Tidying Data with tidyr

In this lesson, you'll learn how to tidy your data with the tidyr package.

Parts of this lesson will require the use of dplyr. If you don't have a basic knowledge of dplyr, you should exit this lesson and begin with the dplyr lessons from earlier in the course.

tidyr was automatically installed (if necessary) and loaded when you started this lesson. Just to build the habit, (re)load the package with *library(tidyr)*.

```
library(tidyr)
```

Hide

The author of tidyr, **Hadley Wickham**, discusses his philosophy of tidy data in his 'Tidy Data' paper: http://vita.had.co.nz/papers/tidy-data.pdf

This paper should be required reading for anyone who works with data, but it's not required in order to complete this lesson.

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

Quiz

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

1. Column headers are values, not variable names
2. Variables are stored in both rows and columns
3. A single observational unit is stored in multiple tables
4. Multiple types of observational units are stored in the same table
5. Multiple variables are stored in one column
6. Every column contains a different variable

Answer: **6**

The incorrect answers to the previous question are the most common symptoms of messy data. Let's work through a simple example of each of these five cases, then tidy some real data.

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
```

Hide

grade <fctr>	male <int>	female <int>
A	1	5
B	5	0
C	5	2
D	5	5
E	7	4

5 rows

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. Pull up the documentation for this function with *?gather*.

> **Description**: *Gather takes multiple columns and collapses into key-value pairs, duplicating all other columns as needed. You use **gather()** when you notice that you have columns that are not variables.*

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.

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

Hide

grade <fctr>	sex <chr>	count <int>
A	male	1
B	male	5
C	male	5
D	male	5
E	male	7
A	female	5
B	female	0
C	female	2
D	female	5
E	female	4

1-10 of 10 rows

Each row of the data now represents exactly one observation, characterized by a unique combination of the grade and sex variables. Each of our variables (*grade, sex*, and *count*) occupies exactly one column. That's tidy data!

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
```

Hide

grade <fctr>	male_1 <int>	female_1 <int>	male_2 <int>	female_2 <int>
A	3	4	3	4
B	6	4	3	5
C	7	4	3	8
D	4	0	8	1
E	1	1	2	7

5 rows

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)
```

Hide

Print res to the console to see what we accomplished.

```
res
```

Hide

grade <fctr>	sex_class <chr>	count <int>
A	male_1	3
B	male_1	6
C	male_1	7
D	male_1	4
E	male_1	1
A	female_1	4
B	female_1	4
C	female_1	4

grade <fctr>	sex_class <chr>	count <int>
D	female_1	0
E	female_1	1

1-10 of 20 rows Previous **1** 2 Next

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. Pull up the help file for **separate()** now.

```
?separate
```

Hide

Description: *Given either regular expression or a vector of character positions,* ***separate()*** *turns a single character 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"))
```

Hide

grade <fctr>	sex <chr>	class <chr>	count <int>
A	male	1	3
B	male	1	6
C	male	1	7
D	male	1	4
E	male	1	1
A	female	1	4
B	female	1	4
C	female	1	4
D	female	1	0
E	female	1	1

1-10 of 20 rows Previous **1** 2 Next

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.)

Tidying *students2* required both **gather()** and **separate()**, causing us to save an intermediate result *(res)*. However, just like with dplyr, you can use the **%>%** operator to chain multiple function calls together.

I've opened an R script for you to give this a try. Follow the directions in the script, then save the script and type **submit()** at the prompt when you are ready. If you get stuck and want to start over, you can type **reset()** to reset the script to its original state.

Script: script1.R

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
```

Hide

grade <fctr>	sex <chr>	class <chr>	count <int>
A	male	1	3
B	male	1	6
C	male	1	7
D	male	1	4
E	male	1	1

grade <fctr>	sex <chr>	class <chr>	count <int>
A	female	1	4
B	female	1	4
C	female	1	4
D	female	1	0
E	female	1	1

1-10 of 20 rows Previous 1 2 Next

```
submit()
```

Hide

A third symptom of messy data is when variables are stored in both rows and columns. *students3* provides an example of this. Print *students3* to the console.

```
students3
```

Hide

name <chr>	test <chr>	class1 <chr>	class2 <chr>	class3 <chr>	class4 <chr>	class5 <chr>
Sally	midterm	A	NA	B	NA	NA
Sally	final	C	NA	C	NA	NA
Jeff	midterm	NA	D	NA	A	NA
Jeff	final	NA	E	NA	C	NA
Roger	midterm	NA	C	NA	NA	B
Roger	final	NA	A	NA	NA	A
Karen	midterm	NA	NA	C	A	NA
Karen	final	NA	NA	C	A	NA
Brian	midterm	B	NA	NA	NA	A
Brian	final	B	NA	NA	NA	C

1-10 of 10 rows

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.

This will require multiple steps, which we will build up gradually using **%>%**. Edit the R script, save it, then type **submit()** when you are ready. Type **reset()** to reset the script to its original state.

Script: script2.R

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.

Remember that when you’re using the **%>%** operator, the value to the left of it gets inserted as the first argument to the function on the right.

Consult *?gather* and/or *?chain* if you get stuck.

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

Hide

	name <chr>	test <chr>	class <chr>	grade <chr>
1	Sally	midterm	class1	A
2	Sally	final	class1	C
9	Brian	midterm	class1	B

	name <chr>	test <chr>	class <chr>	grade <chr>
10	Brian	final	class1	B
13	Jeff	midterm	class2	D
14	Jeff	final	class2	E
15	Roger	midterm	class2	C
16	Roger	final	class2	A
21	Sally	midterm	class3	B
22	Sally	final	class3	C

1-10 of 20 rows Previous 1 2 Next

```
submit()
```

Hide

The next step will require the use of **spread()**. Pull up the documentation for **spread()** now.

```
?spread
```

Hide

Description: *Spread a key-value pair across multiple columns.* **Usage:** *spread(data, key, value, fill = NA, convert = FALSE, drop = TRUE, sep = NULL)

Edit the R script, then save it and type **submit()** when you are ready. Type **reset()** to reset the script to its original state.

Script: script3.R

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
```

Hide

name <chr>	class <chr>	final <chr>	midterm <chr>
Brian	class1	B	B
Brian	class5	C	A
Jeff	class2	E	D
Jeff	class4	C	A
Karen	class3	C	C
Karen	class4	A	A
Roger	class2	A	C
Roger	class5	A	B
Sally	class1	C	A
Sally	class3	C	B

1-10 of 10 rows

```
submit()
```

Hide

readr is required for certain data manipulations, such as **parse_number()**, which will be used in the next question. Let's, (re)load the package with *library(readr)*.

```
library(readr)
```

Hide

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")
```

Hide

```
[1] 5
```

Now, the final step. Edit the R script, then save it and type **submit()** when you are ready. Type **reset()** to reset the script to its original state.

Script: script4.R

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.

Check out *?mutate* and/or *?parse_number* if you need a refresher.

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

Hide

name <chr>	class <dbl>	final <chr>	midterm <chr>
Brian	1	B	B
Brian	5	C	A
Jeff	2	E	D
Jeff	4	C	A
Karen	3	C	C
Karen	4	A	A
Roger	2	A	C
Roger	5	A	B
Sally	1	C	A
Sally	3	C	B

1-10 of 10 rows

```
submit()
```

Hide

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. Take a look at the data now.

```
students4
```

Hide

id <int>	name <chr>	sex <chr>	class <int>	midterm <chr>	final <chr>
168	Brian	F	1	B	B
168	Brian	F	5	A	C
588	Sally	M	1	A	C
588	Sally	M	3	B	C
710	Jeff	M	2	D	E
710	Jeff	M	4	A	C
731	Roger	F	2	C	A
731	Roger	F	5	B	A
908	Karen	M	3	C	C
908	Karen	M	4	A	A

1-10 of 10 rows

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).

Edit the R script, save it, then type **submit()** when you are ready. Type **reset()** to reset the script to its original state.

Script: script5.R

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*.

Hide

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

id <int>	name <chr>	sex <chr>
168	Brian	F
168	Brian	F
588	Sally	M
588	Sally	M
710	Jeff	M
710	Jeff	M
731	Roger	F
731	Roger	F
908	Karen	M
908	Karen	M

1-10 of 10 rows

Hide

```
submit()
```

Notice anything strange about student_info? It contains five duplicate rows! See the script for directions on how to fix this. Save the script and type **submit()** when you are ready, or type **reset()** to reset the script to its original state.

Script: script6.R

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

Like with the call to the **print()** function below, you can omit the parentheses after the function name. This is a nice feature of **%>%** that applies when there are no additional arguments to specify.

Hide

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

	id <int>	name <chr>	sex <chr>
1	168	Brian	F
3	588	Sally	M
5	710	Jeff	M
7	731	Roger	F
9	908	Karen	M

5 rows

Hide

```
submit()
```

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).

Edit the R script, save it, then type **submit()** when you are ready. Type **reset()** to reset the script to its original state.

Script: script7.R

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
```

id <int>	class <int>	midterm <chr>	final <chr>
168	1	B	B
168	5	A	C
588	1	A	C
588	3	B	C
710	2	D	E
710	4	A	C
731	2	C	A
731	5	B	A
908	3	C	C
908	4	A	A

1-10 of 10 rows

```
submit()
```

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.

To illustrate this, we've created two datasets, **passed** and **failed**. Take a look at *passed* now.

```
passed
```

name <chr>	class <int>	final <chr>	status <chr>
Brian	1	B	passed
Roger	2	A	passed
Roger	5	A	passed
Karen	4	A	passed

4 rows

Now view the contents of *failed*.

```
failed
```

name <chr>	class <int>	final <chr>	status <chr>
Brian	5	C	failed
Sally	1	C	failed
Sally	3	C	failed
Jeff	2	E	failed
Jeff	4	C	failed

name <chr>	class <int>	final <chr>	status <chr>
Karen	3	C	failed

6 rows

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.

Hide

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

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

Hide

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

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.

Note: **bind_rows()** is only available in dplyr 0.4.0 or later. If you have an older version of dplyr, please quit the lesson, update dplyr, then restart the lesson where you left off. If you're not sure what version of dplyr you have, type *packageVersion('dplyr')*.

Hide

```
?bind_rows
```

> **Description:** *This is an efficient implementation of the common pattern of* ***do.call(rbind, dfs)*** *or* ***do.call(cbind, dfs)*** *for binding many data frames into one.* ***combine()*** *acts like* ***c()*** *or* ***unlist()*** *but uses consistent dplyr coercion rules.* **Usage:** *bind_rows(…, .id = NULL)*

Hide

```
packageVersion('dplyr')
```

```
[1] '0.7.4'
```

Hide

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

name <chr>	class <int>	final <chr>	status <chr>
Brian	1	B	passed
Roger	2	A	passed
Roger	5	A	passed
Karen	4	A	passed
Brian	5	C	failed
Sally	1	C	failed
Sally	3	C	failed
Jeff	2	E	failed
Jeff	4	C	failed
Karen	3	C	failed

1-10 of 10 rows

Of course, we could arrange the rows however we wish at this point, but the important thing is that each row is an observation, each column is a variable, and the table contains a single observational unit. Thus, the data are tidy.

We've covered a lot in this lesson. Let's bring everything together and tidy a real dataset.

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. Print the dataset now.

Hide

```
sat
```

score_range <chr>	read_male <int>	read_fem <int>	read_total <int>	math_male <int>	math_f... <int>	math_total <int>	write_male <int>	write_fem <int>
700-800	40151	38898	79049	74461	46040	120501	31574	39101
600-690	121950	126084	248034	162564	133954	296518	100963	125368
500-590	227141	259553	486694	233141	257678	490819	202326	247239
400-490	242554	296793	539347	204670	288696	493366	262623	302933
300-390	113568	133473	247041	82468	131025	213493	146106	144381
200-290	30728	29154	59882	18788	26562	45350	32500	24933

6 rows | 1-9 of 10 columns

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.

Script: script8.R

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.

Hide

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

score_range <chr>	part <chr>	sex <chr>	count <int>
700-800	read	male	40151
600-690	read	male	121950
500-590	read	male	227141
400-490	read	male	242554
300-390	read	male	113568
200-290	read	male	30728
700-800	read	fem	38898
600-690	read	fem	126084
500-590	read	fem	259553
400-490	read	fem	296793

1-10 of 36 rows   Previous **1** 2 3 4 Next

```
submit()
```

Hide

Finish off the job by following the directions in the script. Save the script and type **submit()** when you are ready, or type **reset()** to reset the script to its original state.

Script: script9.R

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

Hide

```r
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
```

score_range	part	sex	count	total	prop
<chr>	<chr>	<chr>	<int>	<int>	<dbl>
700-800	read	male	40151	776092	0.05173485
600-690	read	male	121950	776092	0.15713343
500-590	read	male	227141	776092	0.29267278
400-490	read	male	242554	776092	0.31253253
300-390	read	male	113568	776092	0.14633317
200-290	read	male	30728	776092	0.03959324
700-800	read	fem	38898	883955	0.04400450
600-690	read	fem	126084	883955	0.14263622
500-590	read	fem	259553	883955	0.29362694
400-490	read	fem	296793	883955	0.33575578

1-10 of 36 rows Previous **1** 2 3 4 Next

```r
submit()
```

Hide

In this lesson, you learned how to tidy data with *tidyr* and *dplyr*. These tools will help you spend less time and energy getting your data ready to analyze and more time actually analyzing it.

END