SI 506 Lecture 09

Topics

1. Warmup
   1. Challenge 01
   2. Challenge 02
2. Additional control flow statements
   1. `break` statement
   2. `continue` statement
3. Indefinite iteration: `while` loop
   1. Infinite loops
   2. `while` loop `else` condition
   3. `while` loop and conditional statements
   4. `while` loop and the `range` type
   5. Challenge 03
   6. Challenge 04
4. Built-in `input()` function

Vocabulary

- **Boolean**. A type (`bool`) or an expression that evaluates to either `True` or `False`.
- **Conditional Statement**. A statement that determines a computer program's *control flow* or the order in which particular computations are to be executed.
- **Index**. Numeric position of an element or item contained in an ordered sequence. Python indexes are zero-based, i.e., the first element's index value is 0 not 1. `len(< some_list >)` is considered an expression.
- **Iterable**. An object capable of returning its members one at a time. Strings, lists, and tuples are examples of an iterable.
- **Iteration**. Repetition of a computational procedure in order to generate a possible sequence of outcomes. Iterating over a `list` using a `for` loop is an example of iteration.
- **Operator**. A symbol for performing operations on values and variables. The assignment operator (`=`) and arithmetic operators (`+`, `-`, `*`, `/`, `**`, `%`, `//`).
- **Subscript operator**. Square brackets (`[]`) enclosing either an index value or a slicing expression that is used to access individual or groups of sequence characters, elements or items.

Reference

Open the following w3schools reference pages in your browser and bookmark them. The pages provide useful summaries of `str`, `list`, and `tuple` methods.

1. w3schools, "Python Built-in Functions" or python.org "Built-in Functions".
2. w3schools, "Python Operators".

3. w3schools, "Python String Methods"
4. w3schools, "Python List Methods"
5. w3schools, "Python Tuple Methods".

Lecture data

Today's lecture data was retrieved by accessing the US Department of Energy's National Renewable Energy Laboratory (NREL) API (Application Programming Interface). This involved utilizing the third-party Python Requests library to issue an HTTP GET request to retrieve information about electric vehicle (EV) charging stations located in Ann Arbor and Ypsilanti, Michigan.

Below is a list of data retrieved for this week's lectures. Note that it represents a curated selection of information that can be sourced from the NREL's API.

Column	Description
id	Unique identifer assigned to a station
station_name	The name of the station
facility_type	Facility type
access_code	A description of who is allowed to access the station (public
access_days_time	Hours of operation
restricted_access	For public stations, an indication of whether the station has restricted access, given as a boolean (true
city	Municipality in which the station is located.
zip	The ZIP code (postal code) of the station's location.
street_address	The street address of the station's location.
intersection_directions	Brief additional information about how to locate the station.
ev_network	Network that maintains the station.
ev_connector_types	Connector type(s) available at the station.
ev_dc_fast_num	The number of DC Fast EVSE ports.
ev_level1_evse_num	The number of Level 1 EVSE ports.
ev_level2_evse_num	The number of Level 2 EVSE ports.
ev_other_evse	The number and type of additional EVSE ports, such as: SP Inductive - Small paddle inductive, LP Inductive - Large paddle inductive, and/or Avcon Conductive.
ev_pricing	Pricing details.
date_last_confirmed	The date the station's details were last confirmed.

1.0 Warmup

Let's explore the EV `station_data` by completing a couple of challenges.

1.1 Challenge 01

Task: Return a count of all ChargePoint network stations using the `station_data` "headers" list to look up the index position of a station's "ev_network".

1. Access the "headers" list from `station_data` and assign it to the variable named `headers`.

2. Assign zero (`0`) to the variable named `chargepoint_count`.

3. Implement a `for` loop to iterate over the station lists in `station_data`.

4. Inside the loop block, write an `if` statement that tests two strings for **equality**. The comparison to be performed *must* be **case insensitive** and the data to compare is the current station's "ev_network" value and the string "chargepoint network".

5. Instead of hard-coding the ev_network index value in the expression `station[< index >]` employ the `headers` list to look up the "ev_network" index by calling the appropriate `headers` list method. Locate the method call inside the station's subscript operator (`[]`).

6. If conditional statement resolves to `True` increment `chargepoint_count` by one (`1`).

7. Uncomment `print()` and check your work.

1.2 Challenge 02

Task: Update station names that feature the A2DDA acronym with "Ann Arbor Downtown Development Authority -".

1. Implement a `for` loop that iterates over a sequence of numbers provided by the `range` type.

2. Inside the loop block, employ the `headers` list to look up the "station_name" index by calling the appropriate `headers` list method. Assign the index value to a variable named `idx`.

3. Next, write an `if` statement that tests for the existence of a substring in a string (i.e. a membership test). If the substring **"A2DDA"** is found in the station name, call the appropriate string method that will return a new version of the string that swaps out the substring in favor of **"Ann Arbor Downtown Development Authority -"**. Assign the new string to the current station's "station_name" element.

4. Uncomment `print()` and check your work.

2.0 `break` and `continue` statements

You can interrupt control flow inside a loop using the `break` and `continue` statements.

2.1 `break` statement

The `break` statement is employed in a `for` loop to exit the loop and proceed to the next statement in the code. Any statements inside the loop that follow the `break` statement will be ignored. The break statement is usually triggered by a specified condition. Using a `break` statement prevents unnecessary looping and can result in performance gains if the sequence being looped over is large.

For example, if you needed to confirm that today's data set includes Ypsilanti EV station data, the following loop would provide you with the answer:

```python
has_ypsi = False
for station in station_data[1:]:
    if station[6].lower() == 'ypsilanti':
        has_ypsi = True
        break # exit loop
```

💡 Consider leveraging the "headers" list to look up the city column's index value by replacing the hard-coded subscript operator index value with the expression `headers.index('city'):`

```python
headers = station_data[0] # column headers
has_ypsi = False
for station in station_data[1:]:
    if station[headers.index('city')].lower() == 'ypsilanti':
        has_ypsi = True
        break # exit loop
```

2.2 `continue` statement

The `continue` statement is employed in a `for` loop to end the current iteration and proceed directly to the next iteration in the loop (if any), skipping any trailing statements.

In the example below, the goal is to return a list of electric vehicles that represent "outliers" in terms of city driving battery range. If a vehicle's range is between 225 miles and 325 miles (exclusive) the `continue` statement is executed and the trailing `list.append()` operation is skipped. Only vehicles with a city range that falls on either side of the 225 - 325 mile range are added to the `outliers` list.

💡 Use comparison operators arranged as `x < y < z` or `x <= y <= z` to test if a value (typically a number but also a letter) is *between* two values. The expression returns either `True` or `False`.

```
outliers = []
for vehicle in elec_vehicles[1:]:
    vehicle_range = int(vehicle[-1]) # Do not name the variable range (shadows the range() type)
    if 225 < vehicle_range < 325:
        continue # proceed to next iteration (skip)
    outliers.append(f"{vehicle[1]} {vehicle[2]} (range = {vehicle_range} mpge")
```

3.0 Indefinite iteration: the `while` loop

The `while` loop repeats a set of one or more statements *indefinitely*; that is, until a condition is imposed that evaluates to `False` and terminates the loop.

```
while < expression >:
    < statement A >
    < statement B >
```

In the example below, a counter `i` is initialized with a default value of zero (`0`). The `while` loop, once initiated, will continue to iterate over the loop block *indefinitely* until the expression `i < 5` returns `False`. Note that the only way to terminate the looping operation is to increment the counter value by `1` _inside the loop block.

```
i = 0
while i < 5:
    print(i)
    i += 1 # increment (addition assignment operator)
```

Earlier we returned a count of ChargePoint Network stations. We could reimplement the task employing a `while` loop but doing so in my opinion injects unnecessary complexity into the solution and also risks triggering an infinite loop if the counter `i` is not incremented correctly. Nevertheless, note how it is implemented.

```
chargepoint_count = 0
i = 1 # skip the header list
while i < len(station_data):
    if station_data[i][headers.index('ev_network')].lower() == 'chargepoint network':
        chargepoint_count += 1
    i += 1 # increment
```

3.1 Infinite loops

If a `while` loop is implemented incorrectly it will trigger an *infinite loop*, a runaway process that, over time, will consume ever greater memory resources to the detriment of your both your operatings system and hardware (you will hear the fans kick on as the laptop's internal temperature rises). Eventually, your system will crash unless you kill the process.

Typically, a `while` loop is implemented when the number of required iterations is unknown. There are other use cases, one of which we will explore below.

The following example is guaranteed to trigger an infinite loop since the `while` condition remains `True` indefinitely:

```python
while True:
    print("infinite loop triggered") # Don't do this
```

You can tame a `while` loop condition initialized to `True` by adding a conditional statement that includes a `break` statement in the loop code block.

```python
i = 0
while True:
    print('infinite loop triggered')
    if i == 5:
        print('infinite loop terminated\n')
        break # exit the loop
    i += 1 # increment (note indention)
```

❗ if you trigger an infinite loop while running your module in VS Code click the terminal pane's trash can icon in order to kill the session and end the runaway process.

3.2 `while` loop `else` condition

The `while` loop includes a built-in `else` condition that you can use to execute one or more statements after the loop terminates.

```python
i = 0
while i < 5:
    print('I want an EV.')
    i += 1 # increment
else:
    print('Enough said. We believe you.')
```

3.3 `while` loop and conditional statements

You can employ conditional statements inside a `while` loop in order to determine the control flow of each iteration. In the following example the modulus (`%`) operator is used to identify even and odd numbers between 0 and 10.

💡 use the modulus operator to return the remainder after one number is divided by another. If the remainder equals zero the number evaluated is an even number.

```python
i = 0
while i < 10:
    if i % 2 == 0:
        print(f"{i} is an even number.")
    else:
        print(f"{i} is an odd number.")
    i += 1 # increment

i = 10
while i >= 0:
    if i % 2 == 0:
        print(f"{i} is an even number.")
    else:
        print(f"{i} is an odd number.")
    i -= 1 # decrement
```

3.4 `while` loop and the `range` type

You can employ a `while` loop in conjunction with the `range` type to loop over a sequence of numbers. In the example below the `while` loop iterates over the sequence `0, 2, 4, 6, 8` provided by `range(0, 10, 2)`.

❗ note that `i` is incremented by 2 not 1.

```python
i = 0
while i in range(0, 10, 2):
    print(f"{i} is an even number.")
    i += 2 # increment by 2
```

3.5 Challenge 03

Task: The "ev_connector_types" value is, in certain cases, a list masquerading as a string (e.g., `'CHADEMO, J1772COMBO'`). Convert this value to a list for all stations in the `station_data` list.

1. Call the appropriate list method and return the "headers" index value for the element "ev_connector_types". Assign the value to the variable named `idx`.

2. Assign the number one (`1`) to a "counter" variable named `i`. This value will keep track of the number of `while` loop iterations.

3. Implement a `while` loop that employs the `range` type to traverse a sequence of numbers starting with the number one (`1`). Sync the `stop` value to the size of the `station_data` list.

4. Inside the `while` loop call the appropriate string method to convert each station's "ev_connector_type" string value to a list:

   `'CHADEMO, J1772COMBO'` -> `['CHADEMO', 'J1772COMBO']`

   Replace the "ev_connector_types" element with the new `list` value.

   Do this by referencing each nested list's "ev_connector_types" element using subscript operator chaining that leverages the `idx` value. In other words, access the station element, return a list representation of it, and assign it to the current station's "ev_connector_types" element.

5. Inside the `while` loop block increment the variable `i` by one (`1`).

   ❗ In order to avoid trigger an infinite loop you *must* increment the counter `i` inside the loop.

6. Uncomment `print()` and check your work.

3.6 Challenge 04

Task: Return the index value of the first nested list in `station_data` that represents a station located in Ypsilanti, Michigan.

1. Assign `None` to the variable `first_ypsi_station_idx`. An integer representing a nested list's index will later be assigned to this variable.

2. Assign the number one (`1`) to a "counter" variable named `i`. This value will be used to skip the initial "headers" nested list in `station_data` as well as keep track of the number of `while` loop iterations.

3. Implement a `while` loop that employs the `range` type to traverse a sequence of numbers starting with the number one (`1`). Sync the `stop` value to the size of the `station_data` list.

4. Inside the `while` loop write an `if` statement that tests two strings for **equality**. The comparison to be performed *must* be **case insensitive** and the data to compare is the current station's "city" value and the string "ypsilanti". Accessing the each `station_data` nested list's "city" value will require subscript operator chaining.

   💡 Employ the `headers` list to look up the "city" index rather than hard coding the index in the subscript operator.

5. If a match is obtained call the appropriate list method *inside* the `if` statement block to return the index value of the nested list that represents the *first* Ypsilanti station encountered in

`station_data`. Assign the value to a variable name of your own choosing (e.g., `first_ypsi_station_idx`).

```
# First Ypsilanti station in list (return its index value)
 [
     '145371', 'Roundtree Place', None, 'public', '24 hours daily',
None, 'Ypsilanti', '48197', '2539 Ellsworth Rd', None, 'Electrify
America', 'CHADEMO, J1772COMBO', '6', None, None, None, None, '2022-
09-07'
     ]
```

6. Inside the `if` statement block include a control flow statement that will terminate the `while` loop whenever it is encountered.

7. Inside the `while` loop block but *outside* the `if` statement block increment the variable `i` by one (`1`).

8. Uncomment `print()` and check your work.

4.0 Built-in `input()` function

The built-in `input()` function accepts user-supplied strings from the command prompt. It is often positioned inside a `while` loop in order to process user input. The pattern is illustrated in the following example.

In the example below, the built-in function `input()` prompts the user for a street name. The function call is placed inside a `while` loop in order to query the user continuously until a a street name found in the `streets` list is provided.

The `if` statement performs a *case sensitive* membership check. If a match is obtained the boolean `True` is assigned to the variable `is_found`. If an exact match is not obtained (e.g., Ann Street `!=` W Ann St), the `else` statement block provides for *case insensitive* partial matching (e.g., "Ann" `in` "W Ann St). If a partial match is obtained, the boolean `True` is assigned to the variable `is_found`. The `break` statement is then employed to exit the `streets` loop.

Thereafter, if `is_found` resolves to `True` the built-in function `print()` is called and passed the "SUCCESS" string. A second `break` statement is added to exit the `while` loop. If `is_found` remains `False` the built-in function `print()` is called and passed the "FAIL" string. The `while` loop proceeds to the next iteration of the loop and the user is again prompted to provide a street name.

💡 The variable `is_found` is key to the design of the `while` loop. The conditional logic inside the `while` loop requires a way to signal that a match has been obtained and that both the inner `streets` loop and outer `while` loop can be exited. The `is_found` *truth value test* (i.e., `if is_found:`) also eliminates the need to duplicate "SUCCESS" calls to the built-in function `print()`. This is an example of the DRY principle in action.

```python
streets = (
    'Ann Arbor-Saline Rd',
    'Auto Mall Dr',
    'Boardwalk Dr',
    'Broadway St',
    ...
    'W Ann St',
    'W Liberty Rd',
    'W Washington',
    'W William St'
    )

while True:
    is_found = False
    entry = input('\nProvide street name: ')

    # Attempt to obtain an exact match; otherwise attempt to obtain a partial match
    if entry in streets:
        is_found = True # exact match obtained;
    else:
        for street in streets:
            if entry.lower() in street.lower():
                is_found = True
                break # partial match obtained, exit streets loop

    if is_found: # truth value test
        print(f"\nSUCCESS: One or more EV charging stations found on the provided street.")
        break # exit while loop

    print(f"\nFAIL: No EV charging stations found on provided street. Provide a different street name.")
```