Data analysis with dplyr

About the rest of this tutorial:

There are a million different ways to do things in R. This isn't Python, where solutions on StackOverflow get ranked on how "Pythonic" they are. If there's something you like about another workflow in R, there's nothing stopping you from using it!

In this case, there are three main camps on analyzing dataframes in R:

- **"Base R"** - "Base R" means using only functions and stuff built into your base R installation. No external packages or fancy stuff. The focus here is on stability from version to version - your code will never break from an update, but performance and usability aren't always as great.
- **`data.table`** - `data.table` is a dataframe manipulation package known to have very good performance.
- **"The tidyverse"** - The "tidyverse" is a collection of packages that overhauls just about everything in R to use a consistent API. Has comparable performance with `data.table`.

For much of the rest of this tutorial, we'll focus on doing things the "tidyverse" way (with a few exceptions). The biggest reasons is that everything follows a consistent API - everything in the tidyverse works well together. You can often guess how to use a new function because you've used others like it. It's also got pretty great performance. When you use stuff from the tidyverse, you can be reasonably confident that someone has already taken a look at optimizing things to speed things along.

Logical indexing

So far, we've covered how to extract certain pieces of data via indexing. But what we've shown so far only works if we know the exact index of the data we want (`vector[42]`, for example). There is a neat trick to extra certain pieces of data in R known as "logical indexing".

Before we start, we need to know a little about comparing things.

`==` is the equality operator in R.

```
1 == 1
```

```
## [1] TRUE
```

`!` means "not". Not `TRUE` is `FALSE`.

```
!TRUE
```

```
## [1] FALSE
```

Likewise we can check if something is not equal to something else with `!=`

```
TRUE != TRUE
```

```
## [1] FALSE
```

We can also make comparisons with the greater than `>` and less than `<` symbols. Pairing these with an equals sign means "greater than or equal to" (`>=`) or "less than or equal to" (`<=`).

```
4 < 5
```

```
## [1] TRUE
```

```
5 <= 5
```

```
## [1] TRUE
```

```
9 > 999
```

```
## [1] FALSE
```

```
TRUE >= FALSE
```

```
## [1] TRUE
```

The last example worked because `TRUE` and `FALSE` are equal to 1 and 0, respectively.

```
TRUE == 1
```

```
## [1] TRUE
```

```
FALSE == 1
```

```
## [1] FALSE
```

We can even compare strings:

```
"a" == "a"
```

```
## [1] TRUE
```

```
"a" != "b"
```

```
## [1] TRUE
```

This trick also works with vectors, returning `TRUE` or `FALSE` for every element in the vector.

```
example <- 1:7
example >= 4
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

```
another_example <- c("apple", "banana", "banana")
another_example == "banana"
```

```
## [1] FALSE  TRUE  TRUE
```

This trick is ***extremely useful*** for getting specific elements. Watch what happens when we index a vector using a set of boolean values. Using our example from above:

```
example
```

```
## [1] 1 2 3 4 5 6 7
```

```
greater_than_3 <- example > 3
greater_than_3
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

```
example[greater_than_3]
```

```
## [1] 4 5 6 7
```

This can be turned into a one-liner by putting the boolean expression inside the square brackets.

```
example[example > 3]
```

```
## [1] 4 5 6 7
```

We can also get the elements which were not greater than 3 by adding an `!` in front.

```
example[!example > 3]
```

```
## [1] 1 2 3
```

Exercise - Removing NAs from a dataset:

Logical indexing is also a pretty neat trick for removing `NA`s from a vector. Many functions will refuse to work on data with `NA`s present. The `is.na()` function returns `TRUE` or `FALSE` depending on if a value is `NA`.

Using this info, make the following return a number as a result instead of `NA`.

```
ugly_data <- c(1, NA, 5, 7, NA, NA)
mean(ugly_data)
```

```
## [1] NA
```

Exercise - The `na.rm` argument:

Many functions have an `na.rm` argument used to ignore `NA` values. Does this work for `mean()` in the previous example?

Retrieving rows from dataframes

Let's try this out on a bigger dataset. `nycflights13` is an example dataset containing all outbound flights from NYC in 2013. You can get this dataset with `install.packages("nycflights13")`.

Let's take a look at the dataset and see what we've got.

```
library(nycflights13)
head(flights)  # shows the top few rows of a dataset
```

```
## # A tibble: 6 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     1     1      517            515         2      830
## 2  2013     1     1      533            529         4      850
## 3  2013     1     1      542            540         2      923
## 4  2013     1     1      544            545        -1     1004
## 5  2013     1     1      554            600        -6      812
## 6  2013     1     1      554            558        -4      740
## # ... with 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

```
str(flights)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    336776 obs. of  19 variables:
##  $ year          : int  2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
##  $ month         : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ day           : int  1 1 1 1 1 1 1 1 1 1 ...
```

```
## $ dep_time      : int  517 533 542 544 554 554 555 557 557 558 ...
## $ sched_dep_time: int  515 529 540 545 600 558 600 600 600 600 ...
## $ dep_delay     : num  2 4 2 -1 -6 -4 -5 -3 -3 -2 ...
## $ arr_time      : int  830 850 923 1004 812 740 913 709 838 753 ...
## $ sched_arr_time: int  819 830 850 1022 837 728 854 723 846 745 ...
## $ arr_delay     : num  11 20 33 -18 -25 12 19 -14 -8 8 ...
## $ carrier       : chr  "UA" "UA" "AA" "B6" ...
## $ flight        : int  1545 1714 1141 725 461 1696 507 5708 79 301 ...
## $ tailnum       : chr  "N14228" "N24211" "N619AA" "N804JB" ...
## $ origin        : chr  "EWR" "LGA" "JFK" "JFK" ...
## $ dest          : chr  "IAH" "IAH" "MIA" "BQN" ...
## $ air_time      : num  227 227 160 183 116 150 158 53 140 138 ...
## $ distance      : num  1400 1416 1089 1576 762 ...
## $ hour          : num  5 5 5 5 6 5 6 6 6 6 ...
## $ minute        : num  15 29 40 45 0 58 0 0 0 0 ...
## $ time_hour     : POSIXct, format: "2013-01-01 05:00:00" "2013-01-01 05:00:00" ...
```

```
dim(flights)
```

```
## [1] 336776     19
```

A note about tbl_dfs:

`flights` is an example of a "tibble" or `tbl_df`. `tbl_df`s are identical to dataframes for most purposes, but they print out differently (notice how we didn't get all of the columns!).

```
class(flights)
```

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

To force a `tbl_df` to print all columns, you can use `print(some_tbl_df, width=Inf)`

If we ever get annoyed with a `tbl_df`, we can turn it back into a dataframe with `as.data.frame()`.

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

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

The `flights` table clocks in at several hundred thousand rows. That's a fair sized chunk of data. Nevertheless, our tricks from before work just the same.

Using the same technique from before, let's retrieve all of the flights that went to Los Angeles (LAX).

```
rows_with_yvr <- flights$dest == "LAX"
flights[rows_with_yvr, ]
```

```
## # A tibble: 16,174 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      558            600        -2      924
##  2  2013     1     1      628            630        -2     1016
##  3  2013     1     1      658            700        -2     1027
##  4  2013     1     1      702            700         2     1058
##  5  2013     1     1      743            730        13     1107
##  6  2013     1     1      828            823         5     1150
##  7  2013     1     1      829            830        -1     1152
##  8  2013     1     1      856            900        -4     1226
##  9  2013     1     1      859            900        -1     1223
## 10  2013     1     1      921            900        21     1237
## # ... with 16,164 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

```
# and the same, but in one line
result <- flights[flights$dest == "LAX", ]
# checking our work... we should only see "LAX" here
unique(result$dest)
```

```
## [1] "LAX"
```

```
# how many results did we get
nrow(result)
```

```
## [1] 16174
```

Breaking things apart, we look for all instances where the column `dest` was equal to “LAX”. We end up with a vector of whether or not “LAX” was found in each row. We can then use the square brackets to extract every row where the vector is true. Note the addition of a comma in our square brackets. `flights` has 2 dimensions, so our indexing needs to as well!

If we don’t add the comma, R gets upset:

```
flights[flights$dest]
```

```
Error: Length of logical index vector must be 1 or 19 (the number of rows), not 336776
```

One other issue - what happens if we want to grab the flights to either LAX or SEA (Seattle). Let’s try the following:

```
result <- flights[flights$dest == c("LAX", "SEA"), ]
unique(result$dest)
```

```
## [1] "LAX" "SEA"
```

```
nrow(result)
```

```
## [1] 10060
```

Though in both cases we got results corresponding to the cities we wanted, it looks like somethig went wrong. Before, we got 16174 results for just “LAX”. Now we only get 10060, and we even added an extra city worth of flights! So what’s happening here?

When R compares two vectors of different length, it “recycles” the shorter vector until it matches the length of the longer one!

Using a smaller example, this is what just happened:

```
long <- c(1, 1, 1, 2, 2, 2, 3)
short <- c(1, 2)
long == short
```

```
## Warning in long == short: longer object length is not a multiple of shorter
## object length
```

```
## [1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE
```

```
# what R is really doing behind the scenes
short_recycled <- c(1, 2, 1, 2, 1, 2, 1)
long == short_recycled
```

```
## [1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE
```

This is not what we want. We want to know if elements in the long vector were found “in” the shorter vector, not whether or not the two are equal at every point. Fortunately, there is a special `%in%` operator that does just that.

```
long %in% short
```

```
## [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

```
# and using that to subset values
long[long %in% short]
```

```
## [1] 1 1 1 2 2 2
```

If we take the `%in%` operator and apply it to our issue, we get the correct number of rows.

```
res <- flights[flights$dest %in% c("SEA", "LAX"), ]
nrow(res)
```

```
## [1] 20097
```

```
# our results contain the same number of flights bound for LAX
nrow(res[flights$dest == "LAX", ])
```

```
## [1] 16174
```

Filtering rows with `dplyr`

Up to this point, we've done everything using base R. Our code has a lot of crazy symbols in it, and isn't that readable for the average person. It's also not that fun to type out.

Let's try things the "tidyverse" way using `dplyr` (`dplyr` is a package that comes as part of the `tidyverse` package bundle).

To filter out a set of specific rows that match a condition, we use the `filter()` function. The syntax of this function is a bit unusual:

```
library(tidyverse)
```

```
## Loading tidyverse: ggplot2
## Loading tidyverse: tibble
## Loading tidyverse: tidyr
## Loading tidyverse: readr
## Loading tidyverse: purrr
## Loading tidyverse: dplyr
```

```
## Conflicts with tidy packages ----------------------------------------------
```

```
## filter(): dplyr, stats
## lag():    dplyr, stats
```

```
results <- filter(flights, dest == "LAX")
nrow(results)
```

```
## [1] 16174
```

Notice how we just used `dest` all by itself. `filter()` is smart enough to figure out that `dest` is a column name in the `flights` dataframe.

We can also filter multiple things at once using the `&` (AND) and `|` (OR) operators. `&` checks if both conditions are true, `|` checks if just one condition is true:

```
TRUE & TRUE
```

```
## [1] TRUE
```

```
TRUE & FALSE
```

```
## [1] FALSE
```

```
TRUE | FALSE
```

```
## [1] TRUE
```

Using this in an example with `filter()` to fetch all the flights to LAX in February:

```
filter(flights, dest == "LAX" & month == 2)
```

```
## # A tibble: 1,030 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     2     1      554            601        -7      920
##  2  2013     2     1      654            700        -6     1032
##  3  2013     2     1      657            705        -8     1027
##  4  2013     2     1      658            700        -2     1018
##  5  2013     2     1      722            705        17     1040
##  6  2013     2     1      807            730        37     1134
##  7  2013     2     1      826            830        -4     1206
##  8  2013     2     1      857            900        -3     1225
##  9  2013     2     1      859            900        -1     1251
## 10  2013     2     1      901            905        -4     1230
## # ... with 1,020 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Exercise - Filtering data:

Let’s do several more examples to make sure you’re super comfortable with filtering data:

- How many flights left before 6 AM?
- How many flights went to Toronto (YYZ)? Is there anything weird about this dataset?
- What is a typical flight time (air time) when traveling from New York to Chicago O’Hare (ORD)?

Using the “pipe”

The tidyverse heavily encourages the use of a special pipe (`%>%` operator). The pipe sends the output of the last command to the first argument of the next (probably will be a familiar concept for users of bash, the Linux shell). This is a great tool for making our analyses more readable (read: good).

Repeating an earlier example, we can retrieve the number of flights that went to LAX with:

```
# earlier example:
# nrow(filter(flights, dest == "LAX"))

flights %>% filter(dest == "LAX") %>% nrow
```

```
## [1] 16174
```

Our analysis now flows from left to right, instead of inside out. Makes things quite a bit more readable. Many people also put each step on a new line. That way if you want to exclude a step, you can just comment it out.

```
flights %>%
    filter(dest == "LAX") %>%
    nrow()
```

```
## [1] 16174
```

Controlling output

`dplyr` also has its own function for selecting columns: `select()`. To grab the certain columns from a dataframe, we supply their names to `select()` as arguments.

```
flights %>% select(flight, dest, air_time)
```

```
## # A tibble: 336,776 x 3
##    flight  dest air_time
##     <int> <chr>    <dbl>
##  1   1545   IAH      227
##  2   1714   IAH      227
##  3   1141   MIA      160
##  4    725   BQN      183
##  5    461   ATL      116
##  6   1696   ORD      150
##  7    507   FLL      158
##  8   5708   IAD       53
##  9     79   MCO      140
## 10    301   ORD      138
## # ... with 336,766 more rows
```

We can also sort columns using `arrange()`. `arrange()` sorts a dataset by whatever column names you specify.

```
flights %>% arrange(sched_dep_time)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     7    27       NA            106        NA       NA
##  2  2013     1     2      458            500        -2      703
##  3  2013     1     3      458            500        -2      650
##  4  2013     1     4      456            500        -4      631
##  5  2013     1     5      458            500        -2      640
##  6  2013     1     6      458            500        -2      718
##  7  2013     1     7      454            500        -6      637
##  8  2013     1     8      454            500        -6      625
##  9  2013     1     9      457            500        -3      647
## 10  2013     1    10      450            500       -10      634
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

To sort in descending order, we can add the `desc()` function into the mix.

```
flights %>% arrange(desc(sched_dep_time))
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1     2353           2359        -6      425
##  2  2013     1     1     2353           2359        -6      418
##  3  2013     1     1     2356           2359        -3      425
##  4  2013     1     2       42           2359        43      518
##  5  2013     1     2     2351           2359        -8      427
##  6  2013     1     2     2354           2359        -5      413
##  7  2013     1     3       32           2359        33      504
##  8  2013     1     3      235           2359       156      700
##  9  2013     1     3     2349           2359       -10      434
## 10  2013     1     4       25           2359        26      505
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Data analysis

So far we've learned how to to rearrange and select parts of our data. What about actually analyzing it. The `group_by()` and `summarize()` functions, allow us to group by a certain column (say, city or airline), and then perform an operation on every group.

A simple example might be grouping by month and then summarizing by the number of flights (rows) in each group.

```
flights %>%
    group_by(month) %>%
    summarize(length(month))  # number of records in a group
```

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

We can also perform multiple "summarizations" at once and name our columns something informative.

```
flights %>%
    group_by(month) %>%
    summarize(num_flights=length(month),
              avg_flight_time=mean(air_time, na.rm=TRUE))
```

```
## # A tibble: 12 x 3
##    month num_flights avg_flight_time
##    <int>       <int>           <dbl>
##  1     1       27004        154.1874
##  2     2       24951        151.3464
##  3     3       28834        149.0770
##  4     4       28330        153.1011
##  5     5       28796        145.7275
##  6     6       28243        150.3252
##  7     7       29425        146.7283
##  8     8       29327        148.1604
##  9     9       27574        143.4712
## 10    10       28889        148.8861
## 11    11       27268        155.4686
## 12    12       28135        162.5914
```

We can also simply add on a column to a dataset with the `mutate()` function. This is the equivalent of `cats$age <- c(1, 3, 4)` like we did earlier.

```
colnames(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"
```

```
new_flights <- flights %>%
    mutate(plane_speed = distance / air_time)
colnames(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"
```

Exercise - Finding the worst airline:

Which airline has the worst record in terms of delays?

To do this, group our data by carrier, get the average arrival delay for each group, then sort in descending order so that the worst offenders are at the top.

Excercise - Picking an analysis method:

Get the maximum arrival delay in the dataset. You'll want to use the `max()` function. Did you need to use `dplyr`?

Putting dataframes together

In terms of some data, the `flights` table is actually incomplete! What if we wanted to match up the destination airport acronyms to their details (like airports' full names)? This data is actually in another table: `airports`.

```
head(airports)
```

```
## # A tibble: 6 x 8
##     faa                           name      lat       lon   alt    tz
##   <chr>                          <chr>    <dbl>     <dbl> <int> <dbl>
## 1   04G              Lansdowne Airport 41.13047 -80.61958  1044    -5
## 2   06A  Moton Field Municipal Airport 32.46057 -85.68003   264    -6
## 3   06C            Schaumburg Regional 41.98934 -88.10124   801    -6
## 4   06N                Randall Airport 41.43191 -74.39156   523    -5
## 5   09J          Jekyll Island Airport 31.07447 -81.42778    11    -5
## 6   0A9 Elizabethton Municipal Airport 36.37122 -82.17342  1593    -5
## # ... with 2 more variables: dst <chr>, tzone <chr>
```

In order for this information to be useful to us, we need to match it up and "join" it to our flights table. This is a pretty complex operation in base R, but `dplyr` makes it relatively easy.

There are a lot of different types of joins that put together data in different ways. In this case, we're going to do what's called a "left join": one table is on the left side, and we'll keep all of its data. However, on the right side (the table we are joining), we'll only match up and add each entry if there is a corresponding entry on the left side.

```
colnames(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"
```

```
colnames(airports)
```

```
## [1] "faa"   "name"  "lat"   "lon"   "alt"   "tz"    "dst"   "tzone"
```

```
# join syntax:
# left_join(left_table, right_table, by=c("left_colname" = "right_colname"))
# the "by" argument controls which columns in each table are matched up
joined <- left_join(flights, airports, by=c("dest" = "faa"))
colnames(joined)  # joined now contain columns from both
```

```
##  [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"      "name"
## [21] "lat"            "lon"            "alt"            "tz"
## [25] "dst"            "tzone"
```

Let's check our work. SEA should show up as Seattle-Tacoma International Airport. Note: we can use `.` as a placeholder to represent the entire object passed to the summarize function (instead of using just a column name, for instance).

```
joined %>%
    filter(dest == "SEA") %>%
    select(name) %>%
    head(n=1)
```

```
## # A tibble: 1 x 1
##                 name
##                <chr>
## 1 Seattle Tacoma Intl
```

Looks like our join worked!

Exercise - Worst airline, part II:

Find the name of the airline with the biggest arrival delays. You will need to join the `airlines` table to the `flights` table. A suggested workflow is shown below (feel free to reuse code from earlier).

- Calculate the average arrival delays by airline.
- Sort the result by average delay in descending order.
- Find which columns match up between the `airlines` and `flights` tables. Remember, you can use `print(table_name, width=Inf)` to show all columns!
- Join the `airlines` table to the `flights` table based upon their common column.
- The top value is your answer.

Exercise - Writing output:

Write your results from the last problem to a file. Use the `write_csv()` to write the table to a csv file. You can use `?write_csv()` to look up how to use this function.

Next section