Applied Data Science with R: Data Cleaning

Matthias Haber

20 February 2019

Leftovers

JSON

JSON

- Javascript Object Notation
- Lightweight data storage
- Common format for data from application programming interfaces (APIs)
- Similar structure to XML but different syntax
- Data stored as Numbers (double), Strings (double quoted), Boolean (*true* or *false*), Array (ordered, comma separated enclosed in square brackets), Object (unorderd, comma separated collection of key:value pairs in curley brackets $\{\}$)

http://en.wikipedia.org/wiki/JSON

Example JSON file


```
[{
  "_id": {
    "$oid": "5968dd23fc13ae04d9000001"
  },
  "product_name": "sildenafil citrate",
  "supplier": "Wisozk Inc",
  "quantity": 261,
  "unit_cost": "$10.47"
}, {
  "_id": {
    "$oid": "5968dd23fc13ae04d9000002"
  },
  "product_name": "Mountain Juniperus ashei",
  "supplier": "Keebler-Hilpert",
  "quantity": 292,
  "unit_cost": "$8.74"
}, {
  "_id": {
    "$oid": "5968dd23fc13ae04d9000003"
  },
  "product_name": "Dextromathorphan HBr",
  "supplier": "Schmitt-Weissnat",
  "quantity": 211,
  "unit_cost": "$20.53"
}]
```

Reading data from JSON (with jsonlite)

```
url <- paste0("http://mysafeinfo.com/api/",
"data?list=englishmonarchs&format=json")
jsonData <- jsonlite::fromJSON(url)
str(jsonData)
```

```
## 'data.frame':    57 obs. of  5 variables:
##  $ id : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ nm : chr  "Edward the Elder" "Athelstan" "Edmund" "Edred"
##  $ cty: chr  "United Kingdom" "United Kingdom" "United Kingdo
##  $ hse: chr  "House of Wessex" "House of Wessex" "House of We
##  $ yrs: chr  "899-925" "925-940" "940-946" "946-955" ...
```

Writing data frames to JSON

You can use `toJSON()` to convert R data to a JSON object. JSONs can come in mini or pretty format with indentation, whitespace and new lines.

```
# Mini
{"a":1,"b":2,"c":{"x":5,"y":6}}
# Pretty
{
  "a": 1,
  "b": 2,
  "c": {
    "x": 5,
    "y": 6
  }
}
```

Convert back to JSON

```
myJson <- jsonlite::toJSON(iris)
iris2 <- jsonlite::fromJSON(myJson)
head(iris2)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

Last Week's Homework

Reading in Flat files

```
library(tidyverse)
text <- read_csv("x,y,z\n1,2,'a,b'", quote = "'")
text
```

```
## # A tibble: 1 x 3
##       x     y z
##   <int> <int> <chr>
## 1     1     2 a,b
```

Parsing numbers

```
url <- paste0("http://s3.amazonaws.com/assets.datacamp.com/",
              "production/course_1561/datasets/",
              "national_debt.csv")
debt <- read_csv(url)
debt$V2 <- parse_number(debt$V2)
mean(debt$V2)
```

```
## [1] 1.102407e+13
```

Parsing Dates

```
url <- paste0("http://s3.amazonaws.com/assets.datacamp.com/",
              "production/course_1561/datasets/",
              "weather.csv")
weather <- read_csv(url)
weather$date <- parse_date(weather$date, format = "%m/%d/%Y")
```

Reading in Excel

```
library(readxl)
url <- paste0("http://s3.amazonaws.com/assets.datacamp.com/",
              "production/course_1294/datasets/",
              "mbta.xlsx")
download.file(url, "mbta.xlsx")

mbta <- read_excel("mbta.xlsx", range = "C13:N13",
                   col_names = FALSE)
mbta_t <-t(mbta)
mean(mbta_t)
```

```
## [1] 1227.169
```

Week 3: Data Cleaning

Prerequisites

Packages

```
library(tidyverse)
library(readr)
```

Data

```
# base GitHub url
url <- paste0("https://raw.githubusercontent.com/",
              "mhaber/AppliedDataScience/master/",
              "slides/week3/data/")

# link to datasets
pew <- read_csv(paste0(url,"pew.csv"))
billboard <- read_csv(paste0(url,"billboard.csv"))
weather <- read_tsv(paste0(url,"weather.txt"))
```

Piping

The pipe operator %>% (Ctrl/Cmd+Shift+M) allows you to write code in sequences which has several benefits:

- serves the natural way of reading ("First this, then this, . . . ")
- avoids nested function calls
- minimizes the need for local variables and function definitions

Tidy data

Tidy data

Data come in multiple ways. Take the the following datasets: they show the same values of four variables *country*, *year*, *population*, and *cases*, but each dataset organises the values in a different way:

Tidy data

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

Tidy data

country	year	type	count
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

Tidy data

country	year	rate
Afghanistan	1999	745/19987071
Afghanistan	2000	2666/20595360
Brazil	1999	37737/172006362
Brazil	2000	80488/174504898
China	1999	212258/1272915272
China	2000	213766/1280428583

Tidy data

- In tidy data:
  - Each variable forms a column
  - Each observation forms a row
  - Each type of observational unit forms a table

- Any dataset that doesn't satisfy these conditions is considered 'messy'

Tidy data

Why data should be tidy:

1. Consistency: If you have a consistent data structure, it's easier to learn the tools that work with it because they have an underlying uniformity.
2. Conformity: R is a vectorized programming language. Data structures in R are built from vectors and R's operations are optimized to work with vectors. Tidy data takes advantage of both of these traits.
3. Compatibility: dplyr, ggplot2, and all the other packages in the tidyverse are designed to work with tidy data.

Tidy data

Assume that in these data sets, cases refers to the number of people diagnosed with TB per country per year. To calculate the rate of TB cases per country per year (i.e, the number of people per 10,000 diagnosed with TB), you will need to do four operations with the data. You will need to:

- Extract the number of TB cases per country per year
- Extract the population per country per year (in the same order as above)
- Divide cases by population
- Multiply by 10000

Tidy data

```
# Compute rate per 10,000
table1 %>%
  dplyr::mutate(rate = cases / population * 10000)
```

```
## # A tibble: 6 x 5
##   country      year  cases population  rate
##   <chr>       <int>  <int>      <int> <dbl>
## 1 Afghanistan  1999    745   19987071 0.373
## 2 Afghanistan  2000   2666   20595360 1.29
## 3 Brazil       1999  37737  172006362 2.19
## 4 Brazil       2000  80488  174504898 4.61
## 5 China        1999 212258 1272915272 1.67
## 6 China        2000 213766 1280428583 1.67
```

Tidy data

```
# Compute cases per year
table1 %>%
  dplyr::count(year, wt = cases)
```

```
## # A tibble: 2 x 2
##    year      n
##   <int>  <int>
## 1  1999 250740
## 2  2000 296920
```

Exercises

Raw data is rarely tidy and is much harder to work with as a result

1. Compute the `rate` for `table2`. You will need to perform four operations:
    1.1 Extract the number of TB cases per country per year.
    1.2 Extract the matching population per country per year.
    1.3 Divide cases by population, and multiply by 10000.
    1.4 Store back in the appropriate place.

gathering and spreading

`gather()` and `spread()`

The two most important functions in `tidyr` are `gather()` and `spread()`. `tidyr` builds on the idea of a key value pair. A key that explains what the information describes, and a value that contains the actual information (e.g. *Password: 0123456789*).

`gather()` makes wide tables narrower and longer; `spread()` makes long tables shorter and wider.

pew data

religion	<$10k	$10-20k	$20-30k	$30-40k
Agnostic	27	34	60	81
Atheist	12	27	37	52
Buddhist	27	21	30	34

- What variables are in this dataset?
- How does a tidy version of this table look like?

`gather()`

- Problem: Column names are not names of a variable, but *values*
- Goal: Gather the non-variable volumns into a two-column key-value pair

`gather()`

Three parameters:

1. Set of columns that represent values, not variables
2. The name of the variable whose values form the column names (key).
3. The name of the variable whose values are spread over the cells (value).

gather()

```
pewTidy <- pew %>%
  tidyr::gather(key = income, value = frequency,
                -religion)
```

religion	income	frequency
Agnostic	<$10k	27
Atheist	<$10k	12
Buddhist	<$10k	27
Catholic	<$10k	418
Don't know/refused	<$10k	15

Billboard data

year	artist	track	time
2000	2 Pac	Baby Don't Cry (Keep...	04:22:00
2000	2Ge+her	The Hardest Part Of ...	03:15:00
2000	3 Doors Down	Kryptonite	03:53:00
2000	3 Doors Down	Loser	04:24:00
2000	504 Boyz	Wobble Wobble	03:35:00

date.entered	wk1	wk2	wk3	wk4	wk5
2000-02-26	87	82	72	77	87
2000-09-02	91	87	92	NA	NA
2000-04-08	81	70	68	67	66
2000-10-21	76	76	72	69	67
2000-04-15	57	34	25	17	17

Tidying the Billboard data

To tidy this dataset, we first `gather` together all the `wk` columns. The column names give the week and the values are the ranks:

```
billboardTidy <- billboard %>%
  tidyr::gather(key = week, value = rank,
                wk1:wk76, na.rm = TRUE)
```

Tidying the Billboard data

year	artist	track	time
2000	2 Pac	Baby Don't Cry (Keep...	04:22:00
2000	2Ge+her	The Hardest Part Of ...	03:15:00
2000	3 Doors Down	Kryptonite	03:53:00
2000	3 Doors Down	Loser	04:24:00

date.entered	week	rank
2000-02-26	wk1	87
2000-09-02	wk1	91
2000-04-08	wk1	81
2000-10-21	wk1	76

Are we done?

data cleaning with `dplyr()`

Let's turn the week into a numeric variable and create a proper date column

```
billboardTidy2 <- billboardTidy %>%
  dplyr::mutate(week = readr::parse_number(week),
                date = as.Date(date.entered) + 7 *
                  (week - 1)) %>%
  dplyr::select(-date.entered) %>%
  dplyr::arrange(artist, track, week)
```

data cleaning with `dplyr()`

year	artist	track	time
2000	2 Pac	Baby Don't Cry (Keep...	04:22:00
2000	2 Pac	Baby Don't Cry (Keep...	04:22:00
2000	2 Pac	Baby Don't Cry (Keep...	04:22:00
2000	2 Pac	Baby Don't Cry (Keep...	04:22:00

week	rank	date
1	87	2000-02-26
2	82	2000-03-04
3	72	2000-03-11
4	77	2000-03-18

spread()

Spreading is the opposite of gathering. You use it when an observation is scattered across multiple rows. `spread()` turns a pair of key:value columns into a set of tidy columns.

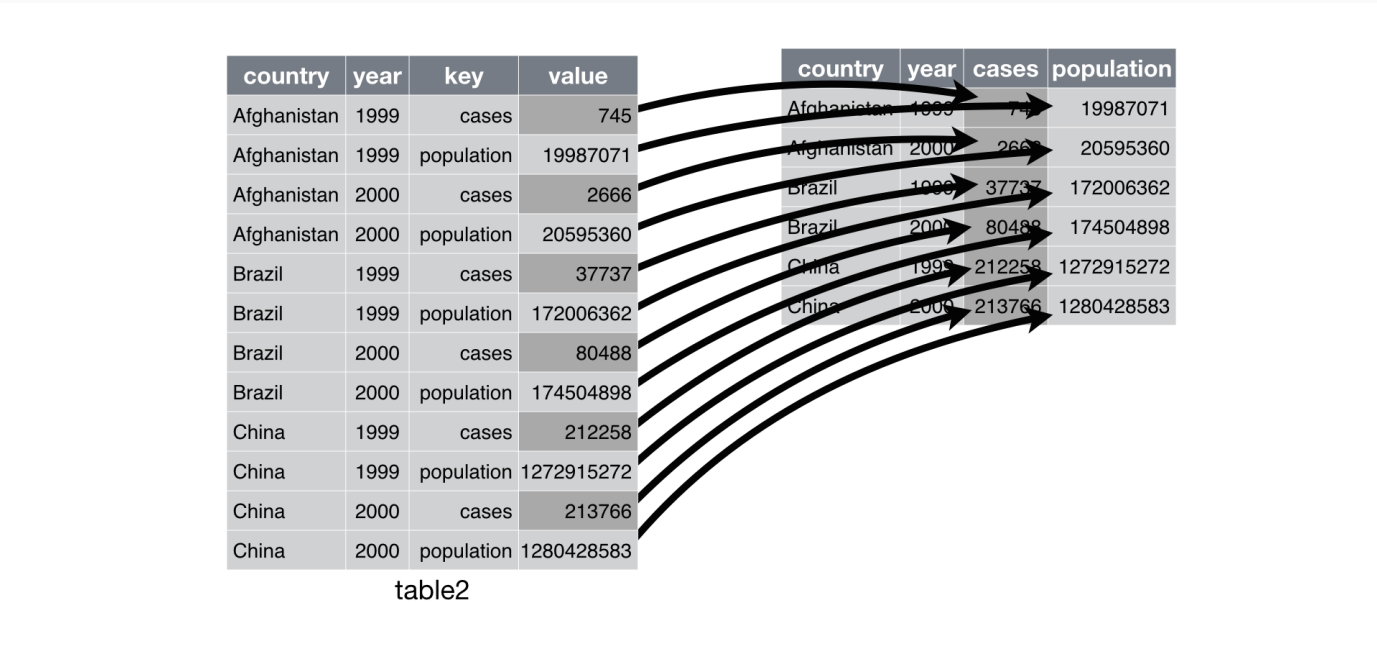
For example, take `table2`: an observation is a country in a year, but each observation is spread across two rows.

country	year	type	count
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488

spread()

To tidy this up, we first analyse the representation in similar way to `gather()`. This time, however, we only need two parameters:

- The column that contains variable names, the `key` column. Here, it's `type`.
- The column that contains values forms multiple variables, the `value` column. Here it's `count`.



spread()

```
table2 %>%
  tidyr::spread(key = type, value = count)
```

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

Exercises

1. Are `gather()` and `spread()` perfectly symmetrical? Carefully consider the following example:

```
stocks <- tibble(
  year   = c(2015, 2015, 2016, 2016),
  half   = c(   1,    2,     1,    2),
  return = c(1.88, 0.59, 0.92, 0.17)
    )
stocks %>%
  tidyr::spread(year, return) %>%
  tidyr::gather("year", "return", `2015`:`2016`)
```

2. Both `spread()` and `gather()` have a `convert` argument. What does it do?

Exercises

3. Why does this code fail?

```
table4a %>%
  tidyr::gather(1999, 2000, key = "year", value = "cases")
```

4.Using the `weather` data: Tidy the day columns `X1-X31` and save the result as `weatherTidy`. Finally, spread the measure column of weatherTidy and save the result as `weatherTidy2`.

Separating and uniting

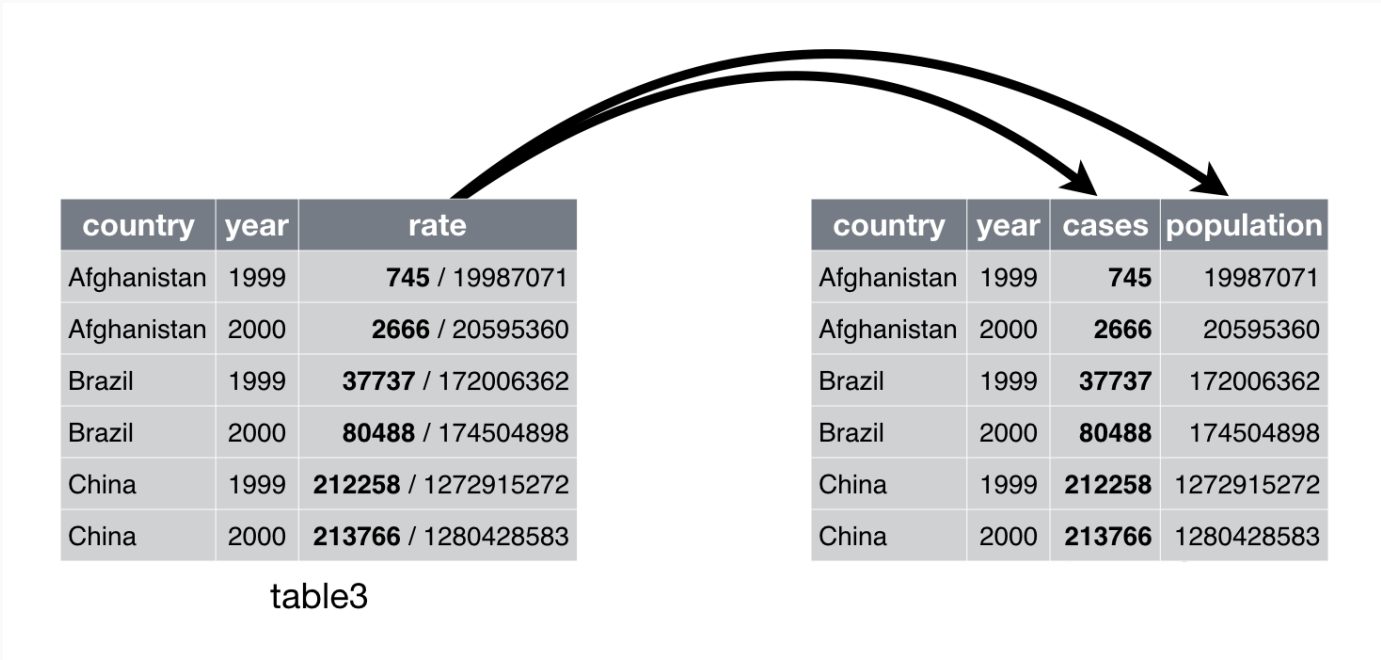
`separate()`

`separate()` pulls apart one column into multiple columns, by splitting wherever a separator character appears. Take `table3`:

country	year	rate
Afghanistan	1999	745/19987071
Afghanistan	2000	2666/20595360
Brazil	1999	37737/172006362
Brazil	2000	80488/174504898
China	1999	212258/1272915272
China	2000	213766/1280428583

separate()

The `rate` column contains both `cases` and `population` variables, and we need to split it into two variables. `separate()` takes the name of the column to separate, and the names of the columns to separate into.



country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

table3

country	year	cases	population
Afghanistan	1999	**745**	19987071
Afghanistan	2000	**2666**	20595360
Brazil	1999	**37737**	172006362
Brazil	2000	**80488**	174504898
China	1999	**212258**	1272915272
China	2000	**213766**	1280428583

separate()

```
table3 %>%
  tidyr::separate(rate, into = c("cases", "population"))
```

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

`separate()`

By default, `separate()` will split values at non-alphanumeric characters (!number, !letter). If you wish to use a specific character to separate a column, you can pass the character to the `sep` argument of `separate()`.

separate()

```
table3 %>%
  tidyr::separate(rate, into = c("cases", "population"),
                  sep = "/")
```

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

`separate()`

You can also pass a vector of integers to `sep`. `separate()` will interpret the integers as positions to split at. Positive values start at 1 on the far-left of the strings; negative value start at -1 on the far-right of the strings.

You can use this arrangement to separate the last two digits of each year. This make this data less tidy, but is useful in other cases, as you'll see in a little bit.

separate()

```
table3 %>%
  tidyr::separate(year, into = c("century", "year"),
                  sep = 2)
```

country	century	year	rate
Afghanistan	19	99	745/19987071
Afghanistan	20	00	2666/20595360
Brazil	19	99	37737/172006362
Brazil	20	00	80488/174504898
China	19	99	212258/1272915272
China	20	00	213766/1280428583

`unite()`

`unite()` is the inverse of `separate()`: it combines multiple columns into a single column. You'll need it much less frequently than `separate()`, but it's still a useful tool to have in your back pocket.

We can use `unite()` to rejoin the *century* and *year* columns in `table5`. `unite()` takes a data frame, the name of the new variable to create, and a set of columns to combine.

unite()

```
table5 %>%
  tidyr::unite(new, century, year)
```

country	new	rate
Afghanistan	19_99	745/19987071
Afghanistan	20_00	2666/20595360
Brazil	19_99	37737/172006362
Brazil	20_00	80488/174504898
China	19_99	212258/1272915272
China	20_00	213766/1280428583

unite()

By default, `unite()` will place an underscore (_) between the values from different columns. If we don't want any separator we use "":

```
table5 %>%
  tidyr::unite(new, century, year, sep = "")
```

country	new	rate
Afghanistan	1999	745/19987071
Afghanistan	2000	2666/20595360
Brazil	1999	37737/172006362
Brazil	2000	80488/174504898
China	1999	212258/1272915272
China	2000	213766/1280428583

Missing values

Missing values

> *"An explicit missing value is the presence of an absence; an implicit missing value is the absence of a presence."*

A value can be missing in one of two possible ways:

- *Explicitly* i.e. flagged with `NA`.
- *Implicitly*, i.e. simply not present in the data.

Let's illustrate this idea with a very simple data set:

```
stocks <- tibble(
  year   = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
  qtr    = c(   1,    2,    3,    4,    2,    3,    4),
  return = c(1.88, 0.59, 0.35,   NA, 0.92, 0.17, 2.66)
)
```

Missing values

There are two missing values in this dataset:

- The return for the 4th quarter of 2015 is explicitly missing (`NA`).
- The return for the 1st quarter of 2016 is implicitly missing.

We can make the implicit missing value explicit by putting years in the columns:

Missing values

```
stocks %>%
  tidyr::spread(year, return)
```

qtr	2015	2016
1	1.88	NA
2	0.59	0.92
3	0.35	0.17
4	NA	2.66

Remove missing values

You can also set `na.rm = TRUE` in `gather()` to turn explicit missing values implicit:

```
stocks %>%
  tidyr::spread(year, return) %>%
  tidyr::gather(year, return, `2015`:`2016`, na.rm = TRUE)
```

qtr	year	return
1	2015	1.88
2	2015	0.59
3	2015	0.35
2	2016	0.92
3	2016	0.17
4	2016	2.66

`complete()`

You can also use `complete()` for making missing values explicit. `complete()` takes a set of columns, and finds all unique combinations; filling in explicit `NA`s where necessary.

complete()

```
stocks %>%
  tidyr::complete(year, qtr)
```

year	qtr	return
2015	1	1.88
2015	2	0.59
2015	3	0.35
2015	4	NA
2016	1	NA
2016	2	0.92
2016	3	0.17
2016	4	2.66

fill()

Finally, you can fill in missing values with `fill()`. It takes a set of columns where you want missing values to be replaced by the most recent non-missing value (sometimes called last observation carried forward).

```
treatment <- tribble(
  ~ person,           ~ treatment, ~response,
  "Derrick Whitmore", 1,           7,
  NA,                 2,           10,
  NA,                 3,           9,
  "Katherine Burke",  1,           4
)
```

fill()

```
treatment %>%
  tidyr::fill(person)
```

person	treatment	response
Derrick Whitmore	1	7
Derrick Whitmore	2	10
Derrick Whitmore	3	9
Katherine Burke	1	4

Homework Exercises

Homework Exercises

For this week's homework exersises go to Moodle and answer the Quiz posted in the week 3 section.

Deadline: Tuesday, February 26.

That's it for today. Questions?