```r
# DTA250
# Spring 2024

# As we learned before, functions are a way to encapsulate a piece of code that
# can be reused.
# In this exercise, we will use functions along with the dplyr library to
# manipulate data.

# Load the tidyverse library
library(tidyverse)

# Vector Functions ----
# Assume you have the following tibble
df <- tibble(
    a = rnorm(5),
    b = rnorm(5),
    c = rnorm(5),
    d = rnorm(5),
)

# TODO
# Mutate a new column and call it a_scaled. In this column, scale the values of
# column a to have a mean of 0 and a standard deviation of 1.
# Use the equation (a - min(a, na.rm=TRUE)) / (max(a, na.rm=TRUE) - min(a, na.rm=TRUE))
# to scale the values.

df |>
    mutate(a_scaled = (a - min(a, na.rm=TRUE)) / (max(a, na.rm=TRUE) - min(a, na.rm=TRUE))

# Now you want to do the same for the rest of the columns.
# However, you don't want to repeat the same code for each column.

##  Writing a function ----

# TODO
# Create a function called scale_column that takes a column as an argument and
# returns the scaled column.

scale_column <- function(column) {
    (column - min(column, na.rm=TRUE)) / (max(column, na.rm=TRUE) - min(column, na.rm=TRUE
}
```

```r
# TODO
# Create the following vector
# c(1, 2, 3, 4, 5)

# TODO
# Use the function to scale the vector.
# Use the |> operator

c(1, 2, 3, 4, 5) |> scale_column()

# TODO
# Use the function to scale the rest of the columns.

df |>
    mutate(
        a_scaled = scale_column(a),
        b_scaled = scale_column(b),
        c_scaled = scale_column(c),
        d_scaled = scale_column(d)
    )

## Improving a function ----

# Let us enhance this function so that we won't be calculating the min and max
# multiple times.

# TODO
# Modify the function so that it uses the range function to calculate the min
# and max values.
# Create a variable inside the function called rng
# Assign to this function the range of the column using the range function
# Make sure to remove NA values using the na.rm argument.

scale_column <- function(column) {
    rng <- range(column, na.rm=TRUE)
    (column - rng[1]) / (rng[2] - rng[1])
}

# TODO
# Test the function again with the vector c(1, 2, 3, 4, 5)
```

```r
c(1, 2, 3, 4, 5) |> scale_column()

## Mutate functions ----

# TODO
# Create a function called clamp that takes a column, a lower bound, and an upper
# bound as arguments and returns the column with the values clamped between the
# lower and upper bounds.
# Use the case_when function to implement the logic.

clamp <- function(column, lower, upper) {
    case_when(
        column < lower ~ lower,
        column > upper ~ upper,
        .default = column
    )
}

# TODO
# Test the function with the vector c(1, 2, 3, 4, 5) and the lower and upper
# bounds 2 and 4.

c(1, 2, 3, 4, 5) |> clamp(2, 4)

# TODO
# Use the function to clamp the rest of the columns between -1 and 1.

df |>
    mutate(
        a_clamped = clamp(a, -1, 1),
        b_clamped = clamp(b, -1, 1),
        c_clamped = clamp(c, -1, 1),
        d_clamped = clamp(d, -1, 1)
    )

## Summary functions ----

# TODO
# Create a function called summary_stats that takes a column as an argument and
# returns a tibble with the mean, median, and standard deviation of the column.
```

```r
summary_stats <- function(column) {
    tibble(
        mean = mean(column, na.rm=TRUE),
        median = median(column, na.rm=TRUE),
        sd = sd(column, na.rm=TRUE)
    )
}

# TODO
# Test the function with the vector c(1, 2, 3, 4, 5)

c(1, 2, 3, 4, 5) |> summary_stats()

# TODO
# Use the function to get the summary statistics for the rest of the columns.
# Use the map_dfr function to apply the function to each column and bind the
# results into a single tibble.

df |>
    map_dfr(summary_stats)


# DataFrame Functions ----

# TODO
# Create a function called summary_stats_df that takes a dataframe as an
# argument and returns a data frame with the summary statistics for each column.
# Follow my lead on this one

summary_state_df <- function(data, var) {
    data |> summarize(
        min = min({{var}}, na.rm=TRUE),
        max = max({{var}}, na.rm=TRUE),
        mean = mean({{var}}, na.rm=TRUE),
        median = median({{var}}, na.rm=TRUE),
        n = n(),
        sd = sd({{var}}, na.rm=TRUE),
        n_miss = sum(is.na({{var}}))
    )
}
```

```r
# TODO
# Test the function with the diamonds dataset to summarize the carat column

diamonds |> summary_state_df(carat)

# Plot Functions ----

# TODO
# Create a function called plot_hist that takes a column as an argument and
# returns a histogram of the column.
# Follow my lead again.

plot_hist <- function(df, x) {
    df |>
        ggplot(aes(x = {{ x }})) +
            geom_histogram() +
            labs(title = sprintf("Histogram of %s",  x),
                    x = x,
                    y = "Frequency"
                )
}

# TODO
# Test the function with the carat column of the diamonds dataset.

diamonds |> plot_hist("carat")

# TODO
# Test the function with the price column of the diamonds dataset.

diamonds |> plot_hist("price")
```