SI 506 Midterm

Exploring player and team shooting performance during the 2023 FIFA Women's World Cup

1.0 Overview

Review the companion `midterm_overview.pdf` document in Canvas *before* attempting this assignment. The document covers due dates, work rules (this is a solo effort), use of generative artificial intelligence (AI) tools, assignment format, code styling rules, Black Formatter line length, Slack etiquette, testing and debugging tips, and submission instructions.

Reminders	**Description**
Start	Thursday, 19 October 2023, 4:00 PM Eastern
End	On or before Saturday, 21 October 2023, 11:59 PM Eastern
Points	Challenges 01-10: 1000 points; Bonus challenges 11-12: 125 points
Authorship	Submitted solution must constitute your own work. Seeking and/or securing the assistance of others is prohibited; assisting other classmates who attempt the assignment is prohibited.
Web resource access	Permitted. Open network, open readings, open notes.
Generative AI tool use	UMGPT and VS Code's GitHub Copilot extension permitted. No other generative AI tools can be used to assist in the completion of this assignment.
Questions/Help	Limited to scheduled office hours and asynchronous interactions with the teaching team over the Slack SI 506 workspace `# midterm` channel. Format messages per the specified rules. Do not send direct messages (DMs) to individual teaching team members unless a personal issue arises that blocks your progress. If an install, configuration, or hardware issues arises post a message to the `# install` channel.
Code styling	Format code using the VS Code Black Formatter extension. Change `line-length` setting from 88 characters (default) to 100 characters. See VS Code install guides for instructions.
Debugging	Use the VS Code debugger, the built-in `print()` function, and/or `assert` statements to check your code. For configuring the debugger see VS Code debugger: launch.json settings.
Submission attempts	Unlimited between start and end dates/times. Late submissions will not be accepted.
Previous score activation	Permitted. If your final submission results in a score that is lower than a previous submission score you will be permitted to activate the earlier submission and claim the higher score.

Reminders	Description
Submission review	Submissions that do not earn 1000 points will be reviewed by the teaching team. Bonus challenge code will **not** be reviewed manually unless bonus code triggers a runtime exception. Partial credit may be awarded for submissions that fail one or more non-bonus autograder tests if the teaching team (at their sole discretion) deems a score adjustment is warranted.

2.0 Data

This midterm involves writing a small program/script that explores player and team shooting performance during the 2023 FIFA Women's World Cup ⚽ hosted by Australia 🇦🇺 and New Zealand 🇳🇿 . Thirty-two (`32`) teams competed in the tournament with Spain 🇪🇸 emerging as the champion.

The data was sourced from FBREF. Before commencing the challenges review the data file. Note how the data is structured. Examine the "headers" row.

❗ Place the data file in the same directory as the README and template `*.py` file.

File	Source	Description
`data-2023-fifa_wwwc-players.csv`	FBREF	Player Shooting 2023 FIFA Women's World Cup.

3.0 Black Formatter

Format your code submissions using VS Code's Black Formatter extension. Change Black's `line-length` setting from 88 to 100 characters and add the `experimental-string-processing` setting. See VS Code install guides for instructions if you have yet to install the extension and update the settings.

4.0 Challenges

The midterm comprises ten (`10`) challenges and two (`2`) optional bonus challenges. Your assignment is to implement a small program/script that surfaces a number of insights regarding player and team shooting performance during the 2023 Women's World Cup. The program features a number of functions including a `main()` function that serves as the entry point to the program and orchestrator of the program's work flow.

❗ Team 506 recommends strongly that you complete each challenge in the order specified in this README document.

4.1 Challenge 01 (60 points)

Task: Access data in a CSV file. Drop "columns" not required for this analysis. Access the mutated "headers" element and mutated "players" elements and assign a variable to each.

1. In the `main()` function, instantiate an instance of `Path` by passing to it a string that represents the name of the data file (e.g., `"< file_name >.< file extension >"`). As you instantiate the `Path` object, call its `absolute()` *or* `resolve()` method to render the path absolute. Assign a variable named `filepath` to the `Path` object.

2. Call the function `read_csv()` and pass it required argument `filepath`. Assign a variable named `data` to the return value.

3. Iterate over the `data` list and mutate each nested list by accessing a subset of the nested list's elements and assigning it "back" to the nested list. The subset of elements represents the "columns" required for this analysis.

   Requirements

   1. Employing a `for` loop and the loop variable `i` iterate over a sequence of integers provided by the `range` type. When instatiating (i.e., creating) the `range()` instance pass it the `data` list length as the **stop** argument.

   2. Inside the loop block, mutate each nested list in `data` by "dropping" nested list elements not required for this analysis of player and team shooting performance. Retain only the first **ten (10) elements** of each nested list. Create an expression that uses the subscript operator `[]` and a positive **slicing expression** to access the target elements in the nested list. Assign the subset of elements "back" to the nested list (this requires subscript operator chaining).

      💡 Recall that you are iterating over a sequence of integers. Create a "chained" expression that accesses each nested list in `data` and then the subset of nested list's elements to retain. Assign the resulting new list "back" to the nested element as specified above.

   3. The loop block that you implement is limited to a **single line of code**.

4. After implementing the loop block uncomment the relevant `print()` calls and `assert` statements. Run your code. Confirm that the loop operation is mutating the nested lists in `data` correctly.

5. Assign a variable named `headers` to the "headers" element in `data`. Employ **slicing** to access a subset of `data` that represents the "player" elements. Assign a variable named `players` to the slice.

4.2 Challenge 02 (105 points)

Task: Implement the functions named `format_player_position()` and `clean_squad()`.

1. Implement `format_player_position()`. Review the function's docstring regarding its expected behavior, parameters, and return value.

   Requirements

   1. Inside the function block call the appropriate `str` method that returns a new version of the passed in `position` string in which all commas (",") have been converted to pipes ("|").

   2. Return the new version of the string directly to the caller *without* first assigning it to a local variable.

   3. The function block that you implement is limited to **a single line of code**.

   4. After implementing the function return to the `main()` function. Uncomment the relevant `assert` statements, run your code, and confirm that the function behaves as expected.

2. Implement `clean_squad()`. Review the function's docstring regarding its expected behavior, parameters, and return value.

   Requirements

   1. Inside the function block create a two-item tuple literal `(< item_01, item_02 >)` comprising the country code and squad/country name derived from the passed in `squad` string as the following examples illustrate.

      Examples

`squad` argument	2-item tuple to return
"es Spain"	`("ES", "Spain")`
"ie Rep. of Ireland"	`("IE", "Rep. of Ireland")`

   2. Employ slicing to access the two letter country code in the string. As part of the expression call the appropriate `str` method to return a version of the slice converted to **upper case** (e.g., "ES"). Embed the expression in the tuple literal as the first item.

   3. Also employ slicing to access the squad name (e.g., "Spain"). Embed the expression in the tuple literal as the second item.

   4. Return the tuple directly to the caller *without* first assigning it to a local variable. In other words, the function block that you implement is limited to **a single line of code**.

   5. After implementing the function return to the `main()` function. Uncomment the relevant `assert` statements, run your code, and confirm that the function behaves as expected.

4.3 Challenge 03 (135 points)

Task: Loop over the `players` list, call the functions `format_player_position()` and `clean_squad()`, and mutate each "player" nested list with the return values provided by the functions. Mutate `headers` to ensure that its elements remains synchronized with the nested lists in `players`.

1. In `main()` call the appropriate `list` method to lookup the index values for each of the following elements in the `headers` list:

   - "Pos" (position)
   - "Squad" (squad/country)

   Assign the variables named `pos_idx` and `squad_idx` to their respective return values.

2. Loop over the `players` list, mutate each nested list by assigning new "Pos", "Country_Code", and "Squad" elements as specified below.

   Requirements

   1. Employ a standard `for` loop. Choose a readable loop variable name (e.g., `player`) to represent each nested list.

   2. Inside the loop block call the function `format_player_position()` and pass it the nested list's "Pos" element as the argument, employing subscript notation (`[]`) and `pos_idx` to access the element. Assign the return value "back" to the nested list's "Pos" element, again employing subscription notation and `pos_idx` to identify the element.

   3. Next, call the function `clean_squad()` and pass it the nested list's "Squad" element as the argument, employing subscript notation (`[]`) and `squad_idx` in the expression to access the element. **Unpack** the return value and assign the variables `code` and `squad` to the unpacked items.

   4. Next, call the appropriate `list` method to add `code` to the nested list in the **fourth position**. Use the `squad_idx` variable to identify the position.

   5. Next, assign `squad` to the nested list's "Squad" element (now in the **fifth** position), employing subscript notation (`[]`) and the appropriate index (`squad_idx + ?`) in the expression on the left-hand side of the assignment operator (`=`) to assign the `squad` value to the "Squad" element".

      💡 You can utilize an arithmetic expression to compute a index value (i.e, `some_sequence[<an_arithmetic_expression>]`).

3. Outside the loop block call the appropriate `list` method to add the string "Country_Code" to the `headers` list in the **fourth position**. This ensures that the `headers` list remains synchronized with each `players` nested list as regards element order and list length.

4. Next, increment `squad_idx` by one (`1`) using **addition assignment**. This ensures that the value assigned to `squad_idx` reflects the new position of the "Squad" element in the `headers` list.

5. Call the function `write_csv()` and pass it the filepath "stu-players.csv" along with the other arguments it requires **by position**.

6. Run your code and confirm that the file was written to the current working directory. Then compare the CSV file to the matching fixture file (`fxt-*.csv`). If the files match proceed to the next challenge.

   💡 In VS Code you can compare or "diff" the file you generate against the appropriate test fixture file. After calling the `write_csv` function and generating a new file do the following:

   1. Hover over your `stu-*.csv` file with your cursor, then right click and choose the "Select for Compare" option.

   2. Next, hover over the appropriate `fxt-*.csv` test fixture file, then right click and choose the "Compare with Selected" option.

   3. Lines highlighted in red indicate row mismatches. If any mismatches are encountered close the comparison pane, revise your code, regenerate your file, and compare it again to the test fixture file. Repeat as necessary until the files match.

   ❗ Your output **must** match the test fixture file line for line and character for character. Review the test fixture file; they are akin to answer keys and should be utilized for comparison purposes as you work your way through the assignment.

4.4 Challenge 04 (95 points)

Task: Implement the function named `get_multi_position_players()`. Retrieve players who play multiple positions and write the data to a CSV file.

1. Implement `get_multi_position_players()`. Review the function's docstring regarding its expected behavior, parameters, and return value.

   Requirements

   1. In the function block implement the accumulator pattern. Assign a local variable (name your choice) to an empty "accumulator" `list`.

   2. Employ a standard `for` loop to iterate over the passed in `players` list. Choose a readable loop variable name to represent each nested list.

   3. Inside the loop block, access each nested_list's "Pos" (i.e., position) element using the passed in `pos_idx` value. As part of the expression call the appropriate `str` method and pass to it the appropriate delimiter/separator value to split the string into a list. Assign a local variable to the return value (name your choice).

   4. Add an `if` statement that evaluates whether or not the "position" list includes multiple position elements (e.g., `["DF", "FW"]`). Perform an arithmetic comparison. If there are multiple elements in the "position" list **append** the nested list representing the player to the accumulator list.

   5. After the loop terminates return the accumulator list to the caller.

2. Return to the `main()` function. Call the function `get_multi_position_players()` and pass it the appropriate arguments. Assign a variable named `multi_position_players` to the return value.

3. Call the function `write_csv()` and pass it the filepath "stu-players-multi_position.csv" along with the other arguments it requires **by position**.

4. Run your code and confirm that the file was written to the current working directory. Then compare the CSV file to the matching fixture file. If the files match proceed to the next challenge.

4.5 Challenge 05 (120 points)

Task: Implement the function named `get_team()`. Test the function by retrieving the Chinese and Moroccan teams and then write each team to a CSV file.

1. Implement `get_team()`. Review the function's docstring regarding its expected behavior, parameters, and return value.

   Requirements

   1. In the function blook implement the accumulator pattern. Assign a local variable (name your choice) to an empty "accumulator" `list`.

   2. Employ a standard `for` loop to iterate over the passed in `players` list. Choose a readable loop variable name to represent each nested list.

    3. Inside the loop block, add an `if` statement that evaluates whether or not the nested list's "Squad" element matches the passed in `squad` string. Perform a *case insensitive* string comparison. If the strings match **append** the nested list representing the player to the accumulator list.

       💡 Employ the subscript operator `[]` and the passed in `squad_idx` value to access the nested list's "Squad" element when constructing your `if` statement.

    4. After the loop terminates return the accumulator list to the caller.

2. Return to the `main()` function. Call the function `get_team()` and pass it the arguments required to return the Chinese women's national football team 🇨🇳 . Assign a variable named `team_china` to the return value.

   ❗ Use VS Code's search feature to scan the `stu-players.csv` file for the Chinese team's "Squad" name to pass as an argument. The word "China" comprises only part of the name. Also use the `squad_idx` variable as an argument.

3. Call the function `write_csv()` and pass it the filepath "stu-team-china.csv" along with the other arguments it requires **by position**.

4. Run your code and confirm that the file was written to the current working directory. Then compare the CSV file to the matching fixture file. If the files match proceed to the next step.

5. Call `get_team()` again and pass it the arguments required to return the Moroccan women's national football team 🇲🇦 employing keyword arguments passed in **reverse order**. Assign a variable named `team_morocco` to the return value.

   ❗ Pass "Morocco" as the `squad` keyword argument value.

6. Call the function `write_csv()` and pass it the filepath "stu-team-morocco.csv" along with the other arguments it requires **by position**.

7. Run your code and confirm that the file was written to the current working directory. Then compare the CSV file to the matching fixture file. If the files match proceed to the next challenge.

4.6 Challenge 06 (80 points)

Task: Implement the function named `get_team_names()`. Retrieve a list of unique team names that can serve double duty as country names.

1. Implement `get_team_names()`. Review the function's docstring regarding its expected behavior, parameters, and return value.

   Requirements

    1. In the function blook implement the accumulator pattern. Assign a local variable (name your choice) to an empty "accumulator" `list`.

    2. Employ a standard `for` loop to iterate over the passed in `players` list.

        3. Inside the loop block, craft an `if` statement that allows you to accumulate a list of team names from `players` that contains **no duplicates**. In other words, append a nested list's "Squad" element to the accumulator list if, and only if, it has not been added previously.

           💡 Each nested list represents a player. Each player is associated with a squad/team/country (e.g., Australia). Given that each team can roster up to 23 players, duplicate "Squad" names exist throughout the `players` list.

        4. After the loop terminates return the list of team names to the caller.

2. Return to the `main()` function. Call the function `get_team_names()` and pass it the arguments required to return a list of unique team names. Since the team names correspond to country names assign a variable named `countries` to the return value.

3. Sort the `countries` list in ascending order, employing either the appropriate `list` method or appropriate built-in function.

   💡 The `list` method performs an in-place sort. It does not return a new list.

4. Uncomment the `print()` call and the `assert` statement. Run your code. Confirm that the function behaves as expected.

4.7 Challenge 07 (95 points)

Task: Implement the function named `get_top_scorer()`. Retrieve the winner(s) of the ⚽ Golden Boot award.

1. Implement `get_top_scorer()`. Review the function's docstring regarding its expected behavior, parameters, and return value.

   Requirements

    1. In the function blook implement the accumulator pattern. Assign a local variable (name your choice) to an empty "accumulator" `list`. This list will hold the top scorer(s). Also assign a local variable (name your choice) to an integer (choose an appropriate value) that will serve as the start value for the most goals scored count.

    2. Employ a standard `for` loop to iterate over the passed in `players` list.

    3. Inside the loop block retrieve each player's "Gls" (goals) element using the subscript operator `[]` and `gls_idx` in your expression. Assign a local variable (name your choice) to the expression (which resolves to a value).

    4. Don't assume that the "Gls" element's you retrieve is a number. Be prepared to convert it to an integer.

    5. After retrieving the player's goals scored value employ `if-elif-else` conditional logic to locate the top goal scorer(s). By now the pattern employed to identify either minimum or maximum values in a sequence as well as account for ties should be familiar to you.

    6. Whenever the accumulator list includes player elements that need to be replaced by a new leading scorer, adjust the **existing** accumulator list. **Do not create a new list**.

7. This challenge requires implementing a minor adjustment to the pattern. There are many World Cup players who did not score any goals during the tournament. The `if-elif-else` conditional logic must account for this possibility and filter out any players with a goal count of zero (`0`). Filtering out such players while continuing to evaluate those who scored one or more goals can be accomplished by implementing a **compound conditional statement** that employs the appropriate logical operator (e.g., `and`, `or`, `not`).

   ❗ Both your `if` and `elif` statements *must* evaluate **two conditions** using the appropriate logical operator (e.g., `and`, `or`, `not`). In other words, the conditional logic *must* exclude players who did not score as well as identify players who scored the most goals during the tournament.

   #### `if` statement conditions

   - Did the player score at least one goal?
   - Is the player's goal count greater than the most goals scored count?

   If `True` update the most goals scored count, remove all players added previously to the accumulator list (without creating a new list), and then append the current player's nested list to the accumulator list. If `False` then evaluate the next condition.

   #### `elif` statement conditions

   - Did the player score at least one goal?
   - Is the player's goal count equal to the most goals scored count?

   If the equality comparison evaluates to `True` append the player's nested list to the accumulator list; otherwise, proceed to the next iteration of the loop.

8. After the loop terminates return the top scorers list to the caller.

2. Return to the `main()` function. Call the appropriate `list` method to lookup the index value for the "Gls" (goals) element in the `headers` list. Assign the variables named `gls_idx` to the return value.

3. Call the function `get_top_scorer()` and pass it the arguments it requires to perform the computation. Assign a variable named `top_scorers` to the return value.

4. Uncomment the `print()` call and run your code. If the object streamed to the terminal is a nested list that contains a list representation of the Japanese player Hinata Miyazawa 🇯🇵, winner of the 2023 Golden Boot Award (5 goals), then your function is behaving as expected.

4.8 Challenge 08 (115 points)

Task: Identify each team's top scorer(s) and write the data to a CSV file.

1. In the `main()` function assign a variable named `team_top_scorers` to an empty list.

2. Next, implement a standard `for` loop that iterates over the `countries` list. Choose a readable loop variable name to represent each nested list.

3. Inside the loop block call the function `get_team()` and pass it the arguments it requires to return a nested list of players who competed for the current country. Assign a variable named `team` to the return value.

4. Also inside the loop block call the function `get_top_scorer()` and pass it the arguments it requires to return a nested list of the current country's top scorer(s). Assign a variable named `top_scorers` to the return value.

5. Once you have the team's top scorers call the appropriate `list` method to add `top_scorers` to the `team_top_scorers` list.

   ❗ Chose your `list` method call wisely. The mutated `team_top_scorers` list *must* match the following structure:

   ```
   [[player_01], [player_02], [player_03], ...]
   ```

6. Call the function `write_csv()` and pass it the filepath "stu-team-top_scorers.csv" along with the other arguments it requires **by position**.

7. Run your code and confirm that the file was written to the current working directory. Then compare the CSV file to the matching fixture file. If the files match proceed to the next challenge.

4.9 Challenge 09 (90 points)

Task: Implement the function named `get_player_shooting_numbers()`.

1. Implement `get_player_shooting_numbers()`. Review the function's docstring regarding its expected behavior, parameters, and return value.

   💡 The shooting-related elements ⚽ are "Gls" (goals), "Sh" (shots), and "SoT" (shots on target). *Shots on target* is defined as a shot taken that would have resulted in a goal if not blocked by the goalkeeper or another player considered the "last defender". Shots that strike the goal post or crossbar 🥅 are not considered a shot on target.

   Requirements

   1. Access the passed in `player` list's shooting-related elements using the subscript operator `[]` and the passed in `slice_` instance. Assign a local variable (name your choice) to the expression which resolves to a list comprising three (3) elements (i.e., the "shooting numbers").

      💡 The `slice` instance is passed to the function as an argument and bound to the parameter `slice_`. Use it as follows:

      ```
      some_val = some_list[slice_]  # equivalent to some_sequence[8:11]

      # Implement loop
      ```

   2. Construct a `for` loop that iterates over a sequence of numbers whose length matches the length of the "shooting numbers" list. Access each element in the target list using the

    appropriate expression and **convert** the element from a string to an integer. Assign the integer "back" to the list element.

    3. After the loop terminates return the "shooting numbers" list to the caller.

2. Return to the `main()` function. Uncomment the following variable assignment:

   ```
   slice_ = slice(gls_idx, len(headers))  #  equivalent to slice(8, 11) or some_sequence[8:11]
   ```

   💡 This is an example of how to create a `slice` instance and then pass it to a function as an argument. Note that the stop value is set to the length of the `headers` list. This ensures that the last three (3) elements in the `player` list indexed 8, 9, and 10 are accessed.

3. Call the function `get_player_shooting_numbers()` and pass it the first "player" nested list in `players` together with the `slice_` instance. **Unpack** the return value and assign the variables `goals`, `shots`, and `shots_on_target` to the unpacked items.

   💡 The Spanish midfielder Teresa Abelleira 🇪🇸 is the first player in the `players` list.

4. Uncomment the `print()` call and the three `assert` statements. Run your code. Confirm that the function and item unpacking behaves as expected.

4.10 Challenge 10 (105 points)

Task: Implement the function named `calculate_shot_conversion_rate()`. Compute the shooting efficiency of all players who competed in the World Cup.

1. Implement `calculate_shot_conversion_rate()`. Review the function's docstring regarding its expected behavior, parameters, and return value.

   Requirements

   1. Employ `try` and `except` blocks to ensure that a passed in shot value of zero (`0`) does not trigger a runtime exception (i.e., a `ZeroDivisionError` exception).

   2. Inside the `try` block divide the passed in `goals` by the passed in `shots`. Round the quotient to the number of decimal places specified by the passed in `precision` value. Return the rounded quotient directly to the caller *without* first assigning it to a local variable.

   3. Inside the `except` block return the floating point value `0.0` directly to the caller.

2. Return to the `main()` function. Loop over the `players` list. Inside the loop block call the function `get_player_shooting_numbers()` and pass it the arguments required to return the current player's shooting numbers. **Unpack** the return value and assign the variables `goals`, `shots`, and `shots_on_target` to the unpacked items.

3. Inside the loop block call the function `calculate_shot_conversion_rate()` and pass it the player's `goals` and `shots` values along with a `precision` value of three (`3`). Pass the arguments **by position**. Append the return value to the player's nested list.

4. Inside the loop block call `calculate_shot_conversion_rate()` a second time, passing it the player's `goals` and `shots_on_target` values along with a `precision` value of three (`3`) using **keyword arguments** passed **by position**. Append the return value to the player's nested list.

5. Outside the loop block add the following list to the `headers` list:

   ```
   ["shots_conv_rate", "shots_on_target_conv_rate"]
   ```

   💡 Adding the above elements to the `headers` list ensures that the list remains synchronized with the mutated nested lists in `players`.

6. Call the function `write_csv()` and pass it the filepath "stu-players-shooting_efficiency.csv" along with the other arguments it requires **by position**.

7. Run your code and confirm that the file was written to the current working directory. Then compare the CSV file to the matching fixture file. If the files match proceed to the next challenge.

4.11 Challenge 11 (Bonus: 50 points)

Task: Implement the function named `get_team_shooting_numbers()`. Compute the shooting efficiency of all teams who competed in the World Cup based on an accumulation of their players shooting numbers.

1. Implement `get_team_shooting_numbers()`. Review the function's docstring regarding its expected behavior, parameters, and return value.

   Requirements

   1. Assign zero (`0`) to three local variables: `goals_count`, `shot_count`, and `shots_on_target_count`.

   2. Loop over the passed in `team` list. In the loop block call the function `get_player_shooting_numbers()` passing it the arguments it needs to return the current player's shooting numbers. **Unpack** the return value and assign the variables `goals`, `shots`, and `shots_on_target` to the unpacked items.

   3. Employ addition assignment and increment the team's `goals_count`, `shot_count`, and `shots_on_target_count` with the player's shooting numbers.

   4. After the loop terminates return a three-item tuple literal comprising the team's `goals_count`, `shot_count`, and `shots_on_target_count` values (ordered as listed).

2. Return to the `main()` function. Create a list literal containing the following strings in the order specified:

   - country
   - goals
   - shots
   - shots_on_target
   - shots_conv_rate

   - shots_on_target_conv_rate

   Assign the variable `team_headers` to the list literal.

   💡 You will pass this list as the `headers` argument the next time you call the `write_csv()` function.

3. Assign an empty list to a variable named `teams`.

4. Loop over the `countries` list. Inside the loop block call the appropriate function and pass it the required arguments to retrieve the current country's team of players. Assign a variable named `team` to the return value.

5. After retrieving the team's players call the appropriate function to retrieve the team's shooting numbers. **Unpack** the return value and assign the variables `goals`, `shots`, and `shots_on_target` to the unpacked items.

6. While still in the loop block create a `list` literal comprising six elements. The first four elements include country/team name, `goals`, `shots`, and `shots_on_target`. The last two elements comprise the team's shooting conversion rates.

   Call the function `calculate_shot_conversion_rate()` twice from **inside** the list literal that you are constructing. The first call will compute the team's **shot conversion rate** while the second call will compute the team's **shots on target conversion rate** based on the arguments that you pass to each function call.

   💡 You can embed not only values and variables in a list literal but also function calls.

7. Assign the list literal to a variable named `team_metrics`. Then append `team_metrics` to the `teams` list.

8. Outside the loop call the function `write_csv()` and pass it the filepath "stu-team-shooting_efficiency.csv" along with the other arguments it requires **by position**.

9. Run your code and confirm that the file was written to the current working directory. Then compare the CSV file to the matching fixture file. If the files match proceed to the next challenge.

4.12 Challenge 12 (Bonus: 75 points)

Task: Rate each team's shots on target conversion rate on a tiered scale and assign the rating to each team's nested list. Then write the results to a CSV file.

1. In the `main()` function loop over `teams`. Access each nested "team" list's **last element** ("shots_on_target_conv_rate") and convert it to a `float`. Assign a variable named `conv_rate` to the number.

2. Inside the loop block employ `if-elif-else` conditional logic to append a "rating" value to each nested "team" list. Assign each team to one of four possible rating tiers: "Top Tier", "Upper Middle Tier", "Lower Middle Tier", or "Bottom Tier." The selection criteria is described below:

Rating	Selection Criteria (shots on target conversion rate)
Top Tier	Greater than or equal to `0.4`.

Rating	Selection Criteria (shots on target conversion rate)
Upper Middle Tier	Greater than or equal to `0.3` but less than `0.4`.
Lower Middle Tier	Greater than or equal to `0.2` but less than `0.3`.
Bottom Tier	Less than `0.2` (i.e., all others).

Inside each `if-elif-else` statement block assign a variable named `rating` to each rating string. Outside the set of conditional statements but *inside* the loop block append the selected `rating` to the team list.

3. After appending a new element to each nested team list in `teams` sync up the `team_headers` list by appending the string "efficiency_rating" to the `team_headers` list.

4. Uncomment the built-in function `sorted()` call that returns a new list sorted by each team's shots on target conversation rate (descending order) and then by the name of the team (ascending order).

   💡 The `sorted()` function employs an anonymous `lambda` function to sort the list. You will learn how to sort lists using `lambda` functions after the midterm.

5. Call the function `write_csv()` and pass it the filepath "stu-team-shooting_efficiency_ratings.csv" along with the other arguments it requires **by position**.

6. Run your code and confirm that the file was written to the current working directory. Then compare the CSV file to the matching fixture file. If the files match submit your code to Gradescope and declare victory.

FINIS 🎉