SI 506 Midterm

Exploring Michigan's county-level COVID-19 case counts and vaccination levels

1.0 Dates

- Release date: Thursday, 20 October 2022, 4:00 PM Eastern
- Due date: on or before Saturday, 22 October 2022, 11:59 AM Eastern (**before noon**)

❗ Note the due date. Late submissions will not be excepted.

2.0 Overview

Review the companion *Midterm Overview* document in Canvas for general details regarding this assignment, including styling rules.

3.0 Points

The midterm is worth 1000 points and you accumulate points by passing a series of auto grader tests.

4.0 Solo effort

The midterm exam is open network, open readings, and open notes. You may refer to code in previous lecture exercises, lab exercises, and problem sets for inspiration. Please abide by the following rules:

1. The midterm assignment that you submit **must constitute your own work**. You are **prohibited from soliciting assistance or accepting assistance** from any person while working to complete the programming assignment. This includes but is not limited to individual classmates, study group members, and tutors.

    - ❗ If you have formed or participated in an SI 506 study group please **suspend all study group activities** for the duration of the midterm assignment.

    - ❗ If you work with a tutor please **suspend contact** with the tutor for the duration of the midterm assignment.

2. Likewise, you are **prohibited from assisting any other student** who is required to complete this assignment. This includes students attempting the assignment during the regular exam period, as well as those who may attempt the assignment at another time and/or place due to scheduling conflicts or other issues.

3. Direct all **questions** regarding the assignment to the Slack SI 506 workspace `# midterm` channel. If you encounter an issue with your code you may request a code review by members of Team 506. Do not post code snippets.

4. **Office hours** held on Friday will be **remote only** conducted over Zoom. Questions can be posed but code cannot be displayed via screen sharing.

5. If a personal issue arises during the assignment period please send a private DM to your GSI or Anthony.

5.0 Data

This midterm involves writing a small program designed to explore the State of Michigan's county-level COVID-19 case counts and vaccination levels.

Data sets are sourced from CovidActNow, the Center for Disease Control and Prevention (CDC), and the National Center for Health Statistics (NCHS).

Before commencing the challenges review each data file. Note how the data is structured. Determine which "columns" permit the data to be linked. Inspect the string values and consider which values can/ought to be cast to a different type (e.g. string -> integer).

File	Source	Description
`mi_county_covid_cases-20221014.csv`	CovidActNow	Michigan county-level COVID-19 case counts as of 14 October 2022 sourced from the COVIDActNow API.
`mi_county_vax_levels-20221005.csv`	CDC	US COVID-19 county-level vaccination rates. The CSV files contains Michigan data only, dated 05 October 2022.
`mi_ur_codes.csv`	NCHS	Urban/rural classification scheme for U.S. counties and county-equivalent entities (2013)
`fxt-mi_ur_county_vax_levels.csv`	SI 506	Test fixture file. Compare your file output to this file. Your file *must* match this file line for line and character for character.

6.0 Debugging

As you write your code take advantage of the built-in `print` function, VS code's debugger, and VS Codes file comparison feature to check your work.

6.1 The built-in `print` function is your friend

As you work through the challenges make frequent use of the built-in `print()` function to check your variable assignments. Recall that you can call `print` from inside a function, loop, or conditional statement. Use f-strings and the newline escape character `\n` to better identify the output that `print` sends to the terminal.

6.2 VS Code debugger

You can also use the debugger to check your code. If you have yet to configure your debugger see the instructions at https://si506.org/guides/. You can then set breakpoints and review your code in action by "stepping over" lines and "stepping into" function calls.

7.0 Gradescope submissions

You may submit your solution to Gradescope as many times as needed before the expiration of the exam time.

❗ Your **final** submission will constitute your exam submission.

8.0 Auto grader / manual scoring

The autograder runs a number of tests against the Python file you submit, which the autograder imports as a module so that it can gain access to and inspect the functions and other objects defined in your code. The functional tests are of two types:

1. The first type will call a function passing in known argument values and then perform an equality comparison between the return value and the expected return value. If the function's return value does not equal the expected return value the test will fail.
2. The second type of test involves checking variable assignments or expressions in `main()` and other functions. This type of test evaluates the code you write, character for character, against an expected line of code using a regular expression to account for permitted variations in the statements that you write. The test searches `main()` or another function for the expected line of code. If the code is not located the test will fail.

If the auto grader is unable to grade your submission successfully with a score of 1000 points the teaching team will grade your submission **manually**. Partial credit **may** be awarded for submissions that fail one or more autograder tests if the teaching team (at their sole discretion) deems a score adjustment warranted.

If you submit a partial solution, feel free to include comments (if you have time) that explain what you were attempting to accomplish in the area(s) of the program that are not working properly. We will review your comments when determining partial credit.

9.0 Challenges

The midterm comprises ten (10) challenges. Your goal is to implement a small program (or script) that can retrieve as well as compute useful county-level information derived from the Michigan COVID-19 data sets. 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.

9.1 Challenge 01

Task: Before commencing work on implementing the program's various functions, demonstrate your knowledge of indexing and slicing operations by accessing elements in the list `washtenaw_vax_rates`. Do so *without* looping over the sequence.

❗ Do not loop your way to the correct values. You are restricted to accessing elements using indexing and slicing **only**. You are also limited to *a single variable assignment* per sub challenge.

1. Employ *slicing* to return the subset of elements in `washtenaw_vax_rates` featuring a `Date` value greater than "12/01/2021". Assign the slice to the variable `vax_2022`.

   💡 Check your work by calling the built-in function `print()` and passing to it as an argument the variable `vax_2022`. Enhance your terminal output's readability by passing an f-string to `print()` formatted as follows:

   ```
   f"\nvax_2022 = {vax_2022}"
   ```

   Consider adopting this general print format as you work through the challenges.

2. Employ *negative slicing* to return the subset of elements in `washtenaw_vax_rates` with a `Series_Complete_Yes` value of less than 200,000. Assign the slice to the variable `vax_under_200k`.

3. Employ slicing to return the subset of elements in `washtenaw_vax_rates` featuring an *even* month in the `Date` string (e.g., February (2), April (4), June (6), ...). Assign the slice to the variable `vax_even_months`.

   ❗ Be sure to exclude the "header" element from your slice.

4. Employ subscript operator *chaining* to return the `Booster_Doses_Vax_Pct` value for the element in `washtenaw_vax_rates` with a `Date` value of "01/01/2022". Assign the value to the variable `jan_2022_booster_pct`.

9.2 Challenge 02

Task: Demonstrate your iteration and control flow fluency by implementing `for` and `while` loops that filter data per a specified condition.

1. Call the function `read_csv()` and pass to it the filepath argument `mi_county_covid_cases-20221014.csv`. Assign the return value to the variable named `case_data`.

2. Assign the `case_data` "headers" element to the variable named `case_headers`.

3. Assign the `case_data` "county" elements slice to the variable named `case_counties`.

4. Implement a `for` loop that loops over `case_counties`.

   Requirements

   1. Inside the loop block write an `if` statement that evaluates whether or not the nested county list includes a name element that commences with the substring "Ingham".

   2. If the conditional statement evaluates to `True` assign the nested county list to the variable named `case_ingham`.

   3. Then add a control statement in the `if` block that *terminates* the loop.

5. Implement a `for i in range()` loop to generate a sequence of numbers to loop over.

   Requirements

   1. Set the `range` object's `stop` argument to the length of the `case_counties` list.

   2. Inside the loop block write an `if` statement that accesses a `case_counties` nested list name element and evaluates whether or not the county name can be found in the `regions` tuple. The `if` statement *must* convert the county name within the nested county list to lowercase.

      💡 Utilize subscript operator `[]` chaining to access the nested list's name element. The loop variable `i` supplies one of the index values you need to access each county's name element.

   3. If a match is obtained append the nested list to the `region_cases` list.

      💡 Michigan Economic Recovery Council (MERC) regional data is sourced from the MI Safe Start Map project.

6. Implement a `while` loop.

   Requirements

   1. Start by assigning a sensible start value to the "counter" variable `i` in the line above your intended `while` loop.

   2. Limit the number of `while` loop iterations to *less than* the length of the `case_counties` list.

   3. Inside the loop block write an `if` statement that accesses a `case_counties` nested list name element and evaluates whether or not the county name *is equal to* the county name assigned to the variable `county_name`.

   💡 Utilize subscript operator `[]` chaining to access the nested list's name element.

4. If the conditional statement evaluates to `True` assign the boolean value `True` to the variable `has_county`. Then add a control statement in the `if` block that *terminates* the loop.

5. Given that the `if` statement could evaluate to `False` add an `else` statement block that assigns the boolean value `False` to the variable `has_county`.

6. Inside the loop block employ the *addition assignment operator* to increment the "counter" variable `i` by `1`. Locate the assignment at the bottom of the loop block.

   ❗ Remember to increment `i`. Failure to do so will trigger an infinite loop.

9.3 Challenge 03

Task: Implement a function that simplifies the retrieval of county data and then test your implementation by returning Jackson County's COVID-19 case count data.

1. Implement the function named `get_county`. Review the function's docstring regarding its expected behavior, parameters, and return value.

   Function requirements and hints

   1. Loop over `counties`. Inside the loop block implement a conditional statement. The `if` statement *must* perform a *case insensitive* string comparison of the nested list's name element against the passed in `county_name` value.

   2. If a name match is obtained return the nested list that represents the county *immediately* (i.e., without first assigning the nested list to a local variable).

   3. If no match is obtained the function should return `None` *implicitly* (i.e., an explicit `return None` statement is *not* required).

2. After implementing `get_county()` return to the `main()` function.

3. Call the function `get_county()` and pass to it the `case_counties` list and the *upper case* string 'JACKSON COUNTY' as arguments. Assign the return value to the variable named `case_jackson`.

   Below is the list you *must* return after calling `get_county()` with the required arguments.

   ```
   [
     '2022-10-14', 'Jackson County', '158510', '2', '3', '51894', '636', '0', '0',
     'https://covidactnow.org/us/michigan-mi/county/jackson_county'
   ]
   ```

9.4 Challenge 04

Task: Your instructor recently tested positive for COVID-19. Add his case to the Jackson County COVID-19 case counts (cases and new_cases).

1. Implement the function named `convert_to_int`. Review the function's docstring regarding its expected behavior, parameter, and return value.

   Function requirements and hints

   1. Implement `try` and `except` statements. Limit the `except` statement to `ValueError` handling only.

   2. In the `try` block attempt to convert the passed in `value` to an integer. Design your `return` statment to return the converted `value` to the caller *immediately* (i.e., without first assigning the new value to a local variable).

   3. If the `try` block triggers a runtime `ValueError` exception, "catch" the error in the `except` block and return the `value` to the caller *unchanged*.

2. After implementing `convert_to_int()` return to the `main()` function.

3. Use `case_headers` to look up the index of the headers "cases" element by calling the appropriate `list` method. Assign the return value (an integer) to the variable named `cases_idx`.

4. Call the function `convert_to_int()` and pass to it the following argument: an expression that resolves to the `case_jackson` "cases" element.

   💡 Access the required `case_jackson` list element using the subscript operator `[]` and the variable `cases_idx`.

5. Then add one (`1`) to the integer returned by the function call. Assign the updated count to the `case_jackson` "cases" element (in other words, replace the element's value).

   ❗ Perform the function call, addition, and the variable assignment *on the same line*.

6. Look up the index of the "new_cases" headers element by utilizing the same approach employed to return the "cases" index. Assign the return value to the variable named `new_cases_idx`.

7. Finally, update the `case_jackson` "new_cases" element utilizing the same workflow employed to update the "cases" element. Call `convert_to_int()`, provide it with an argument that resolves to the "new_cases" element, and then add one (`1`) to the value returned by the function call. Assign the updated count to the `case_jackson` "new_cases" element (in other words, replace the element's value).

   ❗ Perform the function call, addition, and the variable assignment *on the same line*.

   Below is the list you *must* return after calling `convert_to_int()` twice with the required arguments.

   ```
   [
     '2022-10-14', 'Jackson County', '158510', '2', '3', 51895, '636',
   1, '0',
   ```

```
    'https://covidactnow.org/us/michigan-mi/county/jackson_county'
]
```

❗ Take heed: the variable `case_jackson` points to the Jackson County nested list in `case_counties`. The list you mutated in this challenge is an *existing* nested list element and *not* a new list. You can confirm this by adding the following `print()` call to your code in `main()`.

```
print(f"\ncase_counties nested list mutated = {case_counties[37]}")
```

9.5 Challenge 05

Task: The nested lists derived from the midterm `*.csv` files contain a large number of elements. Indeed, the number is such that deriving index values manually is problematic. However, each `*.csv` file includes a "headers" row that can be employed to look up the index value of a target county list element. The function `get_attribute()` adopts the headers "look up" strategy in order to permit easy retrieval of any county element but you must implement the strategy before you can make use of the function. You will test your implementation by returning a county's vaccination "Series_Complete_Yes" value and converting it to an integer.

Implement a function that permits the retrieval of any county element using the CSV "headers" row to look up individual index values.

1. Implement the function named `get_attribute`. Review the function's docstring regarding its expected behavior, parameters, and return value.

   Function requirements and hints

   1. Leverage the passed in `headers` list to look up the index of the passed in `header` element by calling the appropriate `list` method. Use this expression which resolves to a number in combination with the subscript operator `[]` to access the target `county` element. Then `return` the element to the caller.

   2. This function can be implemented with one line of code.

2. After implementing `get_attribute()` return to the `main()` function.

3. Call the function `read_csv()` and pass to it the filepath argument `mi_county_vax_levels-20221005.csv`. Assign the return value to the variable named `vax_county_data`.

4. Assign the `vax_county_data` "headers" element to the variable named `vax_headers`.

   The headers list contains the following elements:

   ```
   [
       'Date', 'Recip_County', 'Series_Complete_Yes',
   'Series_Complete_5Plus', 'Series_Complete_5to17',
   ```

```
'Series_Complete_12Plus', 'Series_Complete_18Plus',
'Series_Complete_65Plus', 'Booster_Doses', 'Booster_Doses_5Plus',
'Booster_Doses_12Plus', 'Booster_Doses_18Plus',
'Booster_Doses_65Plus',
    'Second_Booster_65Plus', 'SVI_CTGY', 'Series_Complete_Pop_Pct_SVI',
    'Series_Complete_Pop_Pct_UR_Equity', 'Booster_Doses_Vax_Pct_SVI',
'Census2019',
    'Census2019_5PlusPop', 'Census2019_5to17Pop',
'Census2019_12PlusPop', 'Census2019_18PlusPop', 'Census2019_65PlusPop'
]
```

5. Assign the `vax_county_data` "county" elements slice to the variable named `vax_counties`.

   A `vax_counties` nested list element represents a county that contain elements described by the the header values above.

   ```
   [
       '10/05/2022', 'Washtenaw County', '266717', '265951', '24754',
   '259140', '241197', '52611',
       '166848', '166844', '162969', '153944', '42918', '18184', 'A', '3',
   '3', '4', '367601',
       '349901', '50064', '323866', '299837', '53369'
   ]
   ```

6. Call the function `get_county()` and pass to it the `vax_counties` list and the string 'washtenaw county' as arguments. Assign the return value to the variable named `washtenaw`.

7. Call the function `get_attribute()` and pass to it as arguments the `washtenaw` list, `vax_headers`, and the header "Series_Complete_Yes". Assign the return value to the variable named `washtenaw_vax_series_complete`.

   💡 The CDC defines the "Series_Complete_Yes" data as:

   > Total number of people who have completed a primary series (have second dose of a two-dose vaccine or one dose of a single-dose vaccine) ....

8. Call the function `convert_to_int()` and pass to it the `washtenaw_vax_series_complete` string as the argument. Assign the return value to the variable named `washtenaw_vax_series_complete`.

Below is the value you *must* return after calling `convert_to_int()` with the required argument.

```
washtenaw_vax_series_complete = 266717
```

💡 Recall that you can check a value's type by passing it to the built-in function `type()`.

9.6 Challenge 06

Task: It's not enough to return counts of county residents who have been vaccinated. Equally, if not more important, is knowing the percentage of residents vaccinated for a given population demographic or group. Implement the function `calculate_vax_pct()` so that percentage values can be computed.

1. Implement the function named `calculate_vax_pct`. Review the function's docstring regarding its expected behavior, parameters, and return value.

   Function requirements and hints

   1. Perform simple division as described in the docstring and then multiply the quotient by 100 to obtain the percentage value. Assign the computed value to a local variable (name your choice).

   2. Write conditional statements that perform the following actions:

      - Evaluate the truth value of `precision`. If the conditional statement evaluates to `True` round the computed value to the number of decimal places specified by `precision` and return the rounded value to the caller.

      - Otherwise, should the truth value of `precision` evaluate to `False` round the computed value without specifying the number of decimal places and then return the resulting integer value to the caller.

2. After implementing `calculate_vax_pct()` return to the `main()` function.

3. Call the function `get_attribute()` and pass to it the appropriate required arguments to return Washtenaw County's "Census2019" population total value. Assign the return value to the variable named `washtenaw_pop_total`.

4. Call the function `convert_to_int()` and pass to it the `washtenaw_pop_total` string as the argument. Assign the return value to the variable named `washtenaw_pop_total`.

5. Call the function `calculate_vax_pct()` and pass to it the arguments `washtenaw_vax_series_complete`, `washtenaw_pop_total`, and a `precision` value that rounds the float returned to the *second* (2nd) decimal place. Assign the return value to the variable named `washtenaw_vax_series_complete_pct`.

   Below is the value you *must* return after calling `calculate_vax_pct()` with the specified arguments.

   ```
   washtenaw_vax_series_complete_pct = 72.56
   ```

9.7 Challenge 07

Task: Reduce the number of calls to the function `convert_to_int()` by implementing a function that handles the conversion process for all whole numbers masquerading as strings in a nested county list.

1. Implement the function named `clean_data`. Review the function's docstring regarding its expected behavior, parameter, and return value.

   Function requirements and hints

   1. Implement a `for i in range()` loop to generate a sequence of numbers to loop over. Set the `range` object's `stop` argument to the length of the `county` list.

   2. During each loop iteration employ the subscript operator `[]` and the loop variable `i` to access the `county` element whose index matches `i`. Pass the element accessed to the function `convert_to_int()` as the argument. Assign the return value to the current element.

   3. After the loop terminates, return the mutated `county` list to the caller.

2. After implementing `clean_data()` return to the `main()` function.

3. Mutate all the nested county lists in `vax_counties`. Do so by implementing another `for i in range()` loop to generate a sequence of numbers to loop over. Set the `range` object's `stop` argument to the length of the `vax_counties` list.

4. During each loop iteration employ the subscript operator `[]` and the loop variable `i` to access the nested list element (a county) whose index matches `i`. Pass the element accessed to the function `clean_data()` as the argument. Assign the return value (a `list`) to the current element.

   💡 Utilize the debugger or the built-in function `print()` to check if the `vax_counties` nested lists have all been mutated.

   Below is an example of a mutated county list:

   ```
   [
      '10/05/2022', 'Oceana County', 14792, 14784, 1076, 14536, 13708,
   4904, 8374, 8374,
      8338, 8159, 3852, 1542, 'D', 14, 6, 16, 26467, 25056, 4500, 22702,
   20556, 5647
   ]
   ```

   😉 This work eliminates the need to call `convert_to_int()` to recast whole numbers derived from `vax_counties`, resulting in a considerable reduction in the number of function calls required by the remaining challenges.

5. Test your work by calling the function `get_county()` and passing to it the appropriate arguments to retrieve the Genesee County nested list. Assign the return value to the variable named `genesee`.

   💡 U-M Flint is located in Genesee County.

6. Call the function `get_attribute()` and pass to it the appropriate arguments required to return Genesee County's "Series_Complete_5to17" value. Assign the return value to the variable named

`genesee_vax_series_complete_5to17`.

💡 The CDC defines the "Series_Complete_5to17" data as:

> Total number of people ages 5-17 years who have completed a primary series (have second dose of a two-dose vaccine or one dose of a single-dose vaccine) ....

7. Call the function `get_attribute()` and return Genesee County's "Census2019_5to17Pop" population total value. Assign the return value to the variable named `genesee_pop_total_5to17`.

8. Call the function `calculate_vax_pct()` and pass to it the arguments `genesee_vax_series_complete_5to17` and `genesee_pop_total_5to17`. *Do not* pass a `precision` argument. Assign the return value to the variable named `genesee_vax_series_complete_5to17_pct`.

   Below is the value you *must* return after calling `calculate_vax_pct()` with the specified arguments.

   ```
   genesee_vax_series_complete_5to17_pct = 23
   ```

9.8 Challenge 08

Task: What are the state-wide vaccination levels for a given population demographic (e.g., all residents, residents 18 years and older, residents 65 years and older). Implement the function `count_vaccinated()` and find out.

1. Implement the function named `count_vaccinated`. Review the function's docstring regarding its expected behavior, parameters, and return value.

   Function requirements and hints

   1. Put the accumulation pattern to work. Create two local "accumulator" variables (names your choice) and assign each a starter value of zero (`0`). A running count of each county's vaccinated residents for a particular demographic will be assigned to one variable while a running count of the corresponding census population will be assigned to the other variable.

   2. Loop over the passed in `counties` list. Inside the loop block call the function `get_attribute()` *twice*: once to retrieve the county's count of vaccinated residents for the specified population demographic, and again to retrieve the corresponding census population count. When calling each function pass the appropriate arguments to it including the relevant "header" string accessed from the `headers_items` tuple.

      💡 The passed in `headers_items` tuple contains the header names that specify the target population demographic, e.g., `( < vaccinated residents header >, < corresponding census population header" >)`.

```
    # Example header_items
    ('Booster_Doses_18Plus', 'Census2019_18PlusPop')
    ```

    3. Each time you call `get_attribute()` *add* the return value (an integer) to the appropriate running count.

       💡 Consider employing addition assignment (`+=`) when you call `get_attribute()` since each expression (recall that a function call is an expression) resolves to an integer.

    4. After the loop terminates `return` a two-item tuple to the caller that contains the statewide (all counties) vaccination count and the census population count for the specified population demographic.

2. After implementing `count_vaccinated` return to the `main()` function.

3. Assign a tuple containing the items "Series_Complete_18Plus" and "Census2019_18PlusPop" to the variable `header_items`.

4. Call the function `count_vaccinated()` and pass to it the arguments `vax_counties`, `vax_headers`, and `header_items`. Unpack the return value (a tuple) and assign the items to the variables `vax_total_18plus` and `census_total_18plus`.

5. Call the function `calculate_vax_pct()` and pass to it the arguments `vax_total_18plus`, `census_total_18plus`, and a `precision` value that rounds the float returned to the *second* (2nd) decimal place. Assign the return value to the variable named `vax_total_18plus_pct`.

   Below is the value you *must* return after calling `calculate_vax_pct()` with the specified arguments.

   ```
   vax_total_18plus_pct = 67.88
   ```

9.9 Challenge 09

Task: The National Center for Health Statistics (NCHS) has developed an Urban-Rural county classification scheme in order to study the health differentials between urban and rural communities. Retrieve the Michigan-specific NCHS urban-rural (UR) county schemes from a CSV file and combine the data with the nested county data in `vax_counties`.

The UR classification schemes are based on the metropolitan (MSA) and micropolitan (µSA) statistical areas formulated by US President's executive branch Office of Management and Budget's (OMB).

ur_code	ur_code_name	description

ur_code	ur_code_name	description
1	Large central metro	Counties in an MSA with a population of 1 million or more that: 1) contain the entire population of the largest principal city of the MSA, or 2) are completely contained within the largest principal city of the MSA, or 3) contain at least 250,000 residents of any principal city in the MSA.
2	Large fringe metro	Counties in an MSA with a population of 1 million or more that do not qualify as large central metro.
3	Medium metro	Counties in an MSA with a population between 250,000-999,999 residents.
4	Small metro	Counties in an MSA with a population less than 250,000 residents.
5	Micropolitan	Counties in a µSA with a population between 10,000-50,000 residents.
6	Noncore	Rural counties not in a µSA.

1. Implement the function named `get_ur_scheme`. Review the function's docstring regarding its expected behavior, parameters, and return value.

   Function requirements and hints

   1. Loop over `ur_codes`. Inside the loop block implement a conditional statement. The `if` statement *must* perform a *case insensitive* string comparison of the nested list's "county" name element against the passed in `county_name` value.

   2. If a name match is obtained return a three-item tuple containing the values listed below and in the specified order:

      1. cbsa_title
      2. ur_code
      3. ur_code_name

   3. Return the tuple from *inside* the `if` block. As you assemble the `return` statement convert the "ur_code" value from a string to an integer by passing it to the function `convert_to_int()`.

   4. If no match is obtained the function should return `None` *implicitly* (i.e., an explicit `return None` statement is *not* required).

2. After implementing `get_ur_scheme()` return to the `main()` function.

3. Call the `read_csv` function and retrieve the Michigan-specific NCHS urban/rural UR data contained in the file `mi_ur_codes.csv`. Assign the return value to a variable named `ur_data`.

4. The "headers" row constitutes the first element in `ur_data`. Access the element and assign it to a variable named `ur_headers`.

5. Next access the remaining "row" elements and assign them to a variable named `ur_schemes`.

6. Implement a `for i in range()` loop to generate a sequence of numbers to loop over. Set the `range` object's `stop` argument to the length of the `vax_counties` list.

7. During each loop iteration employ the subscript operator `[]` and the loop variable `i` to access a `vax_counties` nested list element (a county).

8. Inside the loop block perform the following tasks:

    1. Call the function `get_attribute()` and supply it with the arguments required to retrieve the nested county's name element. Assign the return value to a variable named `county_name`.

    2. Call the function `get_ur_scheme()` and pass it the arguments required to retrieve the county's UR scheme. Unpack the return value (a three-item `tuple`) and assign the items to the variables `cbsa_title`, `ur_code`, and `ur_code_name`.

    3. *Insert* the three UR values **one at a time** into the `vax_counties` nested list element, so that the UR scheme values constitute the third (3rd), fourth (4th), and fifth (5th) elements in the mutated nested list.

        Below is an example of a mutated `vax_counties` list:

        ```
        [
           '10/05/2022', 'Shiawassee County', 'Owosso, MI', 5,
        'Micropolitan', 35412, 35391, 2368,
           34854, 33023, 10955, 20341, 20341, 20204, 19625, 8429, 2488,
        'A', 2, 6, 4, 68122, 64572,
           10630, 59213, 53942, 12971
        ]
        ```

9. After the loop terminates *insert* the corresponding `ur_headers` elements **one at a time** into `vax_headers` so that the UR scheme header names constitute the third (3rd), fourth (4th), and fifth (5th) elements in the mutated `vax_headers` list. This ensures that the number and order of the `vax_headers` elements remain synchronized with the mutated `vax_counties` list.

❗ `ur_headers` is a short list. Employ indexing to access each `ur_headers` element that you need to insert into `vax_headers` when constructing your insert expressions.

💡 Utilize the debugger or the built-in function `print()` to check if the `vax_counties` nested lists have all been mutated.

The mutated `vax_headers` list *must* match the following list:

```
[
   'Date', 'Recip_County', 'cbsa_title', 'ur_code', 'ur_code_name',
'Series_Complete_Yes', 'Series_Complete_5Plus', 'Series_Complete_5to17',
'Series_Complete_12Plus',
   'Series_Complete_18Plus', 'Series_Complete_65Plus', 'Booster_Doses',
'Booster_Doses_5Plus', 'Booster_Doses_12Plus', 'Booster_Doses_18Plus',
'Booster_Doses_65Plus',
```

```
   'Second_Booster_65Plus', 'SVI_CTGY', 'Series_Complete_Pop_Pct_SVI',
   'Series_Complete_Pop_Pct_UR_Equity', 'Booster_Doses_Vax_Pct_SVI',
'Census2019',
   'Census2019_5PlusPop', 'Census2019_5to17Pop', 'Census2019_12PlusPop',
'Census2019_18PlusPop', 'Census2019_65PlusPop'
]
```

9.10 Challenge 10

Task: The final challenge involves accumulating counts of Michigan counties based on their UR classification. Once you have derived the totals you will write the mutated `vax_counties` and `vax_headers` lists to a file.

1. Assign a sensible start value to each of the following variables: `large_central_and_fringe_metro`, `medium_and_small_metro`, `micropolitan`, and `non_core`.

   Each variable represents a county urban-rural (UR) classification. You will check each nested county list's "ur_code" in `vax_counties` and increment the matching UR classification count.

2. Loop over `vax_counties`. Inside the loop block perform the following tasks:

   1. Call the function `get_attribute()` and pass it the arguments required to retrieve a county's "ur_code" value. Assign the return value to a variable named `ur_code`.

   2. Employ `if-elif-else` conditional logic that increments the UR classification counts according to the following rules:

      - Increment the `large_central_and_fringe_metro` count by `1` if the `ur_code` equals `1` or `2`.
      - Increment the `medium_and_small_metro` count by `1` if the `ur_code` equals `3` or `4`.
      - Increment the `micropolitan` count by `1` if the `ur_code` equals `5`.
      - Increment the `non_core` count by `1` if the `ur_code` equals `6`.

      ❗ For this challenge you *must* write **four (4) conditional statements**, one of which must be an `else` condition that increments at least one of the UR classification counts.

   Below are the values that you *must* accumulate:

   ```
   large_central_and_fringe_metro count = 10
   medium_and_small_metro count = 16
   micropolitan count = 25
   noncore count = 32
   ```

3. After the loop terminates call the function `write_csv()` and pass the following values arguments in *reverse* order employing keyword arguments:

- filepath "stu-mi_ur_county_vax_levels.csv"
- `vax_counties`
- `vax_headers`

4. Compare the CSV file you produce to the `fxt-*.csv` test fixture file per the instructions below.

9.11 Compare `stu-*.csv` and `fxt-*.csv` CSV files

As noted above the midterm includes a test fixture CSV file (prefixed with `fxt-`) that represents the correct file output that *must* be generated by the program you write. You should compare the file you produce against the text fixture to confirm that what you produce matches the expected output.

💡 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.

FINIS 🎉