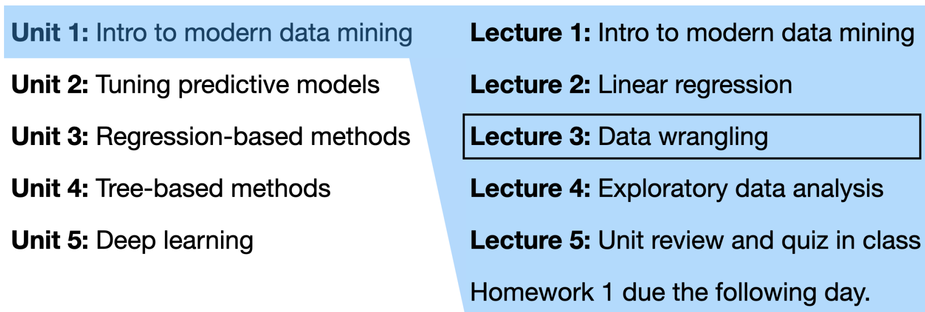
Unit 1 Lecture 3: Data Wrangling

September 7, 2021

Welcome back to STAT 471! We are now in Unit 1 Lecture 2:



This lecture is about *data wrangling*, "the art of getting your data into R in a useful form for visualization and modeling" (R4DS Chapter 9), drawing on Chapters 10, 11, 12, and 15 from the excellent R for Data Science book (direct quotations are presented using block quotes).

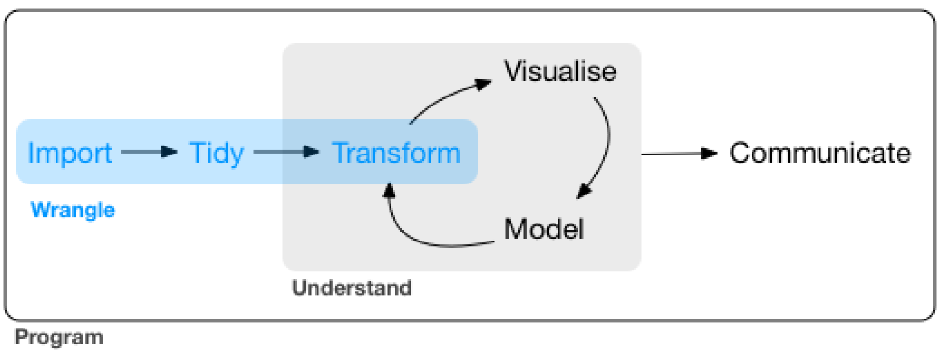


Figure 1: Image source: R4DS Chapter 9.

We will cover:

- Tibbles (a representation of our data matrix in R)
- Data import (getting the data into R)
- Data tidying (getting our data into a format amenable to analysis)

Let's load the `tidyverse`:

```
library(tidyverse)
```

1 Tibbles (R4DS Chapters 10 and 13)

A tibble is an upgraded version of R's data frame. When you read data into R using any `tidyverse` function it will be stored as a tibble. Here is an example of a tibble:

```
mpg
```

```
## # A tibble: 234 x 11
##    manufacturer model      displ  year   cyl trans drv     cty   hwy fl    class
##    <chr>        <chr>      <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
##  1 audi         a4           1.8  1999     4 auto~ f        18    29 p     comp~
##  2 audi         a4           1.8  1999     4 manu~ f        21    29 p     comp~
##  3 audi         a4           2    2008     4 manu~ f        20    31 p     comp~
##  4 audi         a4           2    2008     4 auto~ f        21    30 p     comp~
##  5 audi         a4           2.8  1999     6 auto~ f        16    26 p     comp~
##  6 audi         a4           2.8  1999     6 manu~ f        18    26 p     comp~
##  7 audi         a4           3.1  2008     6 auto~ f        18    27 p     comp~
##  8 audi         a4 quattro   1.8  1999     4 manu~ 4        18    26 p     comp~
##  9 audi         a4 quattro   1.8  1999     4 auto~ 4        16    25 p     comp~
## 10 audi         a4 quattro   2    2008     4 manu~ 4        20    28 p     comp~
## # ... with 224 more rows
```

1.1 Tibble data types

Each column has a *data type*. The most common data types you will find are `logical` (`lgl`), `integer` (`int`), `double` (`dbl`), `character` (`chr`), and `factor` (`fctr`).

The `factor` type is a special type for handling categorical variables. Let's pull out the `class` variable from the data frame to explore:

```
car_classes = mpg %>% pull(class) # pull out class variable
head(car_classes)                 # look at first entries
```

```
## [1] "compact" "compact" "compact" "compact" "compact" "compact"
```

```
table(car_classes)                # tabulate
```

```
## car_classes
##    2seater    compact    midsize    minivan     pickup subcompact        suv
##          5         47         41         11         33         35         62
```

Let's check out the type of `car_classes`:

```
typeof(car_classes)
```

```
## [1] "character"
```

We can convert a character vector to a factor vector as follows:

```
car_classes_factor = factor(car_classes)
head(car_classes_factor)
```

```
## [1] compact compact compact compact compact compact
## Levels: 2seater compact midsize minivan pickup subcompact suv
```

We see that a factor variable has *levels*: a list of the different categories. We can extract a factor's levels as follows:

```
levels(car_classes_factor)
```

```
## [1] "2seater"    "compact"    "midsize"    "minivan"    "pickup"
## [6] "subcompact" "suv"
```

We may want to change one or more of the levels names using `fct_recode`:

```
car_classes_recoded =
  fct_recode(car_classes_factor,
             "two-seater"  = "2seater",
             "SUV"         = "suv")
levels(car_classes_recoded)
```

```
## [1] "two-seater" "compact"    "midsize"    "minivan"    "pickup"
## [6] "subcompact" "SUV"
```

or change the order of the factor levels:

```
car_classes_reordered =
  factor(car_classes_factor,
         levels = c("suv",
                    "subcompact",
                    "pickup",
                    "minivan",
                    "midsize",
                    "compact",
                    "2seater"))
levels(car_classes_reordered)
```

```
## [1] "suv"        "subcompact" "pickup"     "minivan"    "midsize"
## [6] "compact"    "2seater"
```

1.2 Creating tibbles

Sometimes you'll need to create a tibble yourself. You can do so easily using the `tibble` command:

```
tibble(x = rnorm(5),
       y = 1,
       z = x^2 + y)
```

```
## # A tibble: 5 x 3
##         x     y     z
##     <dbl> <dbl> <dbl>
## 1  1.10       1  2.22
## 2  0.0618     1  1.00
## 3 -0.213      1  1.05
## 4  0.282      1  1.08
## 5 -0.418      1  1.17
```

It's possible for a tibble to have column names that are not valid R variable names, aka *non-syntactic* names. For example, they might not start with a letter. To refer to these variables, you need to surround them with backticks:

```
tb <- tibble(
  `:)` = "smile",
  `2000` = "number"
)
tb
```

```
## # A tibble: 1 x 2
##   `:)`  `2000`
##   <chr> <chr>
## 1 smile number
```

Another way to create a tibble is with `tribble()`, short for transposed tibble. `tribble()` is customised for data entry in code: column headings are defined by formulas (i.e. they start with `~`), and entries are separated by commas. This makes it possible to lay out small amounts of data in easy to read form.

```
tribble(
  ~x, ~y, ~z,
  #--/--/----
  "a", 2, 3.6,
  "b", 1, 8.5
)
```

```
## # A tibble: 2 x 3
##   x         y     z
##   <chr> <dbl> <dbl>
## 1 a         2   3.6
## 2 b         1   8.5
```

I often add a comment (the line starting with `#`), to make it really clear where the header is.

2 Data import (R4DS Chapter 11)

Data come in several different formats, e.g. comma-separated values (csv), tab-separated values (tsv), or Excel files. To read files in csv or tsv formats, use `read_csv`and `read_tsv`, respectively. These are both part of the `readr` package, which is part of the `tidyverse`. These functions are very similar to each other. To read Excel files, use the `read_excel` function from the `readxl` package.

Let's see how `read_csv` works. The simplest way of calling it is to specify just one argument (the location of the file you'd like to read):

```
heights = read_csv("../../data/heights.csv")
```

```
## Rows: 1192 Columns: 6
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): sex, race
## dbl (4): earn, height, ed, age
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
heights
```

```
## # A tibble: 1,192 x 6
##     earn height sex       ed   age race
##    <dbl>  <dbl> <chr>  <dbl> <dbl> <chr>
##  1 50000   74.4 male      16    45 white
##  2 60000   65.5 female    16    58 white
##  3 30000   63.6 female    16    29 white
##  4 50000   63.1 female    16    91 other
##  5 51000   63.4 female    17    39 white
##  6  9000   64.4 female    15    26 white
##  7 29000   61.7 female    12    49 white
##  8 32000   72.7 male      17    46 white
##  9  2000   72.0 male      15    21 hispanic
## 10 27000   72.2 male      12    26 white
## # ... with 1,182 more rows
```

The assumption `read_csv` made is that the first line of the file are the column names. If column names are absent, make sure to specify this via `col_names = FALSE`. Note that `read_csv` has guessed the column types. Sometimes, it guesses wrong. Perhaps `sex` and `race` should be factors, and education and age should be integers. If there aren't too many columns, it's good practice to just specify the column types in the initial call to `read_csv` via the `col_types` argument:

```
heights = read_csv("../../data/heights.csv",
                   col_types = "ddfiif")
heights
```

```
## # A tibble: 1,192 x 6
##     earn height sex       ed   age race
##    <dbl>  <dbl> <fct>  <int> <int> <fct>
##  1 50000   74.4 male      16    45 white
##  2 60000   65.5 female    16    58 white
##  3 30000   63.6 female    16    29 white
##  4 50000   63.1 female    16    91 other
##  5 51000   63.4 female    17    39 white
##  6  9000   64.4 female    15    26 white
##  7 29000   61.7 female    12    49 white
##  8 32000   72.7 male      17    46 white
##  9  2000   72.0 male      15    21 hispanic
## 10 27000   72.2 male      12    26 white
## # ... with 1,182 more rows
```

Sometimes the files you'd like to read contain headers, i.e. one or more lines of metadata before the actual data starts. In this case, you can either skip a fixed number of lines (e.g. the first three) via `skip = 3` or skip any lines starting with a certain character (e.g. `#`) via `comment = "#"`.

3 Tidy data (R4DS Chapter 12)

> "Happy families are all alike; every unhappy family is unhappy in its own way." —- Leo Tolstoy
>
> "Tidy datasets are all alike, but every messy dataset is messy in its own way." —- Hadley Wickham
>
> In this chapter, you will learn a consistent way to organise your data in R, an organisation called tidy data. Getting your data into this format requires some upfront work, but that work pays off in the long term. Once you have tidy data and the tidy tools provided by packages in the tidyverse, you will spend much less time munging data from one representation to another, allowing you to spend more time on the analytic questions at hand.

3.1 Tidy data

There are multiple ways to represent the same data:

```
table1
```

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

```r
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

```r
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

```r
table4a
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

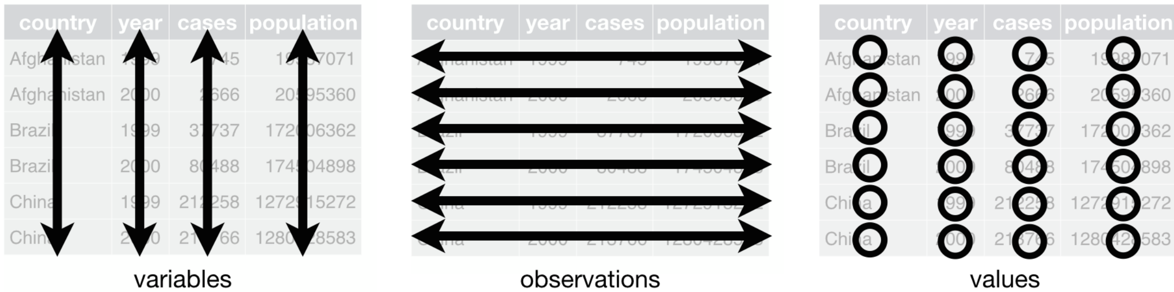
```r
table4b
```

```
## # A tibble: 3 x 3
##   country         `1999`     `2000`
## * <chr>            <int>      <int>
## 1 Afghanistan   19987071   20595360
## 2 Brazil       172006362  174504898
## 3 China       1272915272 1280428583
```

These are all representations of the same underlying data, but they are not equally easy to use. One dataset, the tidy dataset (`table1`), will be much easier to work with inside the tidyverse.

There are three interrelated rules which make a dataset tidy:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell. The figure below shows the rules visually.



All the packages in the `tidyverse` are designed to work with tidy data. The `tidyr` package is designed to get non-tidy data into tidy format.

3.1.1 Exercise

Using prose, describe how the variables and observations are organised in each of the sample tables.

3.2 Pivoting

Once you get a non-tidy dataset, the first step is to figure out what the variables and observations are. Then, you want to get the variables into columns and get observations into rows.

- If one variable is spread across multiple columns, you'll need to `pivot_longer`.
- If one observation is scattered across multiple rows, you'll need to `pivot_wider`.

3.2.1 Longer

A common problem is a dataset where some of the column names are not names of variables, but *values* of a variable. Take `table4a`: the column names `1999` and `2000` represent values of the year variable, the values in the `1999` and `2000` columns represent values of the cases variable, and each row represents two observations, not one.

```
table4a
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

To tidy a dataset like this, we need to **pivot** the offending columns into a new pair of variables. To describe that operation we need three parameters:

- The set of columns whose names are values, not variables. In this example, those are the columns `1999` and `2000`.
- The name of the variable to move the column names to. Here it is `year`.
- The name of the variable to move the column values to. Here it's `cases`.

Together those parameters generate the call to `pivot_longer()`:

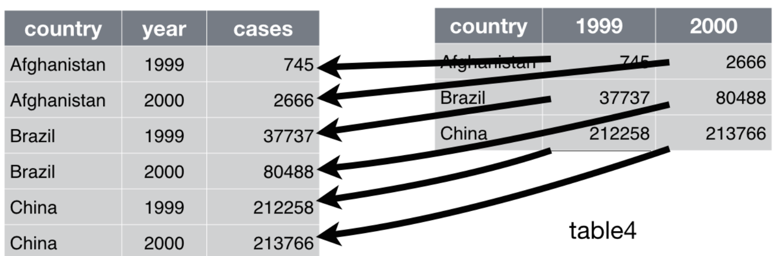
```
table4a %>%
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "cases")
```

```
## # A tibble: 6 x 3
##   country     year   cases
```

```
##   <chr>       <chr>  <int>
## 1 Afghanistan 1999     745
## 2 Afghanistan 2000    2666
## 3 Brazil      1999   37737
## 4 Brazil      2000   80488
## 5 China       1999  212258
## 6 China       2000  213766
```

Note that "1999" and "2000" are non-syntactic names (because they don't start with a letter) so we have to surround them in backticks.

In the final result, the pivoted columns are dropped, and we get new year and cases columns. Otherwise, the relationships between the original variables are preserved. Visually, this is shown in the figure below.



We can use `pivot_longer()` to tidy `table4b` in a similar fashion. The only difference is the variable stored in the cell values:

```
table4b %>%
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "population")
```

```
## # A tibble: 6 x 3
##   country     year  population
##   <chr>       <chr>      <int>
## 1 Afghanistan 1999    19987071
## 2 Afghanistan 2000    20595360
## 3 Brazil      1999   172006362
## 4 Brazil      2000   174504898
## 5 China       1999  1272915272
## 6 China       2000  1280428583
```

To combine the tidied versions of `table4a` and `table4b` into a single tibble, we need to use `dplyr::left_join()`:

```
tidy4a <- table4a %>%
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "cases")
tidy4b <- table4b %>%
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "population")
left_join(tidy4a, tidy4b, by = c("country", "year"))
```

```
## # A tibble: 6 x 4
##   country     year   cases population
##   <chr>       <chr>  <int>      <int>
## 1 Afghanistan 1999     745   19987071
```

```
## 2 Afghanistan 2000    2666   20595360
## 3 Brazil      1999   37737  172006362
## 4 Brazil      2000   80488  174504898
## 5 China       1999  212258 1272915272
## 6 China       2000  213766 1280428583
```

Read more about combining multiple datasets in Chapter 13 of R4DS.

3.2.2 Wider

> `pivot_wider()` is the opposite of `pivot_longer()`. You use it when an observation is scattered across multiple rows. For example, take `table2`: an observation is a country in a year, but each observation is spread across two rows.

```
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

> To tidy this up, we first analyse the representation in similar way to `pivot_longer()`. This time, however, we only need two parameters:
>
> - The column to take variable names from. Here, it's `type`.
> - The column to take values from. Here it's `count`.

Once we've figured that out, we can use `pivot_wider()`.

```
table2 %>%
    pivot_wider(names_from = type, values_from = count)
```

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

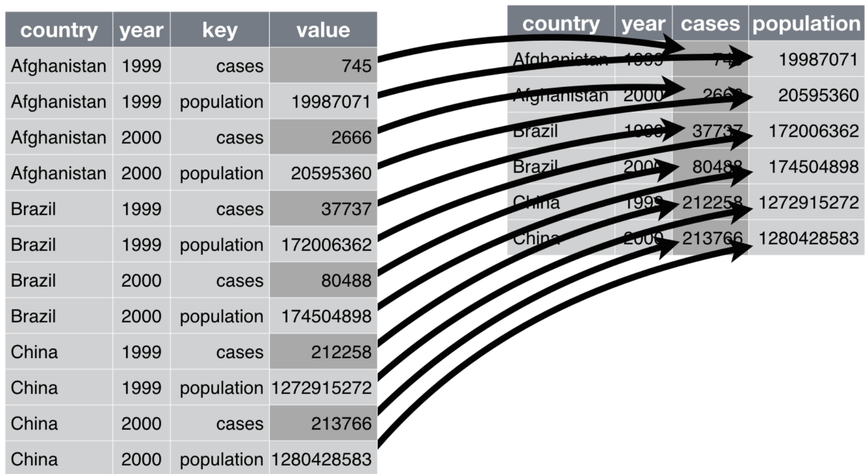


table2

3.2.3 Exercises

1. Why does this code fail?

```
table4a %>%
  pivot_longer(c(1999, 2000), names_to = "year", values_to = "cases")
# Error: Can't subset columns that don't exist.
# Locations 1999 and 2000 don't exist.
# There are only 3 columns.
```

2. What would happen if you widen this table? Why? How could you add a new column to uniquely identify each value?

```
tribble(
  ~name,             ~names,  ~values,
  #-----------------|--------|------
  "Phillip Woods",   "age",       45,
  "Phillip Woods",   "height",   186,
  "Phillip Woods",   "age",       50,
  "Jessica Cordero", "age",       37,
  "Jessica Cordero", "height",   156
)
```

```
## # A tibble: 5 x 3
##   name            names  values
##   <chr>           <chr>   <dbl>
## 1 Phillip Woods   age        45
## 2 Phillip Woods   height    186
## 3 Phillip Woods   age        50
## 4 Jessica Cordero age        37
## 5 Jessica Cordero height    156
```

3. Tidy the simple tibble below. Do you need to make it wider or longer? What are the variables?

```
tribble(
  ~pregnant, ~male, ~female,
  "yes",     NA,    10,
```

```
  "no",       20,    12
)
```

```
## # A tibble: 2 x 3
##   pregnant  male female
##   <chr>    <dbl>  <dbl>
## 1 yes         NA     10
## 2 no          20     12
```

3.3 Separating

So far you've learned how to tidy `table2` and `table4`, but not `table3`. `table3` has a different problem: we have one column (`rate`) that contains two variables (`cases` and `population`). To fix this problem, we'll need the `separate()` function.

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

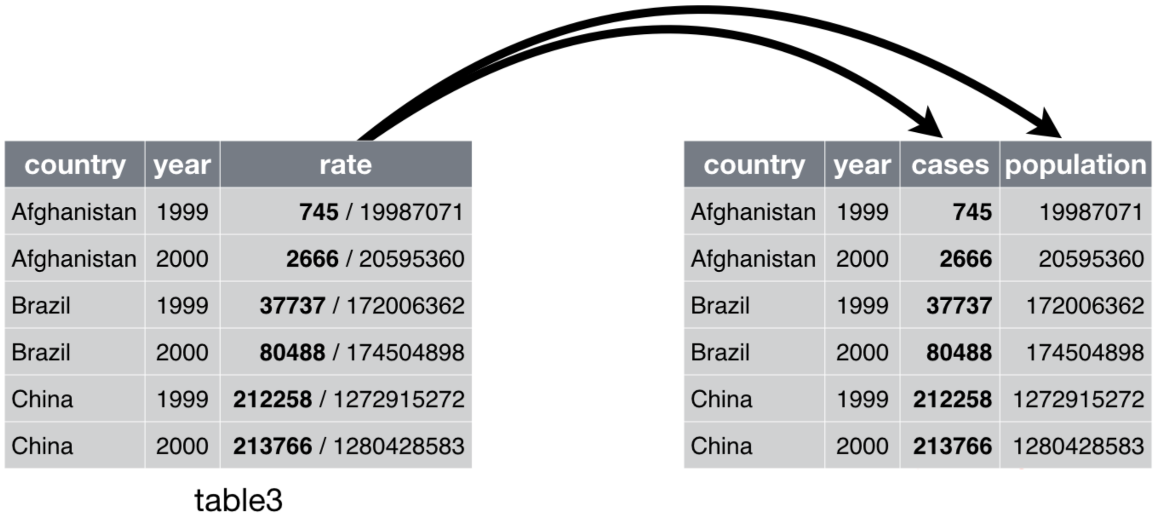
```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

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, as shown below.

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

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



By default, `separate()` will split values wherever it sees a non-alphanumeric character (i.e. a character that isn't a number or letter). For example, in the code above, `separate()` split the values of rate at the forward slash characters. If you wish to use a specific character to separate a column, you can pass the character to the sep argument of `separate()`. For example, we could rewrite the code above as:

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

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

Look carefully at the column types: you'll notice that `cases` and `population` are character columns. This is the default behaviour in `separate()`: it leaves the type of the column as is. Here, however, it's not very useful as those really are numbers. We can ask `separate()` to try and convert to better types using `convert = TRUE`:

```
table3 %>%
  separate(rate, into = c("cases", "population"), convert = TRUE)
```

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

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. When using integers to separate strings, the length of `sep` should be one less than the number of names in into.

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.

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

```
## # A tibble: 6 x 4
##   country     century year  rate
##   <chr>       <chr>   <chr> <chr>
## 1 Afghanistan 19      99    745/19987071
## 2 Afghanistan 20      00    2666/20595360
## 3 Brazil      19      99    37737/172006362
## 4 Brazil      20      00    80488/174504898
## 5 China       19      99    212258/1272915272
## 6 China       20      00    213766/1280428583
```

3.4 Missing values

Changing the representation of a dataset brings up an important subtlety of missing values. Surprisingly, 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)
)
```

There are two missing values in this dataset:

- The return for the fourth quarter of 2015 is explicitly missing, because the cell where its value should be instead contains `NA`.
- The return for the first quarter of 2016 is implicitly missing, because it simply does not appear in the dataset.

One way to think about the difference is with this Zen-like koan: An explicit missing value is the presence of an absence; an implicit missing value is the absence of a presence.

The way that a dataset is represented can make implicit values explicit. For example, we can make the implicit missing value explicit by putting years in the columns:

```
stocks %>%
  pivot_wider(names_from = year, values_from = return)
```

```
## # A tibble: 4 x 3
##     qtr `2015` `2016`
##   <dbl>  <dbl>  <dbl>
## 1     1   1.88  NA
## 2     2   0.59   0.92
## 3     3   0.35   0.17
## 4     4  NA      2.66
```

Because these explicit missing values may not be important in other representations of the data, you can set `values_drop_na = TRUE` in `pivot_longer()` to turn explicit missing values implicit:

3.5 Case study

Let's pull together everything you've learned to tackle a realistic data tidying problem. The tidyr::who dataset contains tuberculosis (TB) cases broken down by year, country, age, gender, and diagnosis method. The data comes from the 2014 World Health Organization Global Tuberculosis Report, available at http://www.who.int/tb/country/data/download/en/.

There's a wealth of epidemiological information in this dataset, but it's challenging to work with the data in the form that it's provided:

```
who
```

```
## # A tibble: 7,240 x 60
##    country     iso2  iso3   year new_sp_m014 new_sp_m1524 new_sp_m2534 new_sp_m3544
##    <chr>       <chr> <chr> <int>       <int>        <int>        <int>        <int>
##  1 Afghanistan AF    AFG    1980          NA           NA           NA           NA
##  2 Afghanistan AF    AFG    1981          NA           NA           NA           NA
##  3 Afghanistan AF    AFG    1982          NA           NA           NA           NA
##  4 Afghanistan AF    AFG    1983          NA           NA           NA           NA
##  5 Afghanistan AF    AFG    1984          NA           NA           NA           NA
##  6 Afghanistan AF    AFG    1985          NA           NA           NA           NA
##  7 Afghanistan AF    AFG    1986          NA           NA           NA           NA
##  8 Afghanistan AF    AFG    1987          NA           NA           NA           NA
##  9 Afghanistan AF    AFG    1988          NA           NA           NA           NA
## 10 Afghanistan AF    AFG    1989          NA           NA           NA           NA
## # ... with 7,230 more rows, and 52 more variables: new_sp_m4554 <int>,
## #   new_sp_m5564 <int>, new_sp_m65 <int>, new_sp_f014 <int>,
## #   new_sp_f1524 <int>, new_sp_f2534 <int>, new_sp_f3544 <int>,
## #   new_sp_f4554 <int>, new_sp_f5564 <int>, new_sp_f65 <int>,
## #   new_sn_m014 <int>, new_sn_m1524 <int>, new_sn_m2534 <int>,
## #   new_sn_m3544 <int>, new_sn_m4554 <int>, new_sn_m5564 <int>,
## #   new_sn_m65 <int>, new_sn_f014 <int>, new_sn_f1524 <int>, ...
```

Let's work together to try to tidy it!!!