SI 506 Lecture 06

Topics

1. Control flow
2. Definite iteration: the `for` loop
3. The `if` statement
4. The accumulator pattern
5. Looping and slicing

Vocabulary

- **Conditional Statement**. A statement that determines a computer program's *control flow* or the order in which particular computations are to be executed.
- **Expression**. An accumulation of values, operators, and/or function calls that return a value.
- **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.
- **Method**. A function defined by and bound to an object. For example the `str` type is provisioned with a number of methods including `str.strip()`.
- **Operator**. A symbol for performing operations on values and variables. The assignment operator (`=`) and arithmetic operators (`+`, `-`, `*`, `/`, `**`, `%`, `//`).
- **Slice**. A subset of a sequence. A slice is created using the subscript notation `[]` with colons separating numbers when several are given, such as in `variable_name[1:3:5]`. The bracket notation uses slice objects internally.

1.0 Control flow

This week we focus on iteration 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` 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.

2.0 Definite iteration: the `for` loop

Given a select list of model year 2021 electric passenger vehicles derived from the US Department of Energy's Fuel Economy vehicle data (18.3 MB) and a task to create a new list called `tesla_cars` that contains all Tesla vehicle data contained in `elec_vehicles` how would you accomplish the task?

```
elec_vehicles = [
    'Ford Mustang Mach-E AWD',
    'Kandi K27',
    'Chevrolet (GM) Bolt EV',
    'Audi (Volkswagen) e-tron',
    'Nissan Leaf (40 kW-hr battery pack)',
    'Tesla Model 3 Performance AWD',
    'Volvo XC40 AWD BEV',
    'Volkswagen ID.4 1st',
    'Polestar (Volvo) 2',
    'BMW i3s',
    'Mini (BMW) Cooper SE Hardtop 2 door',
    'Tesla Model S Performance (19in Wheels)'
]

# TODO Populate the list with Tesla vehicles
tesla_vehicles = [] # add elements
```

One solution would involve referencing each nested Tesla vehicle list by its index position and then appending each vehicle to `tesla_vehicles` by calling the `list.append()` method and passing to it a reference to the nested vehicle list. However, if the number of elements in the list were to grow, calling the `list.append()` method repeatedly across multiple lines of code would prove both time consuming and result in a series of repetitive statements that would prove difficult to maintain as the number of model years and/or vehicle data increases.

Indeed, if the DOE `elec_vehicles` list covered the model years 2010-2021 it would include 245 elements (including the "header" element) and numerous Tesla vehicle nested lists. If you were asked to analyze all the DOE's vehicle fuel economy data for the period 1994 to the present you would confront 43255 records.

A more efficient approach is to utilize a `for` loop to traverse the sequence. Employing a `for` loop that terminates automatically once the last character, element, or item in a sequence is reached is known more generally as **definite iteration**.

The `for` loop employs the keywords `for` and `in` as in the statement `for < element > in < sequence >:` and is terminated by a trailing colon (`:`) that 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 executed.

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

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

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

In order to find related vehicle elements among our list of strings we can employ conditional logic to evaluate each string encountered as we loop over the `elec_vehicles` list. Doing so involves implementing an `if` statement inside the `for` loop.

2.2 The `if` statement

The rationale for writing a `for` loop is usually expressed in one or more conditional statements defined within the body of the loop. The conditional statement determines which computations, if any, are to be performed during the current iteration depending on whether the condition expressed evaluates to `True` or `False`. More generally, conditional statements help determine a computer program's *control flow* or the order in which individual statements are executed.

The `if` statement is written as `if < condition >:` and is terminated with a trailing colon (`:`) that 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`.

```
if < condition >:
    # indented block
    # < statement A.1 >
    # < statement A.2 >
    # ...
```

Below is a second example in which `elec_vehicles` elements are passed to the built-in `print()` function if the string contains the substring "volvo".

```python
for vehicle in elec_vehicles:
    if vehicle.lower().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 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 performs 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.

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

💡 Note the use of `str.lower()` in the above conditional statements. 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 a *case-sensitive* match.

Challenge 01

Task. Print to the terminal screen the strings that represent Teslas in the list `elec_vehicles`. Utilize a `for` loop and a conditional `if` statement to accomplish the task.

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

3.0 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 string encountered by appending the value to an empty list named `teslas`.

```
teslas = [] # accumulator
for vehicle in elec_vehicles:
    if 'tesla' in vehicle.lower():
        teslas.append(vehicle)
```

Another variant of the accumulator pattern is to initialize an accumulator value, assigning it a default value that is then updated in a subsequent `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 with the greatest range in miles. The variable `num` is updated during a loop iteration whenever a vehicle's range is greater than `num`. Likewise, the variable `vehicle_max_range` variable is replaced with a new value if the condition is satisfied. When the loop terminates, the nested vehicle list containing the best range value will have been assigned to `vehicle_max_range`.

💡 Conditional statements often compare two values using comparison operators (`==`, `!=`, `>`, `<`, `>=`, `<=`). The return value of such expressions is either `True` or `False`.

```
# Manufacturer, 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',
    'Polestar (Volvo), 2, 2021, 233',
    'BMW, i3s, 2021, 153',
    'Mini (BMW), Cooper SE Hardtop 2 door, 2021, 110',
    'Tesla, Model S Performance (19in Wheels), 2021, 387'
    ]

vehicle_max_range = None
num = 0
for element in elec_vehicles:
    vehicle = element.split(', ') # split the string
    if int(vehicle[-1]) > num:
        num = int(vehicle[-1]) # WARN: ignores ties
        vehicle_max_range = vehicle
```

Challenge 02

Task. Loop over `elec_vehicles` and leverage the accumulator pattern to return the electric vehicle with the *shortest* range and assign it to the variable `vehicle_min_range`.

```
vehicle_min_range = None
num = None

# TODO Implement loop / conditional statement
```

4.0 Looping and slicing

You can also loop over slice expressions (recall that slicing a sequence returns a new sequence). In the example list below, the first element in `elec_vehicles` is not a representation of a vehicle but a list containing the column names (headers) that describe the data contained in each nested vehicle list. When working with the vehicle data you will want to exclude this element and you can do so by looping over `elec_vehicles[1:]`.

```
elec_vehicles = [

'automaker,brand,model,year,range,range_hwy,range_city,highway_08_mpg,char
ge_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',
    'Volvo Group,Polestar,2,2021,233,222.1,241.9,88,8.0',
    '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'
    ]

for element in elec_vehicles[1:]:
    < statement A >
    < statement B >
    ...
```

❗ Later in the course you will work with CSV (comma-separated value) files. A "header" row of column names is often included in the file in order to render it self-documenting (or nearly so). Be sure to exclude the header row when analyzing the actual data imported from the CSV. You can also extract the header row into its own list rather than simply ignoring or discarding it. The `headers` list elements can then be used to look up corresponding values in the accompanying data.

Challenge 03

Task. Extract the header row into a list named `headers`. Loop over the vehicles (only) in the list `elec_vehicles` and identify vehicles with a range greater than 250 miles per gallon (mpg). Accumulate each vehicle that satisifies the condition by adding the vehicle's name and range to an "accumulator" list.

1. Convert the "headers" element in `elec_vehicles` to a list and assign it to the variable named `headers`.

💡 Employ indexing to access the list element (a string) and the appropriate `str` method and delimiter value to return the list.

2. Loop over the vehicles (only) in the list `elec_vehicles`. Employ slicing to exclude the "headers" element.

3. For each vehicle string encountered in the loop, convert the string to a list in the loop block using the appropriate `str` method and delimiter value. Assign the list to a variable name of your own choosing.

4. Next, write an `if` statement inside the loop block that checks whether or not a vehicle's range is greater than 250 mpg. As you loop over the list, access each vehicle's range value by its index value (or look it up from the `headers` list employing `headers.index(< attribute name >)`) from the list created in the previous step and compare it to the integer 250 in the `if` statement (e.g., `if x > y`).

   ❗ In order to check if an EV's range is greater than 250 mpg, you will need to *convert* the range value (a string) to an integer (`int`). Employ the built-in function `int(< expression >)` to accomplish the task.

5. If the condition evaluates to `True` append the vehicle's *name* and *range* to the "accumulator" list named `hi_range_elec_vehicles`. Access the values using indexing. Format the string as follows:

   `< vehicle name > range: < vehicle range > mpg` e.g., "Bolt EV range: 259 mpg"

   💡 use a formatted string literal (f-string) to build your string.

6. Uncomment the built-in function `print()` and stream the new list to the terminal screen.

```python
# Split "headers" element into a list
headers = None

# Loop over vehicles (only) and accumulate values
hi_range_elec_vehicles = None # accumulator

# TODO Implement for loop / if statement
```