Introduction to Dplyr

Simeon Markind

2017-02-21

Overview of R Packages

Before we begin we will make sure you have the "dplyr" and "nycflights13" packages installed.

So far we have worked in "Base" R, that is to say all of the functions that we have used and the code you have seen is available by default to anyone who downloads R. One of the most powerful aspects of R, and the main reason that R is so powerful is that it is "open-source" meaning that anyone can write new functions to R, post them online and let other people use the new functions. When people want to write R code to share with others they do so by writing up their code as a "package" and hosting it on CRAN, (Comprehensive R Archive Network). There are currently over 10,000 packages on CRAN, and you can download any and all of them for free to utilize whatever functionality they have!

CRAN homepage can be found at: https://cran.r-project.org. Click on 'R packages' on the left side to see all the packages available.

These packages usually include functions that are not available in Base R and allow you to more easily do certain things, such as making graphs. GGplot2 is one of the most popular R packages and is great to making beautiful plots. We will discuss this package later in the course. Today however we will be discussing the Dplyr package which includes functions for improved data subsetting and manipulation.

Ok, so we know that packages live on CRAN but how do we get them from CRAN down to our own computer so we can use them? Great question with a straightforward answer! First we need to install the package to our computer. To do this we use the `install.packages()` function. To this function we include one argument, the quoted name of the package we want to download. So for now you should type `install.packages("dplyr")` into your console and hit enter.

Now we have the dplyr package installed onto our computer but we don't yet have it available for us to use in R. In order for that to happen we need to attach the package to R by using the `library()` function. This function also takes one input, the package name, either quoted or unquoted. In your console type `library(dplyr)` to get the dplyr package to install.

There are many ways to get help in using and understanding a package in R. First you can just type `?packageName` into you console. Type `?dplyr` and see what comes up. To learn more about the package try the function `browseVignettes()`. This is a function that shows some of the help documentation for a package. Try typing `browseVignettes(package = "dplyr")`. In addition, you can look at the package documentation posted to CRAN. I usually just google the package name and the documentation is one of the top results. Here is the pdf for the dplyr package manual which lists the main functions of the package in detail: https://cran.r-project.org/web/packages/dplyr/dplyr.pdf

Now that you know how to install and load packages and you succesfully loaded Dplyr, repeat the process with the "nycflights13" packages.

One thing to note is that you only need to install the package one time. Once the package is installed it is always available to you in R. However, every time you restart R you do need to load in the package with the `library()` function, just because a package is installed does mean you can automatically use it. In addition, packages can and do get updated, in order to get a more updated version of a package you need to reinstall it with `install.packages()` again. This overwrites your old version with the new version. Once you have re-installed the updated version you can once again use it with the `library()` function as per normal.

```r
library(dplyr)
```

Introduction

In this lecture we will cover some of the main features of dplyr, a popular package for manipulating data frames. The dplyr package provides an easy to understand grammar for data manipulation which allows for concise flow for data processing. In this lecture we will go over the main verbs of the dplyr package and then introduce the pipe operator, (which may be familiar to anyone who has used Linux), to put them all together.

dplyr provides:

- a set of functions for efficiently manipulating data sets
- a grammar of data manipulation

To see all functions included in the dplyr package simply type `ls("package:dplyr")` into the command line. As you will see, there are over 200 functions in the package and we will not touch upon all of them.

Example data

How many of you have ever ridden in an airplane? How many of you had to take a plane to get to Howard? How many of you have flown into one of the major New York City airports (LaGuardia, Newark, JFK)? How many of you have ever been on a delayed flight before? Today we are going to look at a dataset put together by United States Bureau of Transportation Statistics located in the nycflights13 package:

```
library(nycflights13)
```

To explore dplyr let's use the nycflights13::flights table.

You may be wondering what nycflights13::flights means. Well, in this case I am specifying both the package and the object within the package that I am looking for. The syntax for this is package::object. This works with data frames, (as we see here), as well as with functions. For example, if I want to specify that I am using the sum function from the base package I could type `base::sum(4,5)` and get 9 out. Normally we don't need to include the package name before a function because we just use the default version of the function. In this case we could just call `flights` at the console because we don't have anything else called flights loaded so R knows that what we mean is `nycflights13::flights`. For the rest of the lecture I will simply refer to the `nycflights13::flights` data as `flights`.

Let's take a look at our data:

```
flights
```

```
Output > # A tibble: 336,776 x 19
Output >     year month   day dep_time sched_dep_time dep_delay arr_time
Output >    <int> <int> <int>    <int>          <int>     <dbl>    <int>
Output > 1   2013     1     1      517            515         2      830
Output > 2   2013     1     1      533            529         4      850
Output > 3   2013     1     1      542            540         2      923
Output > 4   2013     1     1      544            545        -1     1004
Output > 5   2013     1     1      554            600        -6      812
Output > 6   2013     1     1      554            558        -4      740
Output > 7   2013     1     1      555            600        -5      913
Output > 8   2013     1     1      557            600        -3      709
Output > 9   2013     1     1      557            600        -3      838
Output > 10  2013     1     1      558            600        -2      753
Output > # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
Output > #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
Output > #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
Output > #   minute <dbl>, time_hour <time>
```

So we have year, month and day columns as well as departure and arrival times etc. To see a description of all the columns in the data, type `?flights` at the console.

Also, you may notice that our data looks a little different than a regular data frame in R. That's because in addition to being a data frame it is also a tibble! One of the nice features of a tibble is that it prints differently than a regular data frame, only the first few rows get printed so that it does not overwhelm you. Additonally only the columns that fit get printed and the text at the bottom of the table lists all unprinted columns and their data types. If you look at the columns of our data above, underneath the column names you can see what type each variable is. This is useful so that you know if you have to convert something to character, numeric, etc.

To turn a regular data frame into a tibble you will need to have the Dplyr package loaded and use the `tbl_df()` function.

Ok, let's get started with dplyr!

Load the dplyr library and the nycflights13 library, call head(flights) to make sure that the data is loaded correctly

dplyr verbs

dplyr is built around 5 main functions.

Each function takes a data set as input, and returns a data set as output. Each function name is a "verb" that describes the specific action being performed on the data set

These verbs cover the most common data manipulation tasks

- Select certain columns of data.
- Filter your data to select specific rows.
- Arrange the rows of your data into an order.
- Mutate your data frame to contain new columns.
- Summarize chunks of you data in some way.

Let's look at how those work.

Verb 1: Select

Include or exclude specific variables

The flights table has 19 columns and 336776 rows. We can easily find this out using the `dim()` function available in the base package which tells us the rows and columns of an object.

```r
dim(flights)
```

```
Output > [1] 336776     19
```

What other functions could you use to get the number of rows and columns of a data frame?

Include only a few variables by listing their names. Do not quote the variable names. Unlike with Base R, where in order to select columns you need to quote the column names, in Dplyr you do not need to.

```r
## We select columns in Base R by quoting the column names
baseR <- flights[, c("year", "month", "day", "sched_dep_time", "sched_arr_time")]

## Using the Dplyr select function we do not need to quote the column names
flight_cols <- select(flights, year, month, day, sched_dep_time, sched_arr_time)

dim(flight_cols)
```

```
Output > [1] 336776      5
```

```
head(flight_cols)
```

```
Output > # A tibble: 6 x 5
Output >    year month   day sched_dep_time sched_arr_time
Output >   <int> <int> <int>          <int>          <int>
Output > 1  2013     1     1            515            819
Output > 2  2013     1     1            529            830
Output > 3  2013     1     1            540            850
Output > 4  2013     1     1            545           1022
Output > 5  2013     1     1            600            837
Output > 6  2013     1     1            558            728
```

```
identical(flight_cols, baseR)
```

```
Output > [1] TRUE
```

We can also select all variables except for listed unwanted variables by putting a minus sign in front of the variable names we don't want in the select statement:

```
deselect <- select(flights, -carrier, -tailnum)

dim(deselect)
```

```
Output > [1] 336776     17
```

```
head(deselect)
```

```
Output > # A tibble: 6 x 17
Output >    year month   day dep_time sched_dep_time dep_delay arr_time
Output >   <int> <int> <int>    <int>          <int>     <dbl>    <int>
Output > 1  2013     1     1      517            515         2      830
Output > 2  2013     1     1      533            529         4      850
Output > 3  2013     1     1      542            540         2      923
Output > 4  2013     1     1      544            545        -1     1004
Output > 5  2013     1     1      554            600        -6      812
Output > 6  2013     1     1      554            558        -4      740
Output > # ... with 10 more variables: sched_arr_time <int>, arr_delay <dbl>,
Output > #   flight <int>, origin <chr>, dest <chr>, air_time <dbl>,
Output > #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <time>
```

OK, so now that we have the flights data loaded and dplyr package loaded it's time for you to test it out. Create a table called inClass1 where you select the following columns from flights: hour, minute, dep_delay, arr_delay, air_time. Your table should look like this:

```
Output > # A tibble: 336,776 x 5
Output >     hour minute dep_delay arr_delay air_time
Output >    <dbl>  <dbl>     <dbl>     <dbl>    <dbl>
Output > 1      5     15         2        11      227
Output > 2      5     29         4        20      227
Output > 3      5     40         2        33      160
Output > 4      5     45        -1       -18      183
Output > 5      6      0        -6       -25      116
Output > 6      5     58        -4        12      150
Output > 7      6      0        -5        19      158
Output > 8      6      0        -3       -14       53
Output > 9      6      0        -3        -8      140
Output > 10     6      0        -2         8      138
Output > # ... with 336,766 more rows
```

Verb 2: Filter

We used Select to decide whether to include certain columns in our output dataset. Filter allows us to decide to include certain rows in our datset. This is just like subsetting operations we have already learned for data.frames.

The flights table has 336776 rows, way more that we can easily look at. Let's use the filter function to look at a subset of the rows, type `?filter` into your console for detailed information on the function:

What arguments does the filter function take?

```r
## We will filter based on the month values in the flights data.
## First let's take a look at the different values for the month variable in the
## table.

unique(flights$month)
```

```
Output >  [1]  1 10 11 12  2  3  4  5  6  7  8  9
```

```r
#Let's first filter down to only the flights that occurred in January (month == 1)

Jan <- filter(flights, month == 1)

nrow(Jan)
```

```
Output > [1] 27004
```

```r
# Use the nrow function to get the number of rows in a data frame.
# We filtered out most of the rows in the flights data

ncol(Jan)
```

```
Output > [1] 19
```

```r
## Use the ncol function to get the number of columns in a data frame.
## notice that we did not select any columns when we created Jan,
## so we therefore have all the columns the original flights data came with

unique(Jan$month)
```

```
Output > [1] 1
```

```r
# there is only one possible value of month in the table, which we would expect since
# we filtered based on the month value.
```

Notice that in order for the filter to be successful you need to provide a condition for the function to filter on.

What happens if you forget to do that, what does the following command do? `inClass2 <- filter(flights, day)`

So we are in luck, we will never accidentally forget to include a condition.

Filtering on Multiple Columns

The filter function has the ability to take multiple arguments. So we can filter on multiple conditions all at once. Here I filter on three columns at the same time.

```
multiple_filter <- filter(flights, day > 10, dep_delay <= 5, !is.na(arr_time))

dim(multiple_filter)
```

```
Output > [1] 153648     19
```

```
head(multiple_filter)
```

```
Output > # A tibble: 6 x 19
Output >    year month   day dep_time sched_dep_time dep_delay arr_time
Output >   <int> <int> <int>    <int>          <int>     <dbl>    <int>
Output > 1  2013     1    11      453            500        -7      643
Output > 2  2013     1    11      519            525        -6      735
Output > 3  2013     1    11      520            530       -10      817
Output > 4  2013     1    11      530            540       -10      812
Output > 5  2013     1    11      538            540        -2      956
Output > 6  2013     1    11      549            600       -11      642
Output > # ... with 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
Output > #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
Output > #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
Output > #   time_hour <time>
```

So we see that day is greater than 10 and departure delay is less than or equal to 5 and arrival time is not NA. (`is.na()` tests if something is equal to NA, putting the `!` in front of the function negates the function, so `!is.na()` returns TRUE for everything that is not equal to NA).

Ok, now make a data frame, inClass3 of rows from the flights data from only flights in the months after June, and only flights before the 15th of the month with a departure time after noon.

```
Output > # A tibble: 6 x 19
Output >    year month   day dep_time sched_dep_time dep_delay arr_time
Output >   <int> <int> <int>    <int>          <int>     <dbl>    <int>
Output > 1  2013    10     1     1207           1210        -3     1407
Output > 2  2013    10     1     1208           1210        -2     1307
Output > 3  2013    10     1     1209           1215        -6     1419
Output > 4  2013    10     1     1212           1215        -3     1416
Output > 5  2013    10     1     1212           1200        12     1441
Output > 6  2013    10     1     1213           1159        14     1321
Output > # ... with 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
Output > #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
Output > #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
Output > #   time_hour <time>
```

Notice that separating each filter via a comma is the same as using an & in between each filter. Our function to create multiple_filter above is the same as `flights[ flights$day > 10 & flights$dep_delay <= 5 & !is.na(flights$arr_time),]`.

If instead we want to use the OR option for filtering we need to do that without using commas in the filter function. We can do so like this:

```
filter1 <- filter(flights, day == 1 | day == 2, month == 12 | month == 11)

head(filter1)
```

```
Output > # A tibble: 6 x 19
Output >    year month   day dep_time sched_dep_time dep_delay arr_time
Output >   <int> <int> <int>    <int>          <int>     <dbl>    <int>
Output > 1  2013    11     1        5           2359         6      352
Output > 2  2013    11     1       35           2250       105      123
Output > 3  2013    11     1      455            500        -5      641
Output > 4  2013    11     1      539            545        -6      856
Output > 5  2013    11     1      542            545        -3      831
Output > 6  2013    11     1      549            600       -11      912
Output > # ... with 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
Output > #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
Output > #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
Output > #   time_hour <time>
```

Here we take all the flights data with a day of 1 or 2 and a month of 11 or 12.

We can also do similar things using the `%in%` operator, here is the same thing without using the `|` operator. `%in%` compares the vector of input with the vector of options and returns `TRUE` for each element of input that is also in the vector of options. I'll show you an example of `%in%` and then use it to filter the same way as above.

```
inEx <- c(1:5)
inEx %in% c(2,4)
```

```
Output > [1] FALSE  TRUE FALSE  TRUE FALSE
```

```
## Now we use %in% to recreate the filter example from above
filter2 <- filter(flights, day %in% c(1,2), month %in% c(11,12))
```

As a reminder, the `,` in `filter()` take the place of the `&` symbol. We can also rewrite the above filter command using `&` in place of commas.

```
filter3 <- filter(flights, (day == 1 | day == 2) & (month == 11 | month == 12))

## We use the identical function to check that they are all the same output:

identical(filter1, filter2)
```

```
Output > [1] TRUE
```

```
identical(filter1, filter3)
```

```
Output > [1] TRUE
```

Now it's your turn: You can use any of the methods described above to return the following table, inClass4: month of 1, 4, 7, or 9, arrival time before noon or after 6 pm (note the use of 24 hour time in the dataset), a departure delay of greater than 10, and an arrival delay that is not NA.

Your output should have 19818 rows

Verb 3: Arrange

The arrange function allows us to order our data in ascending or descending order based on the columns of our choosing.

```
head(flights)
```

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

```r
arrEx1 <- arrange(flights, dep_time, sched_dep_time, arr_time)

head(arrEx1)
```

```
Output > # A tibble: 6 x 19
Output >    year month   day dep_time sched_dep_time dep_delay arr_time
Output >   <int> <int> <int>    <int>          <int>     <dbl>    <int>
Output > 1  2013     4    10        1           1930       271      106
Output > 2  2013     5    22        1           1935       266      154
Output > 3  2013     6    24        1           1950       251      105
Output > 4  2013     7     1        1           2029       212      236
Output > 5  2013     2    11        1           2100       181      111
Output > 6  2013     1    31        1           2100       181      124
Output > # ... with 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
Output > #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
Output > #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
Output > #   time_hour <time>
```

So we see that, as with filter and select, we can give multiple columns as arguments to the arrange function. The function then arranges in the order of the argument, so it arranges the dep_time first, then sched_dep_time, then arr_time.

Now let's switch up the order of the arrange function to see what I mean, create table inClass5 which should be the flights data arranged first by arr_time, then sched_dep_time, then dep_time.

```
Output > # A tibble: 336,776 x 19
Output >     year month   day dep_time sched_dep_time dep_delay arr_time
Output >    <int> <int> <int>    <int>          <int>     <dbl>    <int>
Output > 1   2013     3     8     2212           1539       393        1
Output > 2   2013     2    11     2159           1630       329        1
Output > 3   2013     6    24     2018           1711       187        1
Output > 4   2013     6    10     2051           1729       202        1
Output > 5   2013     9    12     2109           1730       219        1
Output > 6   2013     8     1     2157           1734       263        1
Output > 7   2013     6    25     2056           1755       181        1
Output > 8   2013    12     8     1909           1804        65        1
Output > 9   2013    12    14     1944           1815        89        1
Output > 10  2013     6    26     2008           1819       109        1
Output > # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
Output > #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
Output > #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
Output > #   minute <dbl>, time_hour <time>
```

The default ordering is ascending, but we can sort descending by using either the `desc()` function around our column name or putting a negative sign in front of the column name.

```
arrEx2 <- arrange(flights, desc(dep_time), -sched_dep_time, arr_time)

arrEx2
```

```
Output > # A tibble: 336,776 x 19
Output >     year month   day dep_time sched_dep_time dep_delay arr_time
Output >    <int> <int> <int>    <int>          <int>     <dbl>    <int>
Output > 1   2013     3    15     2400           2359         1      324
Output > 2   2013    10    30     2400           2359         1      327
Output > 3   2013     4    20     2400           2359         1      338
Output > 4   2013     4     2     2400           2359         1      339
Output > 5   2013     5    21     2400           2359         1      339
Output > 6   2013     8    20     2400           2359         1      354
Output > 7   2013     7    28     2400           2359         1      411
Output > 8   2013     9     2     2400           2359         1      411
Output > 9   2013    12     5     2400           2359         1      427
Output > 10  2013    12     9     2400           2359         1      432
Output > # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
Output > #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
Output > #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
Output > #   minute <dbl>, time_hour <time>
```

Verb 4: Mutate

So now that we know how to filter our data by rows, select the columns we want, and order the data, we want to start being able to do some calculations on the data. This is the where the `mutate()` function comes in which allows us to add new columns to a data set. The function creates a new value for every row.

Mutate adds columns to the end of the dataset, so before we give an example, we will select only a few columns so we can see everything that mutate adds.

Type ?mutate and tell me what the arguments are to the function

```
mData <- select(flights, sched_arr_time, arr_time, arr_delay, dep_delay, air_time)

mData <- mutate(mData,
                overshoot = arr_time - sched_arr_time,
                hours = air_time / 60,
                gain = dep_delay - arr_delay, #time made up in the air
                gain_per_hour = gain/hours)

mData
```

```
Output > # A tibble: 336,776 x 9
Output >    sched_arr_time arr_time arr_delay dep_delay air_time overshoot
Output >             <int>    <int>     <dbl>     <dbl>    <dbl>     <int>
Output > 1             819      830        11         2      227        11
Output > 2             830      850        20         4      227        20
Output > 3             850      923        33         2      160        73
Output > 4            1022     1004       -18        -1      183       -18
Output > 5             837      812       -25        -6      116       -25
Output > 6             728      740        12        -4      150        12
Output > 7             854      913        19        -5      158        59
Output > 8             723      709       -14        -3       53       -14
Output > 9             846      838        -8        -3      140        -8
Output > 10            745      753          8         -2      138          8
Output > # ... with 336,766 more rows, and 3 more variables: hours <dbl>,
Output > #   gain <dbl>, gain_per_hour <dbl>
```

Notice that we are able to use a column we added in a mutate command within the same mutate command if desired. (we calculated the gains column and then use it in the calculation of gains_per_hour within the same function).

We can also use mutate to change existing variables. Right now the air_time variable is in minutes, we instead want it to be in hours.

```
mData <- select(flights, year, month, day, air_time)

mData2 <- mutate(mData, air_time = air_time/60, #update existing variable
                 Quarter = ifelse(month %in% c(1,2,3), "Q1", "Q"), #create new var
                 Quarter = ifelse(month %in% c(4,5,6), "Q2", Quarter), #update var
                 Quarter = ifelse(month %in% c(7,8,9), "Q3", Quarter),
                 Quarter = ifelse(month %in% c(10,11,12), "Q4", Quarter))

head(mData2)
```

```
Output > # A tibble: 6 x 5
Output >    year month   day air_time Quarter
Output >   <int> <int> <int>    <dbl>   <chr>
Output > 1  2013     1     1 3.783333      Q1
Output > 2  2013     1     1 3.783333      Q1
Output > 3  2013     1     1 2.666667      Q1
Output > 4  2013     1     1 3.050000      Q1
Output > 5  2013     1     1 1.933333      Q1
Output > 6  2013     1     1 2.500000      Q1
```

Notice that the air_time variable is now updated and we did not add a new column to the data instead overwriting the old values previously there.

Create a data frame inClass6 where you take the flights data for the columns: dep_time, arr_time, air_time. Create two new columns: "deviance" as arr_time - dep_time and "match" as a boolean value of whether air_time is equal to deviance. Would you expect air time to equal arrival time - departure time?

```
Output > # A tibble: 6 x 7
Output >   dep_time arr_time air_time arr_delay dep_delay deviance match
Output >      <int>    <int>    <dbl>     <dbl>     <dbl>    <int> <lgl>
Output > 1      517      830      227        11         2      313 FALSE
Output > 2      533      850      227        20         4      317 FALSE
Output > 3      542      923      160        33         2      381 FALSE
Output > 4      544     1004      183       -18        -1      460 FALSE
Output > 5      554      812      116       -25        -6      258 FALSE
Output > 6      554      740      150        12        -4      186 FALSE
```

Now that you've created the inClass6 table, let's use it to see what fraction of flights have a deviance equal to their air time. We can do this by summing the "match" column. Since "match" is of type logical, on the tibble we have values corresponding to 1 == TRUE and 0 == FALSE. So let's first find the sum of the match column.

If you type `sum(inClass6$match)` into your console you will get 0. What do you get? What does this mean?

Verb 5: Summarise

Create a new variable aggregating data from multiple observations and reduce the number of observations

Compute the average departure delay, arrival delay, and the average gain.

```
summ <- summarise(flights, avg_dep_delay = mean(dep_delay, na.rm = T),
                  avg_arr_delay = mean(arr_delay, na.rm = T),
                  avg_gain = mean(arr_delay - dep_delay, na.rm = T))

summ
```

```
Output > # A tibble: 1 x 3
Output >   avg_dep_delay avg_arr_delay  avg_gain
Output >           <dbl>         <dbl>     <dbl>
Output > 1      12.63907      6.895377 -5.659779
```

Compute the average air time and standard deviation of air time with the `sd()` function

```
Output > # A tibble: 1 x 2
Output >       time time_sd
Output >      <dbl>   <dbl>
Output > 1 150.6865 93.6883
```

Pipes

dplyr provides another innovation: the ability to chain operations together in sequence with the pipe (%>%) operator.

The following examples are equivalent.

Example Without pipes

Repeatedly input and output a data frame for each step

```
df <- select(flights, year, month, day, arr_delay, dep_delay, air_time)
df <- filter(df, month > 6, day < 5, !is.na(arr_delay))
df <- summarise(df, avg_arr_delay = mean(arr_delay, na.rm = TRUE),
                avg_dep_delay = mean(dep_delay, na.rm = TRUE))
df
```

```
Output > # A tibble: 1 x 2
Output >   avg_arr_delay avg_dep_delay
Output >           <dbl>         <dbl>
Output > 1      5.430873      12.34279
```

Example With pipes

The output from each step is used as the input for the next step

```
df2 <- flights %>%
    select(year, month, day, arr_delay, dep_delay, air_time) %>%
    filter(month > 6, day < 5, !is.na(arr_delay)) %>%
    summarise(avg_arr_delay = mean(arr_delay, na.rm = TRUE),
              avg_dep_delay = mean(dep_delay, na.rm = TRUE))
```

```
df2
```

```
Output > # A tibble: 1 x 2
Output >   avg_arr_delay avg_dep_delay
Output >           <dbl>         <dbl>
Output > 1      5.430873      12.34279
```

```
df2 == df
```

```
Output >      avg_arr_delay avg_dep_delay
Output > [1,]          TRUE          TRUE
```

Pipes explained

The "." is a special object that represents the output from the prior step. By default the pipe operator will replace the first argument in the next function with the output from the previous function, so for our above example it was not needed, but we can use it in order to be more explicit. Going forward we will use the "." in the remainder of the lecture.

```
flights %>%
    select(., year, month, day, arr_delay, dep_delay, air_time) %>%
    filter(., month > 6, day < 5, !is.na(arr_delay)) %>%
    summarise(., avg_arr_delay = mean(arr_delay, na.rm = TRUE),
                avg_dep_delay = mean(dep_delay, na.rm = TRUE))
```

```
Output > # A tibble: 1 x 2
Output >   avg_arr_delay avg_dep_delay
Output >           <dbl>         <dbl>
Output > 1      5.430873      12.34279
```

Same as above but this time we included the "."

Pipes replace nested function calls – without them this would be the only way to avoid explicitly creating a temporary data set for each step or writing an indecipherable nested function call like the below.

```
summarise(
    filter(
        select(
            flights,
            year, month, day, arr_delay, dep_delay, air_time),
        month > 6, day < 5, !is.na(arr_delay)),
    avg_arr_delay = mean(arr_delay, na.rm = TRUE),
    avg_dep_delay = mean(dep_delay, na.rm = TRUE)
)
```

```
Output > # A tibble: 1 x 2
Output >   avg_arr_delay avg_dep_delay
Output >           <dbl>         <dbl>
Output > 1      5.430873      12.34279
```

As you can see, once again we get the same results but this time it is very difficult to understand the steps of what actually happened to create these numbers.

Now that we've seen how pipes work, we can use multiple "verbs" to get more useful results

Aggregate data

group_by allows you to summarize data separately for subgroups of observations. Just like the functions we have seen before you are able to group by multiple columns separated by commas.

What are the arguments to group_by? Type ?group_by

The group_by function does not change anything about your data frame, it merely sets metadata that is used by the summarise function when it aggregates your data.

```
## As you can see, if you run the group_by function by itself, even with argument, there is no change t

group_by(flights, month)
```

```
Output > Source: local data frame [336,776 x 19]
Output > Groups: month [12]
Output >
Output >     year month   day dep_time sched_dep_time dep_delay arr_time
Output >    <int> <int> <int>    <int>          <int>     <dbl>    <int>
Output > 1   2013     1     1      517            515         2      830
Output > 2   2013     1     1      533            529         4      850
Output > 3   2013     1     1      542            540         2      923
Output > 4   2013     1     1      544            545        -1     1004
Output > 5   2013     1     1      554            600        -6      812
Output > 6   2013     1     1      554            558        -4      740
Output > 7   2013     1     1      555            600        -5      913
Output > 8   2013     1     1      557            600        -3      709
Output > 9   2013     1     1      557            600        -3      838
Output > 10  2013     1     1      558            600        -2      753
Output > # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
Output > #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
Output > #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
Output > #   minute <dbl>, time_hour <time>
```

```
## Notice that now at the top of the tibble we have the information about "Groups:"
```

Let's calculate the number of flights, average arrival delay, and average departure delay by airport. How would you do this in Base R?

Yeah, I wouldn't want to think about that either. Below we are able to answer that question in only 5 lines of code and what we did makes sense when you read it! This is the power and beauty of Dplyr.

```
flights %>%
  group_by(origin) %>%
  summarise(flight_count = n(),
            avg_arr_delay = mean(arr_delay, na.rm = T),
            avg_dep_delay = mean(dep_delay, na.rm = T))
```

```
Output > # A tibble: 3 x 4
Output >   origin flight_count avg_arr_delay avg_dep_delay
Output >    <chr>        <int>         <dbl>         <dbl>
Output > 1    EWR       120835      9.107055      15.10795
Output > 2    JFK       111279      5.551481      12.11216
Output > 3    LGA       104662      5.783488      10.34688
```

So it looks like EWR (Newark) has the longest average delays. Let's find out if flights leaving Newark go farther than the other airports.

Create a table inClass8 using the flights data and again group by the origin airport. This time

filter to only data without an NA in the air_time column. Calculate the average air_time for each origin airport.

```
Output > # A tibble: 3 x 2
Output >   origin avg_air_time
Output >    <chr>        <dbl>
Output > 1    EWR     153.3000
Output > 2    JFK     178.3490
Output > 3    LGA     117.8258
```

Remember, you can also use functions on columns you create within the pipeline. Here I create the variable "gains" again in a mutate call and then use it in a summarise call.

```
flights %>%
    mutate(., gains = arr_delay - dep_delay) %>%
    filter(., !is.na(gains)) %>%
    group_by(., carrier, dest) %>%
    summarise(., avg_gains = mean(gains),
              sd_gains = sd(gains),
              count = n()) %>%
    arrange(., dest, -count, carrier)
```

```
Output > Source: local data frame [312 x 5]
Output > Groups: carrier [16]
Output >
Output >    carrier  dest  avg_gains  sd_gains count
Output >      <chr> <chr>      <dbl>     <dbl> <int>
Output > 1       B6   ABQ  -9.358268 28.025708   254
Output > 2       B6   ACK  -1.594697 11.882766   264
Output > 3       EV   ALB  -9.050239  9.160957   418
Output > 4       UA   ANC -15.375000 21.077663     8
Output > 5       DL   ATL  -2.925469 16.229383 10452
Output > 6       FL   ATL   2.475856 16.222273  2278
Output > 7       MQ   ATL   4.700671 17.477659  2235
Output > 8       EV   ATL  -2.622585 15.416770  1656
Output > 9       UA   ATL  -4.166667 17.692692   102
Output > 10      WN   ATL   4.551724 11.116879    58
Output > # ... with 302 more rows
```

Why didn't I need to include the na.rm = T argument in the `mean()` calls?

How do I save output from a pipe?

Assign it to an object

```
avg_gains <- flights %>%
    mutate(., gains = arr_delay - dep_delay) %>%
    filter(., !is.na(gains)) %>%
    group_by(., carrier, dest) %>%
    summarise(., avg_gains = mean(gains),
              sd_gains = sd(gains),
              count = n()) %>%
    arrange(., dest, -count, carrier)

print(head(avg_gains))
```

```
Output > Source: local data frame [6 x 5]
Output > Groups: carrier [5]
Output > 
Output >   carrier  dest  avg_gains  sd_gains count
Output >     <chr> <chr>      <dbl>     <dbl> <int>
Output > 1      B6   ABQ  -9.358268 28.025708   254
Output > 2      B6   ACK  -1.594697 11.882766   264
Output > 3      EV   ALB  -9.050239  9.160957   418
Output > 4      UA   ANC -15.375000 21.077663     8
Output > 5      DL   ATL  -2.925469 16.229383 10452
Output > 6      FL   ATL   2.475856 16.222273  2278
```

Joins

One of the most important things you will do throughout any and all analysis that you perform both in this class and outside will involve the joining of datasets in some way. In fact, there is a good chance that the majority of your work on the final project will be preparing data for joining before running regressions. Dplyr comes with multiple built in join focused functions. These functions are similar to the `merge()` function that comes with Base R but are more specific than the general `merge()`.

For an example of the join functions we will be using the `weather` dataset included with the nycflights13 package. Let's first take a look at the data.

```
names(weather)
```

```
Output >  [1] "origin"     "year"       "month"      "day"        "hour"      
Output >  [6] "temp"       "dewp"       "humid"      "wind_dir"   "wind_speed"
Output > [11] "wind_gust"  "precip"     "pressure"   "visib"      "time_hour" 
```

```
dim(weather)
```

```
Output > [1] 26130    15
```

```
head(weather)
```

```
Output > # A tibble: 6 x 15
Output >   origin  year month   day  hour  temp  dewp humid wind_dir wind_speed
Output >    <chr> <dbl> <dbl> <int> <int> <dbl> <dbl> <dbl>    <dbl>      <dbl>
Output > 1    EWR  2013     1     1     0 37.04 21.92 53.97      230   10.35702
Output > 2    EWR  2013     1     1     1 37.04 21.92 53.97      230   13.80936
Output > 3    EWR  2013     1     1     2 37.94 21.92 52.09      230   12.65858
Output > 4    EWR  2013     1     1     3 37.94 23.00 54.51      230   13.80936
Output > 5    EWR  2013     1     1     4 37.94 24.08 57.04      240   14.96014
Output > 6    EWR  2013     1     1     6 39.02 26.06 59.37      270   10.35702
Output > # ... with 5 more variables: wind_gust <dbl>, precip <dbl>,
Output > #   pressure <dbl>, visib <dbl>, time_hour <time>
```

Ok, so we have year, month, day, hour, and origin columns matching the year, month, day, hour and origin columns of our flights data. So we probably want to be joining such that we match on those columns.

Let's start with a left join where we take two tables, x and y, and essentially add columns from y to our x table. Let's take a brief look at how the join function works and then use dplyr to try and answer the question of how wind speed impacts arrival and departure delays.

For each join function we have the `by =` argument which can take a character vector of the columns on which to join the two tables. (So the function looks for rows in which all columns in the `by =` argument match). If we do not specify `by=` the function will simply use all columns with the same name in each table.

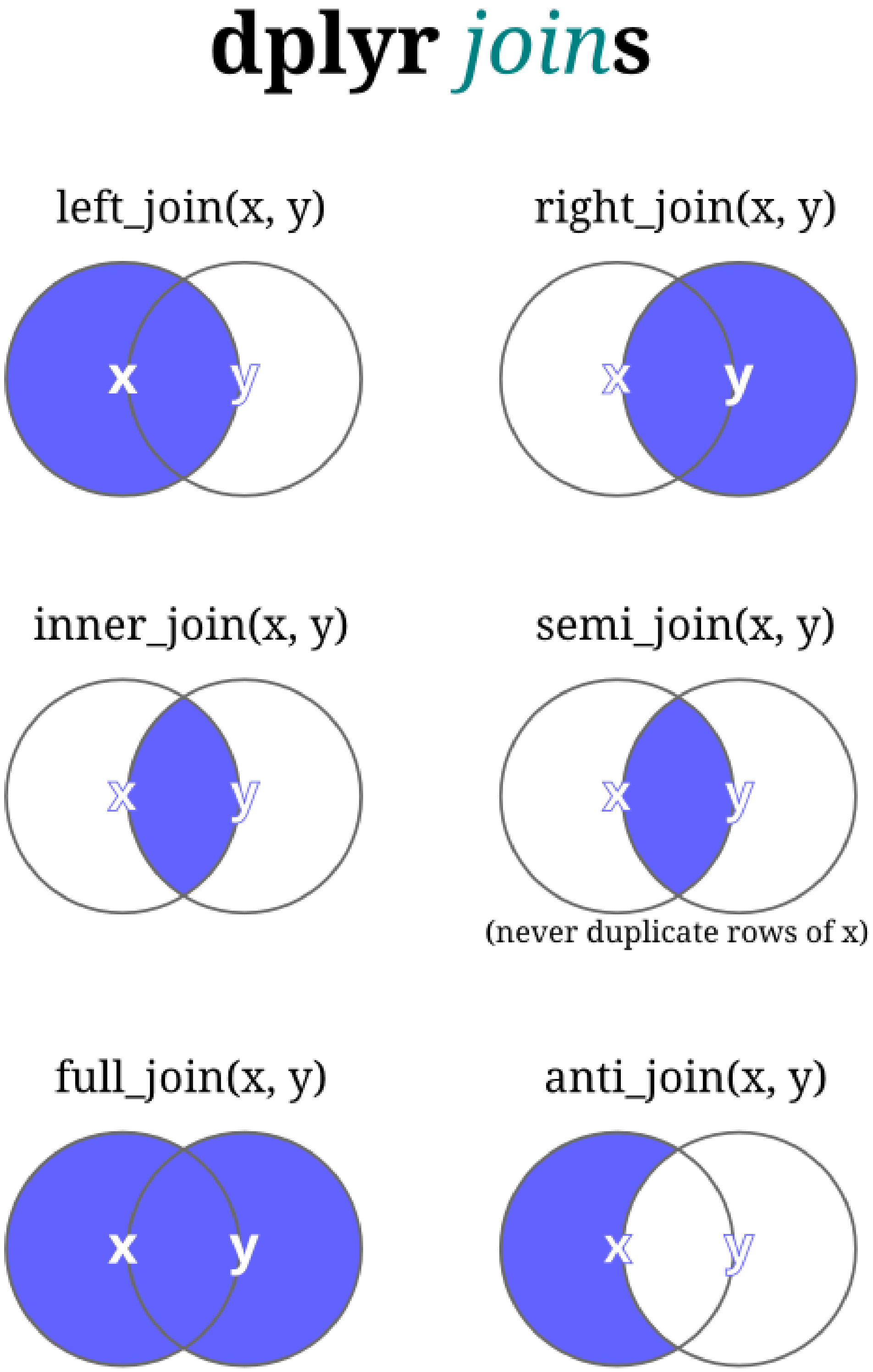

Figure 1: Join Types

```
## Take a sample of the flights data and join to some of the weather data.

samplef <- flights %>%
    filter(., !is.na(dest), !is.na(air_time))%>%
    select(., origin, year, month, day, hour, dest, air_time)

samplew <- weather %>%
    select(., origin, year, month, day, hour, temp, wind_dir)

head(samplew)
```

```
Output > # A tibble: 6 x 7
Output >   origin  year month   day  hour  temp wind_dir
Output >    <chr> <dbl> <dbl> <int> <int> <dbl>    <dbl>
Output > 1    EWR  2013     1     1     0 37.04      230
Output > 2    EWR  2013     1     1     1 37.04      230
Output > 3    EWR  2013     1     1     2 37.94      230
Output > 4    EWR  2013     1     1     3 37.94      230
Output > 5    EWR  2013     1     1     4 37.94      240
Output > 6    EWR  2013     1     1     6 39.02      270
```

```
join <- left_join(samplef, samplew, by = c("origin", "year", "month", "day", "hour"))

nrow(samplef)
```

```
Output > [1] 327346
```

```
nrow(samplew)
```

```
Output > [1] 26130
```

```
nrow(join)
```

```
Output > [1] 327346
```

```
head(join)
```

```
Output > # A tibble: 6 x 9
Output >   origin  year month   day  hour  dest air_time  temp wind_dir
Output >    <chr> <dbl> <dbl> <int> <dbl> <chr>    <dbl> <dbl>    <dbl>
Output > 1    EWR  2013     1     1     5   IAH      227    NA       NA
Output > 2    LGA  2013     1     1     5   IAH      227    NA       NA
Output > 3    JFK  2013     1     1     5   MIA      160    NA       NA
Output > 4    JFK  2013     1     1     5   BQN      183    NA       NA
Output > 5    LGA  2013     1     1     6   ATL      116 39.92      260
Output > 6    EWR  2013     1     1     5   ORD      150    NA       NA
```

So as you can see in the output we have added the dest and air_time columns to the selected weather data. For this function we left join two tables, x and y; `left_join(x,y)`. Because this is a left join we by default keep all the rows of the left table, x, and drop all rows in the right table, y, that do not succesfully merge with the left table. As you can see from above, the rows of samplef (the "x" table) is the same as the rows of join (the "output" table").

Now we will use the join function to try to answer the question of how wind speed seems to impact the average delays

```
## Here you can see how the pipe operator can really make the code flow easy to understand
```

```
wind_speed <- flights %>%
                filter(., !is.na(dep_delay)) %>%
                group_by(., origin, year, month, day, hour) %>%
                summarise(., avg_dep_delay = round(mean(dep_delay),1)) %>%
        left_join(.,
            weather %>%
                filter(., !is.na(wind_speed)) %>%
                select(., origin, year, month, day, hour, wind_speed) %>%
                mutate(., wind_speed = round(wind_speed, 0)),
            by = c("origin", "year", "month", "day", "hour")) %>%
        group_by(., wind_speed) %>%
        summarise(., delay = round(mean(avg_dep_delay, na.rm = T),2)) %>%
        arrange(., wind_speed)

wind_speed
```

```
Output > # A tibble: 36 x 2
Output >    wind_speed delay
Output >         <dbl> <dbl>
Output > 1           0  8.64
Output > 2           3  8.88
Output > 3           5  8.96
Output > 4           6  8.91
Output > 5           7 10.42
Output > 6           8 12.50
Output > 7           9 12.36
Output > 8          10 12.44
Output > 9          12 13.70
Output > 10         13 15.34
Output > # ... with 26 more rows
```

So from this we do see what appears to be a positive correlation between wind speed and delay length. We would want to plot this data and use the `cor()` function to visualize any trend. Let’s use the `cor()` function. Type `?cor`.

What arguments does cor take?

```
cor(wind_speed$wind_speed, wind_speed$delay, use = "complete.obs")
```

```
Output > [1] -0.09438348
```

So we show an extremely weak negative correlation in the data, really nothing strong enough to make any statement about how wind speed impacts delays. It’s a good thing we used the `cor()` function instead of just basing our conclusion on looking at a few rows of data!

Other functions

dplyr also provides a function glimpse() that makes it easy to look at our data in a transposed view. It’s similar to the str() (structure) function, but has a few advantages (see ?glimpse).

```
glimpse(weather)
```

```
Output > Observations: 26,130
Output > Variables: 15
Output > $ origin     <chr> "EWR", "EWR", "EWR", "EWR", "EWR", "EWR", "EWR", "E...
Output > $ year       <dbl> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 201...
Output > $ month      <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
Output > $ day        <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
Output > $ hour       <int> 0, 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ...
Output > $ temp       <dbl> 37.04, 37.04, 37.94, 37.94, 37.94, 39.02, 39.02, 39...
Output > $ dewp       <dbl> 21.92, 21.92, 21.92, 23.00, 24.08, 26.06, 26.96, 28...
Output > $ humid      <dbl> 53.97, 53.97, 52.09, 54.51, 57.04, 59.37, 61.63, 64...
Output > $ wind_dir   <dbl> 230, 230, 230, 230, 240, 270, 250, 240, 250, 260, 2...
Output > $ wind_speed <dbl> 10.35702, 13.80936, 12.65858, 13.80936, 14.96014, 1...
Output > $ wind_gust  <dbl> 11.918651, 15.891535, 14.567241, 15.891535, 17.2158...
Output > $ precip     <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
Output > $ pressure   <dbl> 1013.9, 1013.0, 1012.6, 1012.7, 1012.8, 1012.0, 101...
Output > $ visib      <dbl> 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,...
Output > $ time_hour  <time> 2012-12-31 19:00:00, 2012-12-31 20:00:00, 2012-12-...
```

Conclusion

dplyr makes it easier to write clear working code–which allows you to focus on the details of your data analysis.

For more info

One-page cheatsheet

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

Explanations from creator

http://r4ds.had.co.nz/transform.html

Free online training

https://www.datacamp.com/courses/dplyr-data-manipulation-r-tutorial