Tidying Data with tidyr

Raphael Carvalho

09/07/2019

Tidy data is formatted in a standard way that facilitates exploration and analysis and works seamlessly with other tidy data tools. Specifically, tidy data satisfies three conditions:

1) Each variable forms a column

2) Each observation forms a row

3) Each type of observational unit forms a table

Any dataset that doesn't satisfy these conditions is considered 'messy' data. Therefore, all of the following are characteristics of messy data, EXCEPT...

```
print("6: Every column contains a different variable")
```

```
## [1] "6: Every column contains a different variable"
```

The first problem is when you have column headers that are values, not variable names. I've created a simple dataset called 'students' that demonstrates this scenario. Type students to take a look.

```
students
```

```
##   grade male female
## 1     A    1      5
## 2     B    5      0
## 3     C    5      2
## 4     D    5      5
## 5     E    7      4
```

The first column represents each of five possible grades that students could receive for a particular class. The second and third columns give the number of male and female students, respectively, that received each grade. This dataset actually has three variables: grade, sex, and count. The first variable, grade, is already a column, so that should remain as it is. The second variable, sex, is captured by the second and third column headings. The third variable, count, is the number of students for each combination of grade and sex.

To tidy the students data, we need to have one column for each of these three variables. We'll use the gather() function from tidyr to accomplish this.

```
gather(students, sex, count, -grade)
```

```
##    grade    sex count
## 1      A   male     1
## 2      B   male     5
## 3      C   male     5
## 4      D   male     5
## 5      E   male     7
## 6      A female     5
## 7      B female     0
## 8      C female     2
## 9      D female     5
## 10     E female     4
```

It's important to understand what each argument to gather() means. The data argument, students, gives the name of the original dataset. The key and value arguments – sex and count, respectively – give the column

names for our tidy dataset. The final argument, -grade, says that we want to gather all columns EXCEPT the grade column (since grade is already a proper column variable.)

The second messy data case we'll look at is when multiple variables are stored in one column. Type students2 to see an example of this.

```
students2
```

```
## Error in eval(expr, envir, enclos): objeto 'students2' não encontrado
```

This dataset is similar to the first, except now there are two separate classes, 1 and 2, and we have total counts for each sex within each class. students2 suffers from the same messy data problem of having column headers that are values (male_1, female_1, etc.) and not variable names (sex, class, and count). However, it also has multiple variables stored in each column (sex and class), which is another common symptom of messy data. Tidying this dataset will be a two step process.

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 and the 'value' column count. Save the result to a new variable called res. Consult ?gather again if you need help.

```
res <- gather(students2, sex_class, count, -grade)
```

```
## Error in gather(students2, sex_class, count, -grade): objeto 'students2' não encontrado
```

That got us half way to tidy data, but we still have two different variables, sex and class, stored together in the sex_class column. tidyr offers a convenient separate() function for the purpose of separating one column into multiple columns.

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.

```
separate(res, sex_class, c("sex", "class"))
```

```
## Error in separate(res, sex_class, c("sex", "class")): objeto 'res' não encontrado
```

Conveniently, separate() was able to figure out on its own how to separate the sex_class column. Unless you request otherwise with the 'sep' argument, it splits on non-alphanumeric values. In other words, it assumes that the values are separated by something other than a letter or number (in this case, an underscore.)

Repeat your calls to gather() and separate(), but this time use the %>% operator to chain the commands together without storing an intermediate result. If this is your first time seeing the %>% operator, check out ?chain, which will bring up the relevant documentation. You can also look at the Examples section at the bottom of ?gather and ?separate.

The main idea is that the result to the left of %>% takes the place of the first argument of the function to the right. Therefore, you OMIT THE FIRST ARGUMENT to each function.

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

```
## Error: <text>:5:0: unexpected end of input
## 3:   separate(sex_class, c("sex", "class")) %>%
## 4:   print<
##    ^
```

A third symptom of messy data is when variables are stored in both rows and columns. students3 provides an example of this. In students3, we have midterm and final exam grades for five students, each of whom were enrolled in exactly two of five possible classes.

The first variable, name, is already a column and should remain as it is. The headers of the last five columns, class1 through class5, are all different values of what should be a class variable. The values in the test column, midterm and final, should each be its own variable containing the respective grades for each student.

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. tidyr makes it easy to reference multiple adjacent columns with class1:class5, just like with sequences of numbers.

Since each student is only enrolled in two of the five possible classes, there are lots of missing values (i.e. NAs). Use the argument na.rm = TRUE to omit these values from the final result.

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

```
## Error in eval(lhs, parent, parent): objeto 'students3' não encontrado
```

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).You only need to specify two arguments to spread(). Can you figure out what they are? (Hint: You don't have to specify the data argument since we're using the %>% operator.

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

```
## Error in eval(lhs, parent, parent): objeto 'students3' não encontrado
```

Lastly, we want the values in the class column to simply be 1, 2, ..., 5 and not class1, class2, ..., class5. We can use the parse_number() function from readr to accomplish this. To see how it works, try parse_number("class5").

```
parse_number("class5")
```

```
## Error in parse_number("class5"): não foi possível encontrar a função "parse_number"
```

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(). Hint: You can "overwrite" a column with mutate() by assigning a new value to the existing column instead of creating a new column.

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

```
## Error in eval(lhs, parent, parent): objeto 'students3' não encontrado
```

The fourth messy data problem we'll look at occurs when multiple observational units are stored in the same table. students4 presents an example of this.

students4 is almost the same as our tidy version of students3. The only difference is that students4 provides a unique id for each student, as well as his or her sex (M = male; F = female). At first glance, there doesn't seem to be much of a problem with students4. All columns are variables and all rows are observations. However, notice that each id, name, and sex is repeated twice, which seems quite redundant. This is a hint that our data contains multiple observational units in a single table.

Our solution will be to break students4 into two separate tables – one containing basic student information (id, name, and sex) and the other containing grades (id, class, midterm, final).

selecting the id, name, and sex column from students4 and storing the result in student_info.

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

```
## Error in eval(lhs, parent, parent): objeto 'students4' não encontrado
```

Adding a call to unique()

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

```
## Error in eval(lhs, parent, parent): objeto 'students4' não encontrado
```

select() the id, class, midterm, and final columns (in that order) and store the result in gradebook.

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

```
## Error in eval(lhs, parent, parent): objeto 'students4' não encontrado
```

It's important to note that we left the id column in both tables. In the world of relational databases, 'id' is called our 'primary key' since it allows us to connect each student listed in student_info with their grades listed in gradebook. Without a unique identifier, we might not know how the tables are related. (In this case, we could have also used the name variable, since each student happens to have a unique name.)

The fifth and final messy data scenario that we'll address is when a single observational unit is stored in multiple tables. It's the opposite of the fourth problem.

Teachers decided to only take into consideration final exam grades in determining whether students passed or failed each class. As you may have inferred from the data, students passed a class if they received a final exam grade of A or B and failed otherwise.

The name of each dataset actually represents the value of a new variable that we will call 'status'. Before joining the two tables together, we'll add a new column to each containing this information so that it's not lost when we put everything together.

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.

```
passed <- passed %>% mutate(status = "passed")
```

```
## Error in eval(lhs, parent, parent): objeto 'passed' não encontrado
```

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

```
failed <- failed %>% mutate(status = "failed")
```

```
## Error in eval(lhs, parent, parent): objeto 'failed' não encontrado
```

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.

```
bind_rows(passed, failed)
```

```
## Error in dots_values(...): objeto 'passed' não encontrado
```

The SAT is a popular college-readiness exam in the United States that consists of three sections: critical reading, mathematics, and writing. Students can earn up to 800 points on each section. This dataset presents the total number of students, for each combination of exam section and sex, within each of six score ranges. It comes from the 'Total Group Report 2013', which can be found here: http://research.collegeboard.org/programs/sat/data/cb-seniors-2013

I've created a variable called 'sat' in your workspace, which contains data on all college-bound seniors who took the SAT exam in 2013.

As we've done before, we'll build up a series of chained commands, using functions from both tidyr and dplyr. Edit the R script, save it, then type submit() when you are ready. Type reset() to reset the script to its original state.

Accomplish the following three goals:

1. select() all columns that do NOT contain the word "total", since if we have the male and female data, we can always recreate the total count in a separate column, if we want it.

Hint: Use the contains() function, which you'll find detailed in 'Special functions' section of ?select.

2. gather() all columns EXCEPT score_range, using key = part_sex and value = count.
3. separate() part_sex into two separate variables (columns), called "part" and "sex", respectively. You may need to check the 'Examples' section of ?separate to remember how the 'into' argument should be phrased.

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

```
## Error in eval(lhs, parent, parent): objeto 'sat' não encontrado
```

Append two more function calls to accomplish the following:

1. Use group_by() (from dplyr) to group the data by part and sex, in that order.
2. Use mutate to add two new columns, whose values will be automatically computed group-by-group:

- total = sum(count)
- prop = count / total

```
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
```

```
## Error in eval(lhs, parent, parent): objeto 'sat' não encontrado
```