

Getting / cleaning data 2

Tidy data

All of the material in this section comes directly from Hadley Wickham's paper on tidy data. You will need to read this paper to prepare for the quiz on this section.

Characteristics of tidy data

Characteristics of tidy data are:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Getting your data into a “tidy” format makes it easier to model and plot. By taking the time to tidy your data at the start of an analysis, you will save yourself time, and make it easier to plan out, later steps.

Five common problems

Here are five common problems that Hadley Wickham has identified that keep data from being tidy:

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

In the following slides, I'll give examples of each of these problems.

Five common problems

(1.) Column headers are values, not variable names.

religion	<\$10k	\$10-20k	\$20-30k	\$30-40k	\$40-50k	\$50-75k
Agnostic	27	34	60	81	76	137
Atheist	12	27	37	52	35	70
Buddhist	27	21	30	34	33	58
Catholic	418	617	732	670	638	1116
Don't know/refused	15	14	15	11	10	35
Evangelical Prot	575	869	1064	982	881	1486
Hindu	1	9	7	9	11	34
Historically Black Prot	228	244	236	238	197	223
Jehovah's Witness	20	27	24	24	21	30
Jewish	19	19	25	25	30	95

Five common problems

Solution:

religion	income	freq
Agnostic	<\$10k	27
Agnostic	\$10-20k	34
Agnostic	\$20-30k	60
Agnostic	\$30-40k	81
Agnostic	\$40-50k	76
Agnostic	\$50-75k	137
Agnostic	\$75-100k	122
Agnostic	\$100-150k	109
Agnostic	>150k	84
Agnostic	Don't know/refused	96

Five common problems

(2.) Multiple variables are stored in one column.

country	year	column	cases
AD	2000	m014	0
AD	2000	m1524	0
AD	2000	m2534	1
AD	2000	m3544	0
AD	2000	m4554	0
AD	2000	m5564	0
AD	2000	m65	0
AE	2000	m014	2
AE	2000	m1524	4
AE	2000	m2534	4
AE	2000	m3544	6
AE	2000	m4554	5
AE	2000	m5564	12
AE	2000	m65	10
AE	2000	f014	3

Five common problems

Solution:

country	year	sex	age	cases
AD	2000	m	0-14	0
AD	2000	m	15-24	0
AD	2000	m	25-34	1
AD	2000	m	35-44	0
AD	2000	m	45-54	0
AD	2000	m	55-64	0
AD	2000	m	65+	0
AE	2000	m	0-14	2
AE	2000	m	15-24	4
AE	2000	m	25-34	4
AE	2000	m	35-44	6
AE	2000	m	45-54	5
AE	2000	m	55-64	12
AE	2000	m	65+	10
AE	2000	f	0-14	3

Five common problems

(3.) Variables are stored in both rows and columns.

id	year	month	element	d1	d2	d3	d4	d5	d6	d7	d8
MX17004	2010	1	tmax	—	—	—	—	—	—	—	—
MX17004	2010	1	tmin	—	—	—	—	—	—	—	—
MX17004	2010	2	tmax	—	27.3	24.1	—	—	—	—	—
MX17004	2010	2	tmin	—	14.4	14.4	—	—	—	—	—
MX17004	2010	3	tmax	—	—	—	—	32.1	—	—	—
MX17004	2010	3	tmin	—	—	—	—	14.2	—	—	—
MX17004	2010	4	tmax	—	—	—	—	—	—	—	—
MX17004	2010	4	tmin	—	—	—	—	—	—	—	—
MX17004	2010	5	tmax	—	—	—	—	—	—	—	—
MX17004	2010	5	tmin	—	—	—	—	—	—	—	—

Five common problems

Solution:

id	date	element	value
MX17004	2010-01-30	tmax	27.8
MX17004	2010-01-30	tmin	14.5
MX17004	2010-02-02	tmax	27.3
MX17004	2010-02-02	tmin	14.4
MX17004	2010-02-03	tmax	24.1
MX17004	2010-02-03	tmin	14.4
MX17004	2010-02-11	tmax	29.7
MX17004	2010-02-11	tmin	13.4
MX17004	2010-02-23	tmax	29.9
MX17004	2010-02-23	tmin	10.7

id	date	tmax	tmin
MX17004	2010-01-30	27.8	14.5
MX17004	2010-02-02	27.3	14.4
MX17004	2010-02-03	24.1	14.4
MX17004	2010-02-11	29.7	13.4
MX17004	2010-02-23	29.9	10.7
MX17004	2010-03-05	32.1	14.2
MX17004	2010-03-10	34.5	16.8
MX17004	2010-03-16	31.1	17.6
MX17004	2010-04-27	36.3	16.7
MX17004	2010-05-27	33.2	18.2

Five common problems

(4.) Multiple types of observational units are stored in the same table.

year	artist	time	track	date	week	rank
2000	2 Pac	4:22	Baby Don't Cry	2000-02-26	1	87
2000	2 Pac	4:22	Baby Don't Cry	2000-03-04	2	82
2000	2 Pac	4:22	Baby Don't Cry	2000-03-11	3	72
2000	2 Pac	4:22	Baby Don't Cry	2000-03-18	4	77
2000	2 Pac	4:22	Baby Don't Cry	2000-03-25	5	87
2000	2 Pac	4:22	Baby Don't Cry	2000-04-01	6	94
2000	2 Pac	4:22	Baby Don't Cry	2000-04-08	7	99
2000	2Ge+her	3:15	The Hardest Part Of ...	2000-09-02	1	91
2000	2Ge+her	3:15	The Hardest Part Of ...	2000-09-09	2	87
2000	2Ge+her	3:15	The Hardest Part Of ...	2000-09-16	3	92
2000	3 Doors Down	3:53	Kryptonite	2000-04-08	1	81
2000	3 Doors Down	3:53	Kryptonite	2000-04-15	2	70
2000	3 Doors Down	3:53	Kryptonite	2000-04-22	3	68
2000	3 Doors Down	3:53	Kryptonite	2000-04-29	4	67
2000	3 Doors Down	3:53	Kryptonite	2000-05-06	5	66

Five common problems

Solution:

id	artist	track	time	id	date	rank
1	2 Pac	Baby Don't Cry	4:22	1	2000-02-26	87
2	2Ge+her	The Hardest Part Of ...	3:15	1	2000-03-04	82
3	3 Doors Down	Kryptonite	3:53	1	2000-03-11	72
4	3 Doors Down	Loser	4:24	1	2000-03-18	77
5	504 Boyz	Wobble Wobble	3:35	1	2000-03-25	87
6	98~0	Give Me Just One Nig...	3:24	1	2000-04-01	94
7	A*Teens	Dancing Queen	3:44	1	2000-04-08	99
8	Aaliyah	I Don't Wanna	4:15	2	2000-09-02	91
9	Aaliyah	Try Again	4:03	2	2000-09-09	87
10	Adams, Yolanda	Open My Heart	5:30	2	2000-09-16	92
11	Adkins, Trace	More	3:05	3	2000-04-08	81
12	Aguilera, Christina	Come On Over Baby	3:38	3	2000-04-15	70
13	Aguilera, Christina	I Turn To You	4:00	3	2000-04-22	68
14	Aguilera, Christina	What A Girl Wants	3:18	3	2000-04-29	67
15	Alice Deejay	Better Off Alone	6:50	3	2000-05-06	66

Five common problems

(5.) A single observational unit is stored in multiple tables.

Example: exposure and outcome data stored in different files:

- File 1: Daily mortality counts
- File 2: Daily air pollution measurements

We'll take a break now to do the In-Course Exercise (Section 6.4.1).

Joining datasets

Joining datasets

So far, you have only worked with a single data source at a time. When you work on your own projects, however, you typically will need to merge together two or more datasets to create the a data frame to answer your research question.

For example, for air pollution epidemiology, you will often have to join several datasets:

- Health outcome data (e.g., number of deaths per day)
- Air pollution concentrations
- Weather measurements (since weather can be a confounder)
- Demographic data

`*_join` functions

The `dplyr` package has a family of different functions to join two dataframes together, the `*_join` family of functions. All combine two dataframes, which I'll call `x` and `y` here.

The functions include:

- `inner_join(x, y)`: Keep only rows where there are observations in both `x` and `y`.
- `left_join(x, y)`: Keep all rows from `x`, whether they have a match in `y` or not.
- `right_join(x, y)`: Keep all rows from `y`, whether they have a match in `x` or not.
- `full_join(x, y)`: Keep all rows from both `x` and `y`, whether they have a match in the other dataset or not.

*_join functions

In the examples, I'll use two datasets, x and y. Both datasets include the column course. The other column in x is grade, while the other column in y is day. Observations exist for courses x and y in both datasets, but for w and z in only one dataset.

```
x <- data.frame(course = c("x", "y", "z"),  
                 grade = c(90, 82, 78))  
y <- data.frame(course = c("w", "x", "y"),  
                 day = c("Tues", "Mon / Fri", "Tue"))
```

*_join functions

Here is what these two example datasets look like:

x

```
##    course grade
## 1      x    90
## 2      y    82
## 3      z    78
```

y

```
##    course      day
## 1      w      Tues
## 2      x Mon / Fri
## 3      y      Tue
```

*_join functions

With `inner_join`, you'll only get the observations that show up in both datasets. That means you'll lose data on `z` (only in the first dataset) and `w` (only in the second dataset).

```
inner_join(x, y)
```

```
## Joining, by = "course"
```

```
##   course grade      day
## 1      x    90 Mon / Fri
## 2      y    82      Tue
```

*_join functions

With `left_join`, you'll keep everything in `x` (the “left” dataset), but not keep things in `y` that don't match something in `x`. That means that, here, you'll lose `w`:

```
left_join(x, y)
```

```
## Joining, by = "course"
```

```
##   course grade      day
## 1      x    90 Mon / Fri
## 2      y    82      Tue
## 3      z    78     <NA>
```

*_join functions

`right_join` is the opposite– you keep all observations in the “right” dataframe, but only matching ones in the “left” dataframe:

```
right_join(x, y)
```

```
## Joining, by = "course"
```

```
##   course grade      day
## 1      w    NA      Tues
## 2      x    90 Mon / Fri
## 3      y    82      Tue
```

*_join functions

`full_join` keeps everything from both datasets:

```
full_join(x, y)
```

```
## Joining, by = "course"
```

```
##   course grade      day
## 1      x    90 Mon / Fri
## 2      y    82      Tue
## 3      z    78    <NA>
## 4      w    NA    Tues
```


*_join functions

For some more complex examples of using join, I'll use these example datasets:

```
## # A tibble: 4 x 3
##   course grade student
##   <chr>   <dbl> <chr>
## 1 x         92.   a
## 2 x         90.   b
## 3 y         82.   a
## 4 z         78.   b

## # A tibble: 4 x 3
##   class day      student
##   <chr> <chr>    <chr>
## 1 w     Tues     a
## 2 x     Mon / Fri a
## 3 x     Mon / Fri b
## 4 y     Tue      a
```

*_join functions

If you have two datasets you want to join, but the column names for the joining column are different, you can use the `by` argument:

```
full_join(x, y, by = list(x = "course", y = "class"))
```

```
## # A tibble: 7 x 5
```

```
##   course grade student.x day      student.y
##   <chr>  <dbl> <chr>    <chr>    <chr>
## 1 x      92. a      Mon / Fri a
## 2 x      92. a      Mon / Fri b
## 3 x      90. b      Mon / Fri a
## 4 x      90. b      Mon / Fri b
## 5 y      82. a      Tue      a
## 6 z      78. b      <NA>     <NA>
## 7 w      NA  <NA>     Tues     a
```

A few things to note about this example:

- The joining column name for the “left” dataframe (x in this case) is used as the column name for the joined data
- `student` was a column name in both x and y. If we’re not using it to join the data, the column names are changed in the joined data to `student.x` and `student.y`.
- Values are recycled for rows where there were multiple matches across the dataframe (e.g., rows for course “x”)

*_join functions

Sometimes, you will want to join by more than one column. In this example data, it would make sense to join the data by matching both course and student. You can do this by using a vector of all columns to join on:

```
full_join(x, y, by = list(x = c("course", "student"),  
                           y = c("class", "student")))
```

```
## # A tibble: 5 x 4  
##   course grade student day  
##   <chr>  <dbl> <chr>   <chr>  
## 1 x      92. a      Mon / Fri  
## 2 x      90. b      Mon / Fri  
## 3 y      82. a      Tue  
## 4 z      78. b      <NA>  
## 5 w      NA  a      Tues
```

We'll take a break now to do the In-Course Exercise (Section 6.4.2).

Gathering

There are two functions from the `tidyr` package (another member of the `tidyverse`) that you can use to change between wide and long data: `gather` and `spread`.

Here is a description of these two functions:

- `gather`: Take several columns and gather them into two columns, one with the former column names, and one with the former cell values
- `spread`: Take two columns and spread them into multiple columns. Column names for the new columns will come from one of the two original columns, while cell values will come from the other of the original columns.

The following examples are from `tidyr` help files and show the effects of gathering and spreading a dataset.

Here is some wide data:

```
wide_stocks[1:3, ]
```

##		time	X	Y	Z
## 1	2009-01-01	1.26418181	-1.4808143	-2.97040854	
## 2	2009-01-02	-0.06024213	-0.0122563	-1.95975963	
## 3	2009-01-03	0.23477856	1.4407623	0.09714659	

In the `wide_stocks` dataset, there are separate columns for three different stocks (X, Y, and Z). Each cell gives the value for a certain stock on a certain day.

This data isn't "tidy", because the identify of the stock (X, Y, or Z) is a variable, and you'll probably want to include it as a variable in modeling.

```
wide_stocks[1:3, ]
```

##		time	X	Y	Z
## 1	2009-01-01	1.26418181	-1.4808143	-2.97040854	
## 2	2009-01-02	-0.06024213	-0.0122563	-1.95975963	
## 3	2009-01-03	0.23477856	1.4407623	0.09714659	

gather / spread

If you want to convert the dataframe to have all stock values in a single column, you can use `gather` to convert wide data to long data:

```
long_stocks <- gather(wide_stocks, key = stock,  
                      value = price, -time)  
long_stocks[1:5, ]
```

##		time	stock	price
## 1		2009-01-01	X	1.26418181
## 2		2009-01-02	X	-0.06024213
## 3		2009-01-03	X	0.23477856
## 4		2009-01-04	X	0.77918916
## 5		2009-01-05	X	1.02970742

In this “long” dataframe, there is now one column that gives the identify of the stock (stock) and another column that gives the price of that stock that day (price):

```
long_stocks[1:5, ]
```

##		time	stock	price
## 1	2009-01-01	X	1.26418181	
## 2	2009-01-02	X	-0.06024213	
## 3	2009-01-03	X	0.23477856	
## 4	2009-01-04	X	0.77918916	
## 5	2009-01-05	X	1.02970742	

gather / spread

The format for a gather call is:

```
## Generic code
new_df <- gather(old_df,
                  key = [name of column with old column names],
                  value = [name of column with cell values],
                  - [name of column(s) you want to
                     exclude from gather])
```

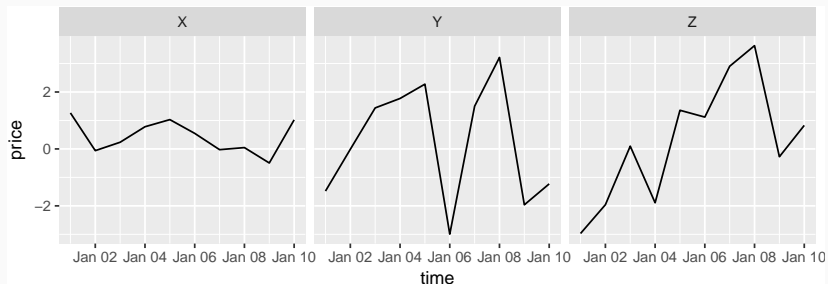
Three important notes:

- Everything is gathered into one of two columns– one column with the old column names, and one column with the old cell values
- With the key and value arguments, you are just providing column names for the two columns that everything's gathered into.
- If there is a column you don't want to gather (date in the example), use - to exclude it in the gather call.

gather / spread

Notice how easy it is, now that the data is gathered, to use `stock` for aesthetics of faceting in a `ggplot2` call:

```
ggplot(long_stocks, aes(x = time, y = price)) +  
  geom_line() +  
  facet_grid(. ~ stock)
```



gather / spread

If you have data in a “long” format and would like to spread it out, you can use `spread` to do that:

```
stocks <- spread(long_stocks, key = stock, value = price)
stocks[1:5, ]
```

##		time	X	Y	Z
## 1		2009-01-01	1.26418181	-1.4808143	-2.97040854
## 2		2009-01-02	-0.06024213	-0.0122563	-1.95975963
## 3		2009-01-03	0.23477856	1.4407623	0.09714659
## 4		2009-01-04	0.77918916	1.7725292	-1.89164625
## 5		2009-01-05	1.02970742	2.2753612	1.35743073

Notice that this reverses the action of `gather`.

“Spread” data is typically not tidy, so you often won’t want to use spread when you are preparing data for analysis. However, spread can be very helpful in creating clean tables for final reports and presentations.

gather / spread

For example, if you wanted to create a table with means and standard deviations for each of the three stocks, you could use `spread` to rearrange the final summary to create an attractive table.

```
stock_summary <- long_stocks %>%  
  group_by(stock) %>%  
  summarize(N = n(), mean = mean(price), sd = sd(price))  
stock_summary
```

```
## # A tibble: 3 x 4  
##   stock      N  mean    sd  
##   <chr> <int> <dbl> <dbl>  
## 1 X         10 0.434 0.580  
## 2 Y         10 0.253 2.08  
## 3 Z         10 0.284 2.13
```



```

stock_summary %>%
  mutate("Mean (Std.dev.)" = paste0(round(mean, 2), " (",
                                     round(sd, 2), ")")) %>%
  select(- mean, - sd) %>%
  gather(key = "Statistic", value = "Value", -stock) %>%
  spread(key = stock, value = Value) %>%
  knitr::kable()

```

Statistic	X	Y	Z
Mean (Std.dev.)	0.43 (0.58)	0.25 (2.08)	0.28 (2.13)
N	10	10	10

Further reading

Chapters 12 and 13 of “R for Data Science” are an excellent supplemental resource if you’d like extra practice on tidy data, gathering / spreading, and joining different datasets.

More with dplyr

So far, you've used several dplyr functions:

- `rename`
- `filter`
- `select`
- `mutate`
- `group_by`
- `summarize`

Some other useful dplyr functions to add to your toolbox are:

- `separate` and `unite`
- `mutate` and other dplyr functions with `group_by`
- `anti_join` and `semi_join`

separate

Sometimes, you want to take one column and split it into two columns. For example, you may have information for two variables in one column:

```
ebola
```

```
## # A tibble: 4 x 1
##   ebola_key
##   <chr>
## 1 Liberia_Cases
## 2 Liberia_Deaths
## 3 Spain_Cases
## 4 Spain_Deaths
```

separate

If you have a consistent “split” character, you can use the `separate` function to split one column into two:

```
ebola %>%  
  separate(col = ebola_key, into = c("country", "outcome"),  
           sep = "_")
```

```
## # A tibble: 4 x 2  
##   country outcome  
##   <chr>    <chr>  
## 1 Liberia Cases  
## 2 Liberia Deaths  
## 3 Spain    Cases  
## 4 Spain    Deaths
```

separate

Here is the generic code for `separate`:

```
## Generic code
separate([dataframe],
         col = [name of the single column you want to split],
         into = [vector of the names of the columns
                 you want to create],
         sep = [the character that designates where
                 you want to split])
```

separate

The default is to drop the original column and only keep the columns into which it was split. However, you can use the argument `remove = FALSE` to keep the first column, as well:

```
ebola %>%  
  separate(col = ebola_key, into = c("country", "outcome"),  
           sep = "_", remove = FALSE)
```

```
## # A tibble: 4 x 3  
##   ebola_key      country outcome  
##   <chr>         <chr>   <chr>  
## 1 Liberia_Cases Liberia Cases  
## 2 Liberia_Deaths Liberia Deaths  
## 3 Spain_Cases   Spain   Cases  
## 4 Spain_Deaths  Spain   Deaths
```


separate

You can use the `fill` argument (`fill = "right"` or `fill = "left"`) to control what happens when there are some observations that do not have the split character.

For example, say your original column looked like this:

```
## # A tibble: 4 x 1
##   ebola_key
##   <chr>
## 1 Liberia_Cases
## 2 Liberia
## 3 Spain_Cases
## 4 Spain_Deaths
```

separate

You can use `fill = "right"` to set how to split observations like the second one, where there is no separator character ("`_`"):

```
ebola %>%  
  separate(col = ebola_key, into = c("country", "outcome"),  
           sep = "_", fill = "right")
```

```
## # A tibble: 4 x 2  
##   country outcome  
##   <chr>    <chr>  
## 1 Liberia Cases  
## 2 Liberia <NA>  
## 3 Spain   Cases  
## 4 Spain   Deaths
```

We'll take a break now to do the In-Course Exercise (Section 6.4.3).

The `unite` function does the reverse of the `separate` function: it lets you join several columns into a single column. For example, say you have data where year, month, and day are split into different columns:

```
## # A tibble: 4 x 3
##   year month   day
##   <dbl> <dbl> <int>
## 1 2016.   10.     1
## 2 2016.   10.     2
## 3 2016.   10.     3
## 4 2016.   10.     4
```

You can use `unite` to join these into a single column:

```
date_example %>%  
  unite(col = date, year, month, day, sep = "-")
```

```
## # A tibble: 4 x 1  
##   date  
##   <chr>  
## 1 2016-10-1  
## 2 2016-10-2  
## 3 2016-10-3  
## 4 2016-10-4
```

If the columns you want to unite are in a row (and in the right order), you can use the `:` syntax with `unite`:

```
date_example %>%  
  unite(col = date, year:day, sep = "-")
```

```
## # A tibble: 4 x 1  
##   date  
##   <chr>  
## 1 2016-10-1  
## 2 2016-10-2  
## 3 2016-10-3  
## 4 2016-10-4
```

Grouping with `mutate` versus `summarize`

So far, we have never used `mutate` with grouping.

You can use `mutate` after grouping— unlike `summarize`, the data will not be collapsed to fewer columns, but the summaries created by `mutate` will be added within each group.

For example, if you wanted to add the mean height and weight by sex to the `nepali` dataset, you could do that with `group_by` and `mutate` (see next slide).

Grouping with mutate versus summarize

```
nepali %>%  
  group_by(sex) %>%  
  mutate(mean_ht = mean(ht, na.rm = TRUE),  
         mean_wt = mean(wt, na.rm = TRUE)) %>%  
  slice(1:3) %>% select(id, sex, wt, ht, mean_ht, mean_wt)
```

```
## # A tibble: 6 x 6  
## # Groups:   sex [2]  
##       id    sex    wt    ht mean_ht mean_wt  
##   <int> <int> <dbl> <dbl>   <dbl>   <dbl>  
## 1 120011     1  12.8  91.2    85.7    11.4  
## 2 120011     1  12.8  93.9    85.7    11.4  
## 3 120011     1  13.1  95.2    85.7    11.4  
## 4 120012     2  14.9 104.     84.6    10.9  
## 5 120012     2  15.1 106.     84.6    10.9  
## 6 120012     2  15.8 108.     84.6    10.9
```


There are also some special functions that work well with `mutate`:

- `lead`: Measured value for following observation
- `lag`: Measured value for previous observation
- `cumsum`: Sum of all values up to this point
- `cummax`: Highest value up to this point
- `cumany`: For TRUE / FALSE, have any been TRUE up to this point

More on mutate

Here is an example of using lead and lag with mutate:

```
library(lubridate)
```

```
## Warning: package 'lubridate' was built under R version 3.4.4
```

```
date_example %>%  
  unite(col = date, year:day, sep = "-") %>%  
  mutate(date = ymd(date),  
         yesterday = lag(date),  
         tomorrow = lead(date))
```

```
## # A tibble: 4 x 3  
##   date      yesterday tomorrow  
##   <date>    <date>    <date>  
## 1 2016-10-01 NA      2016-10-02  
## 2 2016-10-02 2016-10-01 2016-10-03  
## 3 2016-10-03 2016-10-02 2016-10-04
```

slice

You can also group by a factor first using `group_by`. Then, when you use `slice`, you will get the first few rows for each level of the group.

```
nepali %>%  
  group_by(sex) %>%  
  slice(1:2)
```

```
## # A tibble: 4 x 9  
## # Groups:   sex [2]  
##       id    sex    wt    ht  mage    lit  died  alive    age  
##   <int> <int> <dbl> <dbl> <int> <int> <int> <int> <int>  
## 1 120011     1  12.8  91.2    35     0     2     5    41  
## 2 120011     1  12.8  93.9    35     0     2     5    45  
## 3 120012     2  14.9 104.    35     0     2     5    57  
## 4 120012     2  15.1 106.    35     0     2     5    61
```

arrange with group_by

You can also group by a factor before arranging. In this case, all data for the first level of the factor will show up first, in the order given in arrange, then all data from the second level will show up in the specified order, etc.

```
nepali %>%  
  group_by(sex) %>%  
  arrange(desc(ht)) %>%  
  slice(1:2)
```

```
## # A tibble: 4 x 9  
## # Groups:   sex [2]  
##      id    sex    wt    ht  mage   lit  died alive  age  
##   <int> <int> <dbl> <dbl> <int> <int> <int> <int> <int>  
## 1 360302     1  16.8  110.   32     0     1     5    74  
## 2 360392     1  17.2  109.   35     0     1     7    70  
## 3 360791     2  17.5  111.   32     0     0     5    68  
## 4 120112     2  17.0  111.   52     0     0     8    68
```

semi_join and anti_join

These functions allow you to filter one dataframe on only values that **do** have a match in a second dataframe (`semi_join`) or **do not** have a match in a second dataframe (`anti_join`).

I'll show examples with the `x` and `y` dataframes defined before:

```
##   course grade
## 1      x    90
## 2      y    82
## 3      z    78
```

```
##   course      day
## 1      w    Tues
## 2      x Mon / Fri
## 3      y     Tue
```

semi_join and anti_join

The `semi_join` function filters: one dataframe on only values that **do** have a match in a second dataframe:

```
x %>%  
  semi_join(y)
```

```
##   course grade  
## 1      x    90  
## 2      y    82
```

semi_join and anti_join

The `anti_join` function filters one dataframe on only values that **do not** have a match in a second dataframe:

```
y %>%  
  anti_join(x)
```

```
##   course  day  
## 1      w  Tues
```

We'll take a break now to do the In-Course Exercise (Section 6.4.4).

Tidying with dplyr

VADeaths data

For this example, I'll use the VADeaths dataset that comes with R.

This dataset gives the death rates per 1,000 people in Virginia in 1940. It gives death rates by age, gender, and rural / urban:

```
data("VADeaths")
```

```
VADeaths
```

##	Rural	Male	Rural	Female	Urban	Male	Urban	Female
## 50-54		11.7		8.7		15.4		8.4
## 55-59		18.1		11.7		24.3		13.6
## 60-64		26.9		20.3		37.0		19.3
## 65-69		41.0		30.9		54.6		35.1
## 70-74		66.0		54.3		71.1		50.0

There are a few things that make this data untidy:

- One variable (age category) is saved as row names, rather than a column.
- Other variables (gender, rural / urban) are in column names.
- Once you gather the data, you will have two variables (gender, rural / urban) in the same column.

In the following slides, we'll walk through how to tidy this data.

Tidying the VADeaths data

1. One variable (age category) is saved as row names, rather than a column.

To fix this, we need to convert the row names into a new column. We can do this using `mutate` (load `tibble` if needed):

```
VADeaths %>%  
  as.data.frame() %>% ## Convert from matrix to dataframe  
  rownames_to_column(var = "age")
```

##	age	Rural Male	Rural Female	Urban Male	Urban Female
## 1	50-54	11.7	8.7	15.4	8.4
## 2	55-59	18.1	11.7	24.3	13.6
## 3	60-64	26.9	20.3	37.0	19.3
## 4	65-69	41.0	30.9	54.6	35.1
## 5	70-74	66.0	54.3	71.1	50.0

Tidying the VADeaths data

- Two variables (gender, rural / urban) are in column names.

Gather the data to convert column names to a new column:

```
VADeaths %>%  
  as.data.frame() %>%  
  rownames_to_column(var = "age") %>%  
  gather(key = gender_loc, value = mort_rate, - age) %>%  
  slice(1:4)
```

```
## # A tibble: 4 x 3  
##   age   gender_loc mort_rate  
##   <chr> <chr>         <dbl>  
## 1 50-54 Rural Male      11.7  
## 2 55-59 Rural Male      18.1  
## 3 60-64 Rural Male      26.9  
## 4 65-69 Rural Male      41.0
```

Tidying the VADeaths data

3. Two variables (gender, rural / urban) in the same column.

Separate the column into two separate columns for “gender” and “loc” (rural / urban):

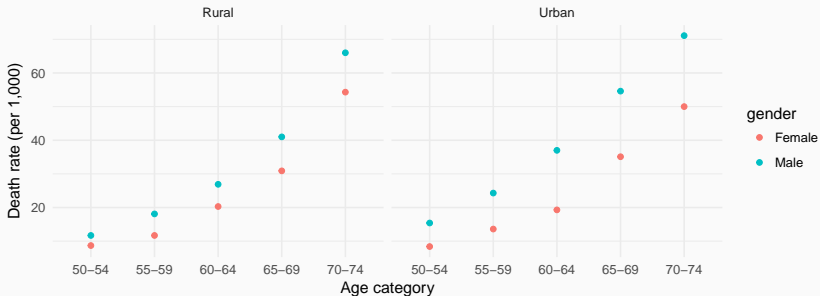
```
VADeaths %>%  
  as.data.frame() %>%  
  rownames_to_column(var = "age") %>%  
  gather(key = gender_loc, value = mort_rate, - age) %>%  
  separate(col = gender_loc, into = c("gender", "loc"),  
           sep = " ") %>%  
  slice(1)
```

```
## # A tibble: 1 x 4  
##   age   gender loc   mort_rate  
##   <chr> <chr> <chr>     <dbl>  
## 1 50-54 Rural  Male      11.7
```

Tidying the VADeaths data

Now that the data is tidy, it's much easier to plot:

```
ggplot(VADeaths, aes(x = age, y = mort_rate,  
                      color = gender)) +  
  geom_point() +  
  facet_wrap( ~ loc, ncol = 2) +  
  xlab("Age category") + ylab("Death rate (per 1,000)") +  
  theme_minimal()
```

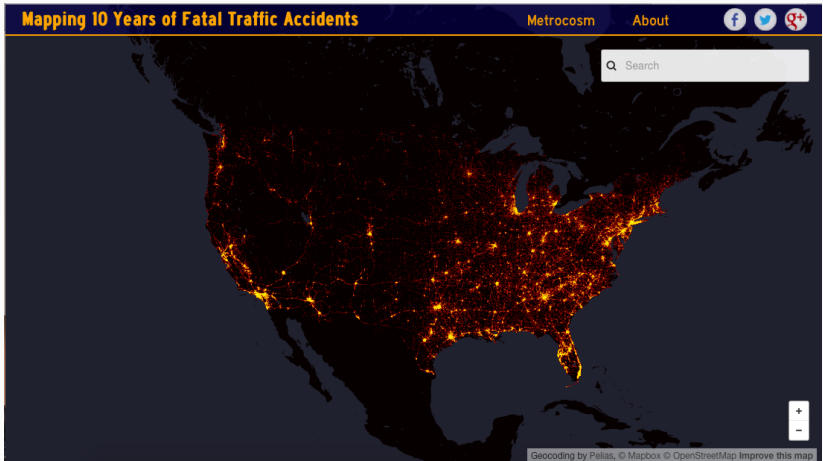


FARS data

The US Department of Transportation runs the Fatality Analysis Reporting System (FARS), which gathers data on all fatal motor vehicle accidents. Here is a description from their documentation:

- Motor vehicle on a public road
- Resulted in a death within 30 days of the crash
- Includes crashes in the 50 states, DC, and Puerto Rico

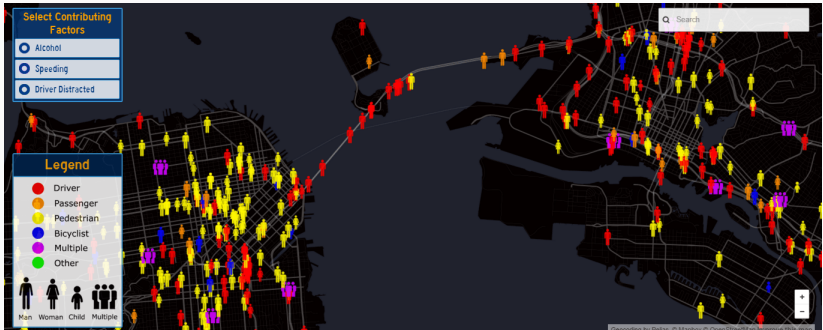
FARS data



Source: Max Galka

<http://metrocosm.com/10-years-of-traffic-accidents-mapped.html>

FARS data



Source: Max Galka

<http://metrocosm.com/10-years-of-traffic-accidents-mapped.html>

We'll take a break now to do the In-Course Exercise (Section 6.4.5).

String operations

String operations

For these examples, we'll use some data on passengers of the Titanic. You can load this data using:

```
# install.packages("titanic")  
library(titanic)  
data("titanic_train")
```

We will be using the stringr package:

```
library(stringr)
```

String operations

This data includes a column called “Name” with passenger names. This column is somewhat messy and includes several elements that we might want to separate (last name, first name, title). Here are the first few values of “Name”:

```
titanic_train %>% select(Name) %>% slice(1:3)
```

```
## # A tibble: 3 x 1
```

```
##   Name
```

```
##   <chr>
```

```
## 1 Braund, Mr. Owen Harris
```

```
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)
```

```
## 3 Heikkinen, Miss. Laina
```

String operations

The `str_trim` function from the `stringr` package allows you to trim leading and trailing whitespace:

```
with_spaces <- c("  a ", "  bob", " gamma")  
with_spaces
```

```
## [1] "  a " "  bob" " gamma"
```

```
str_trim(with_spaces)
```

```
## [1] "a"      "bob"     "gamma"
```

This is rarer, but if you ever want to, you can add leading and / or trailing whitespace to elements of a character vector with `str_pad` from the `stringr` package.

String operations

There are also functions to change a full character string to uppercase, lowercase, or title case:

```
titanic_train$Name[1]
```

```
## [1] "Braund, Mr. Owen Harris"
```

```
str_to_upper(titanic_train$Name[1])
```

```
## [1] "BRAUND, MR. OWEN HARRIS"
```

```
str_to_lower(titanic_train$Name[1])
```

```
## [1] "braund, mr. owen harris"
```

```
str_to_title(str_to_lower(titanic_train$Name[1]))
```

```
## [1] "Braund, Mr. Owen Harris"
```

Regular expressions

Regular expressions

We've already done some things to manipulate strings. For example, if we wanted to separate "Name" into last name and first name (including title), we could actually do that with the `separate` function:

```
titanic_train %>%  
  select(Name) %>%  
  slice(1:3) %>%  
  separate(Name, c("last_name", "first_name"), sep = ", ")
```

```
## # A tibble: 3 x 2  
##   last_name first_name  
##   <chr>      <chr>  
## 1 Braund     Mr. Owen Harris  
## 2 Cumings    Mrs. John Bradley (Florence Briggs Thayer)  
## 3 Heikkinen  Miss. Laina
```

Regular expressions

Notice that `separate` is looking for a regular pattern (",") and then doing something based on the location of that pattern in each string (splitting the string).

There are a variety of functions in R that can perform manipulations based on finding regular patterns in character strings.

Regular expressions

The `str_detect` function will look through each element of a character vector for a designated pattern. If the pattern is there, it will return `TRUE`, and otherwise `FALSE`. The convention is:

```
## Generic code  
str_detect(string = [vector you want to check],  
            pattern = [pattern you want to check for])
```

For example, to create a logical vector specifying which of the Titanic passenger names include “Mrs.”, you can call:

```
mrs <- str_detect(titanic_train$Name, "Mrs\\.")  
head(mrs)
```

```
## [1] FALSE TRUE FALSE TRUE FALSE FALSE
```

Regular expressions

The result is a logical vector, so `str_detect` can be used in `filter` to subset data to only rows where the passenger's name includes "Mrs.":

```
titanic_train %>%  
  filter(str_detect(Name, "Mrs\\\\")) %>%  
  select(Name) %>%  
  slice(1:3)
```

```
## # A tibble: 3 x 1  
##   Name  
##   <chr>  
## 1 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  
## 2 Futrelle, Mrs. Jacques Heath (Lily May Peel)  
## 3 Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
```

Regular expressions

As a note, in regular expressions, all of the following characters are special characters that need to be escaped with backslashes if you want to use them literally:

`. * + ^ ? $ \ | () [] { }`

Regular expressions

There is an older, base R function called `grep1` that does something very similar (although note that the order of the arguments is reversed).

```
titanic_train %>%  
  filter(grep1("Mrs\\.", Name)) %>%  
  select(Name) %>%  
  slice(1:3)
```

```
## # A tibble: 3 x 1  
##   Name  
##   <chr>  
## 1 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  
## 2 Futrelle, Mrs. Jacques Heath (Lily May Peel)  
## 3 Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
```


Regular expressions

The `str_extract` function can be used to extract a string (if it exists) from each value in a character vector. It follows similar conventions to `str_detect`:

```
## Generic code  
str_extract(string = [vector you want to check],  
            pattern = [pattern you want to check for])
```

Regular expressions

For example, you might want to extract “Mrs.” if it exists in a passenger’s name:

```
titanic_train %>%  
  mutate(mrs = str_extract(Name, "Mrs\\.\\.")) %>%  
  select(Name, mrs) %>%  
  slice(1:3)
```

```
## # A tibble: 3 x 2  
##   Name                                     mrs  
##   <chr>                                <chr>  
## 1 Braund, Mr. Owen Harris             <NA>  
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) Mrs.  
## 3 Heikkinen, Miss. Laina             <NA>
```

Notice that now we’re creating a new column (`mrs`) that either has “Mrs.” (if there’s a match) or is missing (`NA`) if there’s not a match.

Regular expressions

For this first example, we were looking for an exact string (“Mrs”). However, you can use patterns that match a particular pattern, but not an exact string. For example, we could expand the regular expression to find “Mr.” or “Mrs.”:

```
titanic_train %>%  
  mutate(title = str_extract(Name, "Mr\\.|Mrs\\.")) %>%  
  select(Name, title) %>%  
  slice(1:3)
```

```
## # A tibble: 3 x 2  
##   Name                                     title  
##   <chr>                                <chr>  
## 1 Braund, Mr. Owen Harris              Mr.  
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) Mrs.  
## 3 Heikkinen, Miss. Laina              <NA>
```

Note that this pattern uses a special operator (|) to find one pattern **or** another. Double backslashes (\\) **escape** the special character “.”.

Regular expressions

Notice that “Mr.” and “Mrs.” both start with “Mr”, end with “.”, and may or may not have an “s” in between.

```
titanic_train %>%  
  mutate(title = str_extract(Name, "Mr(s)*\\.")) %>%  
  select(Name, title) %>%  
  slice(1:3)
```

```
## # A tibble: 3 x 2  
##   Name                                     title  
##   <chr>                                <chr>  
## 1 Braund, Mr. Owen Harris              Mr.  
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) Mrs.  
## 3 Heikkinen, Miss. Laina              <NA>
```

This pattern uses (s)* to match zero or more “s”s at this spot in the pattern.

Regular expressions

In the previous code, we found “Mr.” and “Mrs.”, but missed “Miss.”. We could tweak the pattern again to try to capture that, as well. For all three, we have the pattern that it starts with “M”, has some lowercase letters, and then ends with “.”.

```
titanic_train %>%  
  mutate(title = str_extract(Name, "M[a-z]+\\.\\.")) %>%  
  select(Name, title) %>%  
  slice(1:3)
```

```
## # A tibble: 3 x 2  
##   Name                                     title  
##   <chr>                                <chr>  
## 1 Braund, Mr. Owen Harris              Mr.  
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) Mrs.  
## 3 Heikkinen, Miss. Laina               Miss.
```

Regular expressions

The last pattern used `[a-z]+` to match one or more lowercase letters. The `[a-z]` is a **character class**.

You can also match digits (`[0-9]`), uppercase letters (`[A-Z]`), just some letters (`[aeiou]`), etc.

You can negate a character class by starting it with `^`. For example, `[^0-9]` will match anything that **isn't** a digit.

Regular expressions

Sometimes, you want to match a pattern, but then only subset a part of it. For example, each passenger seems to have a title ("Mr.", "Mrs.", etc.) that comes after ",", and before ".". We can use this pattern to find the title, but then we get some extra stuff with the match:

```
titanic_train %>%  
  mutate(title = str_extract(Name, ",\\s[A-Za-z]*\\.\\s")) %>%  
  select(title) %>%  
  slice(1:3)
```

```
## # A tibble: 3 x 1  
##   title  
##   <chr>  
## 1 ", Mr. "  
## 2 ", Mrs. "  
## 3 ", Miss. "
```

As a note, in this pattern, `\\s` is used to match a space.

Regular expressions

We are getting things like “, Mr.”, when we really want “Mr”. We can use the `str_match` function to do this. We group what we want to extract from the pattern in parentheses, and then the function returns a matrix. The first column is the full pattern match, and each following column gives just what matches within the groups.

```
head(str_match(titanic_train$Name,  
              pattern = ",\\s([A-Za-z]*)\\.\\.\\s"))
```

```
##      [,1]      [,2]  
## [1,] ", Mr. "  "Mr"  
## [2,] ", Mrs. "  "Mrs"  
## [3,] ", Miss. " "Miss"  
## [4,] ", Mrs. "  "Mrs"  
## [5,] ", Mr. "   "Mr"  
## [6,] ", Mr. "   "Mr"
```


Regular expressions

To get just the title, then, we can run:

```
titanic_train %>%  
  mutate(title =  
    str_match(Name, "\\s([A-Za-z]*)\\.\\.\\s")[, 2]) %>%  
  select(Name, title) %>%  
  slice(1:3)
```

```
## # A tibble: 3 x 2  
##   Name                                     title  
##   <chr>                                <chr>  
## 1 Braund, Mr. Owen Harris              Mr  
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) Mrs  
## 3 Heikkinen, Miss. Laina              Miss
```

The `[, 2]` pulls out just the second column from the matrix returned by `str_match`.

Regular expressions

Here are some of the most common titles:

```
titanic_train %>%  
  mutate(title =  
    str_match(Name, "\\s([A-Za-z]*)\\.\\.\\s")[, 2]) %>%  
  group_by(title) %>% summarize(n = n()) %>%  
  arrange(desc(n)) %>% slice(1:5)
```

```
## # A tibble: 5 x 2  
##   title      n  
##   <chr>   <int>  
## 1 Mr      517  
## 2 Miss    182  
## 3 Mrs     125  
## 4 Master   40  
## 5 Dr       7
```

Regular expressions

The following slides have a few other examples of regular expressions in action with this dataset.

Get just names that start with ("^") the letter "A":

```
titanic_train %>%  
  filter(str_detect(Name, "^A")) %>%  
  select(Name) %>%  
  slice(1:3)  
  
## # A tibble: 3 x 1  
##   Name  
##   <chr>  
## 1 Allen, Mr. William Henry  
## 2 Andersson, Mr. Anders Johan  
## 3 Asplund, Mrs. Carl Oscar (Selma Augusta Emilia Johansson)
```

Regular expressions

Get names with “II” or “III” ({2,} says to match at least two times):

```
titanic_train %>%  
  filter(str_detect(Name, "I{2,}")) %>%  
  select(Name) %>%  
  slice(1:3)
```

```
## # A tibble: 2 x 1  
##   Name  
##   <chr>  
## 1 Carter, Master. William Thornton II  
## 2 Roebling, Mr. Washington Augustus II
```

Regular expressions

Get names with “Andersen” or “Anderson” (alternatives in square brackets):

```
titanic_train %>%  
  filter(str_detect(Name, "Anders[eo]n")) %>%  
  select(Name)
```

```
##                               Name  
## 1 Andersen-Jensen, Miss. Carla Christine Nielsine  
## 2                               Anderson, Mr. Harry  
## 3                               Walker, Mr. William Anderson  
## 4                               Olsvigen, Mr. Thor Anderson  
## 5          Soholt, Mr. Peter Andreas Lauritz Andersen
```

Regular expressions

Get names that start with (“^” outside of brackets) the letters “A” and “B”:

```
titanic_train %>%  
  filter(str_detect(Name, "^[AB]")) %>%  
  select(Name) %>%  
  slice(1:3)
```

```
## # A tibble: 3 x 1  
##   Name  
##   <chr>  
## 1 Braund, Mr. Owen Harris  
## 2 Allen, Mr. William Henry  
## 3 Bonnell, Miss. Elizabeth
```

Regular expressions

Get names that end with (“\$”) the letter “b” (either lowercase or uppercase):

```
titanic_train %>%  
  filter(str_detect(Name, "[bB]$")) %>%  
  select(Name)
```

```
##                               Name  
## 1   Emir, Mr. Farred Chehab  
## 2 Goldschmidt, Mr. George B  
## 3           Cook, Mr. Jacob  
## 4           Pasic, Mr. Jakob
```

Regular expression

Some useful regular expression operators include:

Operator	Meaning
.	Any character
*	Match 0 or more times (greedy)
?	Match 0 or more times (non-greedy)
+	Match 1 or more times (greedy)
+?	Match 1 or more times (non-greedy)
^	Starts with (in brackets, negates)
\$	Ends with
[...]	Character classes

For more on these patterns, see:

- Help file for the `stringi-search-regex` function in the `stringi` package (which should install when you install `stringr`)
- Chapter 14 of R For Data Science
- <http://gskinner.com/RegExr>: Interactive tool for helping you build regular expression pattern strings