SI 506: Lecture 09

Topics

1. Compound `if` statements
2. `if-elif-else` statements
3. Challenges

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. 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.

Data

The lecture data set of select Ann Arbor Electric vehicle (EV) charging stations is sourced from the US Dept. of Energy's Alternative Fuels Data Center.

Source: https://afdc.energy.gov/fuels/electricity_locations.html#/analyze?fuel=ELEC&location_mode=address&location=Ann%20Arbor,%20MI

1.0 Compound `if` statements (comparing values using logical operators)

Recall that the expression which comprises an `if` statement returns either `True` or `False`.

You can combine conditions and compare values in a single `if` statement using the logical operators `and` (conjunction), `or` (disjunction) and `not` (negation), either singly or together in various combinations, as occurs in Boolean algebra.

❗ When crafting a compound `if` statement you *must* specify each condition in its entirety.

For example the following compound `if` statement triggers a runtime exception:

```
>>> x = 5
>>> x > 2 and < 10
  File "<stdin>", line 1
    x > 2 and < 10
              ^
SyntaxError: invalid syntax
```

The compound `if` statement *must* be written as follows:

```
>>> x = 5
>>> x > 2 and x < 10
True
```

Or better yet (in this case):

```
>>> x = 5
>>> 2 < x < 10
True
```

For example, if you want to check whether or not a charging station was equipped with between `2` and `4` (inclusive) EVSE units, you should consider carefully how you craft your compound `if` statement lest you trigger a runtime exception.

❗ The "ev_level2_evse_num" data comprise the following unique string values: `''`, `'1'`, `'2'`, `'4'`, `'9'`, and `'10'`. The presence of one or more blank values requires that we first check if the string can be converted to a number `before` we attempt the conversion. Luckily, we can guard against triggering a `ValueError` by adding the `str.isnumeric()` method (evaluates to either `True` or `False`) to the `if` statement.

```
# Separate headers from the data
headers = data[0]
stations = data[1:]

# Incorrect
station_evse = []
idx = headers.index('ev_level2_evse_num') # lookup index value
for station in stations:
    if station[idx].isnumeric() and int(station[idx]) >= 2 and <= 4: # SyntaxError: invalid syntax
        station_evse.append(f"{station[1]}: EVSEs = {station[idx]}")

# Correct
station_evse = []
idx = headers.index('ev_level2_evse_num') # lookup index value
for station in stations:
    if station[idx].isnumeric() and int(station[idx]) >= 2 and int(station[idx]) <= 4:
        station_evse.append(f"{station[1]}: EVSEs = {station[idx]}")

# Pythonic
station_evse = []
idx = headers.index('ev_level2_evse_num') # lookup index value
for station in stations:
    if station[idx].isnumeric() and 2 <= int(station[idx]) <= 4:
        station_evse.append(f"{station[1]}: EVSEs = {station[idx]}")
```

Below are examples of compound conditional statements in action.

1.1 Logical `and` operator

The logical `and` operator combines two or more conditions in a single boolean expression. *All* conditions comprising the expression *must* evaluate to `True` for the expression to evaluates to `True`; otherwise the expression evaluates to `False`.

Examples

```
< condition > and < condition > [and ...]

>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

There are eight (`8`) U-M owned charging stations in the `stations` list. If we needed to filter that group of charging stations by a particular zip code (say, `48104`) we could do so by writing a compound `if` statement that employs the logical `and` operator to join the two conditions.

```
# Separate headers from the data
headers = data[0]
stations = data[1:]

# U-M charging stations
um_count = 0
i = 0
while i < len(stations):
    if stations[i][1].startswith('U-M'):
        um_count += 1
    i += 1

# um_count = 8

# U-M charging stations filtered on a zip code
um_count_48104 = 0
i = 0
while i < len(stations):
    if stations[i][1].startswith('U-M') and int(stations[i][4]) == 48104:
        um_count_48104 += 1
    i += 1

# um_count_48104 = 2
```

Challenge 01

Task. Employ a `while` loop to access a select subset of charging stations.

1. Create an empty "accumulator" list named `um_stations_greene_st`.
2. Implement a `while` loop and an `if` statement that filters on the following charging stations:
    - U-M owned charging stations
    - Greene St locations
3. Add each station that meets *both* of the above specified conditions to the list `um_stations_greene_st`.

1.2 Logical `or` operator

The logical `or` operator combines two or more conditions in a single boolean expression. If *any* condition comprising the expression evaluates to `True` the expression evaluates to `True`; otherwise the expression evaluates to `False`.

Examples

```
< condition > or < condition > [or ...]

>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

In the example below, we can accumulate charging stations owned by Meijor or categorized explicitly as a conveniene store.

```
# EV charging stations located at Meijer or at locations categorized as a "convenience store"
conv_stores = []
i = 0
while i < len(stations):
    if 'meijer' in stations[i][1].lower() or stations[i][2].lower() == 'convenience_store':
        conv_stores.append(stations[i][1])
    i += 1

# count = 7
```

Challenge 02

Task. Employ a `while` loop to access a select subset of charging stations.

1. Create an empty "accumulator" list named `conv_stores`.

2. Implement a `while` loop and an `if` statement that filters on the following charging stations:

   - Shell locations
   - Meijer locations
   - Stations categorized as a convenience store

3. Add each station (name only) that meets *any* of the above specified conditions to the list `conv_stores`.

1.3 Logical `not` operator

The logical `not` operator reverses or negates a boolean expression. If the boolean expression evaluates to `True` the inclusion of the logical `not` operator *reverses* the value to `False`; likewise if the boolean expression evaluates to `False` the inclusion of the logical `not` operator *reverses* the value to `True`.

❗ note that the logical `not` operator reverses only the condition to which it is paired. Reversing multiple conditions requires grouping the conditions with parentheses as described below in the next section.

Examples

```
not < condition >

>>> not True
False
>>> not True and True
False
>>> not True or True
True
>>> not True and False
False
>>> not True or False
False
>>> not False
True
>>> not False and True
True
>>> not False or True
True
>>> not False or False
True
```

Most of the charging stations in the `stations` list are part of the ChargePoint network. If you needed to accumulate a count of charging stations belonging to another network or not affiliated with a network you can employ the logical `not` operator to *reverse* the booelan expression returned by the expression contained in the following `if` statement.

```
# Count EV charging stations that not part of the ChargePoint network
station_count = 0
for station in data[1:]:
    if not station[headers.index('ev_network')] == 'ChargePoint Network':
        station_count += 1

# station_count = 21
```

💡 Employing the comparison operator not equal (`!=`) provides a more readable expression than the local `not` operator in the above example:

```
station[headers.index('ev_network')] != 'ChargePoint Network'
```

Challenge 03

Task. Employ a `while` loop to access a select subset of charging stations in order to accumulate a count.

1. Create an empty "accumulator" list named `station_count`.
2. Implement a `while` loop and an `if` statement that employs the logical `not` operator to *exclude* all charging stations that feature a J1772 or J1772COMBO EV connector type from the count.
3. Accumulate the count to the variable `station_count`.

💡 Extracting the unique "ev_connector_types" values returns the following strings:

```
'J1772', 'CHADEMO J1772COMBO', '', 'J1772 TESLA', 'TESLA', 'CHADEMO J1772
J1772COMBO'
```

1.4 Grouping related expressions

You can employ parentheses `()` to group related conditions that comprise a boolean expression. Pairing the logical `not` operator with a group will reverse the grouped conditions but not conditions outside the group.

❗ Logical operator precedences is `not`, then `and`, then `or`.

Examples

```
< condition > and < condition > or < condition >
is equivalent to
(< condition > and < condition >) or < condition >

However

not < condition > and < condition > or < condition >
is equivalent to
not < condition > and (< condition > or < condition >)

>>> not False and False or False
False
>>> not False and (False or False)
False
```

If you needed to return a list of charging stations located in designated parking garages or lots you could implement the following `while` loop:

```
parking_facilities = []
i = 0
while i < len(stations):
    if (stations[i][2].lower() == 'parking_garage' or
        stations[i][2].lower() == 'pay_garage' or
        stations[i][2].lower() == 'parking_lot' or
        stations[i][1].lower().startswith('washcommcollege parking')):
        parking_facilities.append(stations[i])
    i += 1
```

The compound `if` statement can be further simplified by grouping the "facility_type" strings in a tuple:

```
parking_facilities = []
facility_types = ('parking_garage', 'pay_garage', 'parking_lot')
i = 0
while i < len(stations):
    if (stations[i][2].lower() in facility_types or
        stations[i][1].lower().startswith('washcommcollege parking')):
        parking_facilities.append(stations[i])
    i += 1
```

If there was a need to restrict the results to charging stations open 24 hours daily the `if` statement could be amended as follows:

```
parking_facilities = []
facility_types = ('parking_garage', 'pay_garage', 'parking_lot')
i = 0
while i < len(stations):
    if (stations[i][1].lower().startswith('washcommcollege parking') or
        stations[i][2].lower() in facility_types and
        stations[i][-1].lower() == '24 hours daily'):
        parking_facilities.append(stations[i]) # Washtenaw Comm College
parking INCLUDED
    i += 1
```

However, due to operator precedence the `if` statement fails to filter out several Washetenaw Community College lots which are not open 24 hours daily. Grouping the `or` conditions filters out the offending records.

```
parking_facilities = []
facility_types = ('parking_garage', 'pay_garage', 'parking_lot')
i = 0
while i < len(stations):
    if ((stations[i][1].lower().startswith('washcommcollege parking') or
        stations[i][2].lower() in facility_types) and
        stations[i][-1].lower() == '24 hours daily'):
        parking_facilities.append(stations[i]) # Washtenaw Comm College
parking EXCLUDED
    i += 1
```

2.0 `if-elif-else` conditions

Multiple conditions can be specified by including one or more `elif` conditions in between an `if-else` block. The `if-elif-else` statement chain or ladder is executed from the top downwards.

```
if < condition >:
    # < statement A >
    # ...
elif < condition >:
    < statement B >
    # ...
elif < condition >:
    < statement C >
    # ...
else:
    < statement D >
    # ...
```

The `else` statement is optional but recommended, especially for new programmers, in order to render explicit the conditional logic to be evaluated. You can also nest `if-elif-else` statement blocks. We will explore nested conditional statements during a later lecture.

Note the use of three `elif` statements in the `while` loop below to check for specific EV networks:

```python
chargepoint_count = 0
ev_connect_count = 0
evgo_count = 0
greenlots_count = 0
idx = headers.index('ev_network') # lookup index value

i = 0
while i < len(stations):
    if stations[i][idx].lower() == 'chargepoint network':
        chargepoint_count += 1
    elif stations[i][idx].lower() == 'ev connect':
        ev_connect_count += 1
    elif stations[i][idx].lower() == 'evgo network':
        evgo_count += 1
    elif stations[i][idx].lower() == 'greenlots':
        greenlots_count += 1
    i += 1

# Ann Arbor EV network charging station counts
# ChargePoint count = 28
# EV Connect count = 1
# EVgo count = 1
# Greenlots count = 2
```

Challenge 04

Task. Employ a `while` loop to accumulate counts of networked, non-networked, and network status unknown EV charging stations.

1. Create three empty "accumulator" variables initialized to zero (`0`).

   ```python
   network_count = 0
   non_network_count = 0
   network_unknown_count = 0
   ```

2. Using the `headers` list, lookup up the index value of the element `ev_network`. Assign the value to a variable named `idx`.

3. Implement a `while` loop and `if-elif-else` statements that check whether or not an "ev_network" value matches one of the following criteria

   - value is blank (`''`) -> `network_unknown_count` + 1
   - value equals "non-networked" (case insensitive check) -> `non_network_count` + 1
   - value is a string value other than "Non-Networked" -> `network_count` + 1

   If a condition resolves to `True` increment the count of the relevant "accumulator" variable by one (`1`).

💡 Extracting the unique "ev_network" values returns the following strings:

```
'Non-Networked', 'ChargePoint Network', 'EV Connect', 'Greenlots', '',
'eVgo Network', 'Tesla Destination', 'Tesla'
```