SI 506 Lecture 14

Topics

1. Truth value testing
2. Challenges (debugger demo)

Vocabulary

- **Flow of execution**. The order in which statements in a program are executed. Also referred to as *control flow*.
- **Truth Value**. In Python any object can be tested for its truth value using an `if` or `while` condition or when it is used as an operand in a Boolean operation.

Previous

- **Argument**. A value passed to a function or method that corresponds to a parameter defined for the function or method.
- **Boolean**. A type (`bool`) or an expression that evaluates to either `True` or `False`.
- **Built-in Function**. A function defined by the Standard Library that is always available for use.
- **Caller**. The initiator of a function call.
- **Conditional Statement**. A statement that determines a computer program's *control flow* or the order in which particular computations are to be executed.
- **Deep copying**. For a given mutable object (e.g., `list`) constructs a new compound object and recursively *copies* into it objects found in the original.
- **Dictionary**. An associative array or a map, wherein each specified value is associated with or mapped to a defined key that is used to access the value.
- **Expression**. An accumulation of values, operators, and/or function calls that return a value. `len(< some_list >)` is considered an expression.
- **f-string**. Formatted string literal prefixed with `f` or `F`.
- **File Object**. An object that provides a file-oriented application programming interface (API) to a either a text file, binary file (e.g., image file), or a buffered binary file. File objects include read and write methods for interacting with a file stored locally or remotely.
- **Function**. A defined block of code that performs (ideally) a single task. Functions only run when they are explicitly called. A function can be defined with one or more *parameters* that allow it to accept *arguments* from the caller in order to perform a computation. A function can also be designed to return a computed value. Functions are considered "first-class" objects in the Python eco-system.
- **Immutable**. Object state cannot be modified following creation. Strings are immutable.
- **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()`.
- **Mutable**. Object state can be modified following creation. Lists are mutable.
- **Nested Loop**. A `for` or `while` loop located within the code block of another loop.

- **Operator**. A symbol for performing operations on values and variables. The assignment operator (`=`) and arithmetic operators (`+`, `-`, `*`, `/`, `**`, `%`, `//`).
- **Parameter**. A named entity in a function or method definition that specifies an argument that the function or method accepts.
- **Scope**. The part of a script or program in which a variable and the object to which it is assigned is visible and accessible.
- **Sequence**. An ordered set such as `str`, `list`, or `tuple`, the members of which (e.g., characters, elements, items) can be accessed.
- **Shallow copying**. For a given mutable object (e.g., `list`) constructs a new compound object but inserts *references* (rather than copies) into it of objects found in the original. The `list.copy()` returns a shallow copy of the original list.
- **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.
- **Statement**. An instruction that the Python Interpreter can execute. For example, assigning a variable to a value such as `name = 'arwhyte'` is considered a statement.
- **Tuple**. An ordered sequence that cannot be modified once it is created.
- **Tuple packing**. Assigning items to a tuple.
- **Tuple unpacking**. Assigning tuple items to an equal number of variables in a single assignment. A `list` can also be unpacked.

Data

This week's data features two datasets:

1. English Premier League standings for the 2020-21 season. Each club's win, draw, loss, goals for, goals against, etc. data is stored in the file `english_premier_league-2020-21.csv`.
2. Club information (e.g., full name, nicknames, home grounds, stadium capacity, region). The data is stored in `english_clubs.csv`.

1.0 Truth value testing

In Python every object can be tested for its *truth value*. You can check an object's truth value in an `if` or `while` statement or as an operand (i.e., the value the operator operates on) in an `and`, `or` `not` Boolean operation. A value that evaluates to `True` is considered *truthy* while a value that evaluates to `False` is considered *falsy*.

For SI 506 the following values are considered "truthy" or "falsy":

Type	Value	Truth value
sequence or map (`list`, `tuple`, `str`, `dict`)	non-empty	Truthy
sequence or map (`list`, `tuple`, `str`, `dict`)	empty	Falsy
numeric (`int`, `float`)	non-zero	Truthy
numeric (`int`, `float`)	0, 0.0	Falsy
`range()`	non-empty	Truthy

Type	Value	Truth value
`range()`	empty	Falsy
Nonetype	`None`	Falsy

The following example demonstrates testing a list's truth value:

```
club_names = [] # falsy
if club_names: # evaluates to False
    print(f"\nclub_names list has {len(club_names)} elements.") # not
called
else:
    print('\nclub_names list is empty.')

club_names = ['Arsenal', 'Aston Villa'] # truthy
if club_names: # evaluates to True
    print(f"\nclub_names list has {len(club_names)} elements.") # called
else:
    print('\nclub_names list is empty.')
```

The `while` loop below checks the "truthiness" of the boolean value assigned to the variable `matched`. The loop iterations will continue indefinitely so long as `matched` remains `False` (i.e., `not matched`).

```
# Boolean operation (not)
# Select favorite "Big Six" club (user-supplied input). Run while loop
# until condition evaluates to False.
big_six_clubs = (
    'arsenal',
    'chelsea',
    'liverpool',
    'manchester city',
    'manchester united',
    'tottenham'
    )
prompt = '\nWhich big six club is your favorite?: '

matched = False
while not matched:
    club = input(prompt)
    if club.lower() in big_six_clubs:
        print(f"\nThanks. I like {club.capitalize()} too.\n\nFinis.\n")
        matched = True
```

💡 You can also check if a value is either truthy or falsy using the built-in `bool()` function.

```
club_names = [] # falsy
truth_value = bool(club_names) # returns False
```

```
club_names = ['Bournemouth', 'Southampton'] # truthy
truth_value = bool(club_names) # returns True
```

2.0 Challenges

2.1 Challenge 01

Task: Calculate the total points earned by each club during the 2020-21 season. Assign the points earned to each club's nested `standings` list.

1. Implement the function `calculate_points()`. Read the Docstring for more information.

   💡 The function can be implemented with one line of code.

2. From `main()`, append a new header to the `headers` list named 'points'.

3. Then loop over `standings` and for each club compute the points earned and append the points to the nested club list. Each club should now be represented as follows:

   ```
   ['Arsenal', 'Islington (North London)', 'Emirates Stadium', 60704, 38,
   18, 7, 13, 55, 39, 61]
   ```

4. Retrieve Chelsea's standings from the `standings` employing a function implemented in a previous challenge. Assign the return value to a variable named `chelsea`.

5. Access Chelsea's points using indexing (leverage the headers and perform a lookup of the index value) and assign the return value to a variable named `chelsea_points`.

2.2 Challenge 02

Task: Premier League clubs are *not* uniformly distributed across the nine (9) regions of England. Implement a function that returns clubs by the region in which their home is located. Test function by retrieving all West Midlands clubs.

1. Implement the function `get_region`. Read the Docstring for more information.

2. From `main()` create an empty accumulator list named `west_midland_clubs`.

3. Loop over each club's standings. For each club retrieve their region from the `club_info` list by calling the function `get_region`. Check the return value. If the value equals `West Midlands` append the club's name to `west_midland_clubs`.

2.3 Challenge 03

Task: Implement a function that returns clubs by a specified nickname. Test function by retrieving club(s) known as "The Toffees" and club(s) known as "The Blues".

1. Implement the function `get_club_by_nickname`. Read the Docstring for more information.

2. From `main()`, call `get_club_by_nickname` and pass to it `club_info`, `club_info_headers`, and the nickname "The Toffees". Assign the return value to a variable named `toffees`.

3. Call the function again, but pass the nickname `The Blues` as the third argument. Assign the return value to a variable named `blues`.

2.4 Challenge 04

Task: Return club with smallest grounds (i.e., smallest stadium capacity).

1. In `main()` loop over either the `standings` or the `club_info` list. Write a conditional statement that checks the stadium capacity of the current club against that of the previous club.

2. If the current club's stadium capacity is less the previous club's stadium capacity, assign the capacity value to an accumulator variable named `min_capacity` and assign a *formatted string literal* (f-string) comprising the club's "short name" (e.g., Arsenal), grounds, and capacity count to a second variable named `club_min_capacity` utilizing the following format:

`< short name >: < grounds > (< capacity > seats)`

❗ Revist line 328 (or thereabouts) and cast the capacity value inserted into each club list from `str` to `int`:

```
club.insert(3, int(info[club_info_headers.index('capacity')])) # recast
```

2.5 Challenge 05

Task: Generate a CSV file that provides counts of the number clubs located in each English region.

1. Call the function `read_file` and retrieve the list of regions contained in the file `english_regions.txt`. Assign the return value to a variable named `english_regions`.

2. Create an empty accumulator list named `region_counts`. You *must* store each region and its count of clubs in a tuple (< region >, < count >) that you will append to `region_counts`.

3. Employ two loops (one nested) to traverse `english_regions` and `club_info`. Utilize a counter to hold the count of clubs associated with a region. Once you have obtained a count of all clubs located in a region, append the tuple (< region >, < count >) to `region_counts`.

   ❗ Correct placement of your counter variable (assign zero (`0`) to start) is crucial to achieve the correct regional club counts. Also remember to reset your counter variable back to zero (`0`) before proceeding to the next region in the outer loop.

4. Call `write_csv` and write `region_counts` to the file `english_regions_club_count.csv` along with a header row consisting of "region" and "club_counts".

   The contents of `english_regions_club_count.csv` must match the following rows values as ordered below:

```
region,club_count
South East,2
Greater London,6
East of England,2
East Midlands,1
West Midlands,3
Yorkshire and the Humber,2
North East,1
North West,5
```

2.6 Challenge 06 (Bonus)

Task: Get the relegated clubs (bottom three in terms of points earned).

❗ The task is actually a bit more complex to accomplish because there are tie-breaking rules involving comparing goals for and goals against tallies that are not implemented in the code below. But we'll ignore ties and implement a simpler solution.

Uncomment the Challenge 06 code and review the solution. The solution limits itself to pre-midterm approaches discussed in class (e.g., accumulator pattern, for loops, list sorting, value comparisons).

First, all the club points are accessed and stored in the `club_points` list. The list is then sorted in ascending order (e.g., [23, 26, 28, ...]). Ties are ignored (e.g., if two clubs finished with 28 points tie-breaking rules would come into effect to decide which club gets relegated).

Given the points ordering we can loop over a slice of the points consisting of the first three elements, identify each relegated club by matching on the points, and append the relegated club to the accumulator list `relegated_clubs`.

```
['Sheffield United, (23 pts)', 'West Bromwich Albion, (26 pts)', 'Fulham,
(28 pts)']
```

2.7 Challenge 07 (Bonus)

Task: Return a list of club standings sorted by points (descending).

1. Review the function `determine_place`.
2. From `main`() uncomment the Challenge 07 code and run the file. Check the output streamed to the terminal and written to the file `english_premier_league_standings-2020_21.txt`.