SI 506 Lecture 11

TOPICS

1. Truth Value testing
2. Variable scope
3. Tuple unpacking
4. Challenges

Vocabulary

- **Argument**. A value passed to a function or method that corresponds to a parameter defined for the function or method.
- **Caller**. The initiator of a function call.
- **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.
- **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.
- **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. This feature of the language has now been extended to all *iterables* including lists and sets.

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
Nonetype	`None`	falsy
numeric (`int`, `float`)	non-zero	truthy
numeric (`int`, `float`)	0, 0.0	falsy
boolean	`True`	truthy
boolean	`False`	falsy
sequence (`list`, `range`, `tuple`, `str`)	non-empty	truthy

Type	Value	Truth value
sequence (`list`, `range`, `tuple`, `str`)	empty	falsy
associative array (`dict`)	non-empty	truthy
associative array (`dict`)	empty	falsy

The following example demonstrates testing a list's truth value utilizing the built-in function `bool()` which accepts an object and returns either `True` or `False` per the standard truth value testing rules:

```
cereal = []
truth_value = bool(cereal) # falsy

cereal = ["Cap'n Crunch", 'Quaker Oats Company'],

truth_value = bool(cereal) # truthy
```

The `while` loop below is designed to continue looping indefinitely as long as the `cereal` value remains *falsy*. The only way to break out of the loop is when the provided input passed to the function `find_cereal()` matches a cereal name in the nested tuples in `cereals`.

💡 Note that a `while` loop will only iterate so long as the loop expression evaluates to `True` (e.g., `while True:`). But in this case, looping needs to continue indefinitely while the `cereal` value (i.e., `None`) remains *falsy*. To achieve this, the logical operator `not` is employed to reverse the expression (e.g., `not cereal` evaluates to `True`). When `cereal` is *truthy* (e.g, points to a non-empty tuple) the expression `not cereal` evaluates to `False` and the `while` loop terminates.

```
cereals = (
    ('apple jacks', 12),
    ('cocoa puffs', 10),
    ('honey bunches of oats', 12),
    ('frosted flakes', 11),
    ('fruity pebbles', 11),
    ('raisin bran', 17)
    )


def find_cereal(cereals, cereal_name):
    """Attempts to match a cereal in < cereals > based on passed in <
cereal_name >.
    If match occurs, cereal is returned immediately to the caller and the
loop and
    function are terminated. otherwise None is returned."""

    for cereal in cereals:
        if cereal[0].lower() == cereal_name.lower():
            return cereal
```

```
prompt = '\nPlease name a high sugar content cereal: '
cereal = None
while not cereal:
    name = input(prompt)
    cereal = find_cereal(cereals, name) # attempt to match on name

    # Check truth value of cereal
    if cereal:
        print(f"\n1.0.3 {cereal[0].title()} contains {cereal[1]} grams of
sugar per serving.\n")
    else:
        prompt = '\nCereal not located. Please provide another cereal
name: '
```

2.0 Variable scope

Now that you have begun to write functions it's time to discuss Python's rules for resolving name references (i.e., variables). Accessing a variable and the object to which it is assigned depends in large part on *where* the variable is defined in your program. An object's duration or lifetime also depends in part on *where* in your program it is assigned. A variable's *scope* is limited to those parts of a program in which the variable is visible and can be accessed.

A variable defined *inside* a function is considered *local* to that function. In other words, a local variable can only be accessed from inside the function's code block. On the other hand, a variable defined outside a function in the main part of a program file or module possesses top level or *global* scope. Such a variable is visible throughout the program from the point in which it was first defined. Treat *global* variables carefully. Referencing *global* variables inside functions can have unintended effects.

Python keywords and built-in functions possess a special *built-in* scope and are also available whenever you execute a script or run your program.

In the following example the variables `cocoa_puffs` and `cocoa_puffs_truth_value` possess global scope. In contrast, the function `get_cereal_ingrediants` defines a local variable named] `ingredients`. Attempting to access this variable outside the function block will trigger a runtime `NameError` exception.

```
cocoa_puffs = [
    'General Mills',
    'Cocoa Puffs',
    [
        'Whole Grain Corn',
        'Sugar',
        'Corn Syrup',
        'Cornmeal',
        'Canola and or Rice Bran Oil'
        ]
    ]

if cocoa_puffs: # truth value
    cocoa_puffs_truth_value = True # variable now available globally
```

```
print(f"\n2.0.1: cocoa_puffs truth value = {truth_value}")


def get_cereal_ingredients(cereal):
    ingredients = cereal[2] # variable possesses local scope only
    return ingredients


# Triggers NameError: name 'ingredients' is not defined
print(f"\n2.0.2: variable w/local scope only = {ingredients}")
```

3.0 Tuples

A Python tuple (type: `tuple`) is an ordered sequence of items. Like `list` elements, tuple items can be accessed via indexing and slicings, but unlike lists, tuple values are **immutable**; like a string (type: `str`) a tuple cannot be modified once created. This feature provides optimization opportunities when working with sequences of values that either must not change or form “natural” associations ('Ann Arbor', 'MI', 'USA').

Tuples are typically defined by enclosing the items in parentheses `()` instead of square brackets `[]` as is the case with lists.

❗ A single item tuple **must** include a trailing comma (`,`) or the Python interpreter will consider the expression a string.

In a later lecture we will discuss how to compare two or more tuples using comparison operators ('=', '<', '>') in a conditional statement.

3.1 Tuple packing (item assignment)

Creating a single item tuple requires adding a trailing comma {`,`} after the item.

```
cereal = ('Rice Krispies',) # single item tuple
# cereal = 'Rice Krispies', # no parentheses (legal)
# cereal = ('Rice Krispies') # a string
```

Multiple item tuples do not require a trailing comma.

```
cereals = ('Rice Krispies', 'Corn Flakes', 'Frosted Mini-Wheats')
```

3.2 Tuple immutability

Once created a tuple is *immutable* and cannot be modified. Attempts to modify or replace any tuple item will Trigger a `TypeError` runtime exception.

```
cereals[1] = 'Fruit Loops' # TypeError: 'tuple' object does not support
item assignment
```

That said, you can use tuple concatenation (`+`) to return a new tuple.

```
cereals = (cereals[0],) + ('Fruit Loops',) + (cereals[-1],) # each a
single tuple
```

You can also create a tuple by passing an *iterable* (e.g., a sequence) to the built-in function `tuple()`:

```
cereals = tuple([cereals[0], 'Fruit Loops', cereals[-1]]) # pass a list
```

3.3 Accessing tuple items

You can access individual tuple items using both indexing and slicing.

```
fruit_loops = cereals[1] # returns string

cereals_subset = cereals[-2:] # returns tuple
```

3.4 Tuple unpacking (multiple assignment)

Unpacking in Python involves assigning tuple items to an equal number of comma-separated variables positioned on a single line. This feature of the language has now been extended to all *iterables* including lists and sets.

```
fruity_pebbles = ('fruity pebbles', 'Post Consumer Brands', 11)
cereal_name, manufacturer, sugar_content_gm = fruity_pebbles # unpack
```

❗ Multiple assignment requires that tuple items are mapped (e.g., assigned) to an equal number of variables. Mismatches on either side of the assignment ('=') operator will raise a `ValueError` runtime exception.

```
cereal_name, manufacturer, sugar_content_gm = fruity_pebbles[1:] #
triggers a runtime exception
cereal_name, manufacturer, sugar_content_gm, rating = fruity_pebbles #
triggers a runtime exception
```

4.0 Challenges

```
scale = [('5 stars', 5), ('4 stars', 4), ('3 stars', 3), ('2 stars', 2),
('1 star', 1)]

# Walmart customer reviews (2 March 2021)
cereals = [
    ["Apple Jacks", 'Kellogg Company', (5, 185), (4, 21), (3, 10), (2, 4),
(1, 2)],
    ["Cap'n Crunch", 'Quaker Oats Company', (5, 49), (4, 5), (3, 3), (2,
1), (1, 1)],
    ["Cap'n Crunch's Crunch Berries", 'Quaker Oats Company', (5, 196), (4,
15), (3, 6), (2, 2), (1, 4)],
    ['Cheerios', 'General Mills', (5, 1310), (4, 95), (3, 14), (2, 11),
(1, 28)],
    ['Cinnamon Toast Crunch', 'General Mills', (5, 577), (4, 46), (3, 10),
(2, 5), (1, 19)],
    ['Cocoa Puffs', 'General Mills', (5, 147), (4, 9), (3, 1), (2, 2), (1,
5)],
    ['Corn Flakes', 'Kellogg Company', (5, 467), (4, 45), (3, 9), (2, 3),
(1, 10)],
    ['Frosted Flakes', 'Kellogg Company', (5, 1465), (4, 116), (3, 37),
(2, 11), (1, 35)],
    ['Frosted Mini-Wheats', 'Kellogg Company', (5, 883), (4, 95), (3, 18),
(2, 6), (1, 26)],
    ['Fruit Loops', 'Kellogg Company', (5, 750), (4, 84), (3, 14), (2, 6),
(1, 8)],
    ['Fruity Pebbles', 'Post consumer Brands', (5, 170), (4, 23), (3, 8),
(2, 2), (1, 7)],
    ['Grape-nuts', 'post Consumer Brands', (5, 322), (4, 25), (3, 3), (2,
1), (1, 15)],
    ['Honey Bunches of Oats', 'Post Consumer brands', (5, 95), (4, 7), (3,
3), (2, 1), (1, 2)],
    ['Honey-nut Cheerios', 'General Mills', (5, 814), (4, 64), (3, 22),
(2, 8), (1, 22)],
    ['Lucky Charms', 'General Mills', (5, 388), (4, 38), (3, 12), (2, 3),
(1, 7)],
    ['Raisin Bran', 'Kellogg Company', (5, 946), (4, 79), (3, 21), (2,
14), (1, 30)],
    ["Reese's Puffs", 'General Mills', (5, 184), (4, 14), (3, 10), (2, 4),
(1, 3)],
    ['Rice Krispies', 'Kellogg Company', (5, 429), (4, 31), (3, 11), (2,
5), (1, 13)],
    ['Shredded Wheat', 'post consumer brands', (5, 208), (4, 13), (3, 6),
(2, 5), (1, 11)],
    ['Wheaties', 'General Mills', (5, 215), (4, 18), (3, 5), (2, 2), (1,
12)],
]
```

Challenge 01

Task. Implement a function to retrieve a list representation of a cereal from the `cereals` list.

1. Implement the function named `get_cereal` that defines two parameters:

    - `cereals` (`list`): list of nested lists, each representing a cereal product
    - `cereal_name` (`str`): name of the cereal

   The function *must* check each nested cereal element's name value in the `cereals` list. If a *case insensitive* name match is obtained return the cereal to the caller immediately (i.e., exit the loop and exit the function).

2. After implementing `get_cereal`, call the function and pass the following arguments to it:

    1. the `cereals` list
    2. the string "lucky charms".

   Assign the return value to a variable named `lucky_charms`.

Challenge 02

Task. Implement a function that returns the ratings for a given cereal.

1. Implement a function named `get_ratings` that defines a single parameter:

    - `cereal` (list): represents a cereal brand and its 1 to 5 star ratings.

   The function *must* return a list of the cereal's 1 to 5 star rating tuples (e.g., `(5, 388), ...`).

2. After implementing `get_ratings`, call the function `get_cereal` and pass the following arguments to it in **reverse** order using **keyword arguments**::

    1. the `cereals` list
    2. the string "raisin bran"

   Assign the return value to a variable named `raisin_bran`.

3. Next, call `get_ratings` and pass `raisin_bran` to it as the arguement. Assign the return value to a variable named `raisin_bran_ratings`.

Challenge 03

Task. Loop over the `cereals` list and accumulate values to a new list named `cereal_ratings` that summarize each cereal's ratings as "favorable", "neutral", and "unfavorable".

1. Loop over the `cereals` list. For each cereal encountered call the function `get_ratings` and pass the cereal list to it as the argument.

2. *Unpack* the return value into five variables named: `five`, `four`, `three`, `two`, and `one`.

3. Access the rating counts, sum, and assign the number to a local variable according to the following groupings:

    1. favorable = 5 star plus 4 star rating counts
    2. neutral = 3 star rating count
    3. unfavorable = 2 star plus 1 star rating counts

4. Construct a string using the local variables formatted as follows:

   ```
   <cereal name> ratings: favorable=<favorable count>, neutral=<neutral
   count>, unfavorable=<unfavorable count>"
   ```

5. Append the string to the list `cereal_ratings`.

Challenge 04

Task. Implement a function that computes a favorability rating (percent value) for a given cereal.

1. Implement a function named `compute_favorability_rating` that defines a single parameter:

   - `cereal` (list): represents a cereal brand and its 1 to 5 star ratings.

   The function *must* calculate a cereal's favorability rating based on the following equation:

   ```
   (<5 star rating count> + <4 star rating count>) / < total ratings
   count> * 100
   ```

   ❗ The function *must* delegate the task of retrieving a cereal's ratings to the function `get_ratings()` and the task of summing the 1-5 star rating counts to the function `count_ratings()`.

   💡 The function `count_ratings` is already implemented.

   After calculating the passed in cereal's favorability rating, return the percentage value to the caller.

2. After implementing `compute_favorability_rating`, call the function `get_cereal` and retrieve the nested list in `cereals` that represents the Cheerios brand. Assign the list to the variable named `cheerios`.

3. Next, call the function `compute_favorability_rating` and pass `cheerios` to it as the argument. Assign the return value to the variable `cheerios_fav_pct`.

   💡 The Cheerios favorability rating is `96.36%`.