Chapter 4: Data Manipulation, Wrangling with dplyr

M Affouf

1/8/2018

part1

Manipulating Data

So far, we've covered how to read in data, and select specific rows and columns.

All of these steps help you set up your analysis or data exploration.

Now we are going to cover manipulating your data and summarizing it using basic statistics and visualizations.

Sorting and ordering

`sort(x, decreasing=FALSE)`: 'sort (or order) a vector or factor (partially) into ascending or descending order.' Note that this returns an object that has been sorted/ordered

`order(...,decreasing=FALSE)`: 'returns a permutation which rearranges its first argument into ascending or descending order, breaking ties by further arguments.' Note that this returns the indices corresponding to the sorted data.

Sorting and ordering

```
x = c(1,4,7,6,4,12,9,3)
sort(x)
```

```
## [1]  1  3  4  4  6  7  9 12
```

```
order(x)
```

```
## [1] 1 8 2 5 4 3 7 6
```

Note you would have to assign the sorted variable to a new variable to retain it

Sorting and ordering

```
circ = read.csv("charmcitycirc_reduced.csv", header=TRUE,as
circ2 = circ[,c("day","date", "orangeAverage","purpleAverag
              "greenAverage","bannerAverage","daily")]
head(order(circ2$daily,decreasing=TRUE))
```

```
## [1] 888 887 886 971 880 866
```

```
head(sort(circ2$daily,decreasing=TRUE))
```

```
## [1] 22074.5 21951.0 17580.0 16714.0 16366.5 16149.5
```

The first indicates the rows of `circ2` ordered by daily average ridership. The second displays the actual sorted values of daily average ridership.

Sorting and ordering

```
circSorted = circ2[order(circ2$daily,decreasing=TRUE),]
circSorted[1:5,]
```

```
##           day       date orangeAverage purpleAverage gree
## 888 Saturday 06/16/2012         6322.0         7797.0
## 887   Friday 06/15/2012         6926.5         8089.5
## 886 Thursday 06/14/2012         5617.5         6521.0
## 971   Friday 09/07/2012         5717.5         7007.0
## 880   Friday 06/08/2012         5782.5         6881.5
##     bannerAverage   daily
## 888        4617.0 22074.5
## 887        3450.0 21951.0
## 886        2672.0 17580.0
## 971        1301.0 16714.0
## 880         844.5 16366.5
```

Sorting and ordering

Note that the row names refer to their previous values. You can do something like this to fix:

```
rownames(circSorted)=NULL
circSorted[1:5,]
```

```
##         day       date orangeAverage purpleAverage greenA
## 1 Saturday 06/16/2012         6322.0         7797.0
## 2   Friday 06/15/2012         6926.5         8089.5
## 3 Thursday 06/14/2012         5617.5         6521.0
## 4   Friday 09/07/2012         5717.5         7007.0
## 5   Friday 06/08/2012         5782.5         6881.5
##   bannerAverage   daily
## 1        4617.0 22074.5
## 2        3450.0 21951.0
## 3        2672.0 17580.0
## 4        1301.0 16714.0
## 5         844.5 16366.5
```

Creating categorical variables

However, it's much easier to use `cut()` to create categorical variables from continuous variables.

'cut divides the range of x into intervals and codes the values in x according to which interval they fall. The leftmost interval corresponds to level one, the next leftmost to level two and so on.'

```
cut(x, breaks, labels = NULL, include.lowest = FALSE,
  right = TRUE, dig.lab = 3,
  ordered_result = FALSE, ...)
```

Cut

Now that we know more about factors, `cut()` will make more sense:

```
x = 1:100
cx = cut(x, breaks=c(0,10,25,50,100))
head(cx)
```

```
## [1] (0,10] (0,10] (0,10] (0,10] (0,10] (0,10]
## Levels: (0,10] (10,25] (25,50] (50,100]
```

```
table(cx)
```

```
## cx
##   (0,10]  (10,25]  (25,50] (50,100]
##       10       15       25       50
```

We can also leave off the labels

```
cx = cut(x, breaks=c(0,10,25,50,100), labels=FALSE)
head(cx)
```

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

```
table(cx)
```

```
## cx
##  1  2  3  4
## 10 15 25 50
```

Note that you have to specify the endpoints of the data, otherwise some of the categories will not be created

```
cx = cut(x, breaks=c(10,25,50), labels=FALSE)
head(cx)
```

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

```
table(cx)
```

```
## cx
##  1  2
## 15 25
```

```
table(cx,useNA="ifany")
```

```
## cx
##    1    2 <NA>
##   15   25   60
```

Adding to data frames

```
circ2$riderLevels = cut(circ2$daily,
      breaks = c(0,10000,20000,100000))
circ2[1:2,]
```

```
##       day       date orangeAverage purpleAverage greenAv
## 1  Monday 01/11/2010            952            NA
## 2 Tuesday 01/12/2010            796            NA
##   bannerAverage daily riderLevels
## 1            NA   952   (0,1e+04]
## 2            NA   796   (0,1e+04]
```

```
table(circ2$riderLevels, useNA="always")
```

```
##
##     (0,1e+04] (1e+04,2e+04] (2e+04,1e+05]          <NA>
##           731           280             2           133
```

Other manipulations

- `abs(x)`: absolute value
- `sqrt(x)`: square root
- `ceiling(x)`: ceiling(3.475) is 4
- `floor(x)`: floor(3.475) is 3
- `trunc(x)`: trunc(5.99) is 5
- `round(x, digits=n)`: round(3.475, digits=2) is 3.48
- `signif(x, digits=n)`: signif(3.475, digits=2) is 3.5
- `log(x)`: natural logarithm
- `log10(x)`: common logarithm
- `exp(x)`: e^x

(via: http://statmethods.net/management/functions.html)

Overview

In this module, we will show you how to:

1. Reshaping data from long (tall) to wide (fat)
2. Reshaping data from wide (fat) to long (tall)
3. Merging Data
4. Perform operations by a grouping variable

Setup

We will show you how to do each operation in base R then show you how to use the `dplyr` or `tidyr` package to do the same operation (if applicable).

See the "Data Wrangling Cheat Sheet using `dplyr` and `tidyr`":

- https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

Load the packages/libraries

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(tidyr)
```

Data used: Charm City Circulator

Let's read in the Charm City Circulator data:

```
ex_data = read.csv("Charm_City_Circulator_Ridership.csv", a
head(ex_data, 2)
```

```
##       day       date orangeBoardings orangeAlightings or
## 1  Monday 01/11/2010              877              1027
## 2 Tuesday 01/12/2010              777               815
##   purpleBoardings purpleAlightings purpleAverage greenBo
## 1              NA               NA            NA
## 2              NA               NA            NA
##   greenAlightings greenAverage bannerBoardings bannerAli
## 1              NA           NA              NA
## 2              NA           NA              NA
##   bannerAverage daily
## 1            NA   952
## 2            NA   796
```

Creating a Date class from a character date

The lubridate package is great for dates:

```
library(lubridate) # great for dates!
```

```
##
## Attaching package: 'lubridate'
```

```
## The following object is masked from 'package:base':
##
##     date
```

```
ex_data = mutate(ex_data, date = mdy(date))
nrow(ex_data[ is.na(ex_data$date), ])
```

```
## [1] 0
```

```
head(ex_data$date)
```

```
## [1] "2010-01-11" "2010-01-12" "2010-01-13" "2010-01-14"
```

Making column names a little more separated

We will use `str_replace` from `stringr` to put periods in the column names.

```
library(stringr)
cn = colnames(ex_data)
cn = cn %>%
  str_replace("Board", ".Board") %>%
  str_replace("Alight", ".Alight") %>%
  str_replace("Average", ".Average")
colnames(ex_data) = cn
```

Removing the daily ridership

We want to look at each ridership, and will removet the `daily` column:

```
ex_data$daily = NULL
```

Reshaping data from wide (fat) to long (tall)

See http://www.cookbook-r.com/Manipulating_data/Converting_data_between_wide_and_long_format/

Reshaping data from wide (fat) to long (tall): base R

The `reshape` command exists. It is a **confusing** function. Don't use it.

Reshaping data from wide (fat) to long (tall): tidyr

In `tidyr`, the `gather` function gathers columns into rows.

We want the column names into "`var`" variable in the output dataset and the value in "`number`" variable. We then describe which columns we want to "gather:"

```
long = gather(ex_data, "var", "number",
              starts_with("orange"),
              starts_with("purple"), starts_with("green"),
              starts_with("banner"))
head(long)
```

```
##         day       date                  var number
## 1    Monday 2010-01-11 orange.Boardings    877
## 2   Tuesday 2010-01-12 orange.Boardings    777
## 3 Wednesday 2010-01-13 orange.Boardings   1203
## 4  Thursday 2010-01-14 orange.Boardings   1194
## 5    Friday 2010-01-15 orange.Boardings   1645
## 6  Saturday 2010-01-16 orange.Boardings   1457
```

Reshaping data from wide (fat) to long (tall): tidyr

Now each `var` is boardings, averages, or alightings. We want to separate these so we can have these by line.

```
long = separate_(long, "var", into = c("line", "type"), sep
head(long)
```

```
##         day       date   line      type number
## 1    Monday 2010-01-11 orange Boardings    877
## 2   Tuesday 2010-01-12 orange Boardings    777
## 3 Wednesday 2010-01-13 orange Boardings   1203
## 4  Thursday 2010-01-14 orange Boardings   1194
## 5    Friday 2010-01-15 orange Boardings   1645
## 6  Saturday 2010-01-16 orange Boardings   1457
```

```
table(long$line)
```

```
##
## banner  green orange purple
```

Reshaping data from long (tall) to wide (fat): tidyr

In `tidyr`, the `spread` function spreads rows into columns. Now we have a long data set, but we want to separate the Average, Alightings and Boardings into different columns:

```
# have to remove missing days
wide = filter(long, !is.na(date))
wide = spread(wide, type, number)
head(wide)
```

```
##       day       date   line Alightings Average Boardings
## 1 Friday 2010-01-15 banner          NA      NA         NA
## 2 Friday 2010-01-15  green          NA      NA         NA
## 3 Friday 2010-01-15 orange        1643    1644       1645
## 4 Friday 2010-01-15 purple          NA      NA         NA
## 5 Friday 2010-01-22 banner          NA      NA         NA
## 6 Friday 2010-01-22  green          NA      NA         NA
```

Reshaping data from long (tall) to wide (fat): tidyr

We can use `rowSums` to see if any values in the row is `NA` and keep if the row, which is a combination of date and line type has any non-missing data.

```
# wide = wide %>%
#     select(Alightings, Average, Boardings) %>%
#     mutate(good = rowSums(is.na(.)) > 0)
namat = !is.na(select(wide, Alightings, Average, Boardings)
head(namat)
```

```
##   Alightings Average Boardings
## 1      FALSE   FALSE      FALSE
## 2      FALSE   FALSE      FALSE
## 3       TRUE    TRUE       TRUE
## 4      FALSE   FALSE      FALSE
## 5      FALSE   FALSE      FALSE
## 6      FALSE   FALSE      FALSE
```

```
wide$good = rowSums(namat) > 0
```

Reshaping data from long (tall) to wide (fat): tidyr

Now we can filter only the good rows and delete the good column.

```
wide = filter(wide, good) %>% select(-good)
head(wide)
```

```
##       day       date   line Alightings Average Boardings
## 1 Friday 2010-01-15 orange        1643  1644.0      1645
## 2 Friday 2010-01-22 orange        1388  1394.5      1401
## 3 Friday 2010-01-29 orange        1322  1332.0      1342
## 4 Friday 2010-02-05 orange        1204  1217.5      1231
## 5 Friday 2010-02-12 orange         678   671.0       664
## 6 Friday 2010-02-19 orange        1647  1642.0      1637
```

Data Merging/Append in Base R

- Merging - joining data sets together - usually on key variables, usually "id"
- `merge()` is the most common way to do this with data sets
- `rbind`/`cbind` - row/column bind, respectively
  - `rbind` is the equivalent of "appending" in Stata or "setting" in SAS
  - `cbind` allows you to add columns in addition to the previous ways
- `t()` is a function that will transpose the data

Merging

```
base <- data.frame(id = 1:10, Age= seq(55,60, length=10))
base[1:2,]
```

```
##   id      Age
## 1  1 55.00000
## 2  2 55.55556
```

```
visits <- data.frame(id = rep(1:8, 3), visit= rep(1:3, 8),
                     Outcome = seq(10,50, length=24))
visits[1:2,]
```

```
##   id visit  Outcome
## 1  1     1 10.00000
## 2  2     2 11.73913
```

Merging

```
merged.data <- merge(base, visits, by="id")
merged.data[1:5,]
```

```
##    id       Age visit  Outcome
## 1   1 55.00000     1 10.00000
## 2   1 55.00000     3 23.91304
## 3   1 55.00000     2 37.82609
## 4   2 55.55556     2 11.73913
## 5   2 55.55556     1 25.65217
```

```
dim(merged.data)
```

```
## [1] 24  4
```

Merging

```
all.data <- merge(base, visits, by="id", all=TRUE)
tail(all.data)
```

```
##     id       Age visit  Outcome
## 21   7 58.33333     2 48.26087
## 22   8 58.88889     2 22.17391
## 23   8 58.88889     1 36.08696
## 24   8 58.88889     3 50.00000
## 25   9 59.44444    NA       NA
## 26  10 60.00000    NA       NA
```

```
dim(all.data)
```

```
## [1] 26  4
```

Perform Operations By Groups: base R

The `tapply` command will take in a vector (`X`), perform a function (`FUN`) over an index (`INDEX`):

```
args(tapply)
```

```
## function (X, INDEX, FUN = NULL, ..., default = NA, simpl
## NULL
```

Perform Operations By Groups: base R

Let's get the mean Average ridership by line:

```
tapply(wide$Average, wide$line, mean, na.rm = TRUE)
```

```
##    banner     green    orange    purple
##  827.2685 1957.7814 3033.1611 4016.9345
```

Perform Operations By Groups: dplyr

Let's get the mean Average ridership by line We will use `group_by` to group the data by line, then use `summarize` (or `summarise`) to get the mean Average ridership:

```
gb = group_by(wide, line)
summarize(gb, mean_avg = mean(Average))
```

```
## # A tibble: 4 x 2
##      line  mean_avg
##     <chr>     <dbl>
## 1 banner  827.2685
## 2  green 1957.7814
## 3 orange 3033.1611
## 4 purple 4016.9345
```

Perform Operations By Groups: dplyr with piping

Using piping, this is:

```
wide %>%
  group_by(line) %>%
  summarise(mean_avg = mean(Average))
```

```
## # A tibble: 4 x 2
##      line  mean_avg
##     <chr>     <dbl>
## 1 banner  827.2685
## 2  green 1957.7814
## 3 orange 3033.1611
## 4 purple 4016.9345
```

Perform Operations By Multiple Groups: dplyr

This can easily be extended using group_by with multiple groups. Let's define the year of riding:

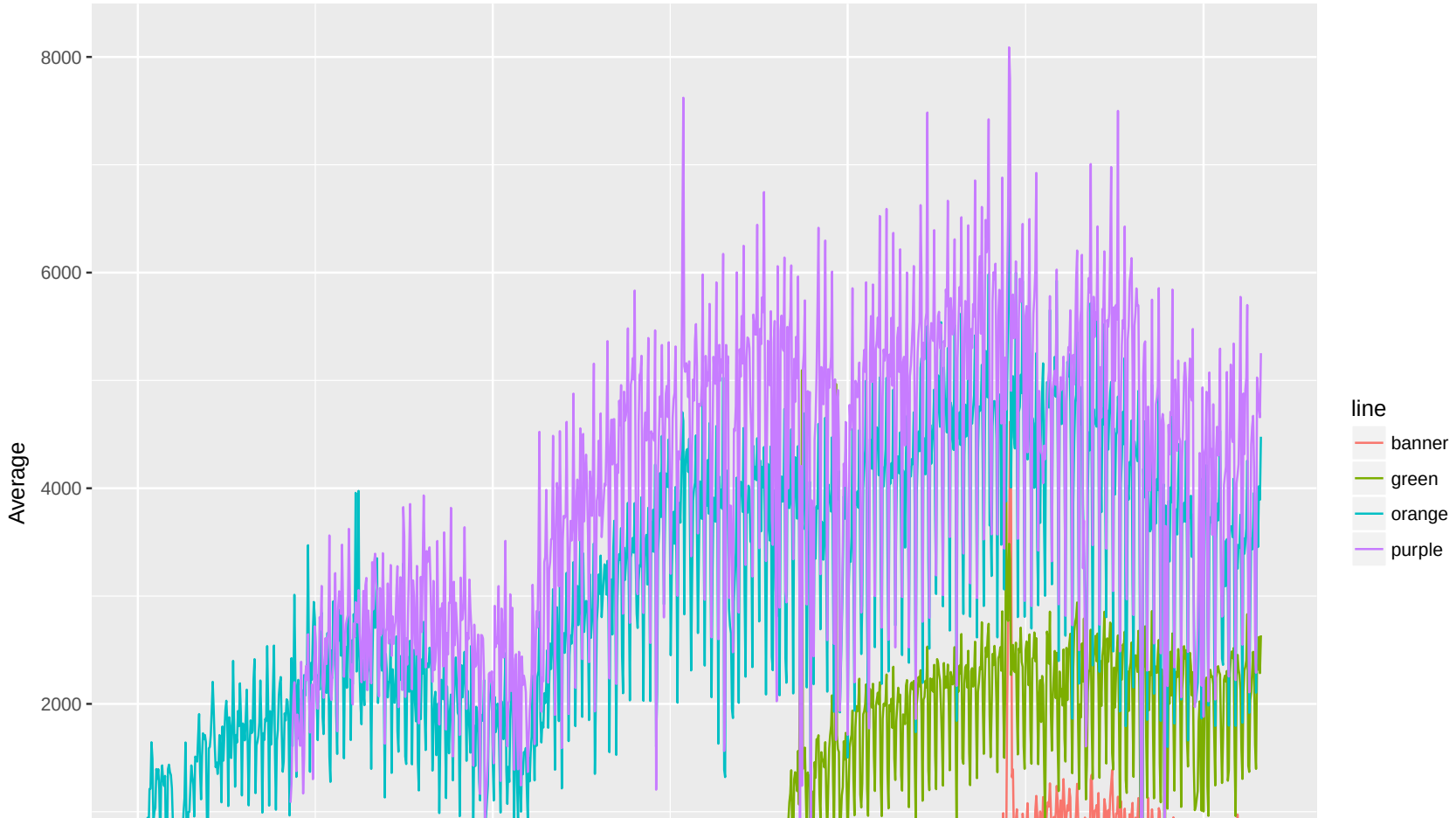
```
wide = wide %>% mutate(year = year(date),
                       month = month(date))
wide %>%
  group_by(line, year) %>%
  summarise(mean_avg = mean(Average))
```

```
## # A tibble: 13 x 3
## # Groups:   line [?]
##      line  year  mean_avg
##     <chr> <dbl>     <dbl>
##  1 banner  2012  882.0929
##  2 banner  2013  635.3833
##  3  green  2011 1455.1667
##  4  green  2012 2028.7740
##  5  green  2013 2028.5250
```

Perform Operations By Multiple Groups: dplyr

We can then easily plot each day over time:

```
library(ggplot2)
ggplot(aes(x = date, y = Average,
                colour = line), data = wide) + geom_line()
```



Perform Operations By Multiple Groups: dplyr

Let's create the middle of the month (the 15th for example), and name it mon.

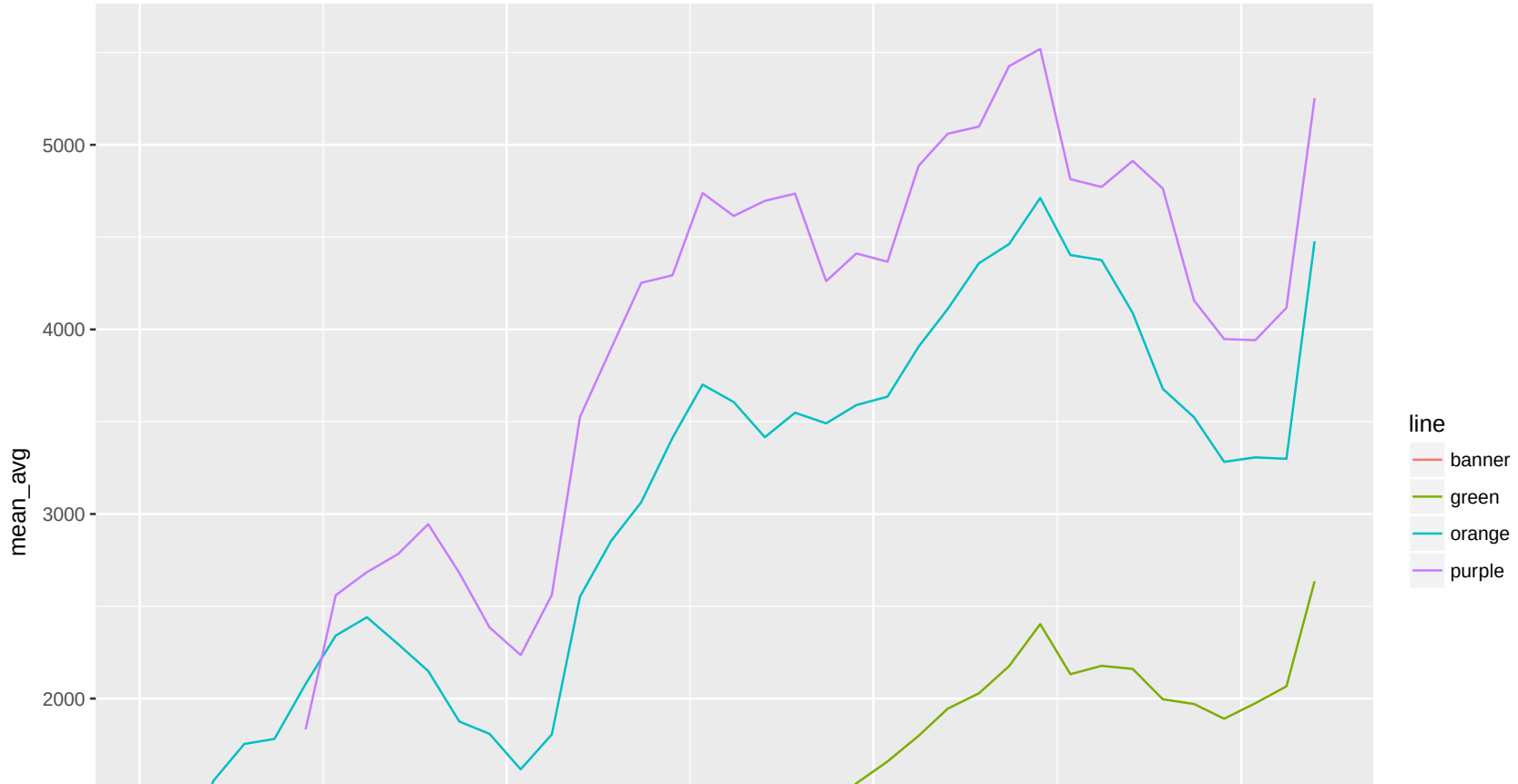
```
mon = wide %>%
  dplyr::group_by(line, month, year) %>%
  dplyr::summarise(mean_avg = mean(Average))
mon = mutate(mon,
             mid_month = dmy(paste0("15-", month, "-", yea
head(mon)
```

```
## # A tibble: 6 x 5
## # Groups:   line, month [6]
##      line month  year  mean_avg  mid_month
##     <chr> <dbl> <dbl>     <dbl>     <date>
## 1 banner     1  2013  610.3226 2013-01-15
## 2 banner     2  2013  656.4643 2013-02-15
## 3 banner     3  2013  822.0000 2013-03-15
## 4 banner     6  2012 1288.1296 2012-06-15
```

Perform Operations By Multiple Groups: dplyr

We can then easily plot the mean of each month to see a smoother output:

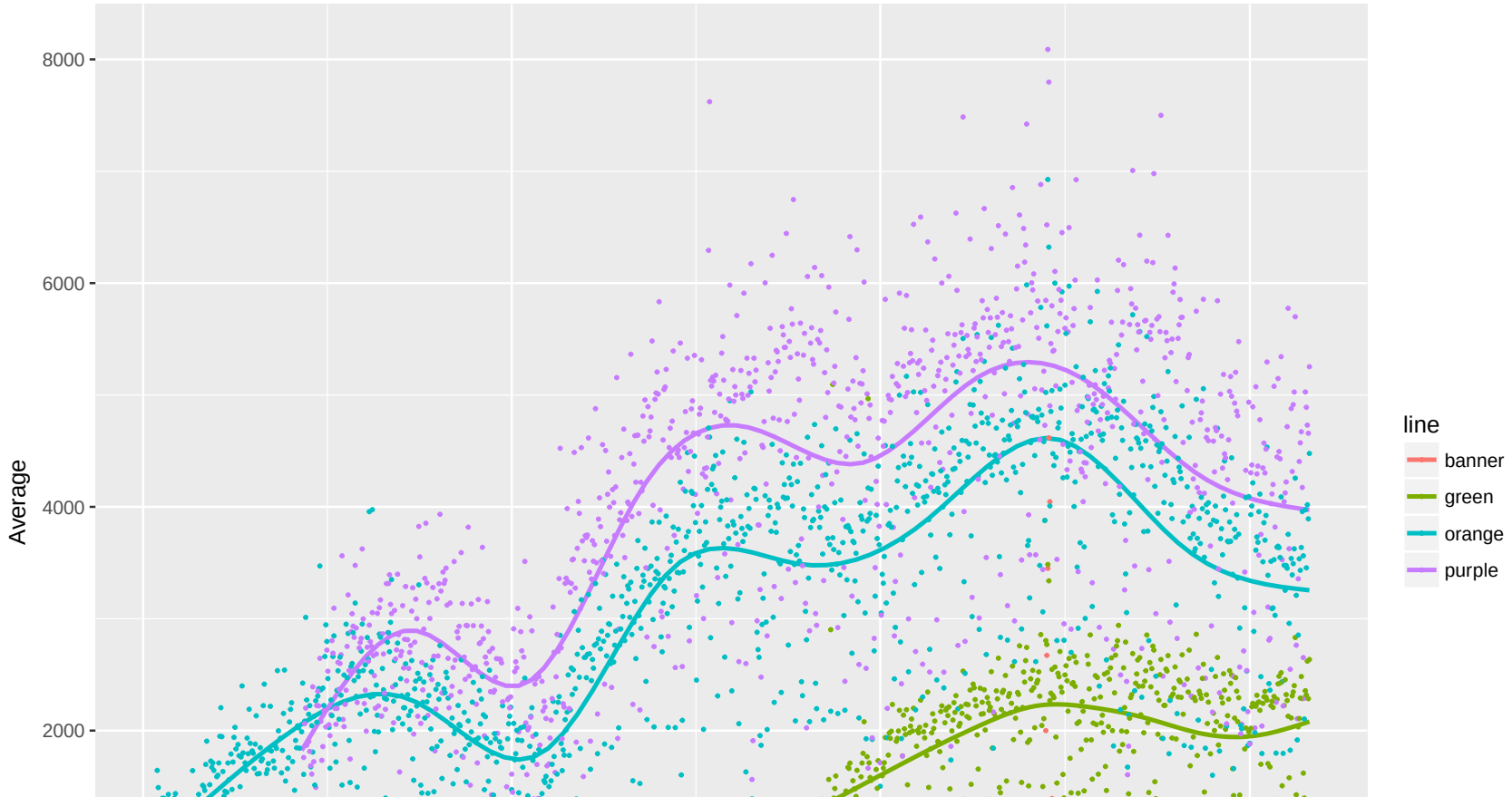
```
ggplot(aes(x = mid_month,
           y = mean_avg,
           colour = line), data = mon) + geom_line()
```



Bonus! Points with a smoother!

```
ggplot(aes(x = date, y = Average, colour = line),
           data = wide) + geom_smooth(se = FALSE) +
  geom_point(size = .5)
```

```
## `geom_smooth()` using method = 'gam'
```



Part2

Packages

```
library(ggplot2)
library(dplyr)
library(nycflights13)
```

These notes are based on the following introduction to dplyr vignette.

For a more thorough discussion, you can look at the Data transformation chapter of R for Data Science

The dplyr and tidyr cheatsheet is another fantastic reference.

Basics of dplyr

The `dplyr` package introduces 5 basic verbs that help to streamline the data manipulation process.

- `filter(<data.frame>, <criteria>)`
  - Selects a subset of rows from a `<data.frame>` based on expressions giving filtering `<criteria>`
- `arrange()`
- `select()`
- `mutate()`
- `summarise()`

It also has several other functions such as `slice()`, `rename()`, `transmute()`, `sample_n()` and `sample_frac()`, all of which you may find useful.

Exploring the nycflights13 data

We'll illustrate the basics of dplyr using the flights data. This dataset contains information on 336776 that departed from New York City in 2013.

```
head(flights)
```

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

Data subsets with `filter()`

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

Let's look at all the flights that departed on January 1st and where the departure time was delayed by at least 15 minutes.

```
filter(flights,
       month == 1,
       day == 1,
       dep_delay >= 15)
```

```
## # A tibble: 163 x 19
##      year month   day dep_time sched_dep_time dep_delay a
##     <int> <int> <int>    <int>          <int>     <dbl>
##  1  2013     1     1      632            608        24
##  2  2013     1     1      732            645        47
##  3  2013     1     1      749            710        39
##  4  2013     1     1      811            630       101
```

Rearrange rows with `arrange()`

You can think of `arrange()` as a "sort by" operation. This function takes a data frame and a set of column names by which to order the data. Later columns are used to break ties (i.e., order within) earlier columns.

Here's an example that arranges the data in order of departure date.

```
arrange(flights, year, month, day)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay a
##    <int> <int> <int>    <int>          <int>     <dbl>
##  1  2013     1     1      517            515         2
##  2  2013     1     1      533            529         4
##  3  2013     1     1      542            540         2
##  4  2013     1     1      544            545        -1
##  5  2013     1     1      554            600        -6
##  6  2013     1     1      554            558        -4
##  7  2013     1     1      555            600        -5
```

Select columns with `select()`

The `select()` function can be thought of as a substitute for the `select =` argument in a `subset()` command. One notable difference is the more flexible syntax offered by `select()`.

```
# Select columns by name
select(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
```

Add new columns with `mutate()`

You can think of `mutate()` as an improved version of the `transform()` command. We'll illustrate a couple of advantages.

```
# Calculate delay reduction in travel (gain) and average sp
mutate(flights,
  gain = arr_delay - dep_delay,
  speed = distance / air_time * 60)
```

```
## # A tibble: 336,776 x 21
##     year month   day dep_time sched_dep_time dep_delay a
##    <int> <int> <int>    <int>          <int>     <dbl>
##  1  2013     1     1      517            515         2
##  2  2013     1     1      533            529         4
##  3  2013     1     1      542            540         2
##  4  2013     1     1      544            545        -1
##  5  2013     1     1      554            600        -6
##  6  2013     1     1      554            558        -4
##  7  2013     1     1      555            600        -5
```

transmute()

If all you want to keep from the `mutate()` are the newly formed variables, you can either chain together a `mutate()` with a `select()`, or you can directly use the `transmute()` command.

```
transmute(flights,
  gain = arr_delay - dep_delay,
  gain_per_hour = gain / (air_time / 60)
)
```

```
## # A tibble: 336,776 x 2
##     gain gain_per_hour
##    <dbl>         <dbl>
##  1     9      2.378855
##  2    16      4.229075
##  3    31     11.625000
##  4   -17     -5.573770
##  5   -19     -9.827586
##  6    16      6.400000
```

Summary tables with summarise()

You can think of `summarise()` as performing a similar operation to the `plyr::ddply()` function. On its own, `summarise()` just returns a 1-line summary data frame.

```
summarise(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       12.63907       6.895377
```

Using group_by()

To obtain summaries within some grouping scheme, you can use the `group_by()` command followed by `summarise()`.

Here we'll illustrate how this approach can be used to better understand the association between arrival delays and distance traveled.

```
# Form a summary table showing the number of flights,
# average distance, and arrival delay for each airplane

by_tailnum <- group_by(flights, tailnum)
delay <- summarise(by_tailnum,
  count = n(),
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE))

# Subset the data to only include frequently flown planes
# and distances < 3000
delay <- filter(delay, count > 20, dist < 3000)
```

Handy summary functions

In addition to functions such as `min()`, `max()`, ..., `median()` etc., you can also use the following, which are enabled by the `dplyr` library:

- `n()`: the number of observations in the current group
- `n_distinct(x)` :the number of unique values in `x`.
- `first(x)`, `last(x)` and `nth(x, n)` - these work similarly to `x[1]`, `x[length(x)]`, and `x[n]` but give you more control over the result if the value is missing.

You can use these functions to, for instance, count the number of planes and number of flights for each possible destination:

```
destinations <- group_by(flights, dest)

summarise(destinations,
  planes = n_distinct(tailnum),
  flights = n()
)
```

Successive summaries

When you group by multiple variables, each summary peels off one level of the grouping. That makes it easy to progressively roll-up a dataset:

```
daily <- group_by(flights, year, month, day)

# Tabulate number of flights on each day
per_day   <- summarise(daily, flights = n())
per_day
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [?]
##     year month   day flights
##    <int> <int> <int>   <int>
##  1  2013     1     1     842
##  2  2013     1     2     943
##  3  2013     1     3     914
##  4  2013     1     4     915
```

distinct()

`distinct()` allows you to identify the unique values of variables (or combinations of variables) in your data.

```
# How many different planes departed from NYC airports
# in 2013?
distinct(flights, tailnum)
```

```
## # A tibble: 4,044 x 1
##    tailnum
##      <chr>
##  1  N14228
##  2  N24211
##  3  N619AA
##  4  N804JB
##  5  N668DN
##  6  N39463
##  7  N516JB
##  8  N829AS
```

rename()

We've done a lot of variable renaming in this class. In most of the cases we've renamed all of the columns all at once. If we want to change only a few column names, this can get frustrating. `rename()` addresses precisely this issue.

```
rename(flights,
       yr = year,
       dep.time = dep_time)
```

```
## # A tibble: 336,776 x 19
##       yr month   day dep.time sched_dep_time dep_delay a
##    <int> <int> <int>    <int>          <int>     <dbl>
##  1  2013     1     1      517            515         2
##  2  2013     1     1      533            529         4
##  3  2013     1     1      542            540         2
##  4  2013     1     1      544            545        -1
##  5  2013     1     1      554            600        -6
##  6  2013     1     1      554            558        -4
```

Piping (chaining)

In this section we'll introduce the %>% ("pipe") command, which you'll quickly find indispensible when chaining together multiple operations.

To illustrate a use case, suppose we wanted to do some grouping, sub-setting, summarizing, and then further filtering of the summary. For instance, we might be interested in identifying days in 2013 where the average arrival or departure delay was especially high.

Here's one approach.

```
# Group by day of the year
a1 <- group_by(flights, year, month, day)

# Select just the arrival and departure delay columns
a2 <- select(a1, arr_delay, dep_delay)
```

```
## Adding missing grouping variables: `year`, `month`, `day
```

```
# Calculate average delays
```

Example: delay gain per hour

```
gain.df <- flights %>%
  mutate(gain = dep_delay - arr_delay,
         gain_per_hour = gain / (air_time / 60)) %>%
  group_by(tailnum) %>%
  summarise(count = n(),
            av_gain = mean(gain_per_hour, na.rm = TRUE),
            av_dep_delay = mean(dep_delay, na.rm = TRUE),
            av_arr_delay = mean(arr_delay, na.rm = TRUE),
            av_dist = mean(distance)
  ) %>%
  filter(count > 10, av_dist < 3000)

ggplot(gain.df, aes(x = av_dist, y = av_gain, size = count)
  geom_point(alpha = 0.3) +
  scale_size_area() +
  geom_smooth(show.legend = FALSE)
```

```
## `geom_smooth()` using method = 'gam'
```

Example: average delay time for each origin, destination

In this example we'll pipe a summary table directly into a ggplot call.

```
flights %>%
  group_by(origin) %>%
  summarise(av_dep_delay = mean(dep_delay, na.rm = TRUE)) %
  ggplot(aes(x = origin, y = av_dep_delay)) +
  geom_bar(stat = "identity") +
  ylab("Average departure delay") +
  xlab("Origin airport")
```