This week's data set heralds from the Museum of Modern Art, which makes their collection data available publically here, though I have already included everything you need in this assignment's repository. The data initially comes in two pieces, with one containing information about the works of art and the other containing information about the artists. As you will see though, some work can be done to clean things up and make the data easier to work with, so this assignment will initially be an exercise in two parts (and problems): cleaning up the data and then gaining some insight from it. **There is a lot of text here, but most of it is explanatory to try to give you plenty of guidance, so don't be intimidated.**

As per usual, you should follow the link here to accept the assignment and get access to the repository:

Assignment link: `https://classroom.github.com/a/oyMuhfhQ`

1. You are starting mostly from scratch here, but I have included template code in the repository that you can copy and run to create the necessary starting tables with data types that will successfully import the data. I will detail the rest of the parts of the clean-up that you should do in the below parts. *For each clean-up step, include the commands you used to perform that operation.*
   (a) (1 point) Create the initial `artists` and `artworks` tables in whatever database you like, and copy the information in from the included CSV files into the appropriate tables. If you are using the template code I provided for you, this should be simple and you need to only include the code you used to copy in the table information. If you made your own tables with your own chosen names, please include the code for them as well.
   (b) (2 points) Perusing through the tables, you will likely notice one thing fairly quickly: there is a lot of duplicated information in the `artworks` table that actually describes the artists (and is also present in the `artists` table. Moreover, if you look around a bit more in the `artworks` table, you will see something unfortunate: some pieces of art have multiple artists, and so all of that information is smashed into compound strings within each column of the `artworks` table. Good table design in relational databases dictates that, in almost all circumstances, you want to have one (and only one) value in each column of a record. In order to accomplish this, we are going to need to generate a third table, which you might call `artwork_artists`, which will have only 2 columns: one containing the artwork `object_ids`, and the other containing the artist `constituent_ids`. If multiple artists worked on the same artwork, then that same artwork `object_id` should simply appear on multiple rows with a different `constituent_id` on each. This is the same idea as to how we combined students and classes earlier in the semester.

   The difficulty lies in how to accomplish this easily. We have all the needed information, but it is buried inside `artworks.constituent_id`. If you have worked with strings in other programming languages, you might want to do something like splitting apart the current `constituent_id` field at the commas, to get each individual constituent id. We can actually do the same in Postgres! The function `string_to_array(some_text, text_to_split_on)` will return an array of the individual pieces of `some_text` split at each occurrence of `text_to_split_on`. The issue then is that you have an array of the constituent id numbers (which are still actually strings). What you really want are these same numbers not in an array, but split across rows. And we've actually already seen a method of doing this, though it only has come up once in the text of Ch 5/6: `UNNEST`. Official documentation here.

Create the `artwork_artists` table such that it has one row for each artwork id and contributing artists.

(c) (1 point) You might think your `artwork_artists` table looks great! You are probably wrong, unfortunately. In theory, each piece of art with a corresponding artist should show up *once* in your created table. Is that the case? Can you figure out why this happened? Fix it by removing the duplicates (or recreating the table with only unique combinations). While you are there, the `constituent_ids` are just increasing integers, so it would be nicer to work with them as actual integers instead of the strings they came in as when you split the original string. (And importantly, this removes any extra spaces that might be still lurking around as well) Change that column data type to be an integer (you may need to use something like `USING constituent_id::INT` at the end of your `ALTER TABLE` statement to get it to go through).

(d) (2 points) You now have three related tables, but no constraints are in place. Alter the tables to add primary keys `object_id` and `constituent_id` to `artworks` and `artists`, respectively. For `artwork_artists`, create a compound primary key that includes both columns, which we can do since we just confirmed that those combinations are all unique. Add in the necessary foreign keys as well to link the `artwork_artists` table columns back to the necessary reference table columns.

(e) (1 point) Now that we have better related the tables, there is really no need for the `artworks` table to still have all the columns containing information that is already in the `artists` table. Remove those 7 columns from `artworks`.

(f) (1 point) If you check for missing information in the `artists` table, you'll realize that there is plenty, but much of it is information that we probably do not have. One thing you might notice though is that there are artists with no recorded `nationality`, but who *do* have something about a nationality mentioned in their bio. These probably could be fixed, but there are a lot of them and no good automatic way to do so, so I'm not going to ask that of you! One other aspect of this table though is that missing or "not yet occurring" birth and death dates are recorded with 0's, which seems potentially problematic if you wanted to do calculations with those columns. Change all instances of 0 in the birth year (`begin_date`) or death year (`end_date`) to `NULL` instead.

(g) (2 points) Moving over to the `artworks` table, take a look at the `date` column, which is when the artwork was supposed created. This column is an absolute mess of formatting. Some are years, some are ranges of years, some are specific days, and everything in between. For the most part though, each record contains **at least one** 4 digit date somewhere amidst the text. While grabbing just this 4 digit date may not be perfectly precise, it would surely be more useful than the current state of things. How can we accomplish that date extraction though? We'll be talking about working with text and regular expressions more soon, but for now, you can extract the first 4 digit date from any string using:

```
substring(date_text FROM '\d{4}')::INT
```

This extracts the part of the string that matches the regular expression `'\d{4}'`, which stands for "four numeric digits adjacent to one another". Add a new column named `date_int` to `artworks` and populate its rows with the first 4 digit date that appears in the row's original `date` column. If you want a quick check that this worked, if I sum all those years in the entire column, I get a value of 263617360.

2. (10 points) Clean up time is over! Now you can proceed to use the data set to answer the following questions. As per usual, show all queries that you used to arrive at your solution.
   (a) What is the title of the piece of art with the most collaborators/contributors at the Museum of Modern Art?
   (b) Who are the top 5 *painters* (those whose work falls within the Painting classification) to have the most artwork acquired by the MoMA?
   (c) What is the median age of a piece of art when the museum acquires it?
   (d) What artist has pieces in the MoMA collection across the greatest number of different classifications?

3. (8 points) Outside of DNA, I think a very good argument can be made for the internet being the greatest treasure trove of information on the planet, and it is important to be able to grab information from existing web resources, either through scraping or use of APIs. In this problem, you'll strive to use both: grabbing related information from two different sources before combining it and outputting it to a CSV file.
   (a) Wikipedia tends to be an excellent source of static tables of information that make a good (and easy) target to scrape. The site here holds information on life expectancies of men and women by country. For this part of the problem, you need to scrape the CIA World Factbook table, which I chose mainly for its recent values and broad global coverage. Scrape all of the information from this table into a data frame for later use.
   (b) Suppose you wanted to investigate whether life expectancy was longer in countries with a coastline as opposed to fully landlocked countries. We have the life expectancy data, but need information about which countries are landlocked. There are a few ways you could go about seeking this, but one option is to use the *restcountries* API, with documentation available here. When this API returns information about a country, one of the included fields is a boolean flag for whether the country

is landlocked or not. Browse the documentation to see your available endpoints and inspect the information you already have available to determine what endpoint might be best to use.

There are different ways the following could be achieved, but I'm going to describe one method below. If you want to use a different method, and it gets you to the same place in the end, that is fine. The available API endpoints mostly seem to query just a single country at a time, but we have many countries we need to look up to determine their landlocked status. One approach would be to define a function which looks up a single country's information and returns only its landlocked status. Then we could map this function to apply to an entire list or column of country names to look up the information for all of them!

i. Start by writing a single function which accepts a country name as input, and which will return a `TRUE` or `FALSE` depending on its landlocked status. If you are rusty at writing functions in R (or if you haven't done so before), the general syntax looks like:

   ```
   funcName <- function (inputVariableName) {
     # your code
     return(outputVariable)
   }
   ```

   which you would then *call* or run using `funcName("Egypt")`. Inside this function, you will need to write the code to paste together the necessary URL from the given API endpoint and your country name. Be aware when doing this that if the country name has a space in it, you can't just blindly insert it into a URL. Instead, I'd recommend using the built-in `URLencode()` function on your country names, which will convert any spaces appropriately. Then you'll need to read the HTML from that URL and parse with `fromJSON` before returning only the single piece of information you want. Note that you want to return just a `TRUE` or `FALSE` here, not an R structure containing that information inside it. Once you've written your function, **test it** in the console! Enter a variety of different country names and ensure they are correctly outputting either `TRUE` or `FALSE`.

ii. What happens to your script if you type in a country name that does not exist? Most likely it will result in an error. This can be problematic, since there may be some entries in our table that do not have corresponding values in the API. What we'd really like to do is attempt to look up the country, and if something goes wrong, just return an `NA` value instead of `TRUE` or `FALSE`. The way this is generally done in R is with a `tryCatch` statement, which has a structure something like below:

   ```
   tryCatch(
     {
       #Code to run, but which may result in an error
     },
   ```

```
  error = function(cond) {
    #Code to run if the above had an error
  }
)
```

In this case, you'll want the original contents of your function to be in that top block of squiggle brackets. All you really need to add in the error block is a `return(NA)`, since that is what you want to happen if the API lookup failed, but you could also print something out if you wanted. For instance, a warning to display on the screen.

Note that, as written, this will "catch" *ALL* errors that might show up in that upper block of code. That may include the one we want to catch, but also any others. So you should generally be careful with `tryCatch` blocks. (There are ways to be more careful about which errors you catch, if you want to read up on it more yourself.)

iii. Now that you have a working function, you just need to use the country names from your existing data frame as the input names to your function. You could do this with loops, or a more direct method is to use the `map` or, in this case, the `map_lgl` function. These functions take two inputs: a list of values and then the *name* of the function you want applied to those values. If you apply your function to your list of country names from your data frame, you should get back a list of `TRUE` and `FALSE` statements. This can take a bit of time, so it might help to add a print statement in your function that prints what country is being looked up, as it can make it easier to see the progress.

(c) Finally, you just need to add a new column to your dataframe to contain these land-locked booleans, and set it equal to your recently computed list that you got from the API. This should be seamless since you computed a value for each country name in your dataframe. Export this dataframe to a CSV named `country_info.csv` and make sure to upload it to GitHub!