Tidy data

(This is an informal and code heavy version of the full tidy data paper. Please refer to that for more details.)

Data tidying

It is often said that 80% of data analysis is spent on the cleaning and preparing data. And it's not just a first step, but it must be repeated many times over the course of analysis as new problems come to light or new data is collected. To get a handle on the problem, this paper focuses on a small, but important, aspect of data cleaning that I call data **tidying**: structuring datasets to facilitate analysis.

The principles of tidy data provide a standard way to organise data values within a dataset. A standard makes initial data cleaning easier because you don't need to start from scratch and reinvent the wheel every time. The tidy data standard has been designed to facilitate initial exploration and analysis of the data, and to simplify the development of data analysis tools that work well together. Current tools often require translation. You have to spend time munging the output from one tool so you can input it into another. Tidy datasets and tidy tools work hand in hand to make data analysis easier, allowing you to focus on the interesting domain problem, not on the uninteresting logistics of data.

Defining tidy data

> Happy families are all alike; every unhappy family is unhappy in its own way — Leo Tolstoy

Like families, tidy datasets are all alike but every messy dataset is messy in its own way. Tidy datasets provide a standardized way to link the structure of a dataset (its physical layout) with its semantics (its meaning). In this section, I'll provide some standard vocabulary for describing the structure and semantics of a dataset, and then use those definitions to define tidy data.

Data structure

Most statistical datasets are data frames made up of **rows** and **columns**. The columns are almost always labeled and the rows are sometimes labeled. The following code provides some data about an imaginary classroom in a format commonly seen in the wild. The table has three columns and four rows, and both rows and columns are labeled.

```
classroom <- read.csv("classroom.csv", stringsAsFactors = FALSE)
classroom
#>     name quiz1 quiz2 test1
#> 1  Billy  <NA>     D     C
#> 2   Suzy     F  <NA>  <NA>
#> 3 Lionel     B     C     B
#> 4  Jenny     A     A     B
```

There are many ways to structure the same underlying data. The following table shows the same data as above, but the rows and columns have been transposed.

```
read.csv("classroom2.csv", stringsAsFactors = FALSE)
#>   assessment Billy  Suzy Lionel Jenny
#> 1      quiz1  <NA> FALSE      B     A
#> 2      quiz2     D    NA      C     A
#> 3      test1     C    NA      B     B
```

The data is the same, but the layout is different. Our vocabulary of rows and columns is simply not rich enough to describe why the two tables represent the same data. In addition to appearance, we need a way to describe the underlying semantics, or meaning, of the values displayed in the table.

Data semantics

A dataset is a collection of **values**, usually either numbers (if quantitative) or strings (if qualitative). Values are organised in two ways. Every value belongs to a **variable** and an **observation**. A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units. An observation contains all values measured on the same unit (like a person, or a day, or a race) across attributes.

A tidy version of the classroom data looks like this: (you'll learn how the functions work a little later)

```
library(tidyr)
library(dplyr)


classroom2 <- classroom %>%
  pivot_longer(quiz1:test1, names_to = "assessment", values_to = "grade") %>%
  arrange(name, assessment)
classroom2
#> # A tibble: 12 × 3
#>   name  assessment grade
#>   <chr> <chr>      <chr>
#> 1 Billy quiz1      <NA>
#> 2 Billy quiz2      D
#> 3 Billy test1      C
#> 4 Jenny quiz1      A
#> 5 Jenny quiz2      A
#> 6 Jenny test1      B
#> # … with 6 more rows
```

This makes the values, variables, and observations more clear. The dataset contains 36 values representing three variables and 12 observations. The variables are:

1. `name`, with four possible values (Billy, Suzy, Lionel, and Jenny).
2. `assessment`, with three possible values (quiz1, quiz2, and test1).
3. `grade`, with five or six values depending on how you think of the missing value (A, B, C, D, F, NA).

The tidy data frame explicitly tells us the definition of an observation. In this classroom, every combination of `name` and `assessment` is a single measured observation. The dataset also informs us of missing values, which can and do have meaning. Billy was absent for the first quiz, but tried to salvage his grade. Suzy failed the first quiz, so she decided to drop the class. To calculate Billy's final grade, we might replace this missing value with an F (or he might get a second chance to take the quiz). However, if we want to know the class average for Test 1, dropping Suzy's structural missing value would be more appropriate than imputing a new value.

For a given dataset, it's usually easy to figure out what are observations and what are variables, but it is surprisingly difficult to precisely define variables and observations in general. For example, if the columns in the classroom data were `height` and `weight` we would have been happy to call them variables. If the columns were `height` and `width`, it would be less clear cut, as we might think of height and width as values of a `dimension` variable. If the columns were `home phone` and `work phone`, we could treat these as two variables, but in a fraud detection environment we might want variables `phone number` and `number type` because the use of one phone number for multiple people might suggest fraud. A general rule of thumb is that it is easier to describe functional relationships between variables (e.g., `z` is a linear combination of `x` and `y`, `density` is the ratio of `weight` to `volume`) than between rows, and it is easier to make comparisons between groups of observations (e.g., average of group a vs. average of group b) than between groups of columns.

In a given analysis, there may be multiple levels of observation. For example, in a trial of new allergy medication we might have three observational types: demographic data collected from each person (`age`, `sex`, `race`), medical data collected from each person on each day (`number of sneezes`, `redness of eyes`), and meteorological data collected on each day (`temperature`, `pollen count`).

Variables may change over the course of analysis. Often the variables in the raw data are very fine grained, and may add extra modelling complexity for little explanatory gain. For example, many surveys ask variations on the same question to better get at an underlying trait. In early stages of analysis, variables correspond to questions. In later stages, you change focus to traits, computed by averaging together multiple questions. This considerably simplifies analysis because you don't need a hierarchical model, and you can often pretend that the data is continuous, not discrete.

Tidy data

Tidy data is a standard way of mapping the meaning of a dataset to its structure. A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types. In **tidy data**:

1. Every column is a variable.
2. Every row is an observation.
3. Every cell is a single value.

This is Codd's 3rd normal form, but with the constraints framed in statistical language, and the focus put on a single dataset rather than the many connected datasets common in relational databases. **Messy data** is any other arrangement of the data.

Tidy data makes it easy for an analyst or a computer to extract needed variables because it provides a standard way of structuring a dataset. Compare the different versions of the classroom data: in the messy version you need to use different strategies to extract different variables. This slows analysis and invites errors. If you consider how many data analysis operations involve all of the values in a variable (every aggregation function), you can see how important it is to extract these values in a simple, standard way. Tidy data is particularly well suited for vectorised programming languages like

R, because the layout ensures that values of different variables from the same observation are always paired.

While the order of variables and observations does not affect analysis, a good ordering makes it easier to scan the raw values. One way of organising variables is by their role in the analysis: are values fixed by the design of the data collection, or are they measured during the course of the experiment? Fixed variables describe the experimental design and are known in advance. Computer scientists often call fixed variables dimensions, and statisticians usually denote them with subscripts on random variables. Measured variables are what we actually measure in the study. Fixed variables should come first, followed by measured variables, each ordered so that related variables are contiguous. Rows can then be ordered by the first variable, breaking ties with the second and subsequent (fixed) variables. This is the convention adopted by all tabular displays in this paper.

Tidying messy datasets

Real datasets can, and often do, violate the three precepts of tidy data in almost every way imaginable. While occasionally you do get a dataset that you can start analysing immediately, this is the exception, not the rule. This section describes the five most common problems with messy datasets, along with their remedies:

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

Surprisingly, most messy datasets, including types of messiness not explicitly described above, can be tidied with a small set of tools: pivoting (longer and wider) and separating. The following sections illustrate each problem with a real dataset that I have encountered, and show how to tidy them.

Column headers are values, not variable names

A common type of messy dataset is tabular data designed for presentation, where variables form both the rows and columns, and column headers are values, not variable names. While I would call this arrangement messy, in some cases it can be extremely useful. It provides efficient storage for completely crossed designs, and it can lead to extremely efficient computation if desired operations can be expressed as matrix operations.

The following code shows a subset of a typical dataset of this form. This dataset explores the relationship between income and religion in the US. It comes from a report produced by the Pew Research Center, an American think-tank that collects data on attitudes to topics ranging from religion to the internet, and produces many reports that contain datasets in this format.

```
relig_income
#> # A tibble: 18 × 11
#>   religion  `<$10k` `$10-20k` `$20-30k` `$30-40k` `$40-50k` `$50-75k` `$75-100k`
#>   <chr>       <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>      <dbl>
#> 1 Agnostic       27        34        60        81        76       137        122
#> 2 Atheist        12        27        37        52        35        70         73
```

```
#> 3 Buddhist       27        21        30        34        33        58         62
#> 4 Catholic      418       617       732       670       638      1116        949
#> 5 Don't kn…      15        14        15        11        10        35         21
#> 6 Evangeli…     575       869      1064       982       881      1486        949
#> # … with 12 more rows, and 3 more variables: $100-150k <dbl>, >150k <dbl>,
#> #   Don't know/refused <dbl>
```

This dataset has three variables, `religion`, `income` and `frequency`. To tidy it, we need to **pivot** the non-variable columns into a two-column key-value pair. This action is often described as making a wide dataset longer (or taller).

When pivoting variables, we need to provide the name of the new key-value columns to create. After defining the colums to pivot (every column except for religion), you will need the name of the key column, which is the name of the variable defined by the values of the column headings. In this case, it's `income`. The second argument is the name of the value column, `frequency`.

```
relig_income %>%
  pivot_longer(-religion, names_to = "income", values_to = "frequency")
#> # A tibble: 180 × 3
#>   religion income  frequency
#>   <chr>    <chr>       <dbl>
#> 1 Agnostic <$10k          27
#> 2 Agnostic $10-20k        34
#> 3 Agnostic $20-30k        60
#> 4 Agnostic $30-40k        81
#> 5 Agnostic $40-50k        76
#> 6 Agnostic $50-75k       137
#> # … with 174 more rows
```

This form is tidy because each column represents a variable and each row represents an observation, in this case a demographic unit corresponding to a combination of `religion` and `income`.

This format is also used to record regularly spaced observations over time. For example, the Billboard dataset shown below records the date a song first entered the billboard top 100. It has variables for `artist`, `track`, `date.entered`, `rank` and `week`. The rank in each week after it enters the top 100 is recorded in 75 columns, `wk1` to `wk75`. This form of storage is not tidy, but it is useful for data entry. It reduces duplication since otherwise each song in each week would need its own row, and song metadata like title and artist would need to be repeated. This will be discussed in more depth in multiple types.

```
billboard
#> # A tibble: 317 × 79
#>   artist   track    date.entered   wk1   wk2   wk3   wk4   wk5   wk6   wk7   wk8
#>   <chr>    <chr>    <date>       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 2 Pac    Baby Do… 2000-02-26      87    82    72    77    87    94    99    NA
#> 2 2Ge+her  The Har… 2000-09-02      91    87    92    NA    NA    NA    NA    NA
#> 3 3 Doors… Krypton… 2000-04-08      81    70    68    67    66    57    54    53
#> 4 3 Doors… Loser    2000-10-21      76    76    72    69    67    65    55    59
#> 5 504 Boyz Wobble … 2000-04-15      57    34    25    17    17    31    36    49
#> 6 98^0     Give Me… 2000-08-19      51    39    34    26    26    19     2     2
#> # … with 311 more rows, and 68 more variables: wk9 <dbl>, wk10 <dbl>,
```

```
#> #   wk11 <dbl>, wk12 <dbl>, wk13 <dbl>, wk14 <dbl>, wk15 <dbl>, wk16 <dbl>,
#> #   wk17 <dbl>, wk18 <dbl>, wk19 <dbl>, wk20 <dbl>, wk21 <dbl>, wk22 <dbl>,
#> #   wk23 <dbl>, wk24 <dbl>, wk25 <dbl>, wk26 <dbl>, wk27 <dbl>, wk28 <dbl>,
#> #   wk29 <dbl>, wk30 <dbl>, wk31 <dbl>, wk32 <dbl>, wk33 <dbl>, wk34 <dbl>,
#> #   wk35 <dbl>, wk36 <dbl>, wk37 <dbl>, wk38 <dbl>, wk39 <dbl>, wk40 <dbl>,
#> #   wk41 <dbl>, wk42 <dbl>, wk43 <dbl>, wk44 <dbl>, wk45 <dbl>, wk46 <dbl>, …
```

To tidy this dataset, we first use `pivot_longer()` to make the dataset longer. We transform the columns from `wk1` to `wk76`, making a new column for their names, `week`, and a new value for their values, `rank`:

```
billboard2 <- billboard %>%
  pivot_longer(
    wk1:wk76,
    names_to = "week",
    values_to = "rank",
    values_drop_na = TRUE
  )
billboard2
#> # A tibble: 5,307 × 5
#>   artist track                   date.entered week   rank
#>   <chr>  <chr>                   <date>       <chr> <dbl>
#> 1 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk1      87
#> 2 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk2      82
#> 3 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk3      72
#> 4 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk4      77
#> 5 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk5      87
#> 6 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk6      94
#> # … with 5,301 more rows
```

Here we use `values_drop_na = TRUE` to drop any missing values from the rank column. In this data, missing values represent weeks that the song wasn't in the charts, so can be safely dropped.

In this case it's also nice to do a little cleaning, converting the week variable to a number, and figuring out the date corresponding to each week on the charts:

```
billboard3 <- billboard2 %>%
  mutate(
    week = as.integer(gsub("wk", "", week)),
    date = as.Date(date.entered) + 7 * (week - 1),
    date.entered = NULL
  )
billboard3
#> # A tibble: 5,307 × 5
#>   artist track                    week  rank date
#>   <chr>  <chr>                   <int> <dbl> <date>
#> 1 2 Pac  Baby Don't Cry (Keep...     1    87 2000-02-26
#> 2 2 Pac  Baby Don't Cry (Keep...     2    82 2000-03-04
#> 3 2 Pac  Baby Don't Cry (Keep...     3    72 2000-03-11
#> 4 2 Pac  Baby Don't Cry (Keep...     4    77 2000-03-18
#> 5 2 Pac  Baby Don't Cry (Keep...     5    87 2000-03-25
```

```
#> 6 2 Pac  Baby Don't Cry (Keep...     6    94 2000-04-01
#> # … with 5,301 more rows
```

Finally, it's always a good idea to sort the data. We could do it by artist, track and week:

```
billboard3 %>% arrange(artist, track, week)
#> # A tibble: 5,307 × 5
#>   artist track                   week  rank date
#>   <chr>  <chr>                  <int> <dbl> <date>
#> 1 2 Pac  Baby Don't Cry (Keep...     1    87 2000-02-26
#> 2 2 Pac  Baby Don't Cry (Keep...     2    82 2000-03-04
#> 3 2 Pac  Baby Don't Cry (Keep...     3    72 2000-03-11
#> 4 2 Pac  Baby Don't Cry (Keep...     4    77 2000-03-18
#> 5 2 Pac  Baby Don't Cry (Keep...     5    87 2000-03-25
#> 6 2 Pac  Baby Don't Cry (Keep...     6    94 2000-04-01
#> # … with 5,301 more rows
```

Or by date and rank:

```
billboard3 %>% arrange(date, rank)
#> # A tibble: 5,307 × 5
#>   artist   track   week  rank date
#>   <chr>    <chr>  <int> <dbl> <date>
#> 1 Lonestar Amazed     1    81 1999-06-05
#> 2 Lonestar Amazed     2    54 1999-06-12
#> 3 Lonestar Amazed     3    44 1999-06-19
#> 4 Lonestar Amazed     4    39 1999-06-26
#> 5 Lonestar Amazed     5    38 1999-07-03
#> 6 Lonestar Amazed     6    33 1999-07-10
#> # … with 5,301 more rows
```

Multiple variables stored in one column

After pivoting columns, the key column is sometimes a combination of multiple underlying variable names. This happens in the `tb` (tuberculosis) dataset, shown below. This dataset comes from the World Health Organisation, and records the counts of confirmed tuberculosis cases by `country`, `year`, and demographic group. The demographic groups are broken down by `sex` (m, f) and `age` (0-14, 15-25, 25-34, 35-44, 45-54, 55-64, unknown).

```
tb <- as_tibble(read.csv("tb.csv", stringsAsFactors = FALSE))
tb
#> # A tibble: 5,769 × 22
#>   iso2   year   m04  m514  m014 m1524 m2534 m3544 m4554 m5564   m65    mu   f04
#>   <chr> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int>
#> 1 AD     1989    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
#> 2 AD     1990    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
#> 3 AD     1991    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
```

```
#> 4 AD     1992    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
#> 5 AD     1993    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
#> 6 AD     1994    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
#> # … with 5,763 more rows, and 9 more variables: f514 <int>, f014 <int>,
#> #   f1524 <int>, f2534 <int>, f3544 <int>, f4554 <int>, f5564 <int>, f65 <int>,
#> #   fu <int>
```

First we use `pivot_longer()` to gather up the non-variable columns:

```
tb2 <- tb %>%
  pivot_longer(
    !c(iso2, year),
    names_to = "demo",
    values_to = "n",
    values_drop_na = TRUE
  )
tb2
#> # A tibble: 35,750 × 4
#>   iso2   year demo      n
#>   <chr> <int> <chr> <int>
#> 1 AD     1996 m014      0
#> 2 AD     1996 m1524     0
#> 3 AD     1996 m2534     0
#> 4 AD     1996 m3544     4
#> 5 AD     1996 m4554     1
#> 6 AD     1996 m5564     0
#> # … with 35,744 more rows
```

Column headers in this format are often separated by a non-alphanumeric character (e.g. ., -, _, :), or have a fixed width format, like in this dataset. `separate()` makes it easy to split a compound variables into individual variables. You can either pass it a regular expression to split on (the default is to split on non-alphanumeric columns), or a vector of character positions. In this case we want to split after the first character:

```
tb3 <- tb2 %>%
  separate(demo, c("sex", "age"), 1)
tb3
#> # A tibble: 35,750 × 5
#>   iso2   year sex   age       n
#>   <chr> <int> <chr> <chr> <int>
#> 1 AD     1996 m     014       0
#> 2 AD     1996 m     1524      0
#> 3 AD     1996 m     2534      0
#> 4 AD     1996 m     3544      4
#> 5 AD     1996 m     4554      1
#> 6 AD     1996 m     5564      0
#> # … with 35,744 more rows
```

Storing the values in this form resolves a problem in the original data. We want to compare rates, not

counts, which means we need to know the population. In the original format, there is no easy way to add a population variable. It has to be stored in a separate table, which makes it hard to correctly match populations to counts. In tidy form, adding variables for population and rate is easy because they're just additional columns.

In this case, we could also do the transformation in a single step by supplying multiple column names to `names_to` and also supplying a grouped regular expression to `names_pattern`:

```
tb %>% pivot_longer(
  !c(iso2, year),
  names_to = c("sex", "age"),
  names_pattern = "(.)(.+)",
  values_to = "n",
  values_drop_na = TRUE
)
#> # A tibble: 35,750 × 5
#>   iso2   year sex   age       n
#>   <chr> <int> <chr> <chr> <int>
#> 1 AD     1996 m     014       0
#> 2 AD     1996 m     1524      0
#> 3 AD     1996 m     2534      0
#> 4 AD     1996 m     3544      4
#> 5 AD     1996 m     4554      1
#> 6 AD     1996 m     5564      0
#> # … with 35,744 more rows
```

Variables are stored in both rows and columns

The most complicated form of messy data occurs when variables are stored in both rows and columns. The code below loads daily weather data from the Global Historical Climatology Network for one weather station (MX17004) in Mexico for five months in 2010.

```
weather <- as_tibble(read.csv("weather.csv", stringsAsFactors = FALSE))
weather
#> # A tibble: 22 × 35
#>   id       year month element    d1    d2    d3    d4    d5    d6    d7    d8
#>   <chr>   <int> <int> <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 MX17004  2010     1 tmax       NA    NA    NA    NA    NA    NA    NA    NA
#> 2 MX17004  2010     1 tmin       NA    NA    NA    NA    NA    NA    NA    NA
#> 3 MX17004  2010     2 tmax       NA  27.3  24.1    NA    NA    NA    NA    NA
#> 4 MX17004  2010     2 tmin       NA  14.4  14.4    NA    NA    NA    NA    NA
#> 5 MX17004  2010     3 tmax       NA    NA    NA    NA  32.1    NA    NA    NA
#> 6 MX17004  2010     3 tmin       NA    NA    NA    NA  14.2    NA    NA    NA
#> # … with 16 more rows, and 23 more variables: d9 <lgl>, d10 <dbl>, d11 <dbl>,
#> #   d12 <lgl>, d13 <dbl>, d14 <dbl>, d15 <dbl>, d16 <dbl>, d17 <dbl>,
#> #   d18 <lgl>, d19 <lgl>, d20 <lgl>, d21 <lgl>, d22 <lgl>, d23 <dbl>,
#> #   d24 <lgl>, d25 <dbl>, d26 <dbl>, d27 <dbl>, d28 <dbl>, d29 <dbl>,
#> #   d30 <dbl>, d31 <dbl>
```

It has variables in individual columns (`id`, `year`, `month`), spread across columns (`day`, d1-d31) and across rows (`tmin`, `tmax`) (minimum and maximum temperature). Months with fewer than 31 days have structural missing values for the last day(s) of the month.

To tidy this dataset we first use pivot_longer to gather the day columns:

```
weather2 <- weather %>%
  pivot_longer(
    d1:d31,
    names_to = "day",
    values_to = "value",
    values_drop_na = TRUE
  )
weather2
#> # A tibble: 66 × 6
#>   id       year month element day   value
#>   <chr>   <int> <int> <chr>   <chr> <dbl>
#> 1 MX17004  2010     1 tmax    d30    27.8
#> 2 MX17004  2010     1 tmin    d30    14.5
#> 3 MX17004  2010     2 tmax    d2     27.3
#> 4 MX17004  2010     2 tmax    d3     24.1
#> 5 MX17004  2010     2 tmax    d11    29.7
#> 6 MX17004  2010     2 tmax    d23    29.9
#> # … with 60 more rows
```

For presentation, I've dropped the missing values, making them implicit rather than explicit. This is ok because we know how many days are in each month and can easily reconstruct the explicit missing values.

We'll also do a little cleaning:

```
weather3 <- weather2 %>%
  mutate(day = as.integer(gsub("d", "", day))) %>%
  select(id, year, month, day, element, value)
weather3
#> # A tibble: 66 × 6
#>   id       year month   day element value
#>   <chr>   <int> <int> <int> <chr>   <dbl>
#> 1 MX17004  2010     1    30 tmax     27.8
#> 2 MX17004  2010     1    30 tmin     14.5
#> 3 MX17004  2010     2     2 tmax     27.3
#> 4 MX17004  2010     2     3 tmax     24.1
#> 5 MX17004  2010     2    11 tmax     29.7
#> 6 MX17004  2010     2    23 tmax     29.9
#> # … with 60 more rows
```

This dataset is mostly tidy, but the `element` column is not a variable; it stores the names of variables. (Not shown in this example are the other meteorological variables `prcp` (precipitation) and `snow` (snowfall)). Fixing this requires widening the data: `pivot_wider()` is inverse of `pivot_longer()`, pivoting `element` and `value` back out across multiple columns:

```
weather3 %>%
  pivot_wider(names_from = element, values_from = value)
#> # A tibble: 33 × 6
#>   id       year month   day  tmax  tmin
#>   <chr>   <int> <int> <int> <dbl> <dbl>
#> 1 MX17004  2010     1    30  27.8  14.5
#> 2 MX17004  2010     2     2  27.3  14.4
#> 3 MX17004  2010     2     3  24.1  14.4
#> 4 MX17004  2010     2    11  29.7  13.4
#> 5 MX17004  2010     2    23  29.9  10.7
#> 6 MX17004  2010     3     5  32.1  14.2
#> # … with 27 more rows
```

This form is tidy: there's one variable in each column, and each row represents one day.

Multiple types in one table

Datasets often involve values collected at multiple levels, on different types of observational units. During tidying, each type of observational unit should be stored in its own table. This is closely related to the idea of database normalisation, where each fact is expressed in only one place. It's important because otherwise inconsistencies can arise.

The billboard dataset actually contains observations on two types of observational units: the song and its rank in each week. This manifests itself through the duplication of facts about the song: `artist` is repeated many times.

This dataset needs to be broken down into two pieces: a song dataset which stores `artist` and `song name`, and a ranking dataset which gives the `rank` of the `song` in each `week`. We first extract a `song` dataset:

```
song <- billboard3 %>%
  distinct(artist, track) %>%
  mutate(song_id = row_number())
song
#> # A tibble: 317 × 3
#>   artist       track                   song_id
#>   <chr>        <chr>                     <int>
#> 1 2 Pac        Baby Don't Cry (Keep...       1
#> 2 2Ge+her      The Hardest Part Of ...       2
#> 3 3 Doors Down Kryptonite                   3
#> 4 3 Doors Down Loser                        4
#> 5 504 Boyz     Wobble Wobble                5
#> 6 98^0         Give Me Just One Nig...      6
#> # … with 311 more rows
```

Then use that to make a `rank` dataset by replacing repeated song facts with a pointer to song details (a unique song id):

```
rank <- billboard3 %>%
  left_join(song, c("artist", "track")) %>%
```

```
  select(song_id, date, week, rank)
rank
#> # A tibble: 5,307 × 4
#>   song_id date        week  rank
#>     <int> <date>     <int> <dbl>
#> 1       1 2000-02-26     1    87
#> 2       1 2000-03-04     2    82
#> 3       1 2000-03-11     3    72
#> 4       1 2000-03-18     4    77
#> 5       1 2000-03-25     5    87
#> 6       1 2000-04-01     6    94
#> # … with 5,301 more rows
```

You could also imagine a `week` dataset which would record background information about the week, maybe the total number of songs sold or similar "demographic" information.

Normalisation is useful for tidying and eliminating inconsistencies. However, there are few data analysis tools that work directly with relational data, so analysis usually also requires denormalisation or the merging the datasets back into one table.

One type in multiple tables

It's also common to find data values about a single type of observational unit spread out over multiple tables or files. These tables and files are often split up by another variable, so that each represents a single year, person, or location. As long as the format for individual records is consistent, this is an easy problem to fix:

1. Read the files into a list of tables.
2. For each table, add a new column that records the original file name (the file name is often the value of an important variable).
3. Combine all tables into a single table.

Purrr makes this straightforward in R. The following code generates a vector of file names in a directory (`data/`) which match a regular expression (ends in `.csv`). Next we name each element of the vector with the name of the file. We do this because will preserve the names in the following step, ensuring that each row in the final data frame is labeled with its source. Finally, `map_dfr()` loops over each path, reading in the csv file and combining the results into a single data frame.

```
library(purrr)
paths <- dir("data", pattern = "\\.csv$", full.names = TRUE)
names(paths) <- basename(paths)
map_dfr(paths, read.csv, stringsAsFactors = FALSE, .id = "filename")
```

Once you have a single table, you can perform additional tidying as needed. An example of this type of cleaning can be found at https://github.com/hadley/data-baby-names which takes 129 yearly baby name tables provided by the US Social Security Administration and combines them into a single file.

A more complicated situation occurs when the dataset structure changes over time. For example, the datasets may contain different variables, the same variables with different names, different file

formats, or different conventions for missing values. This may require you to tidy each file to individually (or, if you're lucky, in small groups) and then combine them once tidied. An example of this type of tidying is illustrated in https://github.com/hadley/data-fuel-economy, which shows the tidying of epa fuel economy data for over 50,000 cars from 1978 to 2008. The raw data is available online, but each year is stored in a separate file and there are four major formats with many minor variations, making tidying this dataset a considerable challenge.