SI 506 Lecture 07

Topics

1. Iteration and control flow
    1. Nested lists
    2. Definite iteration: `for` loop
    3. The `if` statement
    4. Challenge 01
2. The accumulator pattern
    1. Accumulating counts
    2. Challenge 02
3. Looping with the `range` type
    1. `range` behaviors
    2. The `for` loop and `range`
    3. Employing `range` to replace list elements
    4. Chaining subscript operators
    5. Challenge 03

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 String Methods"
2. w3schools, "Python List Methods"

3. w3schools, "Python Tuple Methods".
4. w3schools, "Python Operators"

Lecture data

Today's lecture data is drawn from the US Dept of Energy's Alternative Fuels Data Center information on electric vehicles (EVs).

1.0 Iteration and control flow

This week we focus on *iteration* (i.e., accessing sequence elements employing loops) and *control flow* (i.e., the order in which a program or script executes). You will learn how to iterate or loop over a sequence (`list`, `range`, `str`, `tuple`) using a `for` and a `while` loop. You will also learn how to write conditional statements in order to determine which computations your code must perform. Conditional statements can be placed inside the body of a `for` loop in order to act as data filters or terminate a looping operation if a particular condition has been satisfied.

Conditional statements employ a variety of operators. As noted previously, Python operators are organized into groups. We've touched on arithmetic operators and assignment operators. Starting this week you will begin using other operators when writing conditional statements, especially comparison, identity, and membership operators. Use of logical operators such as `and`, `or`, and `not` will be discussed next week.

1.1 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 collection of *immutable* characters, the `list` data type comprises an ordered collection of *mutable* elements, and the `tuple` data type comprises an ordered collection of *immutable* elements.

The lists and tuples that you've encountered 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'
    ]
```

Accessing individual vehicle data in the list of strings requires splitting each string on the comma and space (`', '`) in order to access the desired vehicle attribute(s) by position via indexing:

```
model = elec_vehicles[0].split(', ')[1] # Mustang Mach-E AWD
```

The above operation suggests that that vehicle data could be represented as a list of lists, with each "inner" list element holding a distinct piece of information. Since lists (and tuples) can reference more complex data types we should consider representinng each EV as a "nested" list rather than a string.

```
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']
    ]
```

Accessing the first element's "model" element involves chaining the index positions.

```
model = elec_vehicles[0][1] # Mustang Mach-E AWD
```

1.2 Definite iteration: the `for` loop

Indexing and slicing sequence elements by position is standard operating procedure for Python programmers. However, if you need to interact with *all* members of a sequence indexing and/or slicing your way to success could prove tedious and inefficient. Python provides a ready solution to this challenge in the guise of the `for` loop.

Writing a `for` loop simplifies sequence traversal. You can loop over the entirety (or a portion) of a sequence confident that the loop will terminate automatically once the the last character, element, or item in the sequence (or subset of the sequence) is reached. This process is known more generally as **definite iteration**.

The keywords `for` and `in` comprise the basic syntax of the `for` loop:

```
for < element > in < sequence >:
    # indented block
    < statement A >
    < statement B >
    # ...
```

Note that the `for` loop statement is terminated by a trailing colon (`:`). The colon indicates the start of the loop's code block. The statement(s) that comprise the loop's code block *must* be indented four (4) spaces. The statements are *local* to the loop and are only executed when the loop is run.

In the following example, each element in `elec_vehicles` is assigned the "loop" variable `vehicle`. During each iteration of the loop the `vehicle` element is passed to the built-in `print()` function. After the last `vehicle` element is printed the loop terminates.

```
for vehicle in elec_vehicles:
    print(vehicle) # indented block
```

❗ Failure to employ Python's indention rules can lead to unexpected computations and/or trigger an `IndentionError`.

You can limit the number of loop iterations by targeting a subset of a sequence. In the following example, slicing is employed to restrict the number of loop iterations to the last four elements of `elec_vehicles`:

```
for vehicle in elec_vehicles[-4:]:
    print(vehicle) # indented block
```

1.3 The `if` statement

Looping just to loop is not all that useful. Data is compiled (usually) in the hope that later statistical analysis will reveal otherwise hidden patterns in the data set. Conditional logic can be embedded in a `for` loop block in order to interact with a sequence's elements in more meaningful ways.

A Python conditional statement evaluates to either `True` or `False`. Conditional statements placed inside a `for` loop block determine which computations, if any, are to be performed during each iteration of the loop. More generally, conditional statements help determine a computer program's *control flow* or the order in which individual statements are executed.

The keyword `if` comprises the basic syntax of a conditional statement:

```

```

```
if < condition >:
    # indented block
    < statement C >
    < statement D >
    # ...
```

Like the `for` loop the `if` statement is terminated with a trailing colon (`:`). The colon indicates the start of the conditional statement's code block. The statement(s) that comprise the `if` statement's code block *must* be indented four (4) spaces. The statements are *local* to the `if statement` and are *only* executed if the statement condition returns `True`.

Below is a second example in which `elec_vehicles` elements are passed to the built-in `print()` function. But in this case, the `if` statement filters on the "automaker" element and only calls `print()` if the vehicle manufacturer is Volvo.

```
for vehicle in elec_vehicles:
    if vehicle[0].find('Volvo') > -1:
        print(vehicle)
```

💡 `str.find()` returns `-1` if no match is obtained; otherwise the method call returns the index of the first occurence of the passed in substring.

If, for example, you needed to identify and print all EVs in `elec_vehicles` produced by Volkswagen you could employ the membership operator `in` as is illustrated in the example below. The conditional statement's use of `str.lower()` guarantees a *case-insensitive* membership check, evaluating whether or not the string "volkswagen" is a *substring* of the `vehicle` string (i.e., is a member of the string sequence). If `True` the built-in function `print()` is called and the `vehicle` value is printed to the terminal screen.

```
for vehicle in elec_vehicles:
    if 'volkswagen' in vehicle[0].lower():
        print(vehicle)
```

The opposite condition can also be evaluated. If you needed to identify and print all EVs in `elec_vehicles` produced by automakers other than Volkswagen you could employ the `not in` membership operator.

```
for vehicle in elec_vehicles:
    if 'volkswagen' not in vehicle[0].lower():
        print(vehicle)
```

💡 Use of `str.lower()` renders the `if` statement *case-insentive* ensuring that possible variations in the manufacturer name (e.g., "Volkswagen", "volkswagen", "VOLKSWAGEN"), will not result in skipping otherwise valid matches. This is an example of "defensive" programming. When working with string data

never assume that the data is "clean" (i.e., uniform and consistent). Note that there will be occasions when you will need to perform *case-sensitive* string matching.

1.4 Challenge 01

Task. Print to the terminal screen the elements that represent Teslas in the list `elec_vehicles`.

1. Implement a `for` loop and a conditional `if` statement to identify vehicle elements that represent Tesla EVs.
2. If the vehicle is manufactured by Tesla, pass the variable that refers to the element to the built-in `print()` function.

💡 There are several different ways that you can write the `if` statement to achieve the required filtering.

2.0 The accumulator pattern

One common programming "pattern" is to traverse a sequence (e.g., a `str`, `list`, or `tuple`), *accumulating* a value during each iteration of the loop and assigning it to another sequence created and assigned to a variable *prior* to implementing the `for` loop.

In the previous challenge instead of printing the Tesla vehicle elements to terminal screen we could have instead "accumulated" each Tesla encountered by appending the value to an empty list named `teslas`.

```
teslas = []
for vehicle in elec_vehicles:
    if vehicle[0].lower().find('tesla') > -1:
        teslas.append(vehicle)
```

Another variant of the accumulator pattern is to initialize an accumulator value, assigning it a default value that is then updated by a `for` loop whenever a certain loop condition is satisfied.

In the example below, two accumulator values are utilized in order to find the electric vehicle featuring the greatest range in miles. When looping over `elec_vehicles` the variable `max_range` is assigned an updated value if a vehicle's range (converted from a string to an integer using the built-in function `int()`) is greater than `max_range`. Likewise, the variable `vehicle_max_range` is assigned a new value whenever the `if` the condition evaluates to `True`. When the loop terminates, the nested vehicle list containing the max range value will have been assigned to `vehicle_max_range`.

```
vehicle_max_range = None
max_range = 0
for vehicle in elec_vehicles:
    vehicle_range = int(vehicle[-1]) # cast str to int
    if vehicle_range > max_range:
```

```
            max_range = vehicle_range
            vehicle_max_range = f"{vehicle[0]} {vehicle[1]}" # automaker model
```

❗ The example above does not handle ties (i.e., multiple vehicles featuring the same max range) You will learn how to handles ties later in the course.

2.1 Accumulating counts

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

In the following example a count 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 'bmw' in vehicle[0].lower():
        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.

2.2 Challenge 02

Task: Return a count of the number of EVs in `elec_vehicles` with a range greater than or equal to 250 miles.

1. Assign zero (`0`) to a variable named `vehicle_count`.
2. Implement a `for` loop and a conditional `if` statement to identify vehicles with a range *greater than or equal to* 250 miles.

   💡 Conditional statements often compare two values using the following comparison operators. The return value of such expressions is either `True` or `False`.

Operator	Description
`==`	equal

Operator	Description
`!=`	not equal
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

3. If the conditional statement evaluates to `True` increment `mpg_count` by one (`1`).

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 zero `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

You can observe how the `range` object behaves by converting the sequence it generates to a list by passing it to the built-in `list()` function.

```
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 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]}")
```

❗ Note that the above `for` loop **does not** loop over `automakers`; it loops over the sequence of numbers generated by `range`. Inside the loop block each number (referenced by the variable `i`) is utilized as an index to access each element in `automakers` by position.

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 regular `for` loop to perform the assignment will not change the underlying element, it only repoints 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
```

If you need to target select elements for modification you can pass `start` and `step` values to `range` as in the following example:

```
# Modify every third element commencing from index 0
for i in range(0, len(automakers), 3):
    automakers[i] = automakers[i].lower() # assigns new string element
```

3.4 Chaining subscript operators

Accessing nested list attributes by position is achieved using subscript operator 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 the subscript operator (`[]`) 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.

3.5 Challenge 03

Task. Use a `for` loop, `range`, the built-in function `len()`, and f-string to modify each vehicle model name in `elec_vehicles`.

1. Utilize `range` to loop over a sequence of numbers keyed to the number of elements in `elec_vehicles`.
2. Inside the `for` loop employ subscript operator chaining to assign a new string to each vehicle's "model" element. Each vehicle list contains the following data:

   `[< automaker >, < brand >, < model >, < year >, < range >]`
3. Construct the new string by combining the "brand" and "model" elements. as follows:

   `f"< brand > < model >"`, e.g. `'Kandi K27'`

   Then assign it to the nested list's target element.
4. The modifed list *must* match the following 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']
]
```