Data Importing and Tidying

Today is an exciting day! Up until today, we've just worked with R's built in datasets. Doing that, we've been able to learn about a tremendous number of tools, and learn a wide variety of useful functions. But to progress on our journey, we need to begin reading in other data into R. Today, we'll cover how to do that, and how to "tidy" the data that we import.

Importing Data

The first step to reading data into R is to figure out how that data is stored. For right now, we'll focus on a few of the most common data types we'll see in this class. Below I've made a chart of the most common types of data storage, their extensions, and the package used to read that type of data into R.

File Extension	Description	Package Used
.csv	Comma separated values	readr
.csv2	Semicolon separated values	readr
.tsv	Tab separated values	readr
.xls or .xlsx	Excel Documents	readxl
.por or .sav	SPSS Files	haven
.sas	SAS Files	haven
.dta	Stata files	haven

While there are obviously other types of data out there, this is a good place to start. There are two ways that we can think about organizing different ways of storing data:

1. By the character used to separate values. For example comma separated values (CSV) files are called that because a comma separates each value (we show an example of one such file below). Likewise, semicolon separated files have a semicolon between each value. These types of delimited files are most commonly exported from spreadsheet programs like Microsoft Excel or Numbers (the Apple spreadsheet program).

2. By the proprietary format generated by a particular program. .xls is Excel's data storage format, and .dta is Stata's file storage format.

In my experience, I typically find most data in .csv (comma-separated) or Excel files, as both of these are outputted from spreadsheet programs, which are typically how people enter data. To read these files, you'll use `readr` or `readxl`. The other files types—SPSS, SAS, or Stata—are other popular statistical analysis packages similar to R (though I think all of them are less flexible, and hence less useful). Nevertheless, because they are all popular programs, you will encounter datasets in these formats from time to time. In a later lecture, we'll read in datasets in these formats.

Because all three of these libraries (`readr`, `haven`, `readxl`) are part of the tidyverse, they all share a common syntactical structure. So once you know how to use one of them, you know how to use the others.

There are other, more complicated types of files that we'll discuss in later courses in this sequence, such as various types of relational databases like SQL, or hierarchical data like XML or JSON. For now, we'll ignore these types of data as they introduce some additional complications. There's also the `rio` package for input, which you can read about on CRAN.

The Basic Syntax of the `readr` Library

We'll first focus on the syntax for `readr`, as that is the most commonly used library for data import. `readr` comes as part of the tidyverse, so to use its functions, you'll need to load the tidyverse. There are 4 main `readr` functions:

- `read_csv()`, which reads comma separated files
- `read_csv2()`, which reads semicolon separated files, which are most common in countries where the comma is used as the decimal place
- `read_tsv()`, which reads tab separated files
- `read_delim()`, which subsumes the three functions above (i.e., it can read comma, semi-colon, or tab separated files)

We'll focus on `read_csv()` since csv files are probably the single most common type of file you'll encounter, but note that the syntax is the same across all four functions. The first, and most essential, argument is the file's path, which is where the file is physically stored on your machine. There are two ways of inputting this:

1. If the file lives online, you can give a URL that points to the file
2. A path on your computer

Even if the file lives online, I recommend downloading a local copy. It's very frustrating when you're working on a project, a website restructures itself, and your URLs no longer work. In this class, we'll typically provide you the file on Canvas, and then you'll download it to your raw data folder. Remember that in our first lecture (which seems so long ago!), we set up our Dropbox folder to have the same structure, so this is "~/Dropbox/DATA101/Data/Raw" on a Mac, or "C:/USERNAME/Dropbox/DATA101/Data/Raw" on a PC. Remember that on a PC, when working in R, you might have to enter the path with the slashes reversed, that is, as "`C:\USERNAME\Dropbox\DATA101\Data\Raw`".

For most csv files, setting the path correctly will allow you to read in the data for analysis in R. But there are two complications that arise sometimes that require an additional note. First, the `read_csv()` functions assumes that your data will be organized in the "standard" way for csv files: the first row is a list of variable names (one for each column), and then each remaining row is the data for one observation: the second row corresponds to observation 1, the third row to observation 2, and so forth. Occasionally, you'll encounter a csv file that does not have any variable names, and the first row is the first observation of

the data. If this is the case, then you'll need to tell R with the `col_names = FALSE` option. For example, if I had a file "nonames.csv" that had no variable names, I'd read it into R with the command: `read_csv(file="nonames.csv",col_names=FALSE)`.

Second, you'll need to look at your data file and see how it records `NA` values (i.e., missing data). This varies somewhat from dataset to dataset, and you'll need to investigate it. Common ways include `NA`, `.`, `-9`, and `-99`, but each dataset is different. You can tell R how your data encodes missingness with the option `na =`. For example, if in my "nonames.csv" file missing data was denoted by -99, I'd read the dataset into R as: `read_csv(file="nonames.csv",col_names=FALSE,na="-99")`.

This will all make more sense when we go through an actual example. Recently, 538 did a study of how earnings vary for college graduates based on their major. 538 is an amazing resource for learning R, because they share their data, and often their R code, on their GitHub. GitHub is a repository for storing data and code on the web, and you'll find a lot of amazing public data and R code (along with other types of computer code) stored there. If you read a 538 story and want to see the data, just click on the "GitHub" link that appears directly below the byline in a story.

Here, we'll use a slightly cleaned up version of the data, which I've saved it to Canvas (the file name is `recent-grads.csv`).

Let's begin by looking at the data, just so you can see what a .csv file looks like:

```
Rank,Major_code,Major,Total,Men,Women,Major_category,ShareWomen,Sample_size,Employed,Full_time,Part_
time,Full_time_year_round,Unemployed,Unemployment_rate,Median,P25th,P75th,College_jobs,Non_college_j
obs,Low_wage_jobs
1,2419,PETROLEUM
ENGINEERING,2339,2057,282,Engineering,0.120564344,36,1976,1849,270,1207,37,0.018380527,110000,95000,
125000,1534,364,193
2,2416,MINING AND MINERAL
ENGINEERING,756,679,77,Engineering,0.101851852,7,640,556,170,388,85,0.117241379,75000,55000,90000,35
0,257,50
3,2415,METALLURGICAL
ENGINEERING,856,725,131,Engineering,0.153037383,3,648,558,133,340,16,0.024096386,73000,50000,105000,
456,176,0
4,2417,NAVAL ARCHITECTURE AND MARINE
ENGINEERING,1258,1123,135,Engineering,0.107313196,16,758,1069,150,692,40,0.050125313,70000,43000,800
00,529,102,0
5,2405,CHEMICAL
```

Figure 1: First Part of the CSV file containing the data on majors and earnings

So note how this file is arranged. The first row contains the variable names. So here, we know that the first variable contains the rank (sorted by earnings), the second is the major code, the third is the name of the major, and so forth. Note that each one is separated by a comma, since its a CSV file. In the second row, we have the data for the the rank = 1 major (so the highest-earning major), which is major 2419: Petroleum Engineering. The third row contains the data for major code 2146: Mining and Metallurgical Engineering, and so forth. Remember, in a CSV file, each row is an observation, and each value is separated by a comma.

So now, we'll download the file from Canvas (recent-grads.csv) and save it to our raw data folder (~/Dropbox/DATA101/Data/Raw). It is important that you save the file here, because below we'll tell R this is where it should find it.

Remember that the first step is always loading the tidyverse library, since the `readr` functions we're using below come from there.

```
library(tidyverse)
setwd("~/Dropbox/DATA101/")
grads <- read_csv(file="Data/Raw/recent-grads.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_integer(),
##   Major = col_character(),
##   Major_category = col_character(),
##   ShareWomen = col_double(),
##   Unemployment_rate = col_double()
## )
```

```
## See spec(...) for full column specifications.
```

Let's walk through this syntax. The first line tells R what I want my working directory to be (hence, the name: it sets the working directory). The working directory tells R where to look for data you wish to read into R (and also where to export graphs and files you wish to save, which we'll cover in a future lecture).

The second line calls `read_csv()`, which is the readr function that reads the data into R. But to do that, it needs to know where to find that file. So remember that we set up our DATA101 folder in Dropbox as follows:

- DATA101
  - Data
    - Raw
    - Processed
  - Code
  - Graph
  - Homework

This is a hierarchical file structure: the "Raw" folder is nested within the "Data" folder, which itself is nested with the "DATA101" folder, which in turn sits in Dropbox. So if I tell R that "~/Dropbox/DATA101" is the working directory, then telling it the file path is "Data/Raw" means it expects to find the folder "Data"" inside "DATA101", and then the folder "Raw" inside "Data", and then the file `recent_grads.csv` there.

Note that in both lines of code, I've enclosed the path in parentheses (R requires this when specifying a file path).

If you see the error message: `Error: 'recent_grads.csv' does not exist in current working directory` then double-check that you've correctly specified your file path (99.999% of the time, this is the error). You should begin by running `getwd()` to tell you what your current working directory is. In this case, it should be ~/Dropbox/DATA101. Then double-check your path to make sure you don't have a spelling error, additional spaces, etc. Then finally check on your machine that you actually did save your data file to the correct location (i.e., verify that `recent_grads.csv` actually lives in ~/Dropbox/DATA101/Data/Raw).

Once R has located the file, the `read_csv()` function then begins to parse each column in my dataset. R goes through each column and figures out whether it is a character (text, like the name of the major), or a number (integer or double, for example, the share of women in each major) and stores it accordingly. Once that happens, the dataset is loaded into memory and we can analyze it.

Note that I've passed the file to an object (`grads`) so that I can call it with other functions. For example, I could sort the data by the highest median income:

```
select(grads,Median,everything()) %>% arrange(desc(Median))
```

```
## # A tibble: 173 x 21
##    Median  Rank Major_code Major Total   Men Women Major_category
##     <int> <int>      <int> <chr> <int> <int> <int> <chr>
##  1 110000     1       2419 PETR…  2339  2057   282 Engineering
##  2  75000     2       2416 MINI…   756   679    77 Engineering
##  3  73000     3       2415 META…   856   725   131 Engineering
##  4  70000     4       2417 NAVA…  1258  1123   135 Engineering
##  5  65000     5       2405 CHEM… 32260 21239 11021 Engineering
##  6  65000     6       2418 NUCL…  2573  2200   373 Engineering
##  7  62000     7       6202 ACTU…  3777  2110  1667 Business
##  8  62000     8       5001 ASTR…  1792   832   960 Physical Scie…
##  9  60000     9       2414 MECH… 91227 80320 10907 Engineering
## 10  60000    10       2408 ELEC… 81527 65511 16016 Engineering
## # … with 163 more rows, and 13 more variables: ShareWomen <dbl>,
## #   Sample_size <int>, Employed <int>, Full_time <int>, Part_time <int>,
## #   Full_time_year_round <int>, Unemployed <int>, Unemployment_rate <dbl>,
## #   P25th <int>, P75th <int>, College_jobs <int>, Non_college_jobs <int>,
## #   Low_wage_jobs <int>
```

Note that the first part of the code (before the pipe) just puts the median income first so we can see it (otherwise, it won't print since only the first few columns are displayed). Perhaps not surprisingly, the major with the highest median income is petroleum engineering at $110,000 (thanks largely to the fracking boom of recent years). The next few majors are other types of engineering (mining/mineral, metallurgical, naval, etc.), all of whom also do quite well. Other high earning majors include actuaries, who help insurance companies assess risk (median income of $62,000), and astronomers/astrophysicists, who also have a median income of $62,000. To put these numbers into context, the median major earns $36,000, so people with these majors do quite well (remember, these data are for recent graduates, so that will skew the data somewhat). You could continue this analysis to learn

more about the link between college major and income with the other functions we've discussed in this class.

If you wanted to see how to read in the file from the URL, that syntax would simply be:

```
grads <-
read_csv("https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2018/2018-10-16/recent-grads.csv")
```

Note that as with the file I saved to my Dropbox folder, I have to enclose the URL in quotation marks.

That's the basic logic of reading in a csv file. Chapter 11 of the textbook covers a number of interesting complications that can arise when reading in different kinds of data. But honestly, for now, I'd recommend just skimming that material, and then returning there when you have an issue. Those type of errors are the sort of things best learned via trial and error. For now, we'll (mostly) avoid those sorts of complications.

A Simple Example using the `readxl` library

In general, I find it easier save my Excel files to .csv files and then read them in with the above, but I think this is an old habit more than anything else. If you want to directly read in the excel file themselves (without first saving it to a csv file), the function is `read_excel()`. It follows the same basic pattern as the `read_csv()` function we discussed above.

The `readxl` library has several built-in examples, so we'll use one of those just to see the syntax. The key thing to note is that Excel files can have more than 1 sheet (these are indicated by the tabs at the bottom of an Excel spreadsheet). By default, the `read_excel()` function only reads in the first sheet, so if your data comes from another sheet, you have to explicitly tell the function that. Let's see this in action:

```
library(readxl)
excel_example <- readxl_example("datasets.xls")
read_excel(excel_example)
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <chr>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # … with 140 more rows
```

The `readxl_example()` function just pulls up several different example datasets (contained in `datasets.xls`), which I then load with the `read_excel()` function. Here, I've loaded data on sepal and petals (this is the famous Iris dataset that comes with R). But note that I can see if this Excel spreadsheet (datasets.xls) contains multiple sheets using the `excel_sheets()` function.

```
excel_sheets(excel_example)
```

```
## [1] "iris"     "mtcars"   "chickwts" "quakes"
```

So there are 4 sheets, each of which is a separate dataset. So if I wanted to load the `mtcars` data, I'd have to tell the `read_excel` function to bring in that spreadsheet:

```
read_excel(excel_example, sheet="mtcars")
```

```
## # A tibble: 32 x 11
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
##  2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
##  3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
##  4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
##  5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
##  6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
##  7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
##  8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
##  9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
## 10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
## # … with 22 more rows
```

I could equivalently tell it to load in the second sheet:

```
read_excel(excel_example, sheet=2)
```

```
## # A tibble: 32 x 11
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
##  2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
##  3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
##  4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
##  5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
##  6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
##  7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
##  8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
##  9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
## 10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
## # … with 22 more rows
```

Likewise, `haven()` uses a very similar syntax to read in SPSS, SAS, or Stata files. We'll read in some of those files in a future lecture, but the syntax is essentially the same.

Test Yourself: Practice Loading another File

Try and load the quakes dataset from the excel spreadsheet containing the same datasets.

Answer:

```
read_excel(excel_example, sheet=4)
```

```
## # A tibble: 1,000 x 5
##      lat  long depth   mag stations
##    <dbl> <dbl> <dbl> <dbl>    <dbl>
##  1 -20.4  182.   562   4.8       41
##  2 -20.6  181.   650   4.2       15
##  3 -26    184.    42   5.4       43
##  4 -18.0  182.   626   4.1       19
##  5 -20.4  182.   649   4         11
##  6 -19.7  184.   195   4         12
##  7 -11.7  166.    82   4.8       43
##  8 -28.1  182.   194   4.4       15
##  9 -28.7  182.   211   4.7       35
## 10 -17.5  180.   622   4.3       19
## # … with 990 more rows
```

Or equivalently:

```
read_excel(excel_example, sheet="quakes")
```

```
## # A tibble: 1,000 x 5
##      lat  long depth   mag stations
##    <dbl> <dbl> <dbl> <dbl>    <dbl>
##  1 -20.4  182.   562   4.8       41
##  2 -20.6  181.   650   4.2       15
##  3 -26    184.    42   5.4       43
##  4 -18.0  182.   626   4.1       19
##  5 -20.4  182.   649   4         11
##  6 -19.7  184.   195   4         12
##  7 -11.7  166.    82   4.8       43
##  8 -28.1  182.   194   4.4       15
##  9 -28.7  182.   211   4.7       35
## 10 -17.5  180.   622   4.3       19
## # … with 990 more rows
```

Where Can I Go to Find Data?

You might be wondering where you can go to find different datasets. Honestly, the world is your oyster. The following is an extremely incomplete listing of some places that you can visit to get started:

- Google Dataset Search Engine: this allows you to search for various datasets through Google

- Pew Research Center: Pew is a major research center that collects a tremendous amount of data on government, politics, and religion. They also maintain a really neat blog about data science, Decoded.
- American National Election Study: One of the longest-running academic studies of the electorate (it dates back to 1948!). This is a key source for information on voter's attitudes and how they've changed over time. The General Social Survey is another great source of data on over-time public opinion data.
- The U.S. Census: There is a whole suite of tools to download and use the data collected by the U.S. Census. Also make sure to check out the open data portal for the federal government, Data.gov. The government collects a mind-boggling amount of data, so its worth poking around this just to see some of the strange datasets they collect.
- Several journalistic outlets—such as 538 and Buzzfeed News—make their data available online
- Many international organizations—such as the World Bank, IMF, and World Health Organization—make much of data available online as well
- Tidy Tuesday is an online repository full of datasets geared toward learning R and the Tidyverse.

And there are many, many more resources online. Do some google sleuthing, and you'll no doubt turn up many, many more datasets to analyze.

Tidying Data

So far, all of the data we've used has been tidy, which is to say it's been cleaned, organized, and easy to use and analyze. Tidy data is a dataset that follows the rules for the tidyverse, and hence interacts well with the functions we've discussed in this class. Sadly, much of the data out there is *not* tidy, and so you'll need to tidy it before using. But once you tidy your data, it becomes *much* easier to work with in R.

Tidy data is data that follows 3 rules:

1. Each variable is its own column
2. Each observation has its own row
3. Each value has its own cell

This seems obvious; wouldn't all data follow these rules? The answer is not really, because many datasets are organized by some other principle, such as to make the data easier to input, or easier to read from the screen. For example, the World Bank collects data on the GDP of every country. I've downloaded the data and put it on Canvas (the file is `wb_gdp.csv`). We can read it into R:

```
read_csv(file="Data/Raw/wb_gdp.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Country = col_character(),
##   Code = col_character()
## )

## See spec(...) for full column specifications.

## # A tibble: 264 x 30
##    Country Code  `1990` `1991` `1992` `1993` `1994` `1995` `1996` `1997`
##    <chr>   <chr>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
##  1 Aruba   ABW      NA     NA     NA     NA     NA     NA     NA     NA
##  2 Afghan… AFG      NA     NA     NA     NA     NA     NA     NA     NA
##  3 Angola  AGO    2217.  2244.  2092.  1575.  1578.  1798.  2020.  2144.
##  4 Albania ALB    2722.  1993.  1903.  2148.  2391.  2782.  3110.  2838.
##  5 Andorra AND      NA     NA     NA     NA     NA     NA     NA     NA
##  6 Arab W… ARB    6754.  6817.  7188.  7390.  7578.  7710.  8025.  8327.
##  7 United… ARE   71376. 70238. 70093. 68691. 71085. 73622. 75543. 79158.
##  8 Argent… ARG    6991.  8025.  9062.  9694. 10342. 10130. 10754. 11689.
##  9 Armenia ARM    2419.  2228.  1350.  1290.  1423.  1585.  1735.  1843.
## 10 Americ… ASM      NA     NA     NA     NA     NA     NA     NA     NA
## # … with 254 more rows, and 20 more variables: `1998` <dbl>, `1999` <dbl>,
## #   `2000` <dbl>, `2001` <dbl>, `2002` <dbl>, `2003` <dbl>, `2004` <dbl>,
## #   `2005` <dbl>, `2006` <dbl>, `2007` <dbl>, `2008` <dbl>, `2009` <dbl>,
## #   `2010` <dbl>, `2011` <dbl>, `2012` <dbl>, `2013` <dbl>, `2014` <dbl>,
## #   `2015` <dbl>, `2016` <dbl>, `2017` <dbl>
```

Here, each country is a row, and each value has it's own cell, so it obeys rules # 2 and # 3. But it does not obey rule #1. There are four variables here: country, country code, year, and GDP. But note that this data is organized so that GDP (a variable) is spread across multiple columns, one for each year, which is not tidy. It's easy to see why the World Bank organized the data this way: it makes it easier to input the data, and you can read across each row to see how GDP changes over time (and you can easily create a graph in Excel). But for working with this dataset in R, it will be easier if the data has been tidied.

When we have messy data, we can (typically) tidy it using 4 related functions:

- `spread()`, for when one observation is spread across multiple rows
- `gather()`, for when one variable is spread across multiple columns (like in our example above)
- `separate()`, for when one column has multiple entries in it
- `unify()`, for when one observation is spread across two columns

So note that `spread()` is the opposite of `gather()` and `separate()` is the opposite of `unify()`. Spread turns rows into columns, and gather turns columns into rows. Separate takes a case where you have 2 things stored in one variable and splits them apart, unify

does the opposite (takes contents from multiple cells and combines them into one cell). Let's work through some examples so we can see how to tidy up a few examples of messy data.

Gather: One Variable, Multiple Columns

The example above of the World Bank GDP data shows us the power of `gather()`. We have 30 columns in our data, but only 4 variables: country, country code, year, and GDP. In particular, GDP is spread across 28 different columns, one for each year in our dataset (1990-2017). So we need to gather these 28 variables and combine them into 2 variables: GDP and year.

```
gdp <- read_csv(file="Data/Raw/wb_gdp.csv")

## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Country = col_character(),
##   Code = col_character()
## )

## See spec(...) for full column specifications.

gdp %>% gather(key=year,value=GDP,c(`1990`:`2017`))

## # A tibble: 7,392 x 4
##    Country              Code  year     GDP
##    <chr>                <chr> <chr>  <dbl>
##  1 Aruba                ABW   1990     NA
##  2 Afghanistan          AFG   1990     NA
##  3 Angola               AGO   1990   2217.
##  4 Albania              ALB   1990   2722.
##  5 Andorra              AND   1990     NA
##  6 Arab World           ARB   1990   6754.
##  7 United Arab Emirates ARE   1990  71376.
##  8 Argentina            ARG   1990   6991.
##  9 Armenia              ARM   1990   2419.
## 10 American Samoa       ASM   1990     NA
## # … with 7,382 more rows
```

In our original data, we have data on 264 geographic units (mostly countries, and a few places like American Samoa that aren't) and 30 variables (country, year, and GDP spread across 28 columns, one per year). After we tidy the data, we have 7,392 country-year rows, with variables for country, GDP, and year. It's the same data, just organized in a different format. In both cases, we have 28 values of GDP for 264 geographical units ($28 \times 264 = 7392$).

Let's dig into the syntax of our code a bit. Again, as above, I have my data in my Raw data folder (~/Dropbox/DATA101/Data/Raw), so R knows to look there for the csv file. Once it reads in the data, I then use the `gather()` command to take all of the columns with the GDP

data and combine them to create a GDP variable, organized by year. `gather()` takes a series of non-variable columns (here, the GDP figures for each year) and gathers them into a two-column variable-key pair.

There are 3 arguments that I need to pass to `gather`:

1. The key, which is the name of the variable whose value forms the column names
2. The value, which is the name of the variable whose values are spread over the cells
3. The set of columns that represent values, not variables

So here, GDP—which is our variable—is spread over 28 columns (1990-2017). The key is the year. I tell R to gather the data from those 28 columns and turn them into a variable called GDP. The key (year) tells us which value of the variable (GDP) is in that row. R knows which year's value is which because I told it year is our key, and so it knows that the value from the column labeled `1990` is the data for 1990 and so forth.

Test Yourself: Using data from Pew

This is an example I found online from the Pew Research Center's Religious Landscape Study, which analyzes various patterns across religious groups in the United States. This chart looks at the educational breakdown of various religious groups in the U.S.

Religious Tradition	HS or Less	Some College	College	Post-Grad
Buddhist	20	33	28	20
Catholic	46	27	16	10
Evangelical Protestant	43	35	14	7
Hindu	12	11	29	48
Historically Black Protestant	52	33	9	6
Jehovah's Witness	63	25	9	3
Jewish	19	22	29	31
Mainline Protestant	37	30	19	14
Mormon	27	40	23	10
Muslim	36	25	23	17
Orthodox Christian	27	34	21	18
Unaffiliated	38	32	18	11

The cells in this table give the percentage of each group with a given education. For example, 20% of Buddhists have a post-graduate education, but only 3% of Jehovah's Witnesses do.

Question 1: Why is this data not tidy?

Answer 1: There are really 3 variables in this dataset: religion, education, and fraction (of that religion with a given education), but our dataset is not organized that way. For

example, while education is one variable, in this data, each level is its own column, so this violates rule # 1 for tidy data (each variable should be a column).

Question 2: I've created a version of this dataset as a csv file and posted it to Canvas (`pew.csv`). Make this into a tidy dataset.

Answer 2: The problem with this data is that the level of education is split across 4 variables: HS or Less, Some College, College, and Post-Grad. We want to create a new variable `education` that records the `fraction` of each religious group that has a given education level. So instead of having HS or Less as a variable, we'll have it as the value of Education (which is what it is!). We'll use `gather()` to do this:

```
pew <- read_csv(file="Data/Raw/pew.csv")

## Parsed with column specification:
## cols(
##   Religion = col_character(),
##   HSLess = col_integer(),
##   SomeCollege = col_integer(),
##   College = col_integer(),
##   PostGrad = col_integer()
## )

pew %>% gather(key=education, value=fraction,
c(HSLess,SomeCollege,College,PostGrad))

## # A tibble: 48 x 3
##    Religion        education fraction
##    <chr>           <chr>        <int>
##  1 Buddhist        HSLess          20
##  2 Catholic        HSLess          46
##  3 Evangelical     HSLess          43
##  4 Hindu           HSLess          12
##  5 HBProtestant    HSLess          52
##  6 JehovahsWitness HSLess          63
##  7 Jewish          HSLess          19
##  8 Mainline        HSLess          37
##  9 Mormon          HSLess          27
## 10 Muslim          HSLess          36
## # … with 38 more rows
```

Notice what I did here. I told R to take the levels of education—HSLess, SomeCollege, College, and PostGrad—and combine them into a variable called Education. For each observation, it tells me what fraction of that religion has that education level. So, for example, we see that 20% of Buddhists have a HS degree or less, as do 46% of Catholics, and so forth. This matches our original data from Pew, it's just now in an easier to use format.

Note that I could write slightly more concise code by writing:

```
pew <- read_csv(file="Data/Raw/pew.csv")
pew %>% gather(education, fraction, -Religion)
```

This is equivalent, and slightly pithier, but maybe less clear, so I'd probably prefer the way I wrote it above.

In my experience, `gather()` is the most commonly used of these four functions for tidying data. This type of setup (like in the GDP case above) is fairly common. Typically, you'll use `gather()` when the column names are not variable *names*, but rather variable *values*.

Spread: One Observation, Multiple Rows

`spread()` is the opposite of `gather()`: we use spread when we have a single observation spread across multiple rows. For example, in the `dsr` library, there is a sample dataset illustrating this property:

```
library(dsr)
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
```

Remember that if you get an error telling you that R cannot find the package, make sure you've first installed it on your machine. Here, the first row of `table2` tells me the number of cases of tuberculosis in Afghanistan in 1999, and the second row tells me the population of Afghanistan in 1999. So these both correspond to the same observation (Afghanistan 1999), but they come from different rows. This violates rule # 2 for tidy data: each observation should be its own row. In essence, the variable `type` is really two variables merged into 1: the number of cases, and the population. It should be two variables, not one.

We can resolve this by spreading the data:

```
table2 %>% spread(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
```

So note what this does: it spreads (separates) the two variables in type (cases and population) into 2 different variables.

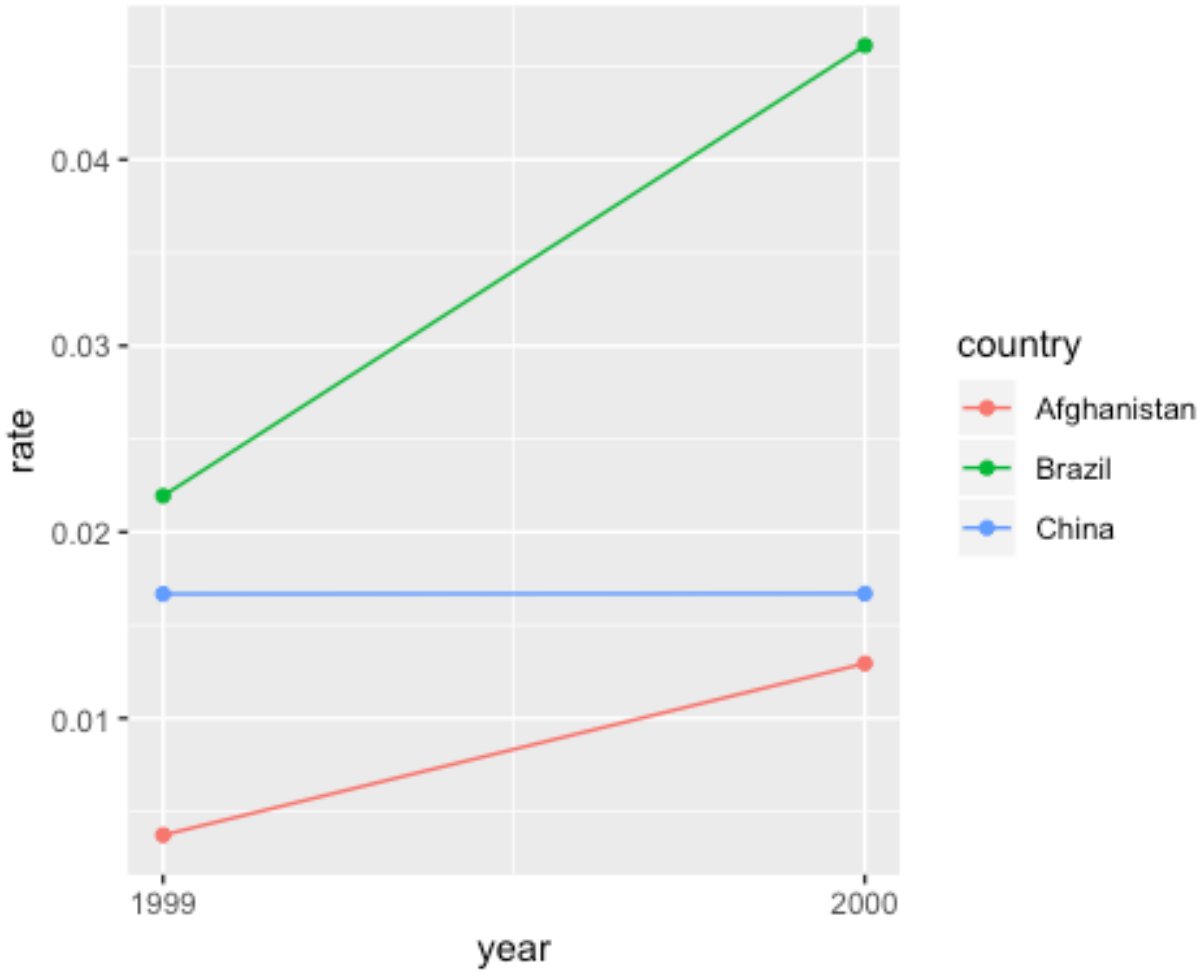
The syntax for `spread()` is similar to `gather()`, but here we only need two pieces of information, not three:

1. The column that contains the multiple variables, which is our key
2. The column that contains values for multiple variables, which is our value

So here, this we take the two variables included in type—cases and population—and spreads them out into their separate variables, putting the values (count) into the appropriate cell.

This also nicely shows us why we want tidy data. Suppose I wanted to calculate the rate of TB in each country, which is just the number of cases divided by the population, and plot the change over time separately for each country in our dataset. In the original form of the data, this would be quite complicated to do. But in our tidy data, this is trivial:

```
table2 %>% spread(key=type, value=count) %>%
mutate(rate=100*(cases/population)) %>%
  ggplot(aes(x=year, y=rate, group=country)) +
  geom_line(aes(color=country)) +
  geom_point(aes(color=country)) +
  scale_x_continuous(breaks=c(1999,2000))
```



So here, we see that the rate of TB had no change between 1999 and 2000 in China, but there was a modest gain in Afghanistan and a rather significant jump (more than 100% increase!) in Brazil. Once we have tidy data, it becomes very easy to use the other functions from the tidyverse to analyze our data.

Separate: Splitting Variables

In other cases, we might have one variable that actually contains multiple pieces of information. For example, the `tb` dataset records cases of tuberculosis (tb), broken down by year, country, and demographic group (these data come from the World Health Organization). Here, the demographic groups are sex (m/f) and age (0-4, 5-14, 0-14 (the sum of the first two), 15-24, 25-34, 35-44, 45-54, 55-64, 65+ and unknown). The data covers 1980-2008, though note that not all countries are included in all years.

```
tb <- read_csv(file="Data/Raw/tb.csv")

## Parsed with column specification:
## cols(
##   .default = col_integer(),
```

```
##   iso2 = col_character()
## )

## See spec(...) for full column specifications.

tb

## # A tibble: 5,769 x 22
##    iso2   year   m04  m514  m014 m1524 m2534 m3544 m4554 m5564   m65    mu
##    <chr> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int>
##  1 AD     1989    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##  2 AD     1990    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##  3 AD     1991    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##  4 AD     1992    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##  5 AD     1993    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##  6 AD     1994    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##  7 AD     1996    NA    NA     0     0     0     4     1     0     0    NA
##  8 AD     1997    NA    NA     0     0     1     2     2     1     6    NA
##  9 AD     1998    NA    NA     0     0     0     1     0     0     0    NA
## 10 AD     1999    NA    NA     0     0     0     1     1     0     0    NA
## # … with 5,759 more rows, and 10 more variables: f04 <int>, f514 <int>,
## #   f014 <int>, f1524 <int>, f2534 <int>, f3544 <int>, f4554 <int>,
## #   f5564 <int>, f65 <int>, fu <int>
```

So this is a bit of a mess. For example, the `m04` variable indicates the number of cases of tb among males ages 0-4 in a given country in a given year, `m514` gives the number of tb cases among males ages 5-14, and so forth. What we'd like is a dataset with 5 variables: country (`iso2`), year (`year`), sex, age, and number of cases. Country and year are basically fine as is, but we'll need to do some work to create the final three variables.

First, we'll use `gather()` to pull together sex and age into one variable `demo`.

```
tb2 <- tb %>% gather(demo, n, c(-iso2,-year))
tb2

## # A tibble: 115,380 x 4
##    iso2   year demo      n
##    <chr> <int> <chr> <int>
##  1 AD     1989 m04      NA
##  2 AD     1990 m04      NA
##  3 AD     1991 m04      NA
##  4 AD     1992 m04      NA
##  5 AD     1993 m04      NA
##  6 AD     1994 m04      NA
##  7 AD     1996 m04      NA
##  8 AD     1997 m04      NA
##  9 AD     1998 m04      NA
## 10 AD     1999 m04      NA
## # … with 115,370 more rows
```

This is the same syntax we used above, and now we have a variable `demo` that records the age and sex groupings. But ideally, we'd want demographic to be split into two pieces of information: sex and age. To do that, we use `separate()`:

```
tb3 <- tb2 %>% separate(col=demo,into=c("sex","age"), sep=1)
tb3
```

```
## # A tibble: 115,380 x 5
##    iso2   year sex   age       n
##    <chr> <int> <chr> <chr> <int>
##  1 AD     1989 m     04       NA
##  2 AD     1990 m     04       NA
##  3 AD     1991 m     04       NA
##  4 AD     1992 m     04       NA
##  5 AD     1993 m     04       NA
##  6 AD     1994 m     04       NA
##  7 AD     1996 m     04       NA
##  8 AD     1997 m     04       NA
##  9 AD     1998 m     04       NA
## 10 AD     1999 m     04       NA
## # … with 115,370 more rows
```

So now we have the dataset with the 5 variables we wanted: country, year, sex, age, and number of cases observed. Here, to make things easier to understand, I've split this into multiple steps, but if you were doing this on your own, you could use the pipe to string these together into one code chunk.

The syntax for `separate()` is quite similar to the other tidying functions above. There are three required pieces of information:

1. The variable I want to separate, which is `col`
2. What the separated pieces should be called, which is given by `into`
3. How to split them, which is given by `sep`

So here, I tell R that I want to separate `demo` into `sex` and `age` by splitting it after the first character (`sep=1`), since the first character is always m or f, indicating the sex. Here, we separated by position, splitting after the first character. But you can also separate by a given character. For example, if the variable has been called `m_04`, then we could separate at the underscore (_).

Test Yourself: A Sample Problem

Question: Below is a messy dataset (it is a version of Table 2 seen above). Explain why it is messy, and then tidy it.

```
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
```

Answer: The data is messy because `rate` contains two values: cases and population. For example, in Afghanistan in 1999, there were 745 cases of disease X, but 19,987,071 people living in the country, separated by `/`. So we need to separate out these values into 2 different variables, which we can do with the following code:

```
table3 %>% separate(rate, 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
```

Unite: Combining Variables

Just as `separate()` splits one cell into two variables (like sexage above), `unite()` does the opposite, and brings two columns together. The main case where you use this involves dates: sometimes you might combine (say) month year, or day and month when analyzing patterns over time (we did something similar to this in an earlier lecture when analyzing the NYC flights data).

Let's see an example of where unite would be useful:

```
table5
```

```
## # 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
```

So notice that year and century are recorded as separate variables, but ideally, they'd be combined into one (so a year was, say, 1999 or 2004). We can do this with unite:

```
table5 %>% unite(col=yr,c(century,year))
```

```
## # A tibble: 6 x 3
##   country     yr    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
```

You need to specify 2 arguments to `unite()`:

1. the new variable we want to create, which is `col` (for column)
2. the variables to be combined

So here, I created a new variable `yr` by combining `century` and `year`. Note that by default, R adds an underscore between the two variables that I'm uniting. But here, that's not what I want: I want `1999`, not `19_99`. Instead, I need to tell R to not put any characters between them. I do that with the `sep=""` option, which tells R not to add any spaces:

```
table5 %>% unite(col=yr,c(century,year),sep="")
```

```
## # A tibble: 6 x 3
##   country     yr    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
```

A Messy Datasest Made Tidy

Let's look at a larger version of the TB data used above (I used a sub-setted version of the data above to make it easier to see the use of separate). The larger version of the data can be found in the `DSR` library, which we loaded above:

```
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 Afghan… AF    AFG    1980          NA           NA           NA
##  2 Afghan… AF    AFG    1981          NA           NA           NA
##  3 Afghan… AF    AFG    1982          NA           NA           NA
##  4 Afghan… AF    AFG    1983          NA           NA           NA
##  5 Afghan… AF    AFG    1984          NA           NA           NA
##  6 Afghan… AF    AFG    1985          NA           NA           NA
##  7 Afghan… AF    AFG    1986          NA           NA           NA
##  8 Afghan… AF    AFG    1987          NA           NA           NA
```

```
##  9 Afghan… AF    AFG    1988          NA           NA           NA
## 10 Afghan… 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>,
## #   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>, new_sn_f2534 <int>, new_sn_f3544 <int>,
## #   new_sn_f4554 <int>, new_sn_f5564 <int>, new_sn_f65 <int>,
## #   new_ep_m014 <int>, new_ep_m1524 <int>, new_ep_m2534 <int>,
## #   new_ep_m3544 <int>, new_ep_m4554 <int>, new_ep_m5564 <int>,
## #   new_ep_m65 <int>, new_ep_f014 <int>, new_ep_f1524 <int>,
## #   new_ep_f2534 <int>, new_ep_f3544 <int>, new_ep_f4554 <int>,
## #   new_ep_f5564 <int>, new_ep_f65 <int>, newrel_m014 <int>,
## #   newrel_m1524 <int>, newrel_m2534 <int>, newrel_m3544 <int>,
## #   newrel_m4554 <int>, newrel_m5564 <int>, newrel_m65 <int>,
## #   newrel_f014 <int>, newrel_f1524 <int>, newrel_f2534 <int>,
## #   newrel_f3544 <int>, newrel_f4554 <int>, newrel_f5564 <int>,
## #   newrel_f65 <int>
```

As before, each cell tells us the number of people diagnosed with TB in a given country in a given year, but we have much more fine-grained information now. From looking at the help file (`?who`), we know:

- The type of diagnosis:
  - `rel` for relapse
  - `ep` for extraplumonary TB
  - `sn` for cases that could not be diagnosed via a pulmonary smear (smear negative)
  - `sp` for cases that could be diagnosed via a pulmonary smear (smear positive)
- The sex of the group of patients, m/f
- The age categories, which are the same as the categories above

As before, we want variables telling us the country, year, sex, age, and then type of diagnosis (relapse, extrapulmonary, etc.).

We'll first use gather, then separate, then spread to break apart these diagnostic codes. Let's begin! We'll start by gathering across all of the different columns of diagnoses and form them into one variable that records the code corresponding to a given category:

```
who1 <- who %>% gather(code,cases,c(-country,-iso2,-iso3,-year))
who1
```

```
## # A tibble: 405,440 x 6
##    country     iso2  iso3   year code        cases
##    <chr>       <chr> <chr> <int> <chr>       <int>
##  1 Afghanistan AF    AFG    1980 new_sp_m014    NA
```

```
##  2 Afghanistan AF    AFG    1981 new_sp_m014    NA
##  3 Afghanistan AF    AFG    1982 new_sp_m014    NA
##  4 Afghanistan AF    AFG    1983 new_sp_m014    NA
##  5 Afghanistan AF    AFG    1984 new_sp_m014    NA
##  6 Afghanistan AF    AFG    1985 new_sp_m014    NA
##  7 Afghanistan AF    AFG    1986 new_sp_m014    NA
##  8 Afghanistan AF    AFG    1987 new_sp_m014    NA
##  9 Afghanistan AF    AFG    1988 new_sp_m014    NA
## 10 Afghanistan AF    AFG    1989 new_sp_m014    NA
## # … with 405,430 more rows
```

So notice that now that I have a variable `code` that tells me the type of diagnosis: so the first row tells me how many cases of TB were diagnosed in males ages 0-14 via a pulmonary smear in Afghanistan in 1980 (unsurprisingly, since Afghanistan was in the midst of the Soviet invasion then, that value is simply missing).

Before we can continue, we need to fix a small inconsistency in the coding (this is annoying, but incredibly realistic to actually doing data science). If you `View` the data, you'll notice that all of our codes take the form `new_TYPE_SEXAGE` where TYPE is the type of diagnosis (i.e., sp, sn, etc.), and SEXAGE is the sex-age combination, with one exception: `newrel` is the code for relapses, rather than `new_rel`. This means that if we try to separate at the underscore (as we discussed above), it won't work for this subset of cases! To avoid this problem, we'll fix it now. Don't worry about this code, we'll cover in a future lecture on strings:

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

```
## # A tibble: 405,440 x 6
##    country     iso2  iso3   year code        cases
##    <chr>       <chr> <chr> <int> <chr>       <int>
##  1 Afghanistan AF    AFG    1980 new_sp_m014    NA
##  2 Afghanistan AF    AFG    1981 new_sp_m014    NA
##  3 Afghanistan AF    AFG    1982 new_sp_m014    NA
##  4 Afghanistan AF    AFG    1983 new_sp_m014    NA
##  5 Afghanistan AF    AFG    1984 new_sp_m014    NA
##  6 Afghanistan AF    AFG    1985 new_sp_m014    NA
##  7 Afghanistan AF    AFG    1986 new_sp_m014    NA
##  8 Afghanistan AF    AFG    1987 new_sp_m014    NA
##  9 Afghanistan AF    AFG    1988 new_sp_m014    NA
## 10 Afghanistan AF    AFG    1989 new_sp_m014    NA
## # … with 405,430 more rows
```

So now let's work on unpacking `code`. Note that there are three chunks in each value, each separated by an underscore (_). For example, `new_sp_m014` has new (here, all cases are new cases, so this isn't terribly useful), sp (the type of diagnosis), and m014 (the age-sex combination). So we can tell separate to pull apart at the underscore with the following code:

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

## # A tibble: 405,440 x 8
##    country     iso2  iso3   year new   diag  sexage cases
##    <chr>       <chr> <chr> <int> <chr> <chr> <chr>  <int>
##  1 Afghanistan AF    AFG    1980 new   sp    m014      NA
##  2 Afghanistan AF    AFG    1981 new   sp    m014      NA
##  3 Afghanistan AF    AFG    1982 new   sp    m014      NA
##  4 Afghanistan AF    AFG    1983 new   sp    m014      NA
##  5 Afghanistan AF    AFG    1984 new   sp    m014      NA
##  6 Afghanistan AF    AFG    1985 new   sp    m014      NA
##  7 Afghanistan AF    AFG    1986 new   sp    m014      NA
##  8 Afghanistan AF    AFG    1987 new   sp    m014      NA
##  9 Afghanistan AF    AFG    1988 new   sp    m014      NA
## 10 Afghanistan AF    AFG    1989 new   sp    m014      NA
## # … with 405,430 more rows
```

Now we have sexage as one variable. We can use separate() to split this into 2 variables, just as we did above:

```
who4 <- who3 %>% separate(sexage,c("sex","age"), sep=1)
who4

## # A tibble: 405,440 x 9
##    country     iso2  iso3   year new   diag  sex   age   cases
##    <chr>       <chr> <chr> <int> <chr> <chr> <chr> <chr> <int>
##  1 Afghanistan AF    AFG    1980 new   sp    m     014      NA
##  2 Afghanistan AF    AFG    1981 new   sp    m     014      NA
##  3 Afghanistan AF    AFG    1982 new   sp    m     014      NA
##  4 Afghanistan AF    AFG    1983 new   sp    m     014      NA
##  5 Afghanistan AF    AFG    1984 new   sp    m     014      NA
##  6 Afghanistan AF    AFG    1985 new   sp    m     014      NA
##  7 Afghanistan AF    AFG    1986 new   sp    m     014      NA
##  8 Afghanistan AF    AFG    1987 new   sp    m     014      NA
##  9 Afghanistan AF    AFG    1988 new   sp    m     014      NA
## 10 Afghanistan AF    AFG    1989 new   sp    m     014      NA
## # … with 405,430 more rows
```

Note that this highlights the two different ways one can use separate:

1. Separating based on a character, such as an underscore
2. Separating at a specific location, such as after the first character in a cell

Because the function is flexible, we can combine the two syntaxes into one. So now we have a tidy dataset that we could use to analyze, say, how rates of TB differ by age, sex, year, etc.

Appendix: All code used to make this document

```
knitr::opts_chunk$set(echo = TRUE)
knitr::opts_knit$set(root.dir = '~/Dropbox/DATA101')
library(tidyverse)
```

```
library(tidyverse)
setwd("~/Dropbox/DATA101/")
grads <- read_csv(file="Data/Raw/recent-grads.csv")
select(grads,Median,everything()) %>% arrange(desc(Median))
grads <-
read_csv("https://raw.githubusercontent.com/rfordatascience/tidytuesday/maste
r/data/2018/2018-10-16/recent-grads.csv")
library(readxl)
excel_example <- readxl_example("datasets.xls")
read_excel(excel_example)
excel_sheets(excel_example)
read_excel(excel_example, sheet="mtcars")
read_excel(excel_example, sheet=2)
read_excel(excel_example, sheet=4)
read_excel(excel_example, sheet="quakes")
read_csv(file="Data/Raw/wb_gdp.csv")
gdp <- read_csv(file="Data/Raw/wb_gdp.csv")
gdp %>% gather(key=year,value=GDP,c(`1990`:`2017`))
pew <- read_csv(file="Data/Raw/pew.csv")
pew %>% gather(key=education, value=fraction,
c(HSLess,SomeCollege,College,PostGrad))
pew <- read_csv(file="Data/Raw/pew.csv")
pew %>% gather(education, fraction, -Religion)
library(dsr)
table2
table2 %>% spread(key = type, value = count)
table2 %>% spread(key=type, value=count) %>%
mutate(rate=100*(cases/population)) %>%
  ggplot(aes(x=year, y=rate, group=country)) +
  geom_line(aes(color=country)) +
  geom_point(aes(color=country)) +
  scale_x_continuous(breaks=c(1999,2000))
tb <- read_csv(file="Data/Raw/tb.csv")
tb
tb2 <- tb %>% gather(demo, n, c(-iso2,-year))
tb2
tb3 <- tb2 %>% separate(col=demo,into=c("sex","age"), sep=1)
tb3
table3
table3 %>% separate(rate, c("cases","population"), sep="/")
table5
table5 %>% unite(col=yr,c(century,year))
table5 %>% unite(col=yr,c(century,year),sep="")
who
who1 <- who %>% gather(code,cases,c(-country,-iso2,-iso3,-year))
who1
who2 <- who1 %>% mutate(code = stringr::str_replace(code, "newrel",
"new_rel"))
who2
who3 <- who2 %>% separate(code,c("new","diag","sexage"), sep="_")
```

```
who3
who4 <- who3 %>% separate(sexage,c("sex","age"), sep=1)
who4
```