SI 506 Lecture 07

Topics

1. Nested lists
2. Looping and counting
3. Looping with the `range` type
4. `if-else` statements

Vocabulary

- **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. Both strings and lists 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 (`+`, `-`, `*`, `/`, `**`, `%`, `//`).

1.0 Nested lists

The Python *sequence* is a container data type that holds objects that can be accessed individually or in groups by their position. Both strings and lists are sequences. The `str` data type comprises an ordered set of *immutable* characters while the `list` data type comprises an ordered set of *mutable* elements.

The lists that you've worked with thus far have consisted of strings and/or numbers. In the example below, each element in the `elec_vehicles` list represents an electric vehicle (EV). Each string contains a set of attributes (automaker, model, model year, range in mpg) that are delineated by use of a comma and space as a separator.

```
# EV attributes: automaker, model, model year, range (miles)
elec_vehicles = [
    'Ford, Mustang Mach-E AWD, 2021, 211',
    'Kandi, K27, 2021, 59',
    'Chevrolet (GM), Bolt EV, 2021, 259',
    'Audi (Volkswagen), e-tron, 2021, 222',
    'Nissan, Leaf (40 kW-hr battery pack), 2021, 149',
    'Tesla, Model 3 Performance AWD, 2021, 315',
    'Volvo, XC40 AWD BEV, 2021, 208',
    'Volkswagen, ID.4 1st, 2021, 250',
    'BMW, i3s, 2021, 153',
    'Mini (BMW), Cooper SE Hardtop 2 door, 2021, 110',
```

```
    'Tesla, Model S Performance (19in Wheels), 2021, 387'
    ]
```

Iterating over the `elec_vehicles` list in order to access each vehicle's attributes requires that each string encountered be split into a list using the `str.split()` method in order to access the desired attribute(s) by position via indexing.

```
models = []
for element in elec_vehicles:
    vehicle = element.split(', ') # a new list
    models.append(f"{vehicle[0]} {vehicle[1]}") # f-string
```

💡 note that you can also split the string into a list and then return a slice of it by appending the slicing notation to the expression.

```
vehicle = element.split(', ')[:2] # a new list comprising the first two
elements only
```

Iterating over a list of strings and generating a list "on the fly" in order to access each string element's delineated attributes suggests that the string-based EV data storage strategy is not efficient. Since lists (and tuples) can hold more complex data types we should instead consider representinng each EV as a list rather than a string. Doing so creates a more efficient data structure: a list of lists or *nested list*.

```
# EV attributes: automaker, model, model year, range (miles)
elec_vehicles = [
    ['Ford', 'Mustang Mach-E AWD', '2021', '211'],
    ['Kandi', 'K27', '2021', '59'],
    ['Chevrolet (GM)', 'Bolt EV', '2021', '259'],
    ['Audi (Volkswagen)', 'e-tron', '2021', '222'],
    ['Nissan', 'Leaf (40 kW-hr battery pack)', '2021', '149'],
    ['Tesla', 'Model 3 Performance AWD', '2021', '315'],
    ['Volvo', 'XC40 AWD BEV', '2021', '208'],
    ['Volkswagen', 'ID.4 1st', '2021', '250'],
    ['BMW', 'i3s', '2021', '153'],
    ['Mini (BMW)', 'Cooper SE Hardtop 2 door', '2021', '110'],
    ['Tesla', 'Model S Performance (19in Wheels)', '2021', '387']
    ]
```

Information in a nested list is accessed by position using subscript notation. No recourse to string splitting is required to access each EV's attributes and store the information in the `models` list.

```
models = []
for vehicle in elec_vehicles:
    models.append(f"{vehicle[0]} {vehicle[1]}")
```

2.0 Looping and counting

The built-in `len()` function provides us with the overall length or size of a list. But if you want to return a count of a subset of a sequence in which slicing cannot be used, then consider using an accumulating "counter" variable to hold a rolling count of the elements that satisfy a given condition.

In the following example a count of the number of vehicles manufactured by BMW is accumulated. A default value of zero (`0`) is assigned to the `bmw_count` variable. The variable is utilized to accumulate a count of the number of nested lists that represent EVs produced by BMW.

💡 Note use of the "assignment addition" operator `+=` to *increment* the count. The expression `bmw_count += 1` is the equivalent to `bmw_count = bmw_count + 1`, an example of Python "syntatic sugar" that I encourage you to use.

```
bmw_count = 0
for vehicle in elec_vehicles:
    if vehicle[0] == 'Bayerische Motoren Werke AG':
        bmw_count += 1 # assignment addition equivalent to bmw_count =
bmw_count + 1
```

:bulb You can also perform "assignment subtraction using the `-=` operator to *decrement* a count.

Challenge 01

Return a count of the number of electric vehicles with a range greater than or equal to 250 miles.

```
mpg_count = 0

# TODO Implement loop / conditional statement
```

3.0 Looping with the `range` type

Although the Python official documentation includes `range()` in its table of built-in functions, the `range` object is actually an immutable sequence type like a `list` or a `tuple`.

The `range` object is employed to generate an *immutable* sequence of numbers. The Default behavior starts the sequence at 0 and then increments by 1 up to but *excluding* the specified stop value passed to the object as an argument. Optional start and step arguments can be passed to `range()` including negative step values that reverse the sequence.

```
range([start,] stop[, step])
```

3.1 `range` behaviors

To work with the `range` object outside a `for` loop, convert the sequence it generates to a list by passing it to the built-in function `list()`.

```
seq = range(10) # instantiate the range object with a stop argument of 10

seq = list(range(10)) # returns [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

seq = list(range(5, 10)) # returns [5, 6, 7, 8, 9]

seq = list(range(5, 21, 5)) # returns [5, 10, 15, 20]

seq = list(range(20, 4, -5)) # returns [20, 15, 10, 5]
```

3.2 The `for` loop and `range`

You can use the `range` object to specify the maximum number of `for` loop iterations. In the following looping example, the expression `range(5)` returns the numeric sequence `0, 1, 2, 3, 4`. Consequently, the built-in function `print()` is called a total of five (5) times before the loop terminates.

```
for i in range(5):
    print("I want to own an EV!")
```

You can also pass the built-in function `len(< sequence >)` to `range` in a `for` loop in order to limit the number of loop iterations to the length of the list. Doing so aligns the numeric sequence returned by `range` (`0, 1, 2, ...`) to the index values of the list elements.

```
automakers = [
    'Bayerische Motoren Werke AG',
    'Ford Motor Co.',
    'General Motors Co.',
    'Kandi Technologies Group',
    'Nissan Motor Co.',
    'Volkswagen AG',
    'Volvo Group',
    'Tesla, Inc.'
    ]

for i in range(len(automakers)):
    print(f"{i} {automakers[i]}")
```

3.3 Employing `range` to replace list elements

If you need to replace a list element with another value utilize a `for i in range():` loop to accomplish the task. Employing a basic `for` loop to perform the assignment will not change the underlying element, it will only repoint the loop variable to the new value leaving the underlying element unchanged:

```
for automaker in automakers:
    automaker = automaker.upper() # assigns new string to loop variable
only

# List elements are unchanged
# [
#     'Bayerische Motoren Werke AG',
#     'Ford Motor Co.',
#     'General Motors Co.',
#     ...
# ]
```

Looping over the numeric sequence generated by `range` permits one to reference each targeted list element by its index. The assignment of a new value can then be performed successfully.

```
for i in range(len(automakers)):
    automakers[i] = automakers[i].upper() # assigns new string to element

# List elements are changed
# [
#     'BAYERISCHE MOTOREN WERKE AG',
#     'FORD MOTOR CO.',
#     'GENERAL MOTORS CO.',
#     ...
# ]
```

3.4 Subscript notation chaining

Accessing nested list attributes by position is achieved using subscript notation chaining. Obtaining the Tesla Model S EV's range value from the `elec_vehicles` list is achieved by first accessing the vehicle list by its index position (`-1` or `10`) and then accessing the vehicle's range value by its index position (`-1` or `3`) in a "chained" expression.

```
elec_vehicles = [
    ['Ford', 'Mustang Mach-E AWD', '2021', '211'],
    ['Kandi', 'K27', '2021', '59'],
    ['Chevrolet (GM)', 'Bolt EV', '2021', '259'],
    ['Audi (Volkswagen)', 'e-tron', '2021', '222'],
    ['Nissan', 'Leaf (40 kW-hr battery pack)', '2021', '149'],
    ['Tesla', 'Model 3 Performance AWD', '2021', '315'],
    ['Volvo', 'XC40 AWD BEV', '2021', '208'],
    ['Volkswagen', 'ID.4 1st', '2021', '250'],
    ['BMW', 'i3s', '2021', '153'],
    ['MINI (BMW)', 'Cooper SE Hardtop 2 door', '2021', '110'],
    ['Tesla', 'Model S Performance (19in Wheels)', '2021', '387']
    ]
```

```
tesla_s_range = elec_vehicles[-1][-1]
# tesla_s_range = elec_vehicles[-1][3] # Alternative
# tesla_s_range = elec_vehicles[10][3] # Alternative
# tesla_s_range = elec_vehicles[10][-1] # Alternative

tesla_s_range = 0
for i in range(len(elec_vehicles)):
    if elec_vehicles[i][1] == 'Model S Performance (19in Wheels)':
        tesla_s_range = elec_vehicles[i][-1]
```

❗ In line with method chaining each chained expression employing subscript notation resolves to a value. Be mindful when calling a method on the value, you can trigger an `AttributeError` if you lose track of the value's type and call a method not possessed by the type.

Challenge 02

Task. Use `range`, the built-in function `len()`, and a `for` loop to replace each vehicle's model name in `elec_vehicles` with a new string that combines the automaker's name with the model name. Format the new string as follows:

`'< automaker > < model >'`, e.g. `'Mustang Mach-E AWD'` -> `'Ford Mustang Mach-E AWD'`

💡 When looping over the numeric sequence employ subscript notation chaining to access the vehicle attributes in the nested lists.

```
# TODO Implement for loop (two lines of code is all that is required)

# Mutated list
[
    ['Ford', 'Ford Mustang Mach-E AWD', '2021', '211'],
    ['Kandi', 'Kandi K27', '2021', '59'],
    ['Chevrolet (GM)', 'Chevrolet (GM) Bolt EV', '2021', '259'],
    ['Audi (Volkswagen)', 'Audi (Volkswagen) e-tron', '2021', '222'],
    ['Nissan', 'Nissan Leaf (40 kW-hr battery pack)', '2021', '149'],
    ['Tesla', 'Tesla Model 3 Performance AWD', '2021', '315'],
    ['Volvo', 'Volvo XC40 AWD BEV', '2021', '208'],
    ['Volkswagen', 'Volkswagen ID.4 1st', '2021', '250'],
    ['BMW', 'BMW i3s', '2021', '153'],
    ['MINI (BMW)', 'MINI (BMW) Cooper SE Hardtop 2 door', '2021', '110'],
    ['Tesla', 'Tesla Model S Performance (19in Wheels)', '2021', '387']
]
```

Challenge 03 (Bonus)

Task: Use `range`, the built-in function `len()`, and a `for` loop to assign *every other* vehicle in `elec_vehicles` to an accumulator list named `select_vehicles`.

```

```

```
elec_vehicles = [
    ['automaker', 'brand', 'model', 'year', 'range', 'range_hwy',
'range_city', 'highway_08_mpg', 'charge_240v_hrs'],
    ['Ford Motor Co.', 'Ford', 'Mustang Mach-E AWD', 2021, 211, 193.7,
225.5, 86, 8.5],
    ['Kandi Technologies Group', 'Kandi', 'K27', 2021, 59, 51.6, 64.3,
102, 7.0],
    ['General Motors Co.', 'Chevrolet', 'Bolt EV', 2021, 259, 235.1,
277.7, 108, 9.3],
    ['Volkswagen AG', 'Audi', 'e-tron', 2021, 222, 221.9408, 222.74, 77,
10.0],
    ['Nissan Motor Co.', 'Nissan', 'Leaf (40 kW-hr battery pack)', 2021,
149, 131.3, 163.2, 99, 8.0],
    ['Tesla Inc.', 'Tesla', 'Model 3 Performance AWD', 2021, 315, 299.0,
328.7, 107, 10.0],
    ['Volvo Group', 'Volvo', 'XC40 AWD BEV', 2021, 208, 188.0, 223.6, 72,
8.0],
    ['Volkswagen AG', 'Volkswagen', 'ID.4 1st', 2021, 250, 230.1587,
266.7659, 89, 7.5],
    ['Bayerische Motoren Werke AG', 'BMW', 'i3s', 2021, 153, 136.4, 166.5,
102, 7.0],
    ['Bayerische Motoren Werke AG', 'MINI', 'Cooper SE Hardtop 2 door',
2021, 110, 101.9, 116.9, 100, 4.0],
    ['Tesla Inc.', 'Tesla','Model S Performance (19in Wheels)', 2021, 387,
373.2, 398.3, 106, 14.7]
]

select_vehicles = []

# TODO Implement for loop (two lines of code is all that is required)
```

💡 Solving this challenge requires passing the correct start, stop, and step arguments to `range()` in the `for` loop. Consider breaking the problem down into sub-problems, solving each sub-problem in turn:

1. Use `range()` to print out all elements in `elec_vehicles`.
2. Adjust the `range()` start and stop arguments to return only the vehicles (skip the header list).
3. Add a step argument to return every other vehicle.

3.0 `if-else` conditions

Execution of an `if` statement's indented code block occurs *only* if the condition to be tested evaluates to `True`. If `False` is returned and a need exists to execute other statements in response an `else` statement can be added togetther with an indented code block.

```
if < condition >:
    < statement A >
    # ...
else:
```

```
    < statement B >
    # ...
```

The `if-else` block below evaluates an electric vehicle's battery charge period; if the period is less than 8 hours it is considered a "short charge" and the element is appended to the `short_charge` list; otherwise the vehicle is appended to the `long_charge` list.

```
headers = elec_vehicles[0] # look up index
short_charge = []
long_charge = []
for vehicle in elec_vehicles[1:]:
    if vehicle[headers.index('charge_240v_hrs')] < 8.0:
        short_charge.append(vehicle)
    else:
        long_charge.append(vehicle)
```

A second example illustrates how to check if a value exists between a range of values. Assume that the automotive industry considers that a standard EV battery charge time range between six (6.0) and ten (10.0) hours *exclusive* (i.e., excludes both the minimum and maximum values).

The expression can be written as follows:

```
6.0 < some_numeric_value < 10.0 # Evaluates to True or False
```

💡 if the minimum and/or maximum values are considered *inclusive* (i.e., part of the the range of values under consideration) use the less than or equal to (`<=`) or greater than or equal to (`>=`) comparison operators in the expression.

```
standard_charge = []
outliers = []
for vehicle in elec_vehicles[1:]:
    if 6.0 < vehicle[headers.index('charge_240v_hrs')] < 10.0:
        standard_charge.append(vehicle)
    else:
        outliers.append(vehicle)
```

Challenge 04

Task. Return a count of the number of EVs in the list `elec_vehicles` with a range greater than or equal to 250 miles and a count of the EVs in `elec_vehicles` with a range that is less than 250 miles. Assign the counts to the variables `range_high` and `range_low` respectively.

```

```

```
range_high = 0
range_low = 0

# TODO Implement loop
```