Introduction to the tidyverse: tidying data with `tidyr`

Christopher Skovron
Northwestern University

July 16, 2018

So what's a tibble, anyway?

Tibbles *are* data frames, but they tweak some older behaviours to make life a little easier.

To learn more, check out `vignette("tibble")`.

```r
library(tidyverse)
```

```
## Warning: package 'tibble' was built under R version 3.4.3
```

```
## Warning: package 'tidyr' was built under R version 3.4.3
```

```
## Warning: package 'stringr' was built under R version 3.4.3
```

Creating tibbles

Almost all of the functions in tidyverse produce tibbles, as tibbles are one of the unifying features of the tidyverse. Most other R packages use regular data frames, so you might want to coerce a data frame to a tibble. You can do that with `as_tibble()`:

```
as_tibble(iris)
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1         5.10        3.50         1.40       0.200 setosa
##  2         4.90        3.00         1.40       0.200 setosa
##  3         4.70        3.20         1.30       0.200 setosa
##  4         4.60        3.10         1.50       0.200 setosa
##  5         5.00        3.60         1.40       0.200 setosa
##  6         5.40        3.90         1.70       0.400 setosa
##  7         4.60        3.40         1.40       0.300 setosa
##  8         5.00        3.40         1.50       0.200 setosa
##  9         4.40        2.90         1.40       0.200 setosa
## 10         4.90        3.10         1.50       0.100 setosa
```

```
## # ... with 140 more rows
```

Creating tibbles

You can create a new tibble from individual vectors with `tibble()`. `tibble()` will automatically recycle inputs of length 1, and allows you to refer to variables that you just created, as shown below.

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

```
## # A tibble: 5 x 3
##       x     y     z
##   <int> <dbl> <dbl>
## 1     1    1.    2.
## 2     2    1.    5.
## 3     3    1.   10.
## 4     4    1.   17.
## 5     5    1.   26.
```

Differences from `data.frame()`

If you're already familiar with `data.frame()`, note that `tibble()` does much less: it never changes the type of the inputs (e.g. it never converts strings to factors!), it never changes the names of variables, and it never creates row names.

tibble column names

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, or they might contain unusual characters like a space. To refer to these variables, you need to surround them with backticks, `:

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

```
## # A tibble: 1 x 3
##    `:)`  ` `   `2000`
##    <chr> <chr> <chr>
## 1 smile space number
```

You'll also need the backticks when working with these variables in other packages, like ggplot2, dplyr, and tidyr.

variables in other packages, like ggplot2, dplyr, and tidyr.

Tibbles vs. data.frame

There are two main differences in the usage of a tibble vs. a classic `data.frame`: printing and subsetting.

Printing

Tibbles have a refined print method that shows only the first 10 rows, and all the columns that fit on screen. This makes it much easier to work with large data. In addition to its name, each column reports its type, a nice feature borrowed from `str()`:

```
tibble(
  a = lubridate::now() + runif(1e3) * 86400,
  b = lubridate::today() + runif(1e3) * 30,
  c = 1:1e3,
  d = runif(1e3),
  e = sample(letters, 1e3, replace = TRUE)
)
```

```
## # A tibble: 1,000 x 5
##    a                   b              c     d e
##    <dttm>              <date>     <int> <dbl> <chr>
##  1 2018-07-16 10:42:26 2018-07-28     1 0.842 u
##  2 2018-07-16 10:01:57 2018-08-07     2 0.447 i
##  3 2018-07-17 00:04:13 2018-07-16     3 0.375 h
##  4 2018-07-16 22:21:20 2018-08-13     4 0.902 w
```

```
##   5 2018-07-16 23:04:16 2018-07-21     5 0.887  y
##   6 2018-07-16 19:36:04 2018-08-01     6 0.952  s
##   7 2018-07-16 21:25:24 2018-08-05     7 0.869  i
##   8 2018-07-17 02:51:52 2018-07-21     8 0.857  b
##   9 2018-07-16 08:05:39 2018-07-25     9 0.482  p
##  10 2018-07-16 10:35:34 2018-07-21    10 0.0322 g
## # ... with 990 more rows
```

Tibbles are designed so that you don't accidentally overwhelm your console when you print large data frames. But sometimes you need more output than the default display. There are a few options that can help.

First, you can explicitly `print()` the data frame and control the number of rows (`n`) and the `width` of the display. `width = Inf` will display all columns:

```
nycflights13::flights %>%
  print(n = 10, width = Inf)
```

Data import with readr

If you're used to base R, you've probably used functions from the `foreign` package to read in `.dta` Stata files, `.csv` files, and other data formats. `foreign` isn't really that bad, but `readr` plays nicely with the tidyverse, is a little more flexible, and can be a lot faster.

Getting started

Most of readr's functions are concerned with turning flat files into data frames:

- `read_csv()` reads comma delimited files, `read_csv2()` reads semicolon separated files (common in countries where `,` is used as the decimal place), `read_tsv()` reads tab delimited files, and `read_delim()` reads in files with any delimiter.

- `read_fwf()` reads fixed width files. You can specify fields either by their widths with `fwf_widths()` or their position with `fwf_positions()`.`read_table()` reads a common variation of fixed width files where columns are separated by white space.

Arguments

The first argument to `read_csv()` is the most important: it's the path to the file to read.

```
heights <- read_csv("data/heights.csv")
```

When you run `read_csv()` it prints out a column specification that gives the name and type of each column.

First line defaults to colnames, but you can change that

1. Sometimes there are a few lines of metadata at the top of the file. You can use `skip = n` to skip the first `n` lines; or use `comment = "#"` to drop all lines that start with (e.g.) #.

```
read_csv("The first line of metadata
  The second line of metadata
  x,y,z
  1,2,3", skip = 2)
```

```
## # A tibble: 1 x 3
##       x     y     z
##   <int> <int> <int>
## 1     1     2     3
```

```
read_csv("# A comment I want to skip
  x,y,z
```

```r
  1,2,3", comment = "#")
```

```
## # A tibble: 1 x 3
##         x     y     z
##     <int> <int> <int>
## 1       1     2     3
```

First line defaults to colnames, but you can change that

1. The data might not have column names. You can use `col_names = FALSE` to tell `read_csv()` not to treat the first row as headings, and instead label them sequentially from `X1` to `Xn`:

```
read_csv("1,2,3\n4,5,6", col_names = FALSE)
```

```
## # A tibble: 2 x 3
##      X1    X2    X3
##   <int> <int> <int>
## 1     1     2     3
## 2     4     5     6
```

First line defaults to colnames, but you can change that

````
Alternatively you can pass `col_names` a character vector which will be
used as the column names:



```r
read_csv("1,2,3\n4,5,6", col_names = c("x", "y", "z"))
```


```
## # A tibble: 2 x 3
##       x     y     z
##   <int> <int> <int>
## 1     1     2     3
## 2     4     5     6
```
````

You might need to fix NAs.

Another option that commonly needs tweaking is `na`: this specifies the value (or values) that are used to represent missing values in your file:

```
read_csv("a,b,c\n1,2,.", na = ".")
```

```
## # A tibble: 1 x 3
##       a     b c
##   <int> <int> <chr>
## 1     1     2 <NA>
```

This is all you need to know to read ~75% of CSV files that you'll encounter in practice. You can also easily adapt what you've learned to read tab separated files with `read_tsv()` and fixed width files with `read_fwf()`. To read in more challenging files, you'll need to learn more about how readr parses each column, turning them into R vectors.

Parsing

readr has lots of specialized tools for parsing different kinds of data at import. If you're already working with nicely cleaned data, you won't need to learn all of these. If you often work with data in non-standard formats, from Arabic-speaking countries, or from the distant past, check out the relevant sections of R4DS.

Parsing decimals across cultures

```
parse_double("1.23")
```

```
## [1] 1.23
```

```
parse_double("1,23", locale = locale(decimal_mark = ","))
```

```
## [1] 1.23
```

Clean junk out of your numbers

`parse_number()` addresses the second problem: it ignores non-numeric characters before and after the number. This is particularly useful for currencies and percentages, but also works to extract numbers embedded in text.

```
parse_number("$100")
```

```
## [1] 100
```

```
parse_number("20%")
```

```
## [1] 20
```

```
parse_number("It cost $123.45")
```

```
## [1] 123.45
```

God, currency data is awful

The final problem is addressed by the combination of `parse_number()` and the locale as `parse_number()` will ignore the “grouping mark”:

```r
# Used in America
parse_number("$123,456,789")
```

```
## [1] 123456789
```

```r
# Used in many parts of Europe
parse_number("123.456.789", locale = locale(grouping_mark = "."))
```

```
## [1] 123456789
```

```r
# Used in Switzerland
parse_number("123'456'789", locale = locale(grouping_mark = "'"))
```

```
## [1] 123456789
```

Study up on your own on character encoding

Factors

R uses factors to represent categorical variables that have a known set of possible values. Give `parse_factor()` a vector of known `levels` to generate a warning whenever an unexpected value is present:

```
fruit <- c("apple", "banana")
parse_factor(c("apple", "banana", "bananana"), levels = fruit)
```

```
## Warning: 1 parsing failure.
## row # A tibble: 1 x 4 col     row   col expected               actual    ex
```

```
## [1] apple  banana <NA>
## attr(,"problems")
## # A tibble: 1 x 4
##     row   col expected           actual
##   <int> <int> <chr>              <chr>
## 1     3    NA value in level set bananana
## Levels: apple banana
```

Dates, date-times, and times

You pick between three parsers depending on whether you want a date (the number of days since 1970-01-01), a date-time (the number of seconds since midnight 1970-01-01), or a time (the number of seconds since midnight). When called without any additional arguments:

- `parse_datetime()` expects an ISO8601 date-time. ISO8601 is an international standard in which the components of a date are organised from biggest to smallest: year, month, day, hour, minute, second.

```
parse_datetime("2010-10-01T2010")
```

```
## [1] "2010-10-01 20:10:00 UTC"
```

```
# If time is omitted, it will be set to midnight
parse_datetime("20101010")
```

```
## [1] "2010-10-10 UTC"
```

Dates, date-times, and times

- `parse_date()` expects a four digit year, a – or /, the month, a – or /, then the day:

```
parse_date("2010-10-01")
```

```
## [1] "2010-10-01"
```

Dates, date-times, and times

- `parse_time()` expects the hour, `:`, minutes, optionally `:` and seconds, and an optional am/pm specifier:

```
library(hms)
```

```
## Warning: package 'hms' was built under R version 3.4.3
```

```
parse_time("01:10 am")
```

```
## 01:10:00
```

```
parse_time("20:10:01")
```

```
## 20:10:01
```

Dates, date-times, and times

```
Base R doesn't have a great built in class for time data, so we use
the one provided in the hms package.
```

Other types of data

- **haven** reads SPSS, Stata, and SAS files.
- **readxl** reads excel files (both `.xls` and `.xlsx`).
- **DBI**, along with a database specific backend (e.g. **RMySQL**, **RSQLite**, **RPostgreSQL** etc) allows you to run SQL queries against a database and return a data frame.

For hierarchical data: use **jsonlite** (by Jeroen Ooms) for json, and **xml2** for XML. Jenny Bryan has some excellent worked examples at https://jennybc.github.io/purrr-tutorial/.

For other file types, try the R data import/export manual and the **rio** package.

Tidy data

Hadley has jokes

“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

Tidy data

You can represent the same underlying data in multiple ways. The example below shows the same data organised in four different ways. Each dataset shows the same values of four variables *country*, *year*, *population*, and *cases*, but each dataset organises the values in a different way.

Table 1

```
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
```

Table 2

```
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
```

Table 3

```
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
```

Table 4

```
# Spread across two tibbles
table4a  # cases
```

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

```
table4b  # population
```

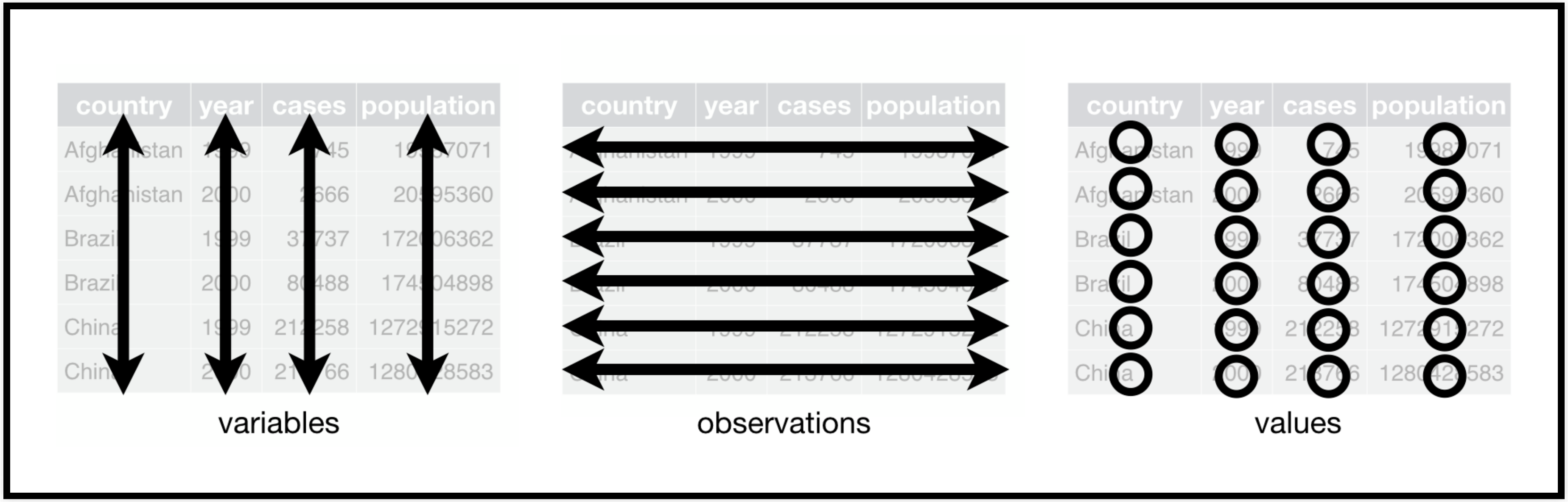
```
## # A tibble: 3 x 3
##   country         `1999`     `2000`
## * <chr>            <int>      <int>
## 1 Afghanistan   19987071   20595360
## 2 Brazil       172006362  174504898
## 3 China       1272915272 1280428583
```

Tidy data principles

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.

Tidy data principles



Tidy data principles

These three rules are interrelated because it’s impossible to only satisfy two of the three. That interrelationship leads to an even simpler set of practical instructions:

1. Put each dataset in a tibble.
2. Put each variable in a column.

Tidy data principles

- Every data task will be different, but tidy data principles will help you keep organized
- Much of your workflow will be getting data inputs into a tidy format
- From the beginning, lay out what you need your dataset to look like to do the analyses you want, then work backward from there
- Hadley's vignette on tidy data principles
- Chapter of R for Data Science on tidy data

Tidy data has

- Each variable forms a column.
- Each observation forms a row.
- Each type of observational unit forms a table.

Tidy data does not have

- 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.

Long vs wide data

dcast formula

```
dcast(aql, month + day ~ variable, value.var = "value")
```

ID variables (left side of formula)	Variable to swing into column names (right side of formula)	Values (value.var)

Long-format data

month	day	variable	value
5	1	ozone	41
5	2	ozone	36
5	3	ozone	12
5	4	ozone	18
5	5	ozone	NA
5	6	ozone	28

Wide-format data

month	day	ozone	solar.r	wind	temp
5	1	41	190	7.4	67
5	2	36	118	8.0	72
5	3	12	149	12.6	74
5	4	18	313	11.5	62
5	5	NA	NA	14.3	56
5	6	28	NA	14.9	66

Relational data

- Some data has structures more complex than simple tables
- For example, Netflix has a database where each user has a table of movies they've watched and a separate table for each movie of the users who have watched it
- This is "relational data"
- It's often, but not always, big
- Requires special tools, usually SQL

Checking up on your data cleaning

- glance at your data:
  - `View()` (but be careful!)
  - `summary()` (be careful on big datasets)
  - `head()` and `tail()`
  - `tibble::glimpse()`
  - `is.na()` and `sum(is.na())`

Spreading and gathering

The principles of tidy data seem so obvious that you might wonder if you'll ever encounter a dataset that isn't tidy. Unfortunately, however, most data that you will encounter will be untidy. There are two main reasons:

1. Most people aren't familiar with the principles of tidy data, and it's hard to derive them yourself unless you spend a *lot* of time working with data.
2. Data is often organised to facilitate some use other than analysis. For example, data is often organised to make entry as easy as possible.

Common problems

1. One variable might be spread across multiple columns.
2. One observation might be scattered across multiple rows.

Typically a dataset will only suffer from one of these problems; it'll only suffer from both if you're really unlucky! To fix these problems, you'll need the two most important functions in tidyr: `gather()` and `spread()`.

Gathering

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, 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
```

Gathering

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

- The set of columns that represent values, not variables. In this example, those are the columns `1999` and `2000`.
- The name of the variable whose values form the column names. I call that the `key`, and here it is `year`.
- The name of the variable whose values are spread over the cells. I call that `value`, and here it's the number of `cases`.

Gathering

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

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

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

The columns to gather are specified with `dplyr::select()` style notation. Here there are only two columns, so we list them individually. 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. To refresh your memory of the other ways to select columns, see select.

Gathering `table4` into a tidy form.

Gathering `table4` into a tidy form.

Gathering

In the final result, the gathered columns are dropped, and we get new `key` and `value` columns. Otherwise, the relationships between the original variables are preserved. Visually, this is shown in Figure @ref(fig:tidy-gather). We can use `gather()` to tidy `table4b` in a similar fashion. The only difference is the variable stored in the cell values:

```
table4b %>%
  gather(`1999`, `2000`, key = "year", value = "population")
```

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

To combine the tidied versions of `table4a` and `table4b` into a single tibble, we need to use `dplyr::left_join()`, which you'll learn about in relational data.

```
tidy4a <- table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
tidy4b <- table4b %>%
  gather(`1999`, `2000`, key = "year", value = "population")
left_join(tidy4a, tidy4b)
```

```
## Joining, by = c("country", "year")
```

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

Spreading

Spreading is the opposite of gathering. 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 `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`.

Once we've figured that out, we can use `spread()`, as shown programmatically below, and visually in Figure @ref(fig:tidy-spread).

```
spread(table2, key = type, value = 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
```

Spreading \`table2\` makes it tidy

Spreading `table2` makes it tidy

As you might have guessed from the common `key` and `value` arguments, `spread()` and `gather()` are complements. `gather()` makes wide tables narrower and longer; `spread()` makes long tables shorter and wider.

Exercises

1. Why are `gather()` and `spread()` not 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 %>%
  spread(year, return) %>%
  gather("year", "return", `2015`:`2016`)
```

(Hint: look at the variable types and think about column *names*.)

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

2. Why does this code fail?

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

```
## Error in inds_combine(.vars, ind_list): Position must be between 0
```

3. Why does spreading this tibble fail? How could you add a new column to fix the problem?

```
people <- tribble(
  ~name,             ~key,    ~value,
  #-----------------|--------|------
  "Phillip Woods",   "age",       45,
  "Phillip Woods",   "height",   186,
  "Phillip Woods",   "age",       50,
  "Jessica Cordero", "age",       37,
  "Jessica Cordero", "height",   156
)
```

4. Tidy the simple tibble below. Do you need to spread or gather it? What are the variables?

```
preg <- tribble(
  ~pregnant, ~male, ~female,
  "yes",     NA,    10,
  "no",      20,    12
)
```

Separating and uniting

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. You'll also learn about the complement of `separate()`: `unite()`, which you use if a single variable is spread across multiple columns.

Separate

`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 in Figure @ref(fig:tidy-separate) and the code 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
```

Separating `table3` makes it tidy

Separating `table3` makes it tidy

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 = "/")
```

(Formally, `sep` is a regular expression, which you'll learn more about in [strings].)

Look carefully at the column types: you'll notice that `case` 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
```

```
## 4 Brazil       20      00     80488/174504898
## 5 China        19      99     212258/1272915272
## 6 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.

Uniting `table5` makes it tidy

Uniting `table5` makes it tidy

We can use `unite()` to rejoin the *century* and *year* columns that we created in the last example. That data is saved as `tidyr::table5`. `unite()` takes a data frame, the name of the new variable to create, and a set of columns to combine, again specified in `dplyr::select()` style:

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

```
## # A tibble: 6 x 3
##   country     new   rate
##   <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
```

In this case we also need to use the `sep` argument. The default will place an underscore (_) between the values from different columns. Here we don't want any separator so we use "":

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

```
## # A tibble: 6 x 3
##   country     new   rate
##   <chr>       <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
```

Exercises

1. What do the `extra` and `fill` arguments do in `separate()`? Experiment with the various options for the following two toy datasets.

```
tibble(x = c("a,b,c", "d,e,f,g", "h,i,j")) %>%
  separate(x, c("one", "two", "three"))

tibble(x = c("a,b,c", "d,e", "f,g,i")) %>%
  separate(x, c("one", "two", "three"))
```

2. Both `unite()` and `separate()` have a `remove` argument. What does it do? Why would you set it to `FALSE`?

3. Compare and contrast `separate()` and `extract()`. Why are there three variations of separation (by position, by separator, and with groups), but only one unite?

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 %>%
  spread(year, return)
```

```
## # A tibble: 4 x 3
##     qtr `2015` `2016`
##   <dbl>  <dbl>  <dbl>
## 1    1.  1.88  NA
## 2    2.  0.590  0.920
## 3    3.  0.350  0.170
## 4    4. NA      2.66
```

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

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

```
## # A tibble: 6 x 3
##     qtr year  return
## * <dbl> <chr>  <dbl>
## 1    1. 2015   1.88
## 2    2. 2015   0.590
## 3    3. 2015   0.350
## 4    2. 2016   0.920
## 5    3. 2016   0.170
## 6    4. 2016   2.66
```

```
## 6     4. 2016   2.66
```

Another important tool for making missing values explicit in tidy data is `complete()`:

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

```
## # A tibble: 8 x 3
##     year   qtr return
##    <dbl> <dbl>  <dbl>
## 1 2015.     1.  1.88
## 2 2015.     2.  0.590
## 3 2015.     3.  0.350
## 4 2015.     4. NA
## 5 2016.     1. NA
## 6 2016.     2.  0.920
## 7 2016.     3.  0.170
## 8 2016.     4.  2.66
```

`complete()` takes a set of columns, and finds all unique combinations. It then ensures the original dataset contains all those values, filling in explicit `NA`s where necessary.

There's one other important tool that you should know for working with missing values. Sometimes when a data source has primarily been used for data entry, missing values indicate that the previous value should be carried forward:

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

You can fill in these 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 %>%
  fill(person)
```

```
## # A tibble: 4 x 3
##   person           treatment response
##   <chr>                <dbl>    <dbl>
```

```
## 1 Derrick Whitmore        1.        7.
## 2 Derrick Whitmore        2.       10.
## 3 Derrick Whitmore        3.        9.
## 4 Katherine Burke         1.        4.
```

Exercises

1. Compare and contrast the `fill` arguments to `spread()` and `complete()`.
2. What does the direction argument to `fill()` do?

Case Study

To finish off the chapter, 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
```

```
##     <chr>        <chr> <chr> <int>        <int>        <int>        <int>
## 
##  1 Afghanistan AF    AFG    1980           NA           NA           NA
##  2 Afghanistan AF    AFG    1981           NA           NA           NA
##  3 Afghanistan AF    AFG    1982           NA           NA           NA
##  4 Afghanistan AF    AFG    1983           NA           NA           NA
##  5 Afghanistan AF    AFG    1984           NA           NA           NA
##  6 Afghanistan AF    AFG    1985           NA           NA           NA
##  7 Afghanistan AF    AFG    1986           NA           NA           NA
##  8 Afghanistan AF    AFG    1987           NA           NA           NA
##  9 Afghanistan AF    AFG    1988           NA           NA           NA
## 10 Afghanistan AF    AFG    1989           NA           NA           NA
## # ... with 7,230 more rows, and 53 more variables: new_sp_m3544 <int>,
## #   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>,
```

This is a very typical real-life example dataset. It contains redundant columns, odd variable codes, and many missing values. In short, `who` is messy, and we'll need multiple steps to tidy it. Like dplyr, tidyr is designed so that each function does one thing well. That means in real-life situations you'll usually need to string together multiple verbs into a pipeline.

The best place to start is almost always to gather together the columns that are not variables. Let's have a look at what we've got:

- It looks like `country`, `iso2`, and `iso3` are three variables that redundantly specify the country.
- `year` is clearly also a variable.
- We don't know what all the other columns are yet, but given the structure in the variable names (e.g. `new_sp_m014`, `new_ep_m014`, `new_ep_f014`) these are likely to be values, not variables.

So we need to gather together all the columns from `new_sp_m014` to `newrel_f65`. We don't know what those values represent yet, so we'll give them the generic name `"key"`. We know the cells represent the count of cases, so we'll use the variable `cases`. There are a lot of missing values in the current representation, so for now we'll use `na.rm` just so we can focus on the values that are present.

```
who1 <- who %>%
  gather(new_sp_m014:newrel_f65, key = "key", value = "cases", na.rm = TI
who1
```

```
## # A tibble: 76,046 x 6
##    country     iso2  iso3   year key          cases
##  * <chr>       <chr> <chr> <int> <chr>        <int>
##  1 Afghanistan AF    AFG    1997 new_sp_m014      0
##  2 Afghanistan AF    AFG    1998 new_sp_m014     30
##  3 Afghanistan AF    AFG    1999 new_sp_m014      8
##  4 Afghanistan AF    AFG    2000 new_sp_m014     52
##  5 Afghanistan AF    AFG    2001 new_sp_m014    129
##  6 Afghanistan AF    AFG    2002 new_sp_m014     90
##  7 Afghanistan AF    AFG    2003 new_sp_m014    127
##  8 Afghanistan AF    AFG    2004 new_sp_m014    139
##  9 Afghanistan AF    AFG    2005 new_sp_m014    151
## 10 Afghanistan AF    AFG    2006 new_sp_m014    193
## # ... with 76,036 more rows
```

We can get some hint of the structure of the values in the new `key` column by counting them:

```r
who1 %>%
  count(key)
```

```
## # A tibble: 56 x 2
##    key              n
##    <chr>        <int>
##  1 new_ep_f014   1032
##  2 new_ep_f1524  1021
##  3 new_ep_f2534  1021
##  4 new_ep_f3544  1021
##  5 new_ep_f4554  1017
##  6 new_ep_f5564  1017
##  7 new_ep_f65    1014
##  8 new_ep_m014   1038
##  9 new_ep_m1524  1026
## 10 new_ep_m2534  1020
## # ... with 46 more rows
```

You might be able to parse this out by yourself with a little thought and some experimentation, but luckily we have the data dictionary handy. It tells us:

1. The first three letters of each column denote whether the column contains new or old cases of TB. In this dataset,

each column contains new cases.

2. The next two letters describe the type of TB:

   - `rel` stands for cases of relapse
   - `ep` stands for cases of extrapulmonary TB
   - `sn` stands for cases of pulmonary TB that could not be diagnosed by a pulmonary smear (smear negative)
   - `sp` stands for cases of pulmonary TB that could be diagnosed be a pulmonary smear (smear positive)

3. The sixth letter gives the sex of TB patients. The dataset groups cases by males (`m`) and females (`f`).

4. The remaining numbers gives the age group. The dataset groups cases into seven age groups:

   - `014` = 0 – 14 years old
   - `1524` = 15 – 24 years old
   - `2534` = 25 – 34 years old

- `2534` = 25 – 34 years old
- `3544` = 35 – 44 years old
- `4554` = 45 – 54 years old
- `5564` = 55 – 64 years old
- `65` = 65 or older

We need to make a minor fix to the format of the column names: unfortunately the names are slightly inconsistent because instead of `new_rel` we have `newrel` (it’s hard to spot this here but if you don’t fix it we’ll get errors in subsequent steps). You’ll learn about `str_replace()` in [strings], but the basic idea is pretty simple: replace the characters “newrel” with “new_rel”. This makes all variable names consistent.

```
who2 <- who1 %>%
  mutate(key = stringr::str_replace(key, "newrel", "new_rel"))
```

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

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

```
who2
```

```
## # A tibble: 76,046 x 6
##    country     iso2  iso3   year key          cases
##    <chr>       <chr> <chr> <int> <chr>        <int>
##  1 Afghanistan AF    AFG    1997 new_sp_m014      0
##  2 Afghanistan AF    AFG    1998 new_sp_m014     30
##  3 Afghanistan AF    AFG    1999 new_sp_m014      8
##  4 Afghanistan AF    AFG    2000 new_sp_m014     52
##  5 Afghanistan AF    AFG    2001 new_sp_m014    129
##  6 Afghanistan AF    AFG    2002 new_sp_m014     90
##  7 Afghanistan AF    AFG    2003 new_sp_m014    127
##  8 Afghanistan AF    AFG    2004 new_sp_m014    139
##  9 Afghanistan AF    AFG    2005 new_sp_m014    151
## 10 Afghanistan AF    AFG    2006 new_sp_m014    193
## # ... with 76,036 more rows
```

We can separate the values in each code with two passes of `separate()`. The first pass will split the codes at each underscore.

```
who3 <- who2 %>%
  separate(key, c("new", "type", "sexage"), sep = "_")
who3
```

```
## # A tibble: 76,046 x 8
##    country     iso2  iso3   year new   type  sexage cases
##    <chr>       <chr> <chr> <int> <chr> <chr> <chr>  <int>
##  1 Afghanistan AF    AFG    1997 new   sp    m014       0
##  2 Afghanistan AF    AFG    1998 new   sp    m014      30
##  3 Afghanistan AF    AFG    1999 new   sp    m014       8
##  4 Afghanistan AF    AFG    2000 new   sp    m014      52
##  5 Afghanistan AF    AFG    2001 new   sp    m014     129
##  6 Afghanistan AF    AFG    2002 new   sp    m014      90
##  7 Afghanistan AF    AFG    2003 new   sp    m014     127
##  8 Afghanistan AF    AFG    2004 new   sp    m014     139
##  9 Afghanistan AF    AFG    2005 new   sp    m014     151
## 10 Afghanistan AF    AFG    2006 new   sp    m014     193
## # ... with 76,036 more rows
```

Then we might as well drop the `new` column because it’s constant in this dataset. While we’re dropping columns, let’s also drop `iso2` and `iso3` since they’re redundant.

```
who3 %>%
  count(new)
```

```
## # A tibble: 1 x 2
##   new       n
##   <chr> <int>
## 1 new   76046
```

```
who4 <- who3 %>%
  select(-new, -iso2, -iso3)
```

Next we'll separate `sexage` into `sex` and `age` by splitting after the first character:

```
who5 <- who4 %>%
  separate(sexage, c("sex", "age"), sep = 1)
who5
```

```
## # A tibble: 76,046 x 6
##    country      year type  sex   age   cases
##    <chr>       <int> <chr> <chr> <chr> <int>
##  1 Afghanistan  1997 sp    m     014       0
##  2 Afghanistan  1998 sp    m     014      30
##  3 Afghanistan  1999 sp    m     014       8
##  4 Afghanistan  2000 sp    m     014      52
##  5 Afghanistan  2001 sp    m     014     129
##  6 Afghanistan  2002 sp    m     014      90
##  7 Afghanistan  2003 sp    m     014     127
##  8 Afghanistan  2004 sp    m     014     139
##  9 Afghanistan  2005 sp    m     014     151
## 10 Afghanistan  2006 sp    m     014     193
## # ... with 76,036 more rows
```

The `who` dataset is now tidy!

The who dataset is now tidy!

I’ve shown you the code a piece at a time, assigning each interim result to a new variable. This typically isn’t how you’d work interactively. Instead, you’d gradually build up a complex pipe:

```
who %>%
  gather(code, value, new_sp_m014:newrel_f65, na.rm = TRUE) %>%
  mutate(code = stringr::str_replace(code, "newrel", "new_rel")) %>%
  separate(code, c("new", "var", "sexage")) %>%
  select(-new, -iso2, -iso3) %>%
  separate(sexage, c("sex", "age"), sep = 1)
```

Exercises

1. In this case study I set `na.rm = TRUE` just to make it easier to
the correct values. Is this reasonable? Think about how missing
represented in this dataset. Are there implicit missing values? \
between an `NA` and zero?

2. What happens if you neglect the `mutate()` step?
(`mutate(key = stringr::str_replace(key, "newre`

3. I claimed that `iso2` and `iso3` were redundant with `country`.

4. For each country, year, and sex compute the total number of ca
informative visualisation of the data.

Non-tidy data

Before we continue on to other topics, it's worth talking briefly about non-tidy data. Earlier in the chapter, I used the pejorative term "messy" to refer to non-tidy data. That's an oversimplification: there are lots of useful and well-founded data structures that are not tidy data. There are two main reasons to use other data structures:

- Alternative representations may have substantial performance or space advantages.
- Specialised fields have evolved their own conventions for storing data that may be quite different to the conventions of tidy data.

Either of these reasons means you'll need something other than a tibble (or data frame). If your data does fit naturally into a rectangular structure composed of observations and variables, I think tidy data should be your default choice. But there are good reasons to use other structures; tidy data is not the only way.

If you'd like to learn more about non-tidy data, I'd highly recommend this thoughtful blog post by Jeff Leek: http://simplystatistics.org/2016/02/17/non-tidy-data/