# Grouping and Chaining with dplyr

Code

In the last lesson, you learned about the five main data manipulation 'verbs' in dplyr: **select(), filter(), arrange(), mutate()**, and **summarize()**. The last of these, **summarize()**, is most powerful when applied to grouped data.

The main idea behind grouping data is that you want to break up your dataset into groups of rows based on the values of one or more variables. The **group_by()** function is reponsible for doing this.

We'll continue where we left off with RStudio's CRAN download log from July 8, 2014, which contains information on roughly 225,000 R package downloads (http://cran-logs.rstudio.com/ (http://cran-logs.rstudio.com/)).

As with the last lesson, the dplyr package was automatically installed (if necessary) and loaded at the beginning of this lesson. Normally, this is something you would have to do on your own. Just to build the habit, type *library(dplyr)* now to load the package again.

Hide

```
library(dplyr)
```

I've made the dataset available to you in a data frame called *mydf*. Put it in a *'data frame tbl'* using the **tbl_df()** function and store the result in a object called *cran*. If you're not sure what I'm talking about, you should start with the previous lesson. Otherwise, practice makes perfect!

Hide

```
cran <- tbl_df(mydf)
```

To avoid confusion and keep things running smoothly, let's remove the original dataframe from your workspace with *rm("mydf")*.

Hide

```
rm("mydf")
```

Print *cran* to the console.

Hide

```
cran
```

| X<br><int> | date<br><chr> | time<br><chr> | size<br><int> | r_version<br><chr> | r_arch<br><chr> | r_os<br><chr> | package<br><chr> | version<br><chr> |
|---|---|---|---|---|---|---|---|---|
| 1 | 2014-07-08 | 00:54:41 | 80589 | 3.1.0 | x86_64 | mingw32 | htmltools | 0.2.4 |
| 2 | 2014-07-08 | 00:59:53 | 321767 | 3.1.0 | x86_64 | mingw32 | tseries | 0.10-32 |
| 3 | 2014-07-08 | 00:47:13 | 748063 | 3.1.0 | x86_64 | linux-gnu | party | 1.0-15 |
| 4 | 2014-07-08 | 00:48:05 | 606104 | 3.1.0 | x86_64 | linux-gnu | Hmisc | 3.14-4 |
| 5 | 2014-07-08 | 00:46:50 | 79825 | 3.0.2 | x86_64 | linux-gnu | digest | 0.6.4 |
| 6 | 2014-07-08 | 00:48:04 | 77681 | 3.1.0 | x86_64 | linux-gnu | randomForest | 4.6-7 |
| 7 | 2014-07-08 | 00:48:35 | 393754 | 3.1.0 | x86_64 | linux-gnu | plyr | 1.8.1 |
| 8 | 2014-07-08 | 00:47:30 | 28216 | 3.0.2 | x86_64 | linux-gnu | whisker | 0.3-2 |
| 9 | 2014-07-08 | 00:54:58 | 5928 | *NA* | *NA* | *NA* | Rcpp | 0.10.4 |
| 10 | 2014-07-08 | 00:15:35 | 2206029 | 3.0.2 | x86_64 | linux-gnu | hflights | 0.1 |

1-10 of 225,468 rows | 1-9 of 11 columns

Previous **1** 2 3 4 5 6 … 100 Next

Our first goal is to group the data by package name. Bring up the help file for **group_by()**.

Hide

```
?group_by
```

Group cran by the package variable and store the result in a new object called *by_package*.

Hide

```
by_package <- group_by(cran, package)
```

Let's take a look at *by_package*. Print it to the console.

Hide

```
by_package
```

| X <int> | date <chr> | time <chr> | size <int> | r_version <chr> | r_arch <chr> | r_os <chr> | package <chr> | version <chr> |
|---|---|---|---|---|---|---|---|---|
| 1 | 2014-07-08 | 00:54:41 | 80589 | 3.1.0 | x86_64 | mingw32 | htmltools | 0.2.4 |
| 2 | 2014-07-08 | 00:59:53 | 321767 | 3.1.0 | x86_64 | mingw32 | tseries | 0.10-32 |
| 3 | 2014-07-08 | 00:47:13 | 748063 | 3.1.0 | x86_64 | linux-gnu | party | 1.0-15 |
| 4 | 2014-07-08 | 00:48:05 | 606104 | 3.1.0 | x86_64 | linux-gnu | Hmisc | 3.14-4 |
| 5 | 2014-07-08 | 00:46:50 | 79825 | 3.0.2 | x86_64 | linux-gnu | digest | 0.6.4 |
| 6 | 2014-07-08 | 00:48:04 | 77681 | 3.1.0 | x86_64 | linux-gnu | randomForest | 4.6-7 |
| 7 | 2014-07-08 | 00:48:35 | 393754 | 3.1.0 | x86_64 | linux-gnu | plyr | 1.8.1 |
| 8 | 2014-07-08 | 00:47:30 | 28216 | 3.0.2 | x86_64 | linux-gnu | whisker | 0.3-2 |
| 9 | 2014-07-08 | 00:54:58 | 5928 | *NA* | *NA* | *NA* | Rcpp | 0.10.4 |
| 10 | 2014-07-08 | 00:15:35 | 2206029 | 3.0.2 | x86_64 | linux-gnu | hflights | 0.1 |

1-10 of 225,468 rows | 1-9 of 11 columns  Previous **1** 2 3 4 5 6 … 100 Next

At the top of the output above, you'll see *'Groups: package'*, which tells us that this tbl has been grouped by the package variable. Everything else looks the same, but now any operation we apply to the grouped data will take place on a per package basis.

Recall that when we applied *mean(size)* to the original tbl_df via **summarize()**, it returned a single number – the mean of all values in the size column. We may care about what that number is, but wouldn't it be so much more interesting to look at the mean download size for each unique package?

That's exactly what you'll get if you use **summarize()** to apply *mean(size)* to the grouped data in *by_package*. Give it a shot.

Hide

```
summarize(by_package, mean(size))
```

| package <chr> | mean(size) <dbl> |
|---|---|
| A3 | 62194.960 |
| abc | 4826665.000 |
| abcdeFBA | 455979.867 |
| ABCExtremes | 22904.333 |
| ABCoptim | 17807.250 |
| ABCp2 | 30473.333 |
| abctools | 2589394.000 |
| abd | 453631.235 |
| abf2 | 35692.615 |
| abind | 32938.884 |

1-10 of 6,023 rows  Previous **1** 2 3 4 5 6 … 100 Next

Instead of returning a single value, **summarize()** now returns the mean size for EACH package in our dataset.

Let's take it a step further. I just opened an R script for you that contains a partially constructed call to **summarize()**. Follow the instructions in the script comments.

When you are ready to move on, save the script and type **submit()**, or type **reset()** to reset the script to its original state.

### Script: summarize1.R

Compute four values, in the following order, from the grouped data:

1. count = n()
2. unique = n_distinct(ip_id)
3. countries = n_distinct(country)
4. avg_bytes = mean(size)

A few thing to be careful of:

1. Separate arguments by commas
2. Make sure you have a closing parenthesis
3. Check your spelling!
4. Store the result in pack_sum (for 'package summary')

You should also take a look at ?n and ?n_distinct, so that you really understand what is going on.

Hide

```
pack_sum <- summarize(by_package,
                      count = n(),
                      unique = n_distinct(ip_id),
                      countries = n_distinct(country),
                      avg_bytes = mean(size))
```

Hide

```
submit()
```

Print the resulting tbl, *pack_sum*, to the console to examine its contents.

Hide

```
pack_sum
```

| package<br><chr> | count<br><int> | unique<br><int> | countries<br><int> | avg_bytes<br><dbl> |
|---|---|---|---|---|
| A3 | 25 | 24 | 10 | 62194.960 |
| abc | 29 | 25 | 16 | 4826665.000 |
| abcdeFBA | 15 | 15 | 9 | 455979.867 |
| ABCExtremes | 18 | 17 | 9 | 22904.333 |
| ABCoptim | 16 | 15 | 9 | 17807.250 |
| ABCp2 | 18 | 17 | 10 | 30473.333 |
| abctools | 19 | 19 | 11 | 2589394.000 |
| abd | 17 | 16 | 10 | 453631.235 |
| abf2 | 13 | 13 | 9 | 35692.615 |
| abind | 396 | 365 | 50 | 32938.884 |

1-10 of 6,023 rows  Previous **1** 2 3 4 5 6 … 100 Next

- The **'count'** column, created with *n()*, contains the total number of rows (i.e. downloads) for each package.
- The **'unique'** column, created with *n_distinct(ip_id)*, gives the total number of unique downloads for each package, as measured by the number of distinct ip_id's.
- The **'countries'** column, created with *n_distinct(country)*, provides the number of countries in which each package was downloaded.
- The **'avg_bytes'** column, created with *mean(size)*, contains the mean download size (in bytes) for each package.

It's important that you understand how each column of *pack_sum* was created and what it means. Now that we've summarized the data by individual packages, let's play around with it some more to see what we can learn.

Naturally, we'd like to know which packages were most popular on the day these data were collected (July 8, 2014). Let's start by isolating the top 1% of packages, based on the total number of downloads as measured by the *'count'* column.

We need to know the value of *'count'* that splits the data into the top 1% and bottom 99% of packages based on total downloads. In statistics, this is called the 0.99, or 99%, sample quantile. Use quantile(pack_sum$count, probs = 0.99) to determine this number.

Hide

```
quantile(pack_sum$count, probs = 0.99)
```

```
   99%
679.56
```

Now we can isolate only those packages which had more than 679 total downloads. Use **filter()** to select all rows from **pack_sum** for which *'count'* is strictly greater (>) than 679. Store the result in a new object called **top_counts**.

Hide

```
top_counts <- filter(pack_sum, count > 679)
```

Let's take a look at **top_counts**. Print it to the console.

Hide

```
top_counts
```

| package<br><chr> | count<br><int> | unique<br><int> | countries<br><int> | avg_bytes<br><dbl> |
|---|---|---|---|---|
| bitops | 1549 | 1408 | 76 | 28715.046 |
| car | 1008 | 837 | 64 | 1229122.307 |
| caTools | 812 | 699 | 64 | 176589.018 |
| colorspace | 1683 | 1433 | 80 | 357411.197 |
| data.table | 680 | 564 | 59 | 1252721.215 |
| DBI | 2599 | 492 | 48 | 206933.250 |
| devtools | 769 | 560 | 55 | 212932.640 |
| dichromat | 1486 | 1257 | 74 | 134731.938 |
| digest | 2210 | 1894 | 83 | 120549.294 |
| doSNOW | 740 | 75 | 24 | 8363.755 |

1-10 of 61 rows Previous **1** 2 3 4 5 6 7 Next

There are only 61 packages in our top 1%, so we'd like to see all of them. Since dplyr only shows us the first 10 rows, we can use the **View()** function to see more.

View all 61 rows with *View(top_counts)*. Note that the 'V' in **View()** is capitalized.

Hide

```
View(top_counts)
```

**arrange()** the rows of *top_counts* based on the *'count'* column and assign the result to a new object called *top_counts_sorted*. We want the packages with the highest number of downloads at the top, which means we want *'count'* to be in descending order. If you need help, check out **?arrange** and/or **?desc.**

Hide

```
top_counts_sorted <- arrange(top_counts, desc(count))
```

Now use **View()** again to see all 61 rows of *top_counts_sorted*.

Hide

```
View(top_counts_sorted)
```

If we use total number of downloads as our metric for popularity, then the above output shows us the most popular packages downloaded from the RStudio CRAN mirror on July 8, 2014. Not surprisingly, ggplot2 leads the pack with 4602 downloads, followed by Rcpp, plyr, rJava, ….

…And if you keep on going, you'll see swirl at number 43, with 820 total downloads. Sweet!

Perhaps we're more interested in the number of *unique* downloads on this particular day. In other words, if a package is downloaded ten times in one day from the same computer, we may wish to count that as only one download. That's what the 'unique' column will tell us.

Like we did with *'count'*, let's find the 0.99, or 99%, quantile for the *'unique'* variable with *quantile(pack_sum$unique, probs = 0.99).*

Hide

```
quantile(pack_sum$unique, probs = 0.99)
```

```
99%
465
```

Apply **filter()** to *pack_sum* to select all rows corresponding to values of *'unique'* that are strictly greater than 465. Assign the result to a object called **top_unique**.

Hide

```
top_unique <- filter(pack_sum, unique > 465)
```

Let's **View()** our top contenders!

Hide

```
View(top_unique)
```

Now **arrange()** *top_unique* by the *'unique'* column, in descending order, to see which packages were downloaded from the greatest number of unique IP addresses. Assign the result to **top_unique_sorted**.

Hide

```
top_unique_sorted <- arrange(top_unique, desc(unique))
```

**View()** the sorted data.

Hide

```
View(top_unique_sorted)
```

Now Rcpp is in the lead, followed by stringr, digest, plyr, and ggplot2. swirl moved up a few spaces to number 40, with 698 unique downloads. Nice!

Our final metric of popularity is the number of distinct countries from which each package was downloaded. We'll approach this one a little differently to introduce you to a method called *'chaining'* (or *'piping'*).

Chaining allows you to string together multiple function calls in a way that is compact and readable, while still accomplishing the desired result. To make it more concrete, let's compute our last popularity metric from scratch, starting with our original data.

I've opened up a script that contains code similar to what you've seen so far. Don't change anything. Just study it for a minute, make sure you understand everything that's there, then **submit()** when you are ready to move on.

## Script: summarize2.R

Don't change any of the code below. Just type *submit()* when you think you understand it.

We've already done this part, but we're repeating it here for clarity.

Hide

```
by_package <- group_by(cran, package)
pack_sum <- summarize(by_package,
                      count = n(),
                      unique = n_distinct(ip_id),
                      countries = n_distinct(country),
                      avg_bytes = mean(size))
```

Here's the new bit, but using the same approach we've been using this whole time.

Hide

```
top_countries <- filter(pack_sum, countries > 60)
result1 <- arrange(top_countries, desc(countries), avg_bytes)
# Print the results to the console.
print(result1)
```

| package<br><chr> | count<br><int> | unique<br><int> | countries<br><int> | avg_bytes<br><dbl> |
|---|---|---|---|---|
| Rcpp | 3195 | 2044 | 84 | 2512100.35 |
| digest | 2210 | 1894 | 83 | 120549.29 |
| stringr | 2267 | 1948 | 82 | 65277.17 |
| plyr | 2908 | 1754 | 81 | 799122.79 |
| ggplot2 | 4602 | 1680 | 81 | 2427716.05 |
| colorspace | 1683 | 1433 | 80 | 357411.20 |
| RColorBrewer | 1890 | 1584 | 79 | 22763.99 |
| scales | 1726 | 1408 | 77 | 126819.33 |
| bitops | 1549 | 1408 | 76 | 28715.05 |
| reshape2 | 2032 | 1652 | 76 | 330128.26 |

Hide

```
submit()
```

It's worth noting that we sorted primarily by *country*, but used *avg_bytes* (in ascending order) as a tie breaker. This means that if two packages were downloaded from the same number of countries, the package with a smaller average download size received a higher ranking.

We'd like to accomplish the same result as the last script, but avoid saving our intermediate results. This requires embedding function calls within one another.

That's exactly what we've done in this script. The result is equivalent, but the code is much less readable and some of the arguments are far away from the function to which they belong. Again, just try to understand what is going on here, then **submit()** when you are ready to see a better solution.

## Script: summarize3.R

Don't change any of the code below. Just type *submit()* when you think you understand it. If you find it confusing, you're absolutely right!

Hide

```
result2 <-
  arrange(
    filter(
      summarize(
        group_by(cran,
                 package
        ),
        count = n(),
        unique = n_distinct(ip_id),
        countries = n_distinct(country),
        avg_bytes = mean(size)
      ),
      countries > 60
    ),
    desc(countries),
    avg_bytes
  )
print(result2)
```

| package<br><chr> | count<br><int> | unique<br><int> | countries<br><int> | avg_bytes<br><dbl> |
|---|---|---|---|---|
| Rcpp | 3195 | 2044 | 84 | 2512100.35 |
| digest | 2210 | 1894 | 83 | 120549.29 |
| stringr | 2267 | 1948 | 82 | 65277.17 |
| plyr | 2908 | 1754 | 81 | 799122.79 |
| ggplot2 | 4602 | 1680 | 81 | 2427716.05 |
| colorspace | 1683 | 1433 | 80 | 357411.20 |
| RColorBrewer | 1890 | 1584 | 79 | 22763.99 |
| scales | 1726 | 1408 | 77 | 126819.33 |
| bitops | 1549 | 1408 | 76 | 28715.05 |
| reshape2 | 2032 | 1652 | 76 | 330128.26 |



Hide

```
submit()
```

In this script, we've used a special chaining operator, **%>%**, which was originally introduced in the **magrittr** R package and has now become a key component of **dplyr**. You can pull up the related documentation with *?chain*. The benefit of **%>%** is that it allows us to chain the function calls in a linear fashion. The code to the right of **%>%** operates on the result from the code to the left of **%>%**.

Once again, just try to understand the code, then type *submit()* to continue.

## Script: summarize4.R

Read the code below, but don't change anything. As you read it, you can pronounce the **%>%** operator as the word *'then'*.

Type *submit()* when you think you understand everything here.

Hide

```
result3 <-
  cran %>%
  group_by(package) %>%
  summarize(count = n(),
            unique = n_distinct(ip_id),
            countries = n_distinct(country),
            avg_bytes = mean(size)
  ) %>%
  filter(countries > 60) %>%
  arrange(desc(countries), avg_bytes)
# Print result to console
print(result3)
```

| package<br><chr> | count<br><int> | unique<br><int> | countries<br><int> | avg_bytes<br><dbl> |
|---|---|---|---|---|
| Rcpp | 3195 | 2044 | 84 | 2512100.35 |
| digest | 2210 | 1894 | 83 | 120549.29 |
| stringr | 2267 | 1948 | 82 | 65277.17 |
| plyr | 2908 | 1754 | 81 | 799122.79 |
| ggplot2 | 4602 | 1680 | 81 | 2427716.05 |
| colorspace | 1683 | 1433 | 80 | 357411.20 |
| RColorBrewer | 1890 | 1584 | 79 | 22763.99 |
| scales | 1726 | 1408 | 77 | 126819.33 |
| bitops | 1549 | 1408 | 76 | 28715.05 |
| reshape2 | 2032 | 1652 | 76 | 330128.26 |

1-10 of 46 rows  Previous **1** 2 3 4 5 Next

Hide

```
submit()
```

So, the results of the last three scripts are all identical. But, the third script provides a convenient and concise alternative to the more traditional method that we've taken previously, which involves saving results as we go along.

Once again, let's **View()** the full data, which has been stored in **result3**.

Hide

```
View(result3)
```

It looks like Rcpp is on top with downloads from 84 different countries, followed by digest, stringr, plyr, and ggplot2. swirl jumped up the rankings again, this time to 27th.

To help drive the point home, let's work through a few more examples of chaining.

Let's build a chain of dplyr commands one step at a time, starting with the script I just opened for you.

## Script: chain1.R

**select()** the following columns from *cran*. Keep in mind that when you're using the chaining operator, you don't need to specify the name of the data tbl in your call to **select()**.

1. ip_id
2. country
3. package
4. size

The call to **print()** at the end of the chain is optional, but necessary if you want your results printed to the console. Note that since there are no additional arguments to **print()**, you can leave off the parentheses after the function name. This is a convenient feature of the **%>%** operator.

Hide

```
cran %>%
  select(ip_id, country, package, size) %>%
    print
```

| ip_id <int> | country <chr> | package <chr> | size <int> |
|---|---|---|---|
| 1 | US | htmltools | 80589 |
| 2 | US | tseries | 321767 |
| 3 | US | party | 748063 |
| 3 | US | Hmisc | 606104 |
| 4 | CA | digest | 79825 |
| 3 | US | randomForest | 77681 |
| 3 | US | plyr | 393754 |
| 5 | US | whisker | 28216 |
| 6 | CN | Rcpp | 5928 |
| 7 | US | hflights | 2206029 |

1-10 of 225,468 rows Previous **1** 2 3 4 5 6 … 100 Next

Hide

```
submit()
```

Let's add to the chain.

## Script: chain2.R

Use **mutate()** to add a column called *size_mb* that contains the size of each download in megabytes (i.e. size / 2^20).

If you want your results printed to the console, add print to the end of your chain.

Hide

```
cran %>%
  select(ip_id, country, package, size) %>%
  mutate(size_mb = size / 2^20) %>%
  print
```

| ip_id <int> | country <chr> | package <chr> | size <int> | size_mb <dbl> |
|---|---|---|---|---|
| 1 | US | htmltools | 80589 | 0.076855659 |
| 2 | US | tseries | 321767 | 0.306860924 |
| 3 | US | party | 748063 | 0.713408470 |
| 3 | US | Hmisc | 606104 | 0.578025818 |
| 4 | CA | digest | 79825 | 0.076127052 |
| 3 | US | randomForest | 77681 | 0.074082375 |
| 3 | US | plyr | 393754 | 0.375513077 |
| 5 | US | whisker | 28216 | 0.026908875 |
| 6 | CN | Rcpp | 5928 | 0.005653381 |
| 7 | US | hflights | 2206029 | 2.103833199 |

1-10 of 225,468 rows Previous **1** 2 3 4 5 6 … 100 Next

Hide

```
submit()
```

A little bit more now.

## Script: chain3.R

Use **filter()** to select all rows for which *size_mb* is less than or equal to (<=) 0.5.

If you want your results printed to the console, add print to the end of your chain.

Hide

```
cran %>%
  select(ip_id, country, package, size) %>%
  mutate(size_mb = size / 2^20) %>%
  filter(size_mb <= 0.5) %>%
  print
```

| ip_id <int> | country <chr> | package <chr> | size <int> | size_mb <dbl> |
|---|---|---|---|---|
| 1 | US | htmltools | 80589 | 0.076855659 |
| 2 | US | tseries | 321767 | 0.306860924 |
| 4 | CA | digest | 79825 | 0.076127052 |
| 3 | US | randomForest | 77681 | 0.074082375 |
| 3 | US | plyr | 393754 | 0.375513077 |
| 5 | US | whisker | 28216 | 0.026908875 |
| 6 | CN | Rcpp | 5928 | 0.005653381 |
| 13 | DE | ipred | 186685 | 0.178036690 |
| 14 | US | mnormt | 36204 | 0.034526825 |
| 16 | US | iterators | 289972 | 0.276538849 |

1-10 of 142,021 rows  Previous **1** 2 3 4 5 6 … 100 Next

Hide

```
submit()
```

And finish it off.

## Script: chain4.R

**arrange()** the result by *size_mb*, in descending order.

If you want your results printed to the console, add print to the end of your chain.

Hide

```
cran %>%
  select(ip_id, country, package, size) %>%
  mutate(size_mb = size / 2^20) %>%
  filter(size_mb <= 0.5) %>%
  arrange(desc(size_mb)) %>%
  print
```

| ip_id <int> | country <chr> | package <chr> | size <int> | size_mb <dbl> |
|---|---|---|---|---|
| 11034 | DE | phia | 524232 | 0.4999466 |
| 9643 | US | tis | 524152 | 0.4998703 |
| 1542 | IN | RcppSMC | 524060 | 0.4997826 |
| 12354 | US | lessR | 523916 | 0.4996452 |
| 12072 | US | colorspace | 523880 | 0.4996109 |
| 2514 | KR | depmixS4 | 523863 | 0.4995947 |
| 1111 | US | depmixS4 | 523858 | 0.4995899 |
| 8865 | CR | depmixS4 | 523858 | 0.4995899 |
| 5908 | CN | RcmdrPlugin.KMggplot2 | 523852 | 0.4995842 |
| 12354 | US | RcmdrPlugin.KMggplot2 | 523852 | 0.4995842 |

1-10 of 142,021 rows  Previous **1** 2 3 4 5 6 … 100 Next

Hide

```
submit()
```

In this lesson, you learned about grouping and chaining using **dplyr**. You combined some of the things you learned in the previous lesson with these more advanced ideas to produce concise, readable, and highly effective code. Welcome to the wonderful world of **dplyr**!

---

END

---