Lab 1: Tidy Data

Lorenzo Decillis

February 19, 2017

Swirl: Getting and Cleaning Data Solutions

Lesson 1: Manipulating Data with dplyr

Q1. I've created a variable called path2csv, which contains the full file path to the dataset. Call read.csv() with two arguments, path2csv and stringsAsFactors | = FALSE, and save the result in a new variable called mydf.

A1. mydf <- read.csv(path2csv, stringsAsFactors = FALSE)

Q2. Use dim() to look at the dimensions of mydf.

A2. dim(mydf)

Q3. Now use head() to preview the data.

A3. head(mydf)

Q4. 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.

A4.library(dplyr)

Q5. It's important that you have dplyr version 0.4.0 or later. To confirm this, type packageVersion("dplyr").

A5. packageVersion("dplyr")

Q6. The first step of working with data in dplyr is to load the data into what the package authors call a 'data frame tbl' or 'tbl_df'.

A6. cran <- tbl_df(mydf)

Q7. As may often be the case, particularly with larger datasets, we are only interested in some of the variables. Use select(cran, ip_id, package, country) to select only the ip_id, package, and country variables from the cran dataset.

A7.select(cran, ip_id, package, country)

Q8.Recall that in R, the : operator provides a compact notation for creating a sequence of numbers. For example, try 5:20.

A8. 5:20

Q9. Normally, this notation is reserved for numbers, but select() allows you to specify a sequence of columns this way, which can save a bunch of typing. Use select(cran, r_arch:country) to select all columns starting from r_arch and ending with country.

A9. select (cran, r_arch:country)

Q10. We can also select the same columns in reverse order. Give it a try.

A10. select (cran, country:r_arch)

Q11. Print the entire dataset again, just to remind yourself of what it looks like. You can do this at anytime during the lesson.

A11. cran

Q12. Instead of specifying the columns we want to keep, we can also specify the columns we want to throw away. To see how this works, do select(cran, -time) to omit the time column.

A12. select(cran, -time)

Q13.The negative sign in front of time tells select() that we DON'T want the time column. Now, let's combine strategies to omit all columns from X through size (X:size). To see how this might work, let's look at a numerical example with -5:20.

A13. -5:20

Q14. we want to negate the entire sequence of numbers from 5 through 20, so that we get -5, -6, -7, … , -18, -19, -20. Try the same thing, except surround 5:20 with parentheses so that R knows we want it to first come up with the sequence of numbers, then apply the negative sign to the whole thing.

A14. -(5:20)

Q15. Use this knowledge to omit all columns X:size using select().

A15. select(cran, -(X:size))

Q16. se filter(cran, package == "swirl") to select all rows for which the package variable is equal to "swirl". Be sure to use two equals signs side-by-side!

A16. filter(cran, package == "swirl")

Q17. You can specify as many conditions as you want, separated by commas. For example filter(cran, r_version == "3.1.1", country == "US") will return all rows of ran corresponding to downloads from users in the US running R version 3.1.1. Try it out.

A17. filter(cran, r_version == "3.1.1", country == "US")

Q18.Edit your previous call to filter() to instead return rows corresponding to users in "IN" (India) running an R version that is less than or equal to "3.0.2". The up arrow on your keyboard may come in handy here. Don't forget your double quotes!

A18. filter(cran, r_version <= "3.0.2", country == "IN")

Q19. Our last two calls to filter() requested all rows for which some condition AND another condition were TRUE. We can also request rows for which EITHER one condition OR another condition are TRUE. For example, filter(cran, country == "US" | country == "IN") will gives us all rows for which the country variable equals either "US" or "IN". Give it a go.

A19. filter(cran, country == "US" | country == "IN")

Q20. Now, use filter() to fetch all rows for which size is strictly greater than (>) 100500 (no quotes, since size is numeric) AND r_os equals "linux-gnu". Hint: You are passing three arguments to filter(): the name of the dataset, the first condition, and the second condition.

A20. filter(cran, size > 100500, r_os == "linux-gnu")

Q21. Okay, ready to put all of this together? Use filter() to return all rows of cran for which r_version is NOT NA. Hint: You will need to use !is.na() as part of your second argument to filter().

A21. filter(cran, !is.na(r_version))

Q22. To see how arrange() works, let's first take a subset of cran. select() all columns from size through ip_id and store the result in cran2.

A22. cran2 <- select(cran, size:ip_id)

Q23. Now, to order the ROWS of cran2 so that ip_id is in ascending order (from small to large), type arrange(cran2, ip_id). You may want to make your console wide enough so that you can see ip_id, which is the last column.

A23. arrange(cran2, ip_id)

Q24. To do the same, but in descending order, change the second argument to desc(ip_id), where desc() stands for 'descending'. Go ahead.

A24. arrange(cran2, desc(ip_id))

Q25. Arrange cran2 by the following three variables, in this order: country (ascending), r_version (descending), and ip_id (ascending).

A25. arrange(cran2, country, desc(r_version), ip_id)

Q26. To illustrate the next major function in dplyr, let's take another subset of our original data. Use select() to grab 3 columns from cran – ip_id, package, and size (in that order) – and store the result in a new variable called cran3.

A26. cran3 <- select(cran, ip_id, package, size)

Q27. One very nice feature of mutate() is that you can use the value computed for your second column (size_mb) to create a third column, all in the same line of code. To see this in action, repeat the exact same command as above, except add a third argument creating a column that is named size_gb and equal to size_mb / 2^10

A27. mutate(cran3, size_mb = size / 2^20, size_gb = size_mb / 2^10)

Q28. Let's try one more for practice. Pretend we discovered a glitch in the system that provided the original values for the size variable. All of the values in cran3 are 1000 bytes less than they should be. Using cran3, create just one new column called correct_size that contains the correct size.

A28. mutate(cran3, correct_size = size + 1000)

Q29. The last of the five core dplyr verbs, summarize(), collapses the dataset to a single row. Let's say we're interested in knowing the average download size.summarize(cran, avg_bytes = mean(size)) will yield the mean value of the size variable. Here we've chosen to label the result 'avg_bytes', but we could have named it anything. Give it a try.

A29. summarize(cran, avg_bytes = mean(size))

Lesson 2: Grouping and Chaining with dplyr

Q1. 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!

A1. cran <- tbl_df(mydf)

Q2. Group cran by the package variable and store the result in a new object called by_package.

A2. by_package <- group_by(cran, package)

Q3. 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.

A3. summarize(by_package, mean(size))

Q4. Compute four values, in the following order, from the grouped data.

A4. pack_sum <- summarize(by_package, count = n(), unique = n_distinct(ip_id), countries = n_distinct(country), avg_bytes = mean(size))

Q5. 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.

A5. top_counts <- filter(pack_sum, count > 679)

Q6. 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.

A6. top_counts_sorted <- arrange(top_counts, desc(count))

Q7. 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.

A7. top_unique <- filter(pack_sum, unique > 465)

Q8. 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.

A8. top_unique_sorted <- arrange(top_unique, desc(unique))

A9. cran %>% select(ip_id, country, package, size) %>% print

Q10. Use mutate() to add a column called size_mb that contains the size of each download in megabytes (i.e. size / 2ˆ20).

A10. cran %>% select(ip_id, country, package, size) %>% mutate(size_mb = size / 2ˆ20)

Q11. Use filter() to select all rows for which size_mb is less than or equal to (<=) 0.5.

A11. cran %>% select(ip_id, country, package, size) %>% mutate(size_mb = size / 2ˆ20) %>% filter(size_mb <= 0.5)

Q12. arrange() the result by size_mb, in descending order.

A12. cran %>% select(ip_id, country, package, size) %>% mutate(size_mb = size / 2ˆ20) %>% filter(size_mb <= 0.5) %>% arrange(desc(size_mb))

Lesson 3: Tidying Data with tidyr

Q1. Using the help file as a guide, call gather() with the following arguments (in order): students, sex, count, -grade. Note the minus sign before grade, which says we want to gather all columns EXCEPT grade.

A1. gather(students, sex, count, -grade)

Q2. Let's start by using gather() to stack the columns of students2, like we just did with students. This time, name the 'key' column sex_class andcthe 'value' column count. Save the result to a new variable called res.cConsult ?gather again if you need help.

A2. res <- gather(students2, sex_class, count, -grade)

Q3. Call separate() on res to split the sex_class column into sex and class. You only need to specify the first three arguments: data = res, col = sex_class, into = c("sex", "class"). You don't have to provide the argument names as long as they are in the correct order.

A3. separate(res, sex_class, c("sex", "class"))

Q4. Repeat your calls to gather() and separate(), but this time use the %>% operator to chain the commands together without storing an intermediate result.

A4. students2 %>% gather(sex_class, count, -grade) %>% separate(sex_class, c("sex", "class")) %>% print

Q5. Call gather() to gather the columns class1 through class5 into a new variable called class. The 'key' should be class, and the 'value' should be grade.

A5. students3 %>% gather(class, grade, class1:class5, na.rm = TRUE) %>% print

Q6. This script builds on the previous one by appending a call to spread(), which will allow us to turn the values of the test column, midterm and final, into column headers (i.e. variables).

A6. students3 %>% gather(class, grade, class1:class5, na.rm = TRUE) %>% spread(test, grade) %>% print

Q7. We want the values in the class columns to be 1, 2, . . . , 5 and not class1, class2, . . . , class5. Use the mutate() function from dplyr along with parse_number()

A7.students3 %>% gather(class, grade, class1:class5, na.rm = TRUE) %>% spread(test, grade) %>% mutate(class = parse_number(class)) %>% print

Q8. Complete the chained command below so that we are selecting the id, name, and sex column from students4 and storing the result in student_info.

A8. student_info <- students4 %>% select(id, name, sex) %>% print

Q9. Add a call to unique() below, which will remove duplicate rows from student_info.

A9. student_info <- students4 %>% select(id, name, sex) %>% unique %>% print

Q10. Now, using the script I just opened for you, create a second table called gradebook using the id, class, midterm, and final columns (in that order).

A10. gradebook <- students4 %>% select(id, class, midterm, final) %>% print

Q11. Use dplyr's mutate() to add a new column to the passed table. The column should be called status and the value, "passed" (a character string), should be the same for all students. 'Overwrite' the current version of passed with the new one.

A11. passed <- passed %>% mutate(status = "passed")

Q12. Now, do the same for the failed table, except the status column should have the value "failed" for all students.

A12. failed <- failed %>% mutate(status = "failed")

Q13. Now, pass as arguments the passed and failed tables (in order) to the dplyr function bind_rows(), which will join them together into a single unit. Check ?bind_rows if you need help.

A13. bind_rows(passed, failed)

Q14. Accomplish the following three goals: 1.select() 2.gather() 3.separate()

A14. sat %>% select(-contains("total")) %>% gather(part_sex, count, -score_range) %>% separate(part_sex, c("part", "sex")) %>% print

Q15. Append two more function calls to accomplish the following: 1.group_by() 2.mutate

A15. sat %>% select(-contains("total")) %>% gather(part_sex, count, -score_range) %>% separate(part_sex, c("part", "sex")) %>% group_by(part, sex) %>% mutate(total = sum(count), prop = count / total) %>% print

Lesson 4: Dates and Times with lubridate

Q1. There are three components to this date. In order, they are year, month, and day. We can extract any of these components using the year(), month(), or day() function, respectively. Try any of those on this_day now.

A1. month(this_day)

Q2. We can also get the day of the week from this_day using the wday() function. It will be represented as a number, such that 1 = Sunday, 2 = Monday, 3 = Tuesday, etc. Give it a shot.

A2.wday(this_day)

Q3. Now try the same thing again, except this time add a second argument, label = TRUE, to display the *name* of the weekday (represented as an ordered factor).

A3. wday(this_day, label = TRUE)

Q4. In addition to handling dates, lubridate is great for working with date and time combinations, referred to as date-times. The now() function returns the | date-time representing this exact moment in time. Give it a try and store the result in a variable called this_moment.

A4. this_moment <- now()

Q5. Just like with dates, we can extract the year, month, day, or day of week. However, we can also use hour(), minute(), and second() to extract specific time information. Try any of these three new functions now to extract one piece of time information from this_moment.

A5. 49.77503

Q6. We can even throw something funky at it and lubridate will often know the right thing to do. Parse 25081985, which is supposed to represent the 25th day of August 1985. Note that we are actually parsing a numeric value here – not a character string – so leave off the quotes.

A6. dmy(25081985)

Q7. What if we have a time, but no date? Use the appropriate lubridate function to parse "03:22:14" (hh:mm:ss).

A7. hms("03:22:14")

Q8. Unless you're a superhero, some time has passed since you first created this_moment. Use update() to make it match the current time, specifying at least | hours and minutes. Assign the result to this_moment, so that this_moment will contain the new time.

A8. this_moment <- update(this_moment, hours = 10, minutes = 16, seconds = 0)

Q9. To find the current date in New York, we'll use the now() function again. This time, however, we'll specify the time zone that we want: "America/New_York". Store the result in a variable called nyc. Check out ?now if you need help.

A9. nyc <- now("America/New_York")

Q10. So now depart contains the date of the day after tomorrow. Use update() to add the correct hours (17) and minutes (34) to depart. Reassign the result to depart.

A10. depart <- update(depart, hours = 17, minutes = 34)

Q11. The first step is to add 15 hours and 50 minutes to your departure time. Recall that nyc + days(2) added two days to the current time in New York. Use the same approach to add 15 hours and 50 minutes to the date-time stored in depart. Store the result in a new variable called arrive.

A11. arrive <- depart + hours(15) + minutes(50)

Q12. Use with_tz() to convert arrive to the "Asia/Hong_Kong" time zone. Reassign the result to arrive, so that it will get the new value.

A12. arrive <- with_tz(arrive, "Asia/Hong_Kong")

Q13. Use the appropriate lubridate function to parse "June 17, 2008", just like you did near the beginning of this lesson. This time, however, you should specify an extra argument, tz = "Singapore". Store the result in a variable called last_time.

A13. last_time <- mdy("June 17, 2008", tz = "Singapore")

Q14. Create an interval() that spans from last_time to arrive. Store it in a new variable called how_long.

A14. how_long <- interval(last_time, arrive)