Grouping and Chaining with dplyr

Raphael Carvalho

05/07/2019

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!

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

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

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

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.

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

```
## # A tibble: 6,023 x 2
##    package     `mean(size)`
##    <chr>              <dbl>
##  1 <NA>             822376.
##  2 A3                62195.
##  3 abc             4826665
##  4 abcdeFBA         455980.
##  5 ABCExtremes       22904.
##  6 ABCoptim          17807.
##  7 ABCp2             30473.
##  8 abctools        2589394
##  9 abd              453631.
## 10 abf2              35693.
## # ... with 6,013 more rows
```

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.

```
# 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.


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

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. And finally, the 'avg_bytes' column, created with mean(size), contains the mean download size (in bytes) for each package.

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

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.

```
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.

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

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.

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

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).

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

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.

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

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.

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

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.

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.

```
# 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.

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.

top_countries <- filter(pack_sum, countries > 60)
result1 <- arrange(top_countries, desc(countries), avg_bytes)

# Print the results to the console.
print(result1)
```

```
## # A tibble: 46 x 5
##    package      count unique countries avg_bytes
##    <chr>        <int>  <int>     <int>     <dbl>
##  1 Rcpp          3195   2044        84  2512100.
##  2 digest        2210   1894        83   120549.
##  3 stringr       2267   1948        82    65277.
##  4 plyr          2908   1754        81   799123.
##  5 ggplot2       4602   1680        81  2427716.
##  6 colorspace    1683   1433        80   357411.
##  7 RColorBrewer  1890   1584        79    22764.
##  8 scales        1726   1408        77   126819.
##  9 bitops        1549   1408        76    28715.
## 10 reshape2      2032   1652        76   330128.
## # ... with 36 more rows
```

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.

```
# 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!

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)
```

```
## # A tibble: 46 x 5
##    package      count unique countries avg_bytes
##    <chr>        <int>  <int>     <int>     <dbl>
##  1 Rcpp          3195   2044        84  2512100.
##  2 digest        2210   1894        83   120549.
##  3 stringr       2267   1948        82    65277.
##  4 plyr          2908   1754        81   799123.
##  5 ggplot2       4602   1680        81  2427716.
##  6 colorspace    1683   1433        80   357411.
##  7 RColorBrewer  1890   1584        79    22764.
##  8 scales        1726   1408        77   126819.
##  9 bitops        1549   1408        76    28715.
## 10 reshape2      2032   1652        76   330128.
## # ... with 36 more rows
```

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.

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

```r
# Type submit() when you think you understand
# everything here.

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)
```

```
## # A tibble: 46 x 5
##    package      count unique countries avg_bytes
##    <chr>        <int>  <int>     <int>     <dbl>
##  1 Rcpp          3195   2044        84  2512100.
##  2 digest        2210   1894        83   120549.
##  3 stringr       2267   1948        82    65277.
##  4 plyr          2908   1754        81   799123.
##  5 ggplot2       4602   1680        81  2427716.
##  6 colorspace    1683   1433        80   357411.
##  7 RColorBrewer  1890   1584        79    22764.
##  8 scales        1726   1408        77   126819.
##  9 bitops        1549   1408        76    28715.
## 10 reshape2      2032   1652        76   330128.
## # ... with 36 more rows
```

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

```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.

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

```
## # A tibble: 225,468 x 4
##    ip_id country package         size
##    <int> <chr>   <chr>          <int>
##  1     1 US      htmltools      80589
##  2     2 US      tseries       321767
##  3     3 US      party         748063
##  4     3 US      Hmisc         606104
##  5     4 CA      digest         79825
##  6     3 US      randomForest   77681
##  7     3 US      plyr          393754
##  8     5 US      whisker        28216
##  9     6 CN      Rcpp            5928
## 10     7 US      hflights     2206029
## # ... with 225,458 more rows
```

Let's add to the chain.

```
# 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.

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

```
## # A tibble: 225,468 x 5
##    ip_id country package         size size_mb
##    <int> <chr>   <chr>          <int>   <dbl>
##  1     1 US      htmltools      80589 0.0769
##  2     2 US      tseries       321767 0.307
##  3     3 US      party         748063 0.713
##  4     3 US      Hmisc         606104 0.578
##  5     4 CA      digest         79825 0.0761
##  6     3 US      randomForest   77681 0.0741
##  7     3 US      plyr          393754 0.376
##  8     5 US      whisker        28216 0.0269
##  9     6 CN      Rcpp            5928 0.00565
## 10     7 US      hflights     2206029 2.10
## # ... with 225,458 more rows
```

A little bit more now.

```
# 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.

cran %>%
  select(ip_id, country, package, size) %>%
  mutate(size_mb = size / 2^20) %>%
  # Your call to filter() goes here
  filter(size_mb <= 0.5)
```

```
## # A tibble: 142,021 x 5
##    ip_id country package        size size_mb
##    <int> <chr>   <chr>         <int>   <dbl>
##  1     1 US      htmltools     80589 0.0769
##  2     2 US      tseries      321767 0.307
##  3     4 CA      digest        79825 0.0761
##  4     3 US      randomForest  77681 0.0741
##  5     3 US      plyr         393754 0.376
##  6     5 US      whisker       28216 0.0269
##  7     6 CN      Rcpp           5928 0.00565
##  8    13 DE      ipred        186685 0.178
##  9    14 US      mnormt        36204 0.0345
## 10    16 US      iterators    289972 0.277
## # ... with 142,011 more rows
```

And finish it off.

```
# 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.

cran %>%
  select(ip_id, country, package, size) %>%
  mutate(size_mb = size / 2^20) %>%
  filter(size_mb <= 0.5) %>%
  # Your call to arrange() goes here
  arrange(desc(size_mb))
```

```
## # A tibble: 142,021 x 5
##    ip_id country package                size size_mb
##    <int> <chr>   <chr>                 <int>   <dbl>
##  1 11034 DE      phia                 524232   0.500
##  2  9643 US      tis                  524152   0.500
##  3  1542 IN      RcppSMC              524060   0.500
##  4 12354 US      lessR                523916   0.500
##  5 12072 US      colorspace           523880   0.500
##  6  2514 KR      depmixS4             523863   0.500
##  7  1111 US      depmixS4             523858   0.500
##  8  8865 CR      depmixS4             523858   0.500
##  9  5908 CN      RcmdrPlugin.KMggplot2 523852   0.500
## 10 12354 US      RcmdrPlugin.KMggplot2 523852   0.500
## # ... with 142,011 more rows
```