Lab 01: Boba Time!

PSTAT 100: Spring 2024 (Instructor: Ethan P. Marzban)

MEMBER 1 (NetID 1) MEMBER 2 (NetID 2)
MEMBER 3 (NetID 3)

April 7, 2024

Required Packages

```
library(ottr)        # for checking test cases (i.e. autograding)
library(pander)      # for nicer-looking formatting of dataframe outputs
library(reshape2)    # for 'melting' data frames
library(tidyverse)   # for graphs, data wrangling, etc.
```

Logistical Details

Logistical Details

- This lab is due by **11:59pm on Wednesday, April 12, 2024**.
- Collaboration is allowed, and encouraged!
  - If you work in groups, list ALL of your group members' names and NetIDs (not Perm Numbers) in the appropriate spaces in the YAML header above.
  - Please delete any "MEMBER X" lines in the YAML header that are not needed.
  - No more than 3 people in a group, please.
- Ensure your Lab properly renders to a `.pdf`; non-`.pdf` submissions will not be graded and will receive a score of 0.
- Ensure all test cases pass (test cases that have passed will display a message stating `"All tests passed!"`)

Lab Overview and Objectives

Welcome to our first "official" PSTAT 100 Lab! In this lab, we will cover the following:

- Reading in data of different filetypes
- Cursory introduction to databases
- An introduction to joining and merging dataframes
- Manipulating and Tidying data using the `tidyverse` package.

How do Labs Work?

Labs assignments are designed to be largely doable within the 50-minutes Lab Sessions on Mondays. Having said that, we would much rather you work through the labs carefully and completely without feeling the need to rush things, which is why the Labs are not due until Wednesdays.

Typically, lab assignments will focus on the programming side of the course. This will often entail extending concepts discussed in Lecture, and occasionally involve introducing some new concepts as well. Please keep in mind that Lab material is potentially testable on the In-Class Assessments.

You are highly encouraged to ask your TA questions during Section and/or Office Hours!

Part I: Reading In Data

Filetypes and File Extensions

Recall that data is most often stored in tabular form. Tables, as we colloquially refer to them, however, contain a lot of extraneous formatting information (e.g. border widths, cell padding, etc.), that don't actually provide all that much information when it comes to computation and/or analysis. As such, when storing data in a **file**, we often strip down the data to its most basic forms.

We really only need to specify two things when structuring a table: how rows are separated, and how cells within each row are separated. In most files, new rows of a table are separated by a new row in the raw data file. There is, however, a fair amount of flexibility in how the separation of *cells* in a dataset are indicated.

Indeed, how cells (i.e .values) are separated in a given file is what is referred to as the **filetype**. Two popular filetypes are:

- **Comma-Separated Values** (CSV): values are separated by commas
- **Tab-Separated Values** (TSV): values are separated by tabs

When saving our file, we use the **file extension** to indicate the filetype: for instance, a file called `my_data.csv` will be stored as a CSV file, whereas a file called `my_data.tsv` will be stored as a TSV file.

Reading In Data

Storing data in raw files is all well and good, but most often we'd like to read the data contained in a file into R! There are actually several functions we can use in R to read in files. The most general one is the `read.table()` function.

! Question 1

Look up the help file for `read.table()` (consult the optional Lab 0 if you need help figuring our how to do this!). Explain, in words, how you specify the filetype of a file when reading it into R using `read.table()` (i.e. what argument controls the filetype? what sorts of values can this argument take?)

Solution:
Replace this line with your answer

There is no autograder for this question; your TA will manually check that your answers are correct.

Because CSV files are so common, there is actually a separate function in R called `read.csv()`, used to read in CSV files. (We encourage you to look up the help file for this function as well!)

Note that both the `read.table()` and `read.csv()` functions have a `file` argument. From the help file, we can see that the `file` argument is:

> "the name of the file which the data are to be read from. Each row of the table appears as one line of the file. If it does not contain an absolute path, the file name is relative to the current working directory, `getwd()`.

What this means is that running `read.csv("my_data.csv")` will attempt to read in a file called `my_data.csv` that is located in your current working directory. If no such file exists, R will return an error.

The reason I highlight this is because it is common to include all relavant data files in a subfolder of your working directory. That is, it is common to have a working directory file structure like:

```
+--- Main Folder
      ¦--- analysis.R
      |--- data
            |--- my_file.csv
```

In this case, if you are in the `analysis.R` file and your working directory is the `Main Folder`, then to read in the `my_file.csv` file you should use `read.csv("data/my_file.csv")`.

Tip

To set the working directory in R, click on the `Session` dropdown menu at the top of `RStudio`, navigate to the `Set Working Directory` submenu, and select the desired option.

Part II: Introduction to Databases

As you might imagine, a single file isn't always enough to capture all of the data necessary for a particular analysis. Sometimes, we need to consider a **database**: for the purposes of this class, we can think of a database as simply being a collection of dataframes, all related in some way.

As a simple example, let's don the persona of a financial analyst working for *GauchoBubble*, a new up-and-coming boba shop. We are particularly interested in examining the revenue generated by sales on a particular day at *GauchoBubble*. Available to us is a database called `GauchoBubble`, which contains two files (each containing a single dataframe): `orders.csv` (which contains information on the actual orders placed), and `inventory.csv` (which contains information on the products sold at *GauchoBubble*).

The variables in the `orders.csv` dataframe are:

- `order_no`: an identifier for each order placed (orders containing multiple items are split across multiple rows, with only one order in each row)
- `item_no`: a unique identifier of each item sold at *GauchoBubble*

The variables in `inventory.csv` are:

- `ITEM_NO`: a unique identifier of each item sold at *GauchoBubble*
- `DESCRIP`: a verbal description of each item sold at *GauchoBubble*
- `PPU`: the price-per-unit of each item sold at *GauchoBubble*

Here are the first few rows of both the `orders` and `inventory` dataframes:

ORDERS:

order_no	item_no
1	T02
2	T02
2	B01
2	T01
2	J01
3	B02

INVENTORY:

ITEM_NO	DESCRIP	PPU
T00	Taro; No Topping	5.25
T01	Taro; Boba	5.75
T02	Taro; Lychee	5.75
T03	Taro; Egg Pudding	5.75
J00	Jasmine Milk; No Topping	6.25
J01	Jasmine Milk; Boba	6.75

! Question 2

Read in the `orders.csv` file and store it in a variable called `orders`. Then, read in the `inventory.csv` file and store it in a variable called `inventory`.

Solution:

```
## replace this line with your code
```

Answer Check:

```
# DO NOT EDIT THIS LINE
invisible({check("tests/q2.R")})
```

```
Test q1 failed:

object 'orders' not found
```

Database Terminology

Databases have a rich and complex theory. Much of the modern-day theory surrounding databases is attributed to Edgar F. Codd, a scientist who developed a framework surrounding **relational databases** while working at IBM, in the 60's and 70's. As this is not meant to be a class in database management, we'll only scratch the surface of databases - If you're curious to read more, much of Codd's work is summarized in his book titled "The Relational Model for Database Management".

Here is the gist of things. Again, we can think of a **database** as being a collection of **relations** (i.e. dataframes/tables), typically related in some fashion. Each relation in a database has a unique **primary key**, which is the smallest set of variables needed to uniquely determine the rows of the relation For example, the primary key of the `INVENTORY` relation is {`ITEM_NO`}, as each item number corresponds to a unique row in the `INVENTORY` relation.

However, the same cannot be said about the `ORDERS` relation. For example, if I tell you "show me the row of the `ORDERS` relation that has `item_no T02`", you won't be able to do so because there are *multiple* rows with `item_no` value equal to `T02`. As such, we actually need *both* `order_no` and `item_no` to uniquely identify the rows of the `ORDERS` relation, and we set the primary key to be the set {`order_no`, `item_no`} (primary keys consisting of more than one variable are sometimes referred to as **compound keys**).

A **foreign key** is a key (or set of keys) in one relation that corresponds to the primary key of a another relation. Note that it is allowed for foreign keys to point to only one of the keys in a compound primary key. For example, note that the `ITEM_NO` column from the `INVENTORY` table corresponds to the `item_no` column in the `ORDERS` table: hence, we would say that the `INVENTORY:ITEM_NO` key (note the syntax `table_name:column_name`) is a foreign key that points to `ORDERS:item_no`.

Joins

When working with databases, it is sometimes necessary/desired to *combine* one or more dataframes in some way. For example, take the `ORDERS` dataframe: it currently only includes the unique *identifier* of each product, and not the actual *name* of the product. Wouldn't it be nice to include the product names with each of the orders?

I think most of us can do this combination manually fairly easily:

order_no	item_no	DESCRIP
1	T02	Taro; Lychee
2	T02	Taro; Lychee
2	B01	Original Milk; Boba
2	T01	Taro; Boba
2	J01	Jasmine Milk; Boba
3	B02	Original Milk

But, how can we make R do this for us? To answer this question, let's break down how we did this combination manually. For example, focusing on the first row:

- We first used the ORDERS dataframe to look up the item_no of the first order.
- We then found that item_no in the INVENTORY dataframe.
- Finally, we looked up the DESCRIP of the corresponding item_no, and added this into the third column of our ORDERS dataframe.

This is an example of what is known as a **join** (also known as a **merge**), in which we combine information from multiple tables. The key idea is that we join on/along a foreign key relationship!

There are two main classes of joins: **mutating joins** and **filtering joins**. For now, let's restrict our considerations to mutating joins. As an illustrative example, we'll consider the following two simple tables:

x

ind_x	var_x
one	a
two	b
two	c
three	c
four	d

y

ind_y	var1_y	var2_y
a	cat	piano
b	dog	piano
c	rabbit	violin
e	snake	viola

We can code these tables into R as dataframes:

```
x <- data.frame(
  ind_x = c("one", "two", "two", "three", "four"),
  var_x = c("a", "b", "c", "c", "d")
)

y <- data.frame(
  ind_y = c("a", "b", "c", "e"),
  var1_y = c("cat", "dog", "rabbit", "snake"),
  var2y = c("piano", "piano", "violin", "viola")
)
```

Left join

Perhaps the most commonly-used join is that of a **left join**, which is used to add additional information to a table. The syntax of a left join is:

```
left_join(
  x,
  y,
  by = join_by(var_x == ind_y)
)
```

```
  ind_x var_x var1_y  var2y
1   one     a    cat  piano
2   two     b    dog  piano
3   two     c rabbit violin
4 three     c rabbit violin
5  four     d   <NA>   <NA>
```

Note that, by default, `left_join(x, y, ...)` includes all rows of `x`, but not necessarily all rows of `y`. This means that the output of `left_join(x, y, ...)` will (almost) always have the same number of rows as `x`.

Caution

Left joins are **not** symmetric! That is: `left_join(x, y)` will not produce the same output as `left_join(y, x)`, as is indicated below:

```
left_join(
  y,
  x,
  by = join_by(ind_y == var_x)
)
```

```
  ind_y var1_y  var2y ind_x
1     a    cat  piano   one
2     b    dog  piano   two
3     c rabbit violin   two
4     c rabbit violin three
5     e  snake  viola  <NA>
```

Inner Joins, Right Joins, and Full Joins

There are three other kinds of mutating joins: **inner joins**, **right joins**, and **full joins**. All of these are similar to left joins in that they are used to augment existing dataframes with information sourced from another dataframe; the key difference is in which rows are kept post-join.

Right joins keep all rows in `y`:

```
right_join(
  x,
  y,
  by = join_by(var_x == ind_y)
)
```

```
  ind_x var_x var1_y  var2y
1   one     a    cat  piano
2   two     b    dog  piano
3   two     c rabbit violin
4 three     c rabbit violin
5  <NA>     e  snake  viola
```

Inner joins keep only rows present in both `x` and `y`:

```
inner_join(
  x,
  y,
  by = join_by(var_x == ind_y)
)
```

```
  ind_x var_x var1_y  var2y
1   one     a    cat  piano
2   two     b    dog  piano
3   two     c rabbit violin
4 three     c rabbit violin
```

Full joins keep rows present in either `x` or `y`:

```
full_join(
  x,
  y,
  by = join_by(var_x == ind_y)
)
```

```
  ind_x var_x var1_y  var2y
1   one     a    cat  piano
2   two     b    dog  piano
3   two     c rabbit violin
4 three     c rabbit violin
5  four     d   <NA>   <NA>
6  <NA>     e  snake  viola
```

Let's return to our *GauchoBubble* database. Suppose we want to join the `OBJECT` and `INVENTORY` dataframes to include the order information as well as the item descriptions and price-per-units, to

obtain a new dataframe with columns `order_no`, `item_no`, `DESCRIP`, and `PPU`, **in that order**. In other words, we want to create a table whose first few rows look like this:

order_no	item_no	DESCRIP	PPU
1	T02	Taro; Lychee	5.75
2	T02	Taro; Lychee	5.75
2	B01	Original Milk; Boba	5.75
2	T01	Taro; Boba	5.75
2	J01	Jasmine Milk; Boba	6.75
3	B02	Original Milk	5.75

! Question 3

A) Which type/s of joins could be used to accomplish this? (If the desired table could be created using several choices of joins, be sure to list them all.)

Solution, part (A):
Replace this line with your answers

B) Now, perform the join. (If you believe there are multiple potential joins we could use, pick one; that is, you do not have to redo the join multiple times)

Solution, part (B) :

```
## replace this line with your code
```

Answer Check:

```
# DO NOT EDIT THIS LINE
invisible({check("tests/q3.R")})
```

```
Test q3 failed:

object 'orders_joined' not found
```

Part III: Transforming and Grouping Dataframes

The majority of our considerations up until now have been pertaining to the structuring of dataframes. Structure, as we know, is only half of the picture - we'd like to start analyzing our data!

`R` has many functions dedicated to the transformation, aggregation, and analysis of dataframes. Many of these can be found in what is known as the **tidyverse**. The tidyverse is technically a collection of several `R` packages: a full list of included packages can be found on the official tidyverse website. The tidyverse is primarily used to clean, manipulate, and tidy datasets. In this section of the lab, we will focus on the data **transformation** functionality of the tidyverse.

I always find examples to be illustrative! As such, here is a (mock) dataset, containing the final scores of 10 students in a fictitious PSTAT course, for illustrative purposes:

```
pstat_grades <- data.frame(
  student_id = 1:10,
  major = c("PSTAT", "PSTAT", "PSTAT", "PSTAT", "PSTAT",
            "Comm", "Comm", "Comm", "Econ", "Econ"),
  final_grade = c(87.2, 89.2, 92.5, 97.7, 40.1, 85.7, 95.5, 77.1, 82.1, 99.1)
)

pstat_grades %>% pander()
```

student_id	major	final_grade
1	PSTAT	87.2
2	PSTAT	89.2
3	PSTAT	92.5
4	PSTAT	97.7
5	PSTAT	40.1
6	Comm	85.7
7	Comm	95.5
8	Comm	77.1
9	Econ	82.1
10	Econ	99.1

Filtering and Rearranging Rows

Suppose we want to filter out rows of a dataset that do not match some constraint. For instance, say we only want to access the rows of the `pstat_grades` dataframe corresponding to students in the PSTAT major. We can do so using the `filter()` function:

```
filter(pstat_grades,
       major == "PSTAT")
```

```
  student_id major final_grade
1          1 PSTAT        87.2
2          2 PSTAT        89.2
3          3 PSTAT        92.5
4          4 PSTAT        97.7
5          5 PSTAT        40.1
```

We can perform more complex filtering by utilizing the logical connectors available to us in `R` (i.e. `&` and `|`). For example, to display only the rows of PSTAT majors scoring above 90, we would run

```
filter(pstat_grades,
       (major == "PSTAT") & (final_grade > 90))
```

```
  student_id major final_grade
1          3 PSTAT        92.5
2          4 PSTAT        97.7
```

Note that the `filter()` function does not change the order of rows. If we wanted to change the order of rows in a dataset, we can use the `arrange()` function. For instance, to rearrange the rows of the `pstat_grades` dataset to be in descending order of `final_grade`, we would use

```
arrange(pstat_grades,
        desc(final_grade))
```

```
   student_id major final_grade
1          10  Econ        99.1
2           4 PSTAT        97.7
3           7  Comm        95.5
4           3 PSTAT        92.5
5           2 PSTAT        89.2
6           1 PSTAT        87.2
7           6  Comm        85.7
8           9  Econ        82.1
9           8  Comm        77.1
10          5 PSTAT        40.1
```

We can actually arrange based on non-numerical columns as well:

```
arrange(pstat_grades,
        desc(major))
```

```
   student_id major final_grade
1           1 PSTAT        87.2
2           2 PSTAT        89.2
3           3 PSTAT        92.5
4           4 PSTAT        97.7
5           5 PSTAT        40.1
6           9  Econ        82.1
7          10  Econ        99.1
8           6  Comm        85.7
9           7  Comm        95.5
10          8  Comm        77.1
```

Can you tell what criterion `R` is using when it rearranges the columns based on a non-numerical column?

! Question 4

Let's return to the `orders_joined` dataframe you created above, by merging the `orders` and `inventory` dataframes. Filter the dataframe to only include rows corresponding to orders of Original Milk Tea with Egg Pudding topping; additionally, sort the rows in descending order of Order Number. Store this in a variable called `og_milk_tea_ep`, and display the first 4 rows of the `og_milk_tea_ep` dataframe.

Solution:

```
## replace this line with your code
```

Answer Check:

```
# DO NOT EDIT THIS LINE
invisible({check("tests/q4.R")})
```

```
Test q4 failed:

object 'og_milk_tea_ep' not found
```

The Pipe Operator

Let's take a quick interlude to discuss what is (arguably) one of the most important operators in `R`: the **pipe operator** (`%>%`).

I like to think of the pipe operator as being akin to a composition of two functions (remember that from Precalculus?) Recall that the composition of two functions $f()$ and $g()$ is notated

$$(f \circ g)(x) := f(g(x))$$

We can see that using the composition operator can help avoid the headache of multiple parentheses.

The pipe operator works much in the same way: it allows us to "unpack" expressions that would otherwise involve series of nested inputs. Loosely speaking, the pipe operator squeezes (pipes) what is on the left hand side to the first argument of whatever is on the RHS. For example:

```
c(1, 2, 3) %>% sum()
```

```
[1] 6
```

is completely equivalent to

```
sum(c(1, 2, 3))
```

```
[1] 6
```

We can get fancy, and use multiple pipe operators in succession:

```
pstat_grades %>%
  filter(major %in% c("PSTAT", "Econ")) %>%
  nrow()
```

```
[1] 7
```

We can see that the pipe operator has an additional advantage over just making our code more readable: it also mimics the workflow that we typically envision while running our code. For instance, the code above is returning the number of students in the `pstat_grades` dataframe whose major was either PSTAT or Econ. Using the pipe operator (like we did) makes our workflow clear:

- take the `pstat_grades` dataframe,
- `filter` out rows to leave only those with `major` value equal to either `"PSTAT"` or `"Econ"`,
- and count the number of rows of the resulting dataframe.

Column-wide Operations

Filtering and arranging can be thought of as row-wide application; that is to say, the `filter()` and `arrange()` functions work by operating on the *rows* of a dataframe. There are a handful of column-wide operations that are of use to us as well. We'll return to these periodically throughout the course - for now, I'd like to introduce you to the `select()` function.

As the name suggests, the `select()` function is used primarily to *select* columns of a dataframe according to a set of specified criteria. I find the `select()` column most useful when we want to select a series of columns by name. Recall that it is quite easy to select a single column of a dataframe, using the `$` operator:

```
pstat_grades$final_grade
```

```
 [1] 87.2 89.2 92.5 97.7 40.1 85.7 95.5 77.1 82.1 99.1
```

If we wanted to select *multiple* columns by name, however, we cannot simply use the `$` operator. (If we knew the column indices of the desired columns we could use indexing/slicing, however with very large datasets it becomes unrealistic to suppose we know the column indices of *any* desired column by name.) We can, however, use `select()`:

```
pstat_grades %>%
  select(major, final_grade)
```

```
   major final_grade
1  PSTAT        87.2
2  PSTAT        89.2
3  PSTAT        92.5
```

```
4  PSTAT        97.7
5  PSTAT        40.1
6   Comm        85.7
7   Comm        95.5
8   Comm        77.1
9   Econ        82.1
10  Econ        99.1
```

! Question 5

Let's return to the `orders_joined` dataframe. Select only the `order_no` and `PPU` columns; store these in a new dataframe called `order_no_ppu`.

Solution:

```
## replace this line with your code
```

Answer Check:

```
# DO NOT EDIT THIS LINE
invisible({check("tests/q5.R")})
```

```
Test q5 failed:

object 'order_no_ppu' not found
```

Grouping a Dataframe

Finally, let's talk about what is perhaps one of the most important dataframe operations: **grouping**.

As an example, let's (again) return to our `pstat_grades` dataframe. Suppose we want to compute the average (mean) final grade within each of the 3 majors represented in the dataset. That is, we'd like to create a table that contains the average final grade of PSTAT students, the average final grade of Communications students, and the average final grade of Economics students.

There are many ways we could do this, one of which includes looping through the different majors. However, the "cleanest" (i.e. most succinct) way to achieve our desired goal is to *group* by major. Admittedly, grouping is a somewhat abstract concept, largely because the `group_by()` function operates almost entirely internally. For example:

```
pstat_grades %>%
  group_by(major)
```

```
# A tibble: 10 x 3
# Groups:   major [3]
   student_id major final_grade
        <int> <chr>       <dbl>
```

```
 1          1 PSTAT        87.2
 2          2 PSTAT        89.2
 3          3 PSTAT        92.5
 4          4 PSTAT        97.7
 5          5 PSTAT        40.1
 6          6 Comm         85.7
 7          7 Comm         95.5
 8          8 Comm         77.1
 9          9 Econ         82.1
10         10 Econ         99.1
```

It doesn't really look like anything has changed! But, that is only because the many chagnes that have taken place took place *behind the scenes*: now, the dataframe is charged and ready to apply functions across groups. For example, to compute the average final grades across majors, we can use:

```
pstat_grades %>%
  group_by(major) %>%
  summarise(avg_grade = mean(final_grade))
```

```
# A tibble: 3 x 2
  major avg_grade
  <chr>     <dbl>
1 Comm       86.1
2 Econ       90.6
3 PSTAT      81.3
```

Appreciating the Pipe

By the way, I'd like to take a moment and have us all appreciate the heavy lifting the pipe operator is doing in the above command! We *could* technically have written the same code without the pipe operator as:

```
summarise(group_by(pstat_grades, major), avg_grade = mean(final_grade))
```

```
# A tibble: 3 x 2
  major avg_grade
  <chr>     <dbl>
1 Comm       86.1
2 Econ       90.6
3 PSTAT      81.3
```

But notice how clunky and awkward that code syntax is - there are a lot of parentheses flying about, and it's a little difficult to see exactly how we can break the code across lines. Additionally, as mentioned previously, the order in which the functions are being applied has been mixed up a bit.

! Question 6

Let's return to the `orders_joined` dataframe. Compute the number of each item (e.g. Taro with Boba, Jasmine Milk with Lychee, etc.) that was sold. (For practice, use the pipe operator.) Store this table in a variable called `num_each_sold`, and ensure that the column names are `description` and `num_units` (as a hint, look up the help file for the `rename()` function!). Then, display the first three rows of your `num_each_sold` table and check that they look like this:

description	num_units
Jasmine Milk; Boba	14
Jasmine Milk; Egg Pudding	7
Jasmine Milk; Lychee	13

Solution:

```
## replace this line with your code
```

Answer Check:

```
# DO NOT EDIT THIS LINE
invisible({check("tests/q6.R")})
```

```
Test q6 failed:

object 'num_each_sold' not found
```

Finally, let's close out by calculating what we set out to calculate: the total amount of revenue generated by sales of each item type.

! Question 7

Obtain a three-column table with column names `item`, `num_units`, and `tot_rev` that displays the number of units sold of each item type along with the total revenue generated by that item type. Store this in a dataframe called `total_revenue`; display the first three rows of your `total_revenue` dataframe, and check that they match the following:

item	num_units	tot_rev
Jasmine Milk; Boba	14	73.5
Jasmine Milk; Egg Pudding	7	40.2
Jasmine Milk; Lychee	13	74.8

Solution:

```
## replace this line with your code
```

Note: this is not the only way to solve this problem.

Answer Check:

```
# DO NOT EDIT THIS LINE
invisible({check("tests/q7.R")})
```

```
Test q7 failed:

object 'total_revenue' not found
```

Part IV: Data Tidying

We got a bit lucky in that both tables in our *GauchoBubble* database were tidy. As we saw in lecture last week, however, not all datasets are tidy! We'll close off this lab by talking about some of the `R` functions commonly used for data tidying.

Firstly, recall that a dataset is said to be **tidy** if:

1) Each variable forms a column
2) Each observation forms a row
3) Each type of observational unit forms a table

We also saw that there are 4 common ways data fails to be tidy:

1) Column headers are values, not variable names
2) Multiple variables are stored in one column
3) Variables are stored in both rows and columns
4) A single observational unit is stored in multiple tables

In Lecture 2, we discussed the theory behind some of the operations used to "fix" the above messes: **melting**, **pivoting**, and **merging**. We talked extensively about merging (remember that this is the same thing as joining!) at the start of this lab; let's talk a bit about melting and pivoting.

Melting a Dataframe

As a simple example, let's consider the following (mock) dataset which displays the number of cases of a particular disease, tracked over time and across two counties:

```r
disease_data <- data.frame(
  County = c("County A", "County B"),
  Day1 = c(109, 150),
  Day2 = c(106, 149),
  Day3 = c(104, 140),
  Day4 = c(102, 145),
  Day5 = c(100, 137)
)

disease_data %>% pander()
```

County	Day1	Day2	Day3	Day4	Day5
County A	109	106	104	102	100
County B	150	149	140	145	137

Here, the observational units are the counties, and the variables are day and number of cases (no observation is fully identified without specifying measurements for each of these two attributes!). Hence, we can see that the column names of this dataframe are values, not variable names: that is, we have case (1) above, and our data is not tidy.

To tidy the dataframe, we can *melt* it:

```r
disease_data %>%
  melt(
    id.vars = "County",        # which variables to retain for each row?
    variable.name = "Day",     # name for the variable containing column names
    value.name = "Num.Cases"   # name for the variable containing values
  ) %>%
  pander()
```

County	Day	Num.Cases
County A	Day1	109
County B	Day1	150
County A	Day2	106
County B	Day2	149
County A	Day3	104
County B	Day3	140
County A	Day4	102
County B	Day4	145
County A	Day5	100
County B	Day5	137

Alternatively, we can use the `pivot_longer()` function:

```r
disease_data %>%
  pivot_longer(
    cols = !County,
    names_to = "Day",
    values_to = "Num.Cases"
  ) %>%
  pander()
```

County	Day	Num.Cases
County A	Day1	109
County A	Day2	106
County A	Day3	104
County A	Day4	102
County A	Day5	100
County B	Day1	150
County B	Day2	149
County B	Day3	140
County B	Day4	145
County B	Day5	137

Note that we obtain the mostly same output using either `melt()` or `pivot_longer()`, with only the order of rows differing across the two.

! Question 8

Suppose we really want to make the outputs of `melt()` and `pivot_longer()` *exactly* match. Look up the help file for `pivot_longer()`, and figure out which additional arguments need to be specified in order to make the outputs of `melt()` and `pivot_longer()` match exactly. Then, pipe the `disease_data` dataframe into a call to `pivot_longer()` with these additional arguments specified, and visually confirm that this new output is the same as the output obtained using `melt()` above.

Solution:

```r
## replace this line with your code
```

Answer Check:
There is no autograder for this question; your TA will manually check that your answers are correct.

Submission Details

Congrats on finishing the first PSTAT 100 lab! Please carry out the following steps:

Submission Details

1) Check that all of your tables, plots, and code outputs are rendering correctly in your final `.pdf`.

2) Check that you passed all of the test cases (on questions that have autograders). You'll know that you passed all tests for a particular problem when you get the message "All tests passed!".

3) Submit **ONLY** your `.pdf` to Gradescope. Make sure to match pages to your questions - we'll be lenient on the first few labs, but after a while failure to match pages will result in point penalties.