so long as the function can take a vector of data and return a single number. R contains many *aggregating functions*, as `dplyr` calls them. Here are some of the most useful:

- `min(x)` - minimum value of vector `x`.
- `max(x)` - maximum value of vector `x`.
- `mean(x)` - mean value of vector `x`.
- `median(x)` - median value of vector `x`.
- `quantile(x, p)` - pth quantile of vector `x`.
- `sd(x)` - standard deviation of vector `x`.
- `var(x)` - variance of vector `x`.
- `IQR(x)` - Inter Quartile Range (IQR) of vector `x`.
- `diff(range(x))` - total range of vector `x`.

Examples:

```r
# Calculate summarizing statistics for flights that have an ArrDelay that is not NA
temp1 <- filter(hflights, !is.na(ArrDelay))
```
s1 <- summarise(temp1,
   earliest = min(ArrDelay),
   average = mean(ArrDelay),
   latest = max(ArrDelay),
   sd = sd(ArrDelay))

kable(head(s1), align = 'c')

<table>
<thead>
<tr>
<th>earliest</th>
<th>average</th>
<th>latest</th>
<th>sd</th>
</tr>
</thead>
<tbody>
<tr>
<td>-70</td>
<td>7.094334</td>
<td>978</td>
<td>30.70852</td>
</tr>
</tbody>
</table>

# Calculate the maximum taxiing difference for flights that have taxi data available

temp2 <- filter(hflights, !is.na(TaxiIn), !is.na(TaxiOut))
s2 <- summarise(temp2, max_taxi_diff = max(abs(TaxiIn - TaxiOut)))

print(s2)

## Source: local data frame [1 x 1]
##
##   max_taxi_diff
## 1           160

dplyr aggregate functions

dplyr provides several helpful aggregate functions of its own, in addition to the ones that are already defined in R. These include:

- first(x) - The first element of vector x.
- last(x) - The last element of vector x.
- nth(x, n) - The nth element of vector x.
- n() - The number of rows in the data.frame or group of observations that summarise() describes.
- n_distinct(x) - The number of unique values in vector x.

Next to these dplyr-specific functions, you can also turn a logical test into an aggregating function with sum() or mean(). A logical test returns a vector of TRUE's and FALSE's. When you apply sum() or mean() to such a vector, R coerces each TRUE to a 1 and each FALSE to a 0. This allows you to find the total number or proportion of observations that passed the test, respectively.

Examples:

# Calculate the summarizing statistics of hflights
\[
\begin{align*}
\text{s1} & \leftarrow \text{summarise(hflights, n_obs = n(),} \\
& \quad \text{n_carrier = n_distinct(UniqueCarrier),} \\
& \quad \text{n_dest = n_distinct(Dest),} \\
& \quad \text{dest100 = nth(Dest, 100))}
\end{align*}
\]

\[
\begin{array}{cccc}
\text{n_obs} & \text{n_carrier} & \text{n_dest} & \text{dest100} \\
227496 & 15 & 116 & \text{DFW}
\end{array}
\]

# Calculate the summarizing statistics for flights flown by American Airlines
\[
\begin{align*}
\text{aa} & \leftarrow \text{filter(hflights, UniqueCarrier == "American")} \\
\text{s2} & \leftarrow \text{summarise(aa, n_flights = n(),} \\
& \quad \text{n_canc = sum(Cancelled == 1),} \\
& \quad \text{p_canc = mean(Cancelled == 1) * 100,} \\
& \quad \text{avg_delay = mean(ArrDelay, na.rm = TRUE))}
\end{align*}
\]

\[
\begin{array}{ccccc}
\text{n_flights} & \text{n_canc} & \text{p_canc} & \text{avg_delay} \\
3244 & 60 & 1.849568 & 0.8917558
\end{array}
\]

**Overview of pipe operator syntax**

Using the pipe operator `%>%` the following two statements are completely analogous:

\[
\begin{align*}
\text{mean(c(1, 2, 3, NA), na.rm = TRUE)} \\
\text{c(1, 2, 3, NA) %>% mean(na.rm = TRUE)}
\end{align*}
\]

The `%>%` operator allows you to extract the first argument of a function from the arguments list and put it in front of it, thus solving the *Dagwood sandwich problem*.

**Example:**

\[
\begin{align*}
\text{# Write the 'piped' version of the English sentences.} \\
\text{p} & \leftarrow \text{hflights %>%} \\
& \quad \text{mutate(diff = TaxiOut - TaxiIn) %>%} \\
& \quad \text{filter(!is.na(diff)) %>%} \\
& \quad \text{summarise(avg = mean(diff))}
\end{align*}
\]

\[
\text{print(p)}
\]
Drive or fly?

You can answer sophisticated questions by combining the verbs of dplyr. Over the next few examples we will examine whether it sometimes makes sense to drive instead of fly. We will begin by making a data set that contains relevant variables. Then, we find flights whose equivalent average velocity is lower than the velocity when traveling by car.

Example:

```r
# Part 1, concerning the selection and creation of columns
d <- hflights %>%
  select(Dest, UniqueCarrier, Distance, ActualElapsedTime) %>%
  mutate(RealTime = ActualElapsedTime + 100, mph = Distance / RealTime * 60)
kable(head(d), align = 'c')
```

<table>
<thead>
<tr>
<th>Dest</th>
<th>UniqueCarrier</th>
<th>Distance</th>
<th>ActualElapsedTime</th>
<th>RealTime</th>
<th>mph</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFW</td>
<td>American</td>
<td>224</td>
<td>60</td>
<td>160</td>
<td>84.0000</td>
</tr>
<tr>
<td>DFW</td>
<td>American</td>
<td>224</td>
<td>60</td>
<td>160</td>
<td>84.0000</td>
</tr>
<tr>
<td>DFW</td>
<td>American</td>
<td>224</td>
<td>70</td>
<td>170</td>
<td>79.0588</td>
</tr>
<tr>
<td>DFW</td>
<td>American</td>
<td>224</td>
<td>70</td>
<td>170</td>
<td>79.0588</td>
</tr>
<tr>
<td>DFW</td>
<td>American</td>
<td>224</td>
<td>62</td>
<td>162</td>
<td>82.9629</td>
</tr>
<tr>
<td>DFW</td>
<td>American</td>
<td>224</td>
<td>64</td>
<td>164</td>
<td>81.9512</td>
</tr>
</tbody>
</table>

# Part 2, concerning flights that had an actual average speed of < 70 mph.

d %>%
  filter(!is.na(mph), mph < 70) %>%
  summarise(n_less = n(),
            n_dest = n_distinct(Dest),
            min_dist = min(Distance),
            max_dist = max(Distance))

<table>
<thead>
<tr>
<th>n_less</th>
<th>n_dest</th>
<th>min_dist</th>
<th>max_dist</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The previous example suggested that some flights might be less efficient than driving in terms of speed. But is speed all that matters? Flying imposes burdens on a traveler that driving does not. For example, airplane tickets are very expensive. Air travelers also need to limit what they bring on their trip and arrange for a pick up or a drop off. Given these burdens we might demand that a flight provide a large speed advantage over driving.

Example:

```r
# Solve the exercise using a combination of dplyr verbs and %>%
hflights %>%
  select(Dest, Cancelled, Distance, ActualElapsedTime, Diverted) %>%
  mutate(RealTime = ActualElapsedTime + 100, mph = Distance / RealTime *
        filter(mph < 105 | Cancelled == 1 | Diverted == 1) %>%
        summarise( n_non = n(),
                    p_non = n_non / nrow(hflights) * 100,
                    n_dest = n_distinct(Dest),
                    min_dist = min(Distance),
                    max_dist = max(Distance))
```

```
 n_non  p_non  n_dest  min_dist  max_dist
42400 18.63769 113 79 3904
```

Advanced piping

One more example in using piping operator.

```r
# Count the number of overnight flights
hflights %>%
  filter(!is.na(DepTime), !is.na(ArrTime), DepTime > ArrTime) %>%
  summarise(n = n())
```

Group_by and working with databases

Unite and conquer using group_by

`group_by()` lets you define groups within your data set. Its influence becomes clear when calling on a grouped dataset: summarizing statistics are calculated for the different groups separately.
The syntax for this function is again straightforward:

```r
group_by(data, Var0, Var1, ...)
```

Here, `data` is the tbl dataset you work with, and `Var0, Var1, ...` are the variables you want to group by. If you pass on several variables as arguments, the number of separate sets of grouped observations will increase, but their size will decrease.

Example:

```r
# Make the calculations to end up with ordered statistics per carrier
hflights %>%
  group_by(UniqueCarrier) %>%
  summarise(n_flights = n(),
    n_canc = sum(Cancelled == 1),
    p_canc = mean(Cancelled == 1) * 100,
    avg_delay = mean(ArrDelay, na.rm = TRUE)) %>%
  arrange(avg_delay, p_canc)
```

<table>
<thead>
<tr>
<th>UniqueCarrier</th>
<th>n_flights</th>
<th>n_canc</th>
<th>p_canc</th>
<th>avg_delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>US_Airways</td>
<td>4082</td>
<td>46</td>
<td>1.1268986</td>
<td>-0.6307692</td>
</tr>
<tr>
<td>American</td>
<td>3244</td>
<td>60</td>
<td>1.8495684</td>
<td>0.8917558</td>
</tr>
<tr>
<td>AirTran</td>
<td>2139</td>
<td>21</td>
<td>0.9817672</td>
<td>1.8536239</td>
</tr>
<tr>
<td>Alaska</td>
<td>365</td>
<td>0</td>
<td>0.0000000</td>
<td>3.1923077</td>
</tr>
<tr>
<td>Mesa</td>
<td>79</td>
<td>1</td>
<td>1.2658228</td>
<td>4.0128205</td>
</tr>
<tr>
<td>Delta</td>
<td>2641</td>
<td>42</td>
<td>1.5903067</td>
<td>6.0841374</td>
</tr>
<tr>
<td>Continental</td>
<td>70032</td>
<td>475</td>
<td>0.6782614</td>
<td>6.0986983</td>
</tr>
<tr>
<td>American_Eagle</td>
<td>4648</td>
<td>135</td>
<td>2.9044750</td>
<td>7.1529751</td>
</tr>
<tr>
<td>Atlantic_Southeast</td>
<td>2204</td>
<td>76</td>
<td>3.4482759</td>
<td>7.2569543</td>
</tr>
<tr>
<td>Southwest</td>
<td>45343</td>
<td>703</td>
<td>1.5504047</td>
<td>7.5871430</td>
</tr>
<tr>
<td>Frontier</td>
<td>838</td>
<td>6</td>
<td>0.7159905</td>
<td>7.6682692</td>
</tr>
<tr>
<td>ExpressJet</td>
<td>73053</td>
<td>1132</td>
<td>1.5495599</td>
<td>8.1865242</td>
</tr>
<tr>
<td>SkyWest</td>
<td>16061</td>
<td>224</td>
<td>1.3946828</td>
<td>8.6934922</td>
</tr>
<tr>
<td>JetBlue</td>
<td>695</td>
<td>18</td>
<td>2.5899281</td>
<td>9.8588410</td>
</tr>
<tr>
<td>United</td>
<td>2072</td>
<td>34</td>
<td>1.6409266</td>
<td>10.4628628</td>
</tr>
</tbody>
</table>

# Answer the question: Which day of the week is average total taxiing time hig
hflights %>%
group_by(DayOfWeek) %>%
summarise(avg_taxi = mean(TaxiIn + TaxiOut, na.rm=TRUE)) %>%
arrange(desc(avg_taxi))

<table>
<thead>
<tr>
<th>DayOfWeek</th>
<th>avg_taxi</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21.77027</td>
</tr>
<tr>
<td>2</td>
<td>21.43505</td>
</tr>
<tr>
<td>4</td>
<td>21.26076</td>
</tr>
<tr>
<td>3</td>
<td>21.19055</td>
</tr>
<tr>
<td>5</td>
<td>21.15805</td>
</tr>
<tr>
<td>7</td>
<td>20.93726</td>
</tr>
<tr>
<td>6</td>
<td>20.43061</td>
</tr>
</tbody>
</table>

**Combine group_by with mutate**

You can also combine `group_by()` with `mutate()`. When you mutate grouped data, `mutate()` will calculate the new variables independently for each group. This is particularly useful when `mutate()` uses the `rank()` function, which calculates within group rankings. `rank()` takes a group of values and calculates the rank of each value within the group, e.g.

```r
rank(c(21, 22, 24, 23))
```

has output

```
[1] 1 2 4 3
```

As with `arrange()`, `rank()` ranks values from the largest to the smallest and this behavior can be reversed with the `desc()` function.

Example:

```r
# Part 1
hflights %>%
group_by(UniqueCarrier) %>%
filter(!is.na(ArrDelay)) %>%
summarise(p_delay = mean(ArrDelay > 0)) %>%
mutate(rank = rank(p_delay)) %>%
arrange(rank)
```
# Part 2

```
hflights %>%
  group_by(UniqueCarrier) %>%
  filter(!is.na(ArrDelay), ArrDelay > 0) %>%
  summarise(avg = mean(ArrDelay)) %>%
  mutate(rank = rank(avg)) %>%
  arrange(rank)
```

<table>
<thead>
<tr>
<th>UniqueCarrier</th>
<th>avg</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesa</td>
<td>18.67</td>
<td>1</td>
</tr>
<tr>
<td>Frontier</td>
<td>18.69</td>
<td>2</td>
</tr>
<tr>
<td>US_Airways</td>
<td>20.70</td>
<td>3</td>
</tr>
<tr>
<td>Continental</td>
<td>22.13</td>
<td>4</td>
</tr>
<tr>
<td>Alaska</td>
<td>22.91</td>
<td>5</td>
</tr>
<tr>
<td>SkyWest</td>
<td>24.14</td>
<td>6</td>
</tr>
<tr>
<td>ExpressJet</td>
<td>24.19</td>
<td>7</td>
</tr>
</tbody>
</table>
## Advanced `group_by`

This section is an all-encompassing review of the concepts in `dplyr`.

```r
# Which plane (by tail number) flew out of Houston the most times? How many times?
adv1 <- hflights %>%
  group_by(TailNum) %>%
  summarise(n = n()) %>%
  filter(n == max(n))

# How many airplanes only flew to one destination from Houston? adv2
adv2 <- hflights %>%
  group_by(TailNum) %>%
  summarise(ndest = n_distinct(Dest)) %>%
  filter(ndest == 1) %>%
  summarise(nplanes = n())

# Find the most visited destination for each carrier: adv3
adv3 <- hflights %>%
  group_by(UniqueCarrier, Dest) %>%
  summarise(n = n()) %>%
  mutate(rank = rank(desc(n))) %>%
  filter(rank == 1)

# Find the carrier that travels to each destination the most: adv4
adv4 <- hflights %>%
  group_by(Dest, UniqueCarrier) %>%
  summarise(n = n()) %>%
  mutate(rank = rank(desc(n))) %>%
  filter(rank == 1)
```
dplyr deals with different types

For this section `hflights2` is a copy of `hflights` that is saved as a data table using following code:

```r
library(data.table)
hflights2 <- as.data.table(hflights)
```

`hflights2` contains all of the same information as `hflights`, but the information is stored in a different data structure.

Even though `hflights2` is a different data structure, you can use the same dplyr functions to manipulate `hflights2` as you used to manipulate `hflights`.

Example:

```r
# Use summarise to calculate n_carrier
s2 <- hflights2 %>%
    summarise(n_carrier = n_distinct(UniqueCarrier))
```

dplyr and mySQL databases

`nycflights` is a mySQL database exist on the DataCamp server. It contains information about flights that departed from New York City in 2013. This data is similar to the data in `hflights`, but it does not contain information about cancellations or diversions (you can access the same data in the `nycflights13` R package).

`nycflights`, an R object that stores a connection to the nycflights tbl that lives outside of R on the datacamp server, will be created for you on the right. You can use such connection objects to pull data from databases into R. This lets you work with datasets that are too large to fit in R.

You can learn a connection language to make sophisticated queries from such a database, or you can simply use dplyr. When you run a dplyr command on a database connection, dplyr will convert the command to the database's native language and do the query for you. As such, just the data that you need from the database will be retrieved. This will usually be a fraction of the total data, which will fit in R withouth memory issues.

For example, we can easily retrieve a summary of how many carriers and how many flights flew in and out of New York City in 2013 with the code (note that in `nycflights`, the `UniqueCarrier` variable is named `carrier`):

```r
summarise(nycflights,
    n_carriers = n_distinct(carrier),
```
```r
n_flights = n()

Exmaple:

```r
# set up a src that connects to the mysql database (src_mysql is provided by dplyr)
my_db <- src_mysql(dbname = "dpilyr",
                   host = "dpilyr.csrrinzquibik.us-east-1.rds.amazonaws.com",
                   port = 3306, user = "dpilyr", password = "dpilyr")

# and reference a table within that src
nycflights <- tbl(my_db, "dpilyr")

# nycflights is now available as an R object that references to the remote nycflights table.

glimpse(nycflights)

```r
## Observations: 336776  
## Variables:  
## $ id (int) 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,...  
## $ month (int) 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...  
## $ day (int) 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...  
## $ dep_time (int) 517, 533, 542, 544, 554, 554, 555, 557, 557, 558,...  
## $ dep_delay (int) 2, 4, 2, -1, -6, -4, -5, -3, -2, -2, -2, -2, -2,...  
## $ arr_time (int) 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753,...  
## $ arr_delay (int) 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3,...  
## $ carrier (chr) "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6"...  
## $ tailnum (chr) "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N...  
## $ flight (int) 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301,...  
## $ origin (chr) "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA...  
## $ dest (chr) "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IA...  
## $ air_time (int) 227, 227, 160, 183, 116, 150, 158, 53, 140, 138,...  
## $ distance (int) 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944,...  
## $ hour (int) 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,...  
## $ minute (int) 17, 33, 42, 44, 54, 54, 55, 57, 57, 58, 58, 58, 58,...  

# Calculate the grouped summaries detailed in the instructions.

```r
dbsumm <- nycflights %>%
  group_by(carrier) %>%
  summarise(n_flights = n(), avg_delay = mean(arr_delay)) %>%
```
```r
arrange(avg_delay)
```

## Source: mysql 5.6.21-log [dplyr@dplyr.csrrinzqubik.us-east-1.rds.amazonaws.com:/dplyr]
## From: <derived table> [?? x 3]
## Arrange: avg_delay

## Warning in .local(conn, statement, ...): Decimal MySQL column 2 imported as numeric

```r
## carrier n_flights avg_delay
## 1 AS 714 -9.8613
## 2 HA 342 -6.9152
## 3 AA 32729 0.3556
## 4 DL 48110 1.6289
## 5 VX 5162 1.7487
## 6 US 20536 2.0565
## 7 UA 58665 3.5045
## 8 9E 18460 6.9135
## 9 B6 54635 9.3565
## 10 WN 12275 9.4675
## .. ... ... ...
```

### Reference

- R & Data Science courses offered thorough DataCamp.