12 Managing Data Frames with the `dplyr` package

Watch a video of this chapter

12.1 Data Frames

The ***data frame*** is a key data structure in statistics and in R. The basic structure of a data frame is that there is one observation per row and each column represents a variable, a measure, feature, or characteristic of that observation. R has an internal implementation of data frames that is likely the one you will use most often. However, there are packages on CRAN that implement data frames via things like relational databases that allow you to operate on very very large data frames (but we won't discuss them here).

Given the importance of managing data frames, it's important that we have good tools for dealing with them. In previous chapters we have already discussed some tools like the `subset()` function and the use of `[` and `$` operators to extract subsets of data frames. However, other operations, like filtering, re-ordering, and collapsing, can often be tedious operations in R whose syntax is not very intuitive. The `dplyr` package is designed to mitigate a lot of these problems and to provide a highly optimized set of routines specifically for dealing with data frames.

12.2 The `dplyr` Package

The `dplyr` package was developed by Hadley Wickham of RStudio and is an optimized and distilled version of his `plyr` package. The `dplyr` package does not provide any "new" functionality to R per se, in the sense that everything `dplyr` does could already be done with base R, but it ***greatly*** simplifies existing functionality in R.

One important contribution of the `dplyr` package is that it provides a "grammar" (in particular, verbs) for data manipulation and for operating on data frames. With this grammar, you can sensibly communicate what it is that you are doing to a data frame that other people can understand (assuming they also know the grammar). This is useful because it provides an abstraction for data manipulation that previously did not exist. Another useful contribution is that the `dplyr` functions are **very** fast, as many key operations are coded in C++.

12.3 `dplyr` Grammar

Some of the key "verbs" provided by the `dplyr` package are

- `select` : return a subset of the columns of a data frame, using a flexible notation
- `filter` : extract a subset of rows from a data frame based on logical conditions
- `arrange` : reorder rows of a data frame
- `rename` : rename variables in a data frame
- `mutate` : add new variables/columns or transform existing variables
- `summarise / summarize` : generate summary statistics of different variables in the data frame, possibly within strata
- `%>%` : the "pipe" operator is used to connect multiple verb actions together into a pipeline

The `dplyr` package as a number of its own data types that it takes advantage of. For example, there is a handy `print` method that prevents you from printing a lot of data to the console. Most of the time, these additional data types are transparent to the user and do not need to be worried about.

12.3.1 Common `dplyr` Function Properties

All of the functions that we will discuss in this Chapter will have a few common characteristics. In particular,

1. The first argument is a data frame.

2. The subsequent arguments describe what to do with the data frame specified in the first argument, and you can refer to columns in the data frame directly without using the $ operator (just use the column names).
3. The return result of a function is a new data frame
4. Data frames must be properly formatted and annotated for this to all be useful. In particular, the data must be tidy. In short, there should be one observation per row, and each column should represent a feature or characteristic of that observation.

12.4 Installing the `dplyr` package

The `dplyr` package can be installed from CRAN or from GitHub using the `devtools` package and the `install_github()` function. The GitHub repository will usually contain the latest updates to the package and the development version.

To install from CRAN, just run

```
> install.packages("dplyr")
```

To install from GitHub you can run

```
> install_github("hadley/dplyr")
```

After installing the package it is important that you load it into your R session with the `library()` function.

```
> 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
```

You may get some warnings when the package is loaded because there are functions in the `dplyr` package that have the same name as functions in other packages. For now you can ignore the warnings.

12.5 `select()`

For the examples in this chapter we will be using a dataset containing air pollution and temperature data for the city of Chicago in the U.S. The dataset is available from my web site.

After unzipping the archive, you can load the data into R using the `readRDS()` function.

```
> chicago <- readRDS("chicago.rds")
```

You can see some basic characteristics of the dataset with the `dim()` and `str()` functions.

```
> dim(chicago)
[1] 6940    8
> str(chicago)
'data.frame':   6940 obs. of  8 variables:
 $ city      : chr  "chic" "chic" "chic" "chic" ...
 $ tmpd      : num  31.5 33 33 29 32 40 34.5 29 26.5 32.5 ...
 $ dptp      : num  31.5 29.9 27.4 28.6 28.9 ...
 $ date      : Date, format: "1987-01-01" "1987-01-02" ...
 $ pm25tmean2: num  NA NA NA NA NA NA NA NA NA NA ...
 $ pm10tmean2: num  34 NA 34.2 47 NA ...
 $ o3tmean2  : num  4.25 3.3 3.33 4.38 4.75 ...
 $ no2tmean2 : num  20 23.2 23.8 30.4 30.3 ...
```

The `select()` function can be used to select columns of a data frame that you want to focus on. Often you'll have a large data frame containing "all" of the data, but any ***given*** analysis might only use a subset of variables or observations. The `select()` function allows you to get the few columns you might need.

Suppose we wanted to take the first 3 columns only. There are a few ways to do this. We could for example use numerical indices. But we can also use the names directly.

```
> names(chicago)[1:3]
[1] "city" "tmpd" "dptp"
> subset <- select(chicago, city:dptp)
> head(subset)
  city tmpd   dptp
1 chic 31.5 31.500
2 chic 33.0 29.875
3 chic 33.0 27.375
4 chic 29.0 28.625
5 chic 32.0 28.875
6 chic 40.0 35.125
```

Note that the `:` normally cannot be used with names or strings, but inside the `select()` function you can use it to specify a range of variable names.

You can also ***omit*** variables using the `select()` function by using the negative sign. With `select()` you can do

```
> select(chicago, -(city:dptp))
```

which indicates that we should include every variable ***except*** the variables `city` through `dptp`. The equivalent code in base R would be

```
> i <- match("city", names(chicago))
> j <- match("dptp", names(chicago))
> head(chicago[, -(i:j)])
```

Not super intuitive, right?

The `select()` function also allows a special syntax that allows you to specify variable names based on patterns. So, for example, if you wanted to keep every variable that ends with a "2", we could do

```
> subset <- select(chicago, ends_with("2"))
> str(subset)
'data.frame':   6940 obs. of  4 variables:
 $ pm25tmean2: num  NA NA NA NA NA NA NA NA NA NA ...
 $ pm10tmean2: num  34 NA 34.2 47 NA ...
 $ o3tmean2  : num  4.25 3.3 3.33 4.38 4.75 ...
 $ no2tmean2 : num  20 23.2 23.8 30.4 30.3 ...
```

Or if we wanted to keep every variable that starts with a "d", we could do

```
> subset <- select(chicago, starts_with("d"))
> str(subset)
'data.frame':   6940 obs. of  2 variables:
 $ dptp: num  31.5 29.9 27.4 28.6 28.9 ...
 $ date: Date, format: "1987-01-01" "1987-01-02" ...
```

You can also use more general regular expressions if necessary. See the help page (`?select`) for more details.

12.6 `filter()`

The `filter()` function is used to extract subsets of rows from a data frame. This function is similar to the existing `subset()` function in R but is quite a bit faster in my experience.

Suppose we wanted to extract the rows of the `chicago` data frame where the levels of PM2.5 are greater than 30 (which is a reasonably high level), we could do

```
> chic.f <- filter(chicago, pm25tmean2 > 30)
> str(chic.f)
'data.frame':   194 obs. of  8 variables:
 $ city      : chr  "chic" "chic" "chic" "chic" ...
 $ tmpd      : num  23 28 55 59 57 57 75 61 73 78 ...
 $ dptp      : num  21.9 25.8 51.3 53.7 52 56 65.8 59 60.3 67.1 ...
 $ date      : Date, format: "1998-01-17" "1998-01-23" ...
 $ pm25tmean2: num  38.1 34 39.4 35.4 33.3 ...
 $ pm10tmean2: num  32.5 38.7 34 28.5 35 ...
 $ o3tmean2  : num  3.18 1.75 10.79 14.3 20.66 ...
 $ no2tmean2 : num  25.3 29.4 25.3 31.4 26.8 ...
```

You can see that there are now only 194 rows in the data frame and the distribution of the `pm25tmean2` values is.

```
> summary(chic.f$pm25tmean2)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  30.05   32.12   35.04   36.63   39.53   61.50
```

We can place an arbitrarily complex logical sequence inside of `filter()` , so we could for example extract the rows where PM2.5 is greater than 30 ***and*** temperature is greater than 80 degrees Fahrenheit.

```
> chic.f <- filter(chicago, pm25tmean2 > 30 & tmpd > 80)
> select(chic.f, date, tmpd, pm25tmean2)
         date tmpd pm25tmean2
1  1998-08-23   81   39.60000
2  1998-09-06   81   31.50000
3  2001-07-20   82   32.30000
4  2001-08-01   84   43.70000
5  2001-08-08   85   38.83750
6  2001-08-09   84   38.20000
7  2002-06-20   82   33.00000
8  2002-06-23   82   42.50000
9  2002-07-08   81   33.10000
10 2002-07-18   82   38.85000
11 2003-06-25   82   33.90000
12 2003-07-04   84   32.90000
13 2005-06-24   86   31.85714
14 2005-06-27   82   51.53750
15 2005-06-28   85   31.20000
16 2005-07-17   84   32.70000
17 2005-08-03   84   37.90000
```

Now there are only 17 observations where both of those conditions are met.

12.7 `arrange()`

The `arrange()` function is used to reorder rows of a data frame according to one of the variables/columns. Reordering rows of a data frame (while preserving corresponding order of other columns) is normally a pain to do in R. The `arrange()` function simplifies the process quite a bit.

Here we can order the rows of the data frame by date, so that the first row is the earliest (oldest) observation and the last row is the latest (most recent) observation.

```
> chicago <- arrange(chicago, date)
```

We can now check the first few rows

```
> head(select(chicago, date, pm25tmean2), 3)
        date pm25tmean2
1 1987-01-01         NA
2 1987-01-02         NA
3 1987-01-03         NA
```

and the last few rows.

```
> tail(select(chicago, date, pm25tmean2), 3)
           date pm25tmean2
6938 2005-12-29    7.45000
6939 2005-12-30   15.05714
6940 2005-12-31   15.00000
```

Columns can be arranged in descending order too by useing the special `desc()` operator.

```
> chicago <- arrange(chicago, desc(date))
```

Looking at the first three and last three rows shows the dates in descending order.

```
> head(select(chicago, date, pm25tmean2), 3)
        date pm25tmean2
1 2005-12-31   15.00000
2 2005-12-30   15.05714
3 2005-12-29    7.45000
> tail(select(chicago, date, pm25tmean2), 3)
           date pm25tmean2
6938 1987-01-03         NA
6939 1987-01-02         NA
6940 1987-01-01         NA
```

12.8 rename()

Renaming a variable in a data frame in R is surprisingly hard to do! The `rename()` function is designed to make this process easier.

Here you can see the names of the first five variables in the `chicago` data frame.

```
> head(chicago[, 1:5], 3)
  city tmpd dptp       date pm25tmean2
1 chic   35 30.1 2005-12-31   15.00000
2 chic   36 31.0 2005-12-30   15.05714
3 chic   35 29.4 2005-12-29    7.45000
```

The `dptp` column is supposed to represent the dew point temperature adn the `pm25tmean2` column provides the PM2.5 data. However, these names are pretty obscure or awkward and probably be renamed to something more sensible.

```
> chicago <- rename(chicago, dewpoint = dptp, pm25 = pm25tmean2)
> head(chicago[, 1:5], 3)
  city tmpd dewpoint       date     pm25
1 chic   35     30.1 2005-12-31 15.00000
2 chic   36     31.0 2005-12-30 15.05714
3 chic   35     29.4 2005-12-29  7.45000
```

The syntax inside the `rename()` function is to have the new name on the left-hand side of the `=` sign and the old name on the right-hand side.

I leave it as an exercise for the reader to figure how you do this in base R without `dplyr` .

12.9 mutate()

The `mutate()` function exists to compute transformations of variables in a data frame. Often, you want to create new variables that are derived from existing variables and `mutate()` provides a clean interface for doing that.

For example, with air pollution data, we often want to ***detrend*** the data by subtracting the mean from the data. That way we can look at whether a given day's air pollution level is higher than or less than average (as opposed to looking at its absolute level).

Here we create a `pm25detrend` variable that subtracts the mean from the `pm25` variable.

```
> chicago <- mutate(chicago, pm25detrend = pm25 - mean(pm25, na.rm = TRUE))
> head(chicago)
  city tmpd dewpoint       date     pm25 pm10tmean2  o3tmean2 no2tmean2
1 chic   35     30.1 2005-12-31 15.00000       23.5  2.531250  13.25000
2 chic   36     31.0 2005-12-30 15.05714       19.2  3.034420  22.80556
3 chic   35     29.4 2005-12-29  7.45000       23.5  6.794837  19.97222
4 chic   37     34.5 2005-12-28 17.75000       27.5  3.260417  19.28563
5 chic   40     33.6 2005-12-27 23.56000       27.0  4.468750  23.50000
6 chic   35     29.6 2005-12-26  8.40000        8.5 14.041667  16.81944
  pm25detrend
1   -1.230958
2   -1.173815
3   -8.780958
4    1.519042
5    7.329042
6   -7.830958
```

There is also the related `transmute()` function, which does the same thing as `mutate()` but then ***drops all non-transformed variables***.

Here we detrend the PM10 and ozone (O3) variables.

```
> head(transmute(chicago,
+                pm10detrend = pm10tmean2 - mean(pm10tmean2, na.rm = TRUE),
+                o3detrend = o3tmean2 - mean(o3tmean2, na.rm = TRUE)))
  pm10detrend  o3detrend
1  -10.395206 -16.904263
2  -14.695206 -16.401093
3  -10.395206 -12.640676
4   -6.395206 -16.175096
5   -6.895206 -14.966763
6  -25.395206  -5.393846
```

Note that there are only two columns in the transmuted data frame.

12.10 group_by()

The `group_by()` function is used to generate summary statistics from the data frame within strata defined by a variable. For example, in this air pollution dataset, you might want to know what the average annual level of PM2.5 is. So the stratum is the year, and that is something we can derive from the `date` variable. In conjunction with the `group_by()` function we often use the `summarize()` function (or `summarise()` for some parts of the world).

The general operation here is a combination of splitting a data frame into separate pieces defined by a variable or group of variables (`group_by()`), and then applying a summary function across those subsets (`summarize()`).

First, we can create a `year` varible using `as.POSIXlt()` .

```
> chicago <- mutate(chicago, year = as.POSIXlt(date)$year + 1900)
```

Now we can create a separate data frame that splits the original data frame by year.

```
> years <- group_by(chicago, year)
```

Finally, we compute summary statistics for each year in the data frame with the `summarize()` function.

```
> summarize(years, pm25 = mean(pm25, na.rm = TRUE),
+           o3 = max(o3tmean2, na.rm = TRUE),
+           no2 = median(no2tmean2, na.rm = TRUE))
`summarise()` ungrouping output (override with `.groups` argument)
# A tibble: 19 x 4
    year  pm25    o3   no2
   <dbl> <dbl> <dbl> <dbl>
 1  1987 NaN    63.0  23.5
 2  1988 NaN    61.7  24.5
 3  1989 NaN    59.7  26.1
 4  1990 NaN    52.2  22.6
 5  1991 NaN    63.1  21.4
 6  1992 NaN    50.8  24.8
 7  1993 NaN    44.3  25.8
 8  1994 NaN    52.2  28.5
 9  1995 NaN    66.6  27.3
10  1996 NaN    58.4  26.4
11  1997 NaN    56.5  25.5
12  1998  18.3  50.7  24.6
13  1999  18.5  57.5  24.7
14  2000  16.9  55.8  23.5
15  2001  16.9  51.8  25.1
16  2002  15.3  54.9  22.7
17  2003  15.2  56.2  24.6
18  2004  14.6  44.5  23.4
19  2005  16.2  58.8  22.6
```

`summarize()` returns a data frame with `year` as the first column, and then the annual averages of `pm25`, `o3`, and `no2`.

In a slightly more complicated example, we might want to know what are the average levels of ozone (`o3`) and nitrogen dioxide (`no2`) within quintiles of `pm25`. A slicker way to do this would be through a regression model, but we can actually do this quickly with `group_by()` and `summarize()`.

First, we can create a categorical variable of `pm25` divided into quintiles.

```
> qq <- quantile(chicago$pm25, seq(0, 1, 0.2), na.rm = TRUE)
> chicago <- mutate(chicago, pm25.quint = cut(pm25, qq))
```

Now we can group the data frame by the `pm25.quint` variable.

```
> quint <- group_by(chicago, pm25.quint)
```

Finally, we can compute the mean of `o3` and `no2` within quintiles of `pm25`.

```
> summarize(quint, o3 = mean(o3tmean2, na.rm = TRUE),
+           no2 = mean(no2tmean2, na.rm = TRUE))
`summarise()` ungrouping output (override with `.groups` argument)
# A tibble: 6 x 3
  pm25.quint     o3   no2
  <fct>       <dbl> <dbl>
1 (1.7,8.7]    21.7  18.0
2 (8.7,12.4]   20.4  22.1
3 (12.4,16.7]  20.7  24.4
4 (16.7,22.6]  19.9  27.3
5 (22.6,61.5]  20.3  29.6
6 <NA>         18.8  25.8
```

From the table, it seems there isn't a strong relationship between `pm25` and `o3`, but there appears to be a positive correlation between `pm25` and `no2`. More sophisticated statistical modeling can help to provide precise answers to these questions, but a simple application of `dplyr` functions can often get you most of the way there.

12.11 %>%

The pipeline operater `%>%` is very handy for stringing together multiple `dplyr` functions in a sequence of operations. Notice above that every time we wanted to apply more than one function, the sequence gets buried in a sequence of nested function calls that is difficult to read, i.e.

```
> third(second(first(x)))
```

This nesting is not a natural way to think about a sequence of operations. The `%>%` operator allows you to string operations in a left-to-right fashion, i.e.

```
> first(x) %>% second %>% third
```

Take the example that we just did in the last section where we computed the mean of `o3` and `no2` within quintiles of `pm25`. There we had to

1. create a new variable `pm25.quint`
2. split the data frame by that new variable
3. compute the mean of `o3` and `no2` in the sub-groups defined by `pm25.quint`

That can be done with the following sequence in a single R expression.

```
> mutate(chicago, pm25.quint = cut(pm25, qq)) %>%
+         group_by(pm25.quint) %>%
+         summarize(o3 = mean(o3tmean2, na.rm = TRUE),
+                   no2 = mean(no2tmean2, na.rm = TRUE))
`summarise()` ungrouping output (override with `.groups` argument)
# A tibble: 6 x 3
  pm25.quint     o3   no2
  <fct>       <dbl> <dbl>
1 (1.7,8.7]    21.7  18.0
2 (8.7,12.4]   20.4  22.1
3 (12.4,16.7]  20.7  24.4
4 (16.7,22.6]  19.9  27.3
5 (22.6,61.5]  20.3  29.6
6 <NA>         18.8  25.8
```

This way we don't have to create a set of temporary variables along the way or create a massive nested sequence of function calls.

Notice in the above code that I pass the `chicago` data frame to the first call to `mutate()`, but then afterwards I do not have to pass the first argument to `group_by()` or `summarize()`. Once you travel down the pipeline with `%>%`, the first argument is taken to be the output of the previous element in the pipeline.

Another example might be computing the average pollutant level by month. This could be useful to see if there are any seasonal trends in the data.

```
> mutate(chicago, month = as.POSIXlt(date)$mon + 1) %>%
+         group_by(month) %>%
+         summarize(pm25 = mean(pm25, na.rm = TRUE),
+                   o3 = max(o3tmean2, na.rm = TRUE),
+                   no2 = median(no2tmean2, na.rm = TRUE))
`summarise()` ungrouping output (override with `.groups` argument)
# A tibble: 12 x 4
   month  pm25    o3   no2
   <dbl> <dbl> <dbl> <dbl>
 1     1  17.8  28.2  25.4
 2     2  20.4  37.4  26.8
 3     3  17.4  39.0  26.8
 4     4  13.9  47.9  25.0
 5     5  14.1  52.8  24.2
 6     6  15.9  66.6  25.0
 7     7  16.6  59.5  22.4
 8     8  16.9  54.0  23.0
 9     9  15.9  57.5  24.5
10    10  14.2  47.1  24.2
11    11  15.2  29.5  23.6
12    12  17.5  27.7  24.5
```

Here we can see that `o3` tends to be low in the winter months and high in the summer while `no2` is higher in the winter and lower in the summer.

12.12 Summary

The `dplyr` package provides a concise set of operations for managing data frames. With these functions we can do a number of complex operations in just a few lines of code. In particular, we can often conduct the beginnings of an exploratory analysis with the powerful combination of `group_by()` and `summarize()`.

Once you learn the `dplyr` grammar there are a few additional benefits

- `dplyr` can work with other data frame "backends" such as SQL databases. There is an SQL interface for relational databases via the DBI package
- `dplyr` can be integrated with the `data.table` package for large fast tables

The `dplyr` package is handy way to both simplify and speed up your data frame management code. It's rare that you get such a combination at the same time!