Lab 01: Welcome to the `tidyverse`! SOLUTIONS

PSTAT 100, Summer Session A 2025 with Ethan P. Marzban

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

June 24, 2025

Required Packages

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

Logistical Details

Logistical Details

- This lab is due by **11:59pm on Wednesday, June 25, 2025**.
- 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

- Manipulating and summarizing data using the `tidyverse` package.

How do Labs Work?

You are encouraged to complete as much of the lab as possible during our biweekly 50-minute Lab Sessions. 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 rectangular data is most often stored in tabular form. Tables, as we colloquially refer to them, however, contain extraneous formatting information (e.g. border widths, cell padding, etc.), that don't actually provide 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.

Filetype refers to how how cells (i.e .values) are separated in a given file. 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:
It is the `sep` argument that allows us to differentiate different filetyeps. Specifically, we pass the separator value to the `sep` argument; e.g. `sep = ","` for .CSV files, `sep = "\t"` for .TSV files, etc.

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.

Cats Dataset

In lecture, we briefly discussed a dataset collected by several UK veterenarians. From the **data dictionary** (accessible, along with the raw data, at this site), the dataset contains the following variables:

- `Cat ID`: a unique identifier for each cat
- `RSD`: confirmed/unconfirmed cases of recurrent seizure status
- `Epilepsy`: whether the given cat had epilepsy or not
- `Breed`: the breed of the cat
- `Age`: age of the cat, in years
- `Sex`: sex of the cat
- `Neutered`: whether the cat was neutered or not
- `Insured`: whether the cat was insured or not

! Question 2

The above-described dataset is included in a file called `cats_data.csv`, located in the `data` subfolder. Read in the `cats` dataset, and assign it to a variable called `cats`. (We will use this dataset later in the lab.)

Solution:

```
cats <- read.csv("data/cats_data.csv")
```

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

```
All tests passed!
```

Part II: Transforming and Grouping Dataframes

`R` has many functions dedicated to the transformation, aggregation, and analysis of dataframes. Many of these can be found in in the **`dplyr`** package (often pronounced like 'd-plier'), which is part of the so-called **`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), and 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:

```r
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

```r
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

```r
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:

```r
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 3

Let's return to the `cats` dataframe we created above. Filter the dataframe to only include information only from Domestic Medium Hair (note the spelling and the spaces!) cats; additionally, sort the rows in descending order of age Store this in a variable called `dmh_sorted_by_age`, and display the first 4 rows of the `dmh_sorted_by_age` dataframe.

Solution:

```r
dmh_sorted_by_age <- arrange(
  filter(cats, Breed == "Domestic Medium Hair"),
  desc(Age)
)
```

Answer Check:

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

```
All tests passed!
```

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** (`%>%`), more formally known as the **`magrittr` 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:

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

```
[1] 6
```

is completely equivalent to

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

```
[1] 6
```

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

```r
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.

Note

There are actually *two* famous `R` pipe operators: the `magrittr` pipe (`%>%`, named after the `magrittr` package in which it is found) and the base pipe (`|>`), which can be used without loading in any additional packages. (Technically, the `magrittr` package contains *even more* pipes, but these are less commonly-used.)

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:

```r
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()`:

```r
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 4

In our cats dataset, we aren't particularly concerned with the RSD (recurrent seizure disorder) status of each cat. Additionally, the `Insured` column doesn't seem to have much information. As such, select all columns except the `RSD` and `Insured` columns, and assign the new dataframe to a variable called `cats_simplified`.

Solution:

```
cats_simplified <- cats %>% select(!c(RSD, Insured))
```

Answer Check:

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

```
All tests passed!
```

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 changes 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:

```r
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:

```r
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 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 5

Let's return to the original `cats` dataframe. Compute the median age of each breed of cat represented in the dataset. (For practice, use the pipe operator.) Store this table in a variable called `median_ages`, and ensure that the column names are `Breed_Name` and `Median_Age` (as a hint, look up the help file for the `rename()` function!). **Important:** do not try to resolve any issues pertaining to missing values; we'll handle those in the next few questions.

Solution:

```r
median_ages <- cats %>%
  group_by(Breed) %>%
  summarise(Median_Age = median(Age)) %>%
  rename(Breed_Name = Breed)
```

Here are the first few rows:

```r
head(median_ages, 10)
```

```
# A tibble: 10 x 2
   Breed_Name           Median_Age
   <chr>                     <dbl>
 1 *****                     0.153
 2 -                         2.26
 3 ???                       0.233
 4 Abbysian                  8.14
 5 Abysinnian Cross         17.7
 6 Abyssinian               NA
 7 Abyssinian X             NA
 8 Abyssinian X Bengal      10.2
 9 Abyssinian x Siamese      3.79
10 Abyssinian/Tabby          5.18
```

Answer Check:

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

```
All tests passed!
```

This reveals that there are missing values in the both the `Age` and `Breed` columns.

! Question 6

How are missing values encoded in the `Breed` column? (In other words, what symbols are being used to indicate that a particular `Breed` value is missing?) Are there any missing values in the `Age` column? How can you tell?

Solution:
From a cursory glance at our `median_ages` table, it seems that missing values in the `Breed` column are indicated by a series of different symbols, including `*****`, `-`, and `???`. We can also tell that there were missing values in the `Age` column, since, by default, the `median()` function in `R` returns a value of `NA` when any values are missing.

We will talk more about missing values later in this course. For now, let's address the missingness in the `Age` column:

! Question 7

Re-do the computation from Question 5 above (i.e. computing the median age of each breed of cat), but now exclude any missing `Age` values from the computation. Assign this to a new variable called `median_ages_no_na`. **Hint:** Look up the help file for the `median()` function; is there an argument that might help us here?

Solution:

```
median_ages_no_na <- cats %>%
  group_by(Breed) %>%
  summarise(Median_Age = median(Age, na.rm = T)) %>%
  rename(Breed_Name = Breed)
```

Here are the first few rows:

```
head(median_ages_no_na, 10)
```

```
# A tibble: 10 x 2
   Breed_Name           Median_Age
   <chr>                     <dbl>
 1 *****                     0.153
 2 -                         2.26
 3 ???                       0.233
 4 Abbysian                  8.14
 5 Abysinnian Cross         17.7
 6 Abyssinian                6.83
 7 Abyssinian X              4.89
 8 Abyssinian X Bengal      10.2
 9 Abyssinian x Siamese      3.79
10 Abyssinian/Tabby          5.18
```

Answer Check:

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

```
All tests passed!
```

The missingness in the `Breed` column is a little more difficult to handle, so we'll save that endeavor for a later time in the course.

! Question 8

Does it appear that, on average, epilepsy is more common among older cats than younger cats? **Hint:** Think about how you can create a numerical summary of the dataset (using `tidyverse` functions) to help answer this question. Make sure you provide some sort of justification for your answer! **Later in the course,** we will work on providing more formal justification for our claims.

Solution:
Here's the general idea: let's compute the median age among epileptic cats and non-epileptic cats, and compare these values:

```
cats %>%
  group_by(Epilepsy) %>%
  summarise(Median_Age = median(Age, na.rm = T))
```

```
# A tibble: 3 x 2
  Epilepsy Median_Age
  <chr>         <dbl>
1 Case           7.67
2 Non-case       4.43
3 <NA>          11.2
```

Though there appear to be some cats whose epilepsy condition were not recorded, it does appear that the median age among epileptic cats is much higher than the median age among non-epileptic cats. Hence, it does appear (upon first glance) that epilepsy is more common among older cats.

Of course, to make this statement more rigorous (and to give it some more statistical justifcation), we need more sophisticated tools - we will discuss these tools later in the course.

Submission Details

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

ℹ 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.