Introduction to Programming W7

Tuples and sets
Practice with loops, lists, and dictionaries

2022/05/20

1 Tuples

Tuples are another type of containers for a collection of objects. It is important that tuples are although **ordered** as lists, they are **immutable** (cannot be modified after creation). Tuples can have duplicate values, and its elements can be accessed using their indices (same way as with lists). Tuples are defined using parentheses (or "round brackets").

```
language_tuple = ("FORTRAN", "COBOL", "LISP")
print("initial tuple:", language_tuple, "\ndata type:", type(language_tuple))
print("second element of the tuple:", language_tuple[1])
#trying to change an item from a tuple like it was a list will result in an
#error: tuples are immutable
#the following line would result in a TypeError: 'tuple' object does not support item assignment
#language_tuple[1] = "C++" you cannot do this!
```

Tuple unpacking can be used to assign multiple variables from a tuple:

```
my_tuple = (10, 20, 30, 40) #define tuple
#using tuple unpacking:
number_1, number_2, number_3, number_4 = my_tuple
print("For example, number 2:", number_2)
```

You have already encountered tuples before: the `.items()` dictionary method returns the dictionary keys and values in tuples:

```
scientist_dictionary = {'Da Vinci':1452, 'Galilei':1564, 'Newton':1642}
elements = list(scientist_dictionary.items())
print("List of dictionary items in a list of tuples:", elements)
print("First element:", elements[0]) #this is a tuple!
```

Above, the contents of the list `elements` are tuples.

2 Sets

Unlike tuples, sets are **unordered** (so no indexing), and all of its elements are unique (cannot have duplicates). Sets are **mutable** because you can add new items or remove existing ones, but you cannot modify its items (again, no indexing). Sets can be defined using curly brackets (same as with a dictionary, but sets do not have keys and values).

```
my_set = {2,3,4,"apples"}
print("My set:", my_set)
#the next line would result in a TypeError if executed, because you cannot use indexing with sets
#print(my_set[1])
```

You can add new elements to a set using the `.add()` method.

```
language_set = {"FORTRAN", "COBOL", "LISP"}
print("Original language set:", language_set)
language_set.add("C++")
print("Modified language set:", language_set)
#next, since the string FORTRAN is already in the set, the set won't be modified
```

```
#sets have only unique elements!
language_set.add("FORTRAN")
print("Language set was not modified:", language_set)
```

The `.remove()` method can be used to remove a specified element from a set. While `.pop()` is a valid method to use with sets, since sets are not indexed, you cannot provide an index as an argument, and a random element will be removed and returned.

```
language_set = {"FORTRAN", "COBOL", "LISP"}
print("Original language set:", language_set)
language_set.remove("COBOL")
print("Modified language set:", language_set)
language_set.pop() # you cannot define an index!
print("Language set modified again:", language_set) #random element removed
```

For sets, the `.update()` method updates the set by adding the elements from another set (remember that all elements of a set are unique). The `.intersection()` and `.difference()` methods are equivalent to their set theory definitions.

```
language_set1 = {"FORTRAN", "COBOL", "LISP"}
language_set2 = {"C++", "COBOL"}
#bellow, the comma as a string is just a separator
print("Original language sets:", language_set1, ",", language_set2)
inter = language_set1.intersection(language_set2)
diff = language_set1.difference(language_set2)
print("Elements that exist in both language set 1 and language set 2:", inter)
print("Elements that only exist in language set 1, and not in set language set 2:", diff)
language_set1.update(language_set2)
print("Updated language set 1:", language_set1) #language_set1 already has COBOL!
```

3 Practice

One of the most useful applications of a set is to remove duplicates from a list, using the `set()` constructor:

```
my_list = ["FORTRAN", "COBOL", "FORTRAN", 1,2,2]
my_set = set(my_list)
print("Original list:", my_list, '\ntype:', type(my_list))
print("Duplicates removed:", my_set, '\ntype:', type(my_set))
#if you don't wish to assign it to a new variable and also keep list data type
my_list = list(set(my_list))
print("The list with duplicates removed:", my_list, "\ntype:", type(my_list))
```

The only caveat is since sets are unordered, the order of elements might be different from the original list. To remove duplicates while keeping the order, you must use the `.fromkeys()` dictionary method that makes a new dictionary from a sequence of elements, such as a list. The syntax is the following:

```
my_list = ["FORTRAN", "COBOL", "FORTRAN", 1,2,2]
my_list = list(dict.fromkeys(my_list))
print("Duplicates removed and order preserved:", my_list)
```

This works because before using the `list()` constructor, the dictionary keeps the order of elements (Python 3.7+). The `in`, `not in` operators can be used on tuples and sets the same way as on lists:

```
my_tuple = ("apple", "apple", "orange")
#notice that below, even if we define our set with a duplicate, it won't have it!
my_set = {"apple", "apple", "orange"}
print("My tuple:", my_tuple, "\nMy set:", my_set)
print('The word (string) "orange" is an element the tuple:', "orange" in my_tuple)
print('The word (string) "lemon" is not an element of the set:', "lemon" not in my_tuple)
```

The following nested list `my_nested_list` has lists with duplicates. As a first step, remove the duplicates, so the nested list prints: `[[1, 3], ['orange', 'apple'], [0], [0]]`. The order does not need to be preserved inside the smaller lists. One way to do it is using the `enumerate` function to access the elements of the big list by their index and modify the small lists by using a set and list constructor.

```
my_nested_list = [[1,1,3], ["apple", "orange", "orange"], [0], [0,0,0]]
print("Original list:", my_nested_list)
#remove the duplicates from the lists inside the list my_nested_list !

for index, a_list in enumerate(my_nested_list):
    #complete the for loop body
    my_nested_list[index] =

print("Dups inside small lists removed:", my_nested_list)
```

At this point, the list has *duplicate lists*: `[0]`. Removing duplicate lists from a list is not as straightforward as calling `list(set(my_nested_list))`, that would result in an error. However, we can do it using the `.append()` list method with a conditional checking membership relations. At the end, you should have the list `[[1, 3], ['orange', 'apple'], [0]]`. Complete the code and make sure you understood every step.

```
#just continue the previous code snippet
temp_list = [] #define empty list we will populate with unique lists
for element in my_nested_list: #for all the small lists in the nested list
    if element not in temp_list: #if the current small list is still not in the temporary list
        #append it the temporary list

my_nested_list = temp_list #if we want to keep the original name for the list
print("Duplicate lists are removed:", my_nested_list)
```

Of course, one can use `temp_list` as a separate list, but if you wish to use the original variable name `my_nested_list`, reassign the variable name as shown above.

The `.split()` method splits a string into a list where each word is a list item. If you do not provide additional arguments, the default value for the separator is a whitespace. If you would use for example `.split(,)`, the separator would be a comma. In the following exercise, a string is split into a list of strings `split_string`. Count the number of word occurrences in the dictionary `string_counts`, and print out the results. At the end, the dictionary should print `{'the': 3, 'phenomenon': 1, 'time': 2, 'dilation': 1, 'is': 1, 'difference': 1, 'in': 1, 'elapsed': 1, 'as': 1, 'measured': 1, 'by': 1, 'two': 1, 'clocks': 1}`. Using the `.get()` method to check how may times the string "time" was in the list should print `2`. Complete the code and make sure you understood every step.

```
my_string = "the phenomenon time dilation is the difference in the elapsed time as measured
↪  by two clocks"
split_string = my_string.split() #split a string by the whitespaces to a list
print("The split string in a list:", split_string)

string_counts = {} #define empty dictionary
for elem in split_string: #for the strings inside our list of strings
    if elem in string_counts: #if the current one is already in our counting dictionary (key)
        ##increment the count number, where key is the string and value is the count
        string_counts[elem]
    else: #every other case: if the string is not yet in our dictionary (key)
        #assign the count (dictionary value) to 1, since it's the first time
        string_counts[elem]

print("String count dictionary:", string_counts)
#in the next line, use the get method
print('How many times we encountered the string "time"?', string_counts   )
```