---
title: "Chapter 4: Data Manipulation, Wrangling with dplyr "
author: "M Affouf"
date: "2/13/2018"
output:output: html_document

#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

```{r sortVorder}
x = c(1,4,7,6,4,12,9,3)
sort(x)
order(x)
```

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

Sorting and ordering {.smaller}

```{r sortVorder1}
circ = read.csv("charmcitycirc_reduced.csv", header=TRUE,as.is=TRUE)
circ2 = circ[,c("day","date", "orangeAverage","purpleAverage",
              "greenAverage","bannerAverage","daily")]
head(order(circ2$daily,decreasing=TRUE))
head(sort(circ2$daily,decreasing=TRUE))
```
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 {.smaller}

```{r sortVorder2}
circSorted = circ2[order(circ2$daily,decreasing=TRUE),]
```

circSorted[1:5,]
```

## Sorting and ordering {.smaller}

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

```{r sortVorder3}
rownames(circSorted)=NULL
circSorted[1:5,]
```

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:

```{r cut1}
x = 1:100
cx = cut(x, breaks=c(0,10,25,50,100))
head(cx)
table(cx)
```

We can also leave off the labels

```{r cut2}
cx = cut(x, breaks=c(0,10,25,50,100), labels=FALSE)
head(cx)
table(cx)
```

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

```{r cut3}
cx = cut(x, breaks=c(10,25,50), labels=FALSE)

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

Adding to data frames {.smaller}

```{r addingVar}
circ2$riderLevels = cut(circ2$daily,
      breaks = c(0,10000,20000,100000))
circ2[1:2,]
table(circ2$riderLevels, useNA="always")
```

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

```{r}
library(dplyr)
library(tidyr)
```
```

Data used: Charm City Circulator

Let's read in the Charm City Circulator data:

```{r}
ex_data = read.csv("Charm_City_Circulator_Ridership.csv", as.is = TRUE)
head(ex_data, 2)
```

Creating a Date class from a character date

The `lubridate` package is great for dates:

```{r}
library(lubridate) # great for dates!
ex_data = mutate(ex_data, date = mdy(date))
nrow(ex_data[ is.na(ex_data$date), ])
head(ex_data$date)
class(ex_data$date)
```

Making column names a little more separated

We will use `str_replace` from `stringr` to put periods in the column names.
```{r}
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:
```{r}
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 {.smaller}

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:"
```{r}
long = gather(ex_data, "var", "number",
              starts_with("orange"),
              starts_with("purple"), starts_with("green"),
              starts_with("banner"))
head(long)
table(long$var)
```

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.

```{r}
long = separate_(long, "var", into = c("line", "type"), sep = "[.]")
head(long)
table(long$line)
table(long$type)
```

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:

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

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.

```{r}
# wide = wide %>%
#     select(Alightings, Average, Boardings) %>%
#     mutate(good = rowSums(is.na(.)) > 0)
namat = !is.na(select(wide, Alightings, Average, Boardings))
head(namat)
wide$good = rowSums(namat) > 0
head(wide, 3)
```

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

Now we can filter only the good rows and delete the `good` column.
```{r}
wide = filter(wide, good) %>% select(-good)
head(wide)
```


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 {.smaller}

```{r merging}
base <- data.frame(id = 1:10, Age= seq(55,60, length=10))
base[1:2,]
visits <- data.frame(id = rep(1:8, 3), visit= rep(1:3, 8),
                    Outcome = seq(10,50, length=24))
visits[1:2,]
```

Merging {.smaller}

```{r merging2}
merged.data <- merge(base, visits, by="id")
merged.data[1:5,]
dim(merged.data)
```


Merging {.smaller}

```{r mergeall}
all.data <- merge(base, visits, by="id", all=TRUE)
tail(all.data)
dim(all.data)
```


Perform Operations By Groups: base R

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

```{r}
args(tapply)
```


Perform Operations By Groups: base R

Let's get the mean Average ridership by line:
```{r}
tapply(wide$Average, wide$line, mean, na.rm = TRUE)
```

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:
```{r}
gb = group_by(wide, line)
summarize(gb, mean_avg = mean(Average))
```

Perform Operations By Groups: dplyr with piping

Using piping, this is:

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

Perform Operations By Multiple Groups: dplyr

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

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

Perform Operations By Multiple Groups: dplyr {.smaller}

We can then easily plot each day over time:

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

Perform Operations By Multiple Groups: dplyr {.smaller}

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

```{r}
mon = wide %>%
  dplyr::group_by(line, month, year) %>%
```

````
  dplyr::summarise(mean_avg = mean(Average))
mon = mutate(mon,
             mid_month = dmy(paste0("15-", month, "-", year)))
head(mon)
```



## Perform Operations By Multiple Groups: dplyr {.smaller}

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

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




## Bonus! Points with a smoother! {.smaller}
```{r}
ggplot(aes(x = date, y = Average, colour = line),
           data = wide) + geom_smooth(se = FALSE) +
  geom_point(size = .5)
```

#Part2

##Packages

```{r, message = FALSE}
library(ggplot2)
library(dplyr)
library(nycflights13)
```

> These notes are based on the following  [introduction to dplyr vignette]
(https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html).

> For a more thorough discussion, you can look at the [Data transformation
chapter](http://r4ds.had.co.nz/transform.html) of R for Data Science

> The [dplyr and tidyr cheatsheet](https://www.rstudio.com/wp-
content/uploads/2015/02/data-wrangling-cheatsheet.pdf) 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 `r nrow(flights)` that departed from New York City in 2013.

```{r}
head(flights)
summary(flights)
```

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.

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

How does this compare to other syntax we've learned about?

```{r}
# This gets clunky fast...
flights[flights$month == 1 & flights$day == 1 & flights$dep_delay >= 15, ]
```

Better, but the dplyr syntax
```{r}
subset(flights, month == 1 & day == 1 & dep_delay >= 15)
```

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.

```{r}
arrange(flights, year, month, day)
```

You can also add expressions to the `arrange()` command. For instance, if you wanted to sort the flights in *descending* order of departure delay, you could use the `desc()` command:

```{r}
arrange(flights, desc(dep_delay))
```

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()`.

```{r}
# Select columns by name
select(flights, year, month, day)

# Select all columns between year and day (inclusive)
select(flights, year:day)

# Select all columns except those from year to day (inclusive)
select(flights, -(year:day))
```

You can use helper functions such as `starts_with()`, `ends_with()`, `matches()` and `contains()` as part of your select call.

- `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".

- `matches("(.)\\1")`: selects variables that match a regular expression. This one matches any variables that contain repeated characters. You'll learn more about regular expressions in strings.

- `num_range("x", 1:3)` matches x1, x2 and x3.

```{r}
# Pull all of the departure-related columns
select(flights, contains("dep"))
```

```{r}
# Pull all of the arrival and departure related columns
select(flights,
       contains("dep"),
       contains("arr"))
```

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.

```{r}
# Calculate delay reduction in travel (gain) and average speed
```

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

An interesting thing that you can do with `mutate()` but not `transform()` is to create columns based on transformations of new columns that you just created within the same command. Here's an example.

```{r}
mutate(flights,
  gain = arr_delay - dep_delay,
  gain_per_hour = gain / (air_time / 60)
)
```

Here's what would happen if we tried doing the same thing with the `transform()` command:

```{r, eval = FALSE}
transform(flights,
  gain = arr_delay - dep_delay,
  gain_per_hour = gain / (air_time / 60)
)
# Error in eval(expr, envir, enclos) : object 'gain' not found
```

`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.

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

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.

```{r}
summarise(flights,
          mean_dep_delay = mean(dep_delay, na.rm = TRUE),
          mean_arr_delay = mean(arr_delay, na.rm = TRUE)
          )
```

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.

```{r}
# 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)

# Plot
ggplot(delay, aes(dist, delay)) +
  geom_point(aes(size = count), alpha = 1/2) +
  geom_smooth() +
  scale_size_area()
```

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:

```{r}
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:

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

# Tabulate number of flights on each day
```

```r
per_day   <- summarise(daily, flights = n())
per_day

# Tabulate number of flights on each month
per_month <- summarise(per_day, flights = sum(flights))
per_month

# Total number of flights that year
per_year  <- summarise(per_month, flights = sum(flights))
per_year
```

`distinct()`

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

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

# How many distinct (origin, dest) pairs were there?
distinct(flights, origin, dest)
```

`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.

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

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.

```{r}
# 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)
```

````
# Calculate average delays
a3 <- summarise(a2,
  mean_arr_delay = mean(arr_delay, na.rm = TRUE),
  mean_dep_delay = mean(dep_delay, na.rm = TRUE))

# Filter to the days where the average delay was at least 30 mins
a4 <- filter(a3, mean_arr_delay > 30 | mean_dep_delay > 30)
```

Here's another approach, which wraps all of the functions together to avoid
having to create intermediate variables (`a1`, `a2` and `a3`) during the
computation.

```{r}
filter(
  summarise(
    select(
      group_by(flights, year, month, day),
      arr_delay, dep_delay
    ),
    mean_arr_delay = mean(arr_delay, na.rm = TRUE),
    mean_dep_delay = mean(dep_delay, na.rm = TRUE)
  ),
  mean_arr_delay > 30 | mean_dep_delay > 30
)
```

While this performs the exact same operation, it's nearly impossible to
read.  This is largely due to the fact that you have to parse the operation
from the inside out, rather than left-to-right or top-to-bottom.

A much better approach is to use `%>%`, which is automatically loaded when
you load `dplyr`.  Essentially, given a function `f(x, y)`, `x %>% f(y)` is
interpreted as `f(x, y)`.  This allows us to chain operations together using
much more readable syntax.

```{r}
flights %>%
  group_by(year, month, day) %>%
  select(arr_delay, dep_delay) %>%
  summarise(
    mean_arr_delay = mean(arr_delay, na.rm = TRUE),
    mean_dep_delay = mean(dep_delay, na.rm = TRUE)
  ) %>%
  filter(mean_arr_delay > 30 | mean_dep_delay > 30)
```

## Example: delay gain per hour

```{r}
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)

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

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

## Example: average delay time for each origin, destination

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

```{r, fig.height = 5, fig.width = 5}
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")
```

```{r, fig.width = 12, fig.height = 6}
flights %>%
  group_by(dest) %>%
  summarise(av_dep_delay = mean(dep_delay, na.rm = TRUE),
            origin = first(origin),
            count = n()) %>%
  mutate(dest = reorder(dest, av_dep_delay)) %>%
  filter(count > 50) %>%
  ggplot(aes(x = dest, y = av_dep_delay,
             colour = origin, size = count)) +
  geom_point(alpha = 0.5) +
  scale_size_area() +
  ylab("Average departure delay") +
  xlab("Destination airport") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```
````
````