SI 506: Lecture 12

Topics

1. Shallow and Deep copying (`copy` module)
2. Tuples
3. Challenges (working with CSV files and nested loops)

Vocabulary

This week

- **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.
- **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.
- **Nested Loop**. A `for` or `while` loop located within the code block of another loop.
- **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.
- **Tuple packing**. Assigning items to a tuple.
- **Tuple unpacking**. Assigning tuple items to an equal number of variables in a single assignment.

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.
- **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`.
- **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.
- **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.
- **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.

Data

- **resnick-citations.csv**. A comma-separated values (CSV) delimited text file containing biblometric data (e.g., citation report) of Professor Paul Resnick's articles, book chapters, and conference papers. Data sourced from the Web of Science database and exported into Endnote. Beyond illustrating this week's topic on reading from/writing to files, the data set also helps illustrate UMSI scholarly output, scholarly connections within UMSI (e.g., UMSI co-authors) as well as scholarly “influence" (citation counts).
- **umsi-faculty.csv**. List of UMSI faculty (last name, first name) as of January 2021.

2.0 Deep copies of mutable objects

When working with mutable objects such as lists one must approach copy operations carefully, especially when working with nested lists. Recall that variables are nothing more than pointers to objects. Objects can hold references to other objects, and if both object types are *mutable* working with (faux) object "copies" can result in unintended mutations of the original object.

The standard library provides a remedy in the guise of the `copy` module. The module provides a `deepcopy` function that returns a new compound object that contains copies of (rather than references to) objects contained in the original.

The following example using the Python shell illustrates the concern and remedy nicely.

```
>>> import copy

>>> nums = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> print(nums)
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> nums2 = nums # not a copy
```

```
>>> nums2.append([10, 11, 12])
>>> print(nums2)
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

>>> print(nums)
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]


>>> nums3 = nums.copy() # shallow copy
>>> nums3.append([13, 14, 15])
>>> print(nums3)
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15]]

>>> print(nums)
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

>>> nums3[0][0] = 1000
>>> print(nums3)
[[1000, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15]]

>>> print(nums)
[[1000, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]


>>> nums4 = copy.deepcopy(nums)
>>> print(nums4)
[[1000, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

>>> nums4[0][1] = 2000
>>> print(nums4)
[[1000, 2000, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

>>> print(nums)
[[1000, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

Observations

1. `nums2` is *not* a copy of `nums`. `nums2` is nothing more than a second pointer/label/name to the `list` named `nums`.
2. Mutating `nums2` mutates the `list` named `nums`.
3. `nums3` is a *shallow copy* of `nums`. This means that while you can add *new* elements to `nums3` without mutating `nums`, you cannot mutate *nested* list elements derived from `nums` without also mutating `nums`. Shallow copies contain *references* to objects inserted into it *not* copies.
4. `nums4` is a *deep copy* of nums returned by passing `nums` to the `copy` module's `deepcopy` function. `nums4` is a true copy of `nums` without residual references to the original list. Mutating `nums4` element values does not mutate the values contained in the original list.

2.0 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.

2.1 Tuple assignment (packing)

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

```
faculty = ('Paul Resnick',) # single item tuple
# faculty = 'Paul Resnick', # no parentheses (legal)
# faculty = ('Paul Resnick') # a string
```

Multiple item tuples do not require a trailing comma.

```
faculty = ('Paul Resnick', 'Tawanna DILLAHUNT', 'Barbara Ericson')
```

2.2 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.

```
faculty[1] = 'Tawanna Dillahunt' # TypeError: 'tuple' object does not
support item assignment
```

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

```
faculty = (faculty[0],) + ('Tawanna Dillahunt',) + (faculty[-1],) # each a
single tuple
```

2.3 Accessing tuple items

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

```
tawanna = faculty[1] # returns string

tawanna_barb = faculty[-2:] # returns tuple
```

2.4 Tuple multiple assignment (unpacking)

A distinctive feature of the tuple (and lists) involves multiple assignment; i,e., the ability to assign or "unpack" tuple items to an equal number of comma-separated variables positioned on a single line.

💡 Use of parentheses to define the tuple is optional though recommended.

```
paul, tawanna, barb = faculty
```

❗ 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.

```
paul, tawanna, barb = faculty[1:] # triggers a runtime exception
paul, tawanna, barb, chris = faculty # triggers a runtime exception
```

4.0 Challenges

Today's challenges focus on reading from and writing to CSV files, implementing functions, and working with nested loops.

Recall that a nested loop refers to a loop located within the code block of another loop. During each iteration of the "outer" loop, the "inner" loop will execute, completing all its iterations (unless terminated early) *prior* to the outer loop commencing its next iteration, if any.

```
for < element > in < sequence >:
    # indented block
    for < element > in < element (itself a sequence) >
        < statement A >
        < statement B >
        ...
```

`resnick-citations.csv` headers

```
headers = [
    'Title', 'Authors', 'Book Editors', 'Source Title', 'Publication
Date', 'Publication Year', 'Volume', 'Issue', 'Part Number', 'Supplement',
'Special Issue', 'Beginning Page', 'Ending Page', 'Article Number', 'DOI',
'Conference Title', 'Conference Date', 'Total Citations', 'Average per
```

```
Year', '1995', '1996', '1997', '1998', '1999', '2000', '2001', '2002',
'2003', '2004', '2005', '2006', '2007', '2008', '2009', '2010', '2011',
'2012', '2013', '2014', '2015', '2016', '2017', '2018', '2019', '2020'
    ]
```

Challenge 01

Task: Read `resnick_citations.csv` and create a deep copy of the list returned.

1. Call the function `read_csv` and return a list of headers and publications contained in the file `resnick-citations.csv`. Assign the return value to a variable named `data`.
2. Create a "deep copy" of `data` using the `copy` module. Assign the return value to a variable named `publications`.
3. Access the "headers" element in `publications`. Assign the return value to a variable named `headers`.

Challenge 02

Task: Convert all caps titles, authors, and source title instances to mixed case. This is an exercise in data "cleaning".

1. Implement the `format_title` function. Review the Docstring to better understand the function's expected behavior.
2. Review the code below. Note similar statements that are repeated. "Refactor" (i.e., rewrite) the `for` loop in such a way that the loop block eliminates the unnecessary statement duplication (3 like statements -> 1 statement).

    Think nested `for` loop along with a new tuple to hold certain values.

   ```
   for publication in publications[1:]:
       title_idx = headers.index('Title')
       authors_idx = headers.index('Authors')
       source_idx = headers.index('Source Title')

       if publication[title_idx].isupper():
           publication[title_idx] = format_title(publication[title_idx])
       if publication[authors_idx].isupper():
           publication[authors_idx] =
   format_title(publication[authors_idx])
       if publication[source_idx].isupper():
           publication[source_idx] =
   format_title(publication[source_idx])
   ```

3. Call the function `write_csv` and write the updated `publications` list to the file `resnick-citations-cleaned.csv` employing the `headers` list as the headers argument.

Challenge 03

Task: Return total citation counts per year, 1995-2020.

1. Implement the function `get_attribute`. Review the Docstring to better understand the function's expected behavior.

2. Return a slice of `headers` that includes all year elements ('1995'-'2000'). Assign the list to a variable named `years`.

   💡 look up the index value for the element `1995` and use it in your slicing notation.

3. Create an accumulator listed named `annual_counts`. The list will hold two-item tuples that comprise the year and total citations for the year. Both values *must* be cast to an `int` before appending them to the tuple.

   ```
   [
       (< year >, < total citations >)
       . . .
   ]
   ```

4. Loop over the `years` and for each year loop over the `publications`. Call `get_attribute` and return the year's citation count for the publication. Increment an accumulator variable named `citation_count` with the citation total returned for each publication for the given year. Append the annual counts to `annual_counts` in the form of a tuple as described above.

   ❗ Type conversion is required since all values derived from the CSV file are of type `str`. Also, don't forget to reset the `citation_count` to zero (`0`) after looping over the publications and moving on to the next year.

5. Call the function `write_csv` and write the updated `annual_counts` list to the file `resnick-citations-annual_counts.csv` employing `['year', 'citations']` as the headers argument.

Challenge 04

Task: Return list of UMSI coauthors filtering out all duplicates.

1. Call `read_csv` and return a list of UMSI faculty contained in the file `umsi-faculty.csv`. Assign the return value to a variable named `faculty`.

2. Implement the function `filter_authors`. Review the Docstring to better understand the function's expected behavior.

3. Create an accumulator list named `umsi_coauthors`. The list will hold UMSI faculty who have coauthored a publication with Professor Paul Resnick.

4. Loop over the publications. For each publication return its string of authors by calling the function `get_attribute`. Assign the return value to a variable named `authors`. Split the `authors` into a list.

Pass the list to the function `filter_authors` as the first argument and the list `['Resnick, Paul']` as the second argument. Assign the return value to `authors`.

5. Inside the publications loop, loop over `authors`, and inside the authors loop, loop over `faculty`. Write a conditional statement that tests if the coauthor both matches a faculty member *and* has not previously been added to `umsi_coauthors`. If both conditions evaluate to `True` append the faculty member (a `list`) to `umsi_coauthors`.

   💡 use the logical operator `and` and the membership operator `not in` in your conditional statement.

6. Call the function `write_csv` and write the updated `umsi_coauthors` list to the file `resnick-citations-umsi_coauthors.csv` employing `['last_name', 'first_name']` as the headers argument.

Challenge 05

Task: Return list of UMSI-coauthored publications.

1. Mimic the steps in Challenge 04 above with the following changes:
2. Create an accumulator list named `umsi_coauthored`. The list will hold UMSI-coauthored publications.
3. Loop over the publications accessing each publication's authors. Inside the publications loop, loop over `authors`, and inside the authors loop, loop over `faculty`. Write a conditional statement that tests if the coauthor matches a faculty member *and* the coauthor's publication has not previously been added to `umsi_coauthored`. If both conditions evaluate to `True` append the publication (a `list`) to `umsi_coauthored`.
4. Call the function `write_csv` and write the updated `umsi_coauthored` list to the file `resnick-citations-umsi_coauthored.csv` employing `headers` as the headers argument.

Challenge 06

Task: Return the publication(s) with the highest citation count.

1. Create two accumulator variables `max_citations` (`list`) and `max_count` (`int`) to which you will assign the publication(s) with the highest number of citations recorded between 1995-2020 and the accompanying citations count. Ties, if any, *must* be recorded.
2. Loop over the publications and for each publication access its `Total Citations` value by calling `get_attribute` and passing to it the required arguments. Assign the return value to a ariable named `citation_count`.
3. If the publication's `citation_count` is greater than the `max_count`, remove any previous publications added to `max_citations`, append the new leading publication, and update `max_count` with the new citation's count value. If the publication's `citation_count` is equal to `max_count` append the publication to `max_citations`.
4. Call the function `write_csv` and write the updated `max_citations` list to the file `resnick-citations-max_citations.csv` employing `headers` as the headers argument.