Module 9 – Tuples and Dictionaries

Instructions

1) If not already open, from the Anaconda Navigator Home screen, launch Sypder

2) In Spyder, go to “File”, then “Open”, then navigate to the location where you saved: “9_TuplesDictionaries.py”, and select it to open the script file.

3) Introduction

This module will wrap up the introduction to each of the fundamental data types that you will work with in Python. These two come into play often, especially when using an external package that is tailored to use in a specific application area. As I mentioned at the beginning, it’s nearly impossible to cover the application areas of Python, especially in an introductory workshop. Most of you will work with a specific set of packages that are applicable to your research area (and you’ll see one example of mine in geographic analysis in the final module). The point here is that it is not uncommon to encounter tuples or dictionaries as outputs produced by some of the blocks of codes in these modules. Knowing what to do with them when they are generated is useful.

For starters, there actually are fewer allowable operations attached to these two data types compared to lists. The **append** and **sort** methods that we’ve used with lists don’t work with tuples. But tuple offer a number of advantages as means to store data effectively and in a more compact manner, and as should now be clear, you can always convert between types if need be.

4) Working with Tuples

Tuples can be identified by the ( ) designations. They – like lists – store a sequence of elements starting from index position 0. Unlike lists, though, tuples are immutable. They also only have two accessible methods/operations: **count** and **index**.

Run lines 17 and 18. Note that a tuple is initiated and stored through the use of parenthesis on the right side of the assignment operator. This tuple holds four string data elements. The print statement reads and reports the element at index position 1. You can search for the index position of an element by searching the value (**Run line 19**).

Run lines 23 and 24, which provide an example of the immutability of tuples. While this kind of operation would have overwritten ‘Tamarack’ with ‘Snowflower’ if CarsonRange was a list, it won’t work in this case. Running these lines will prompt an error message to that effect.

You can, though, convert a tuple to a list if you want, where you can then access the allowable operations that list data provide. **Run line 26** to do this, then view the output in the console by entering CR.

Lines 36-37 show how to assign a number of variable names to values in order to place them in a tuple. **Run them to confirm**

5) Tuples Example

Before you do anything, **run line 40.** The example provided between Lines 42 and 53 shows how tuples can fit into common data workflows. If it's not clear yet, it's worth noting that you rarely work with one data type throughout a program, so having a working of knowledge what each can and cannot do – and how they interact – is helpful.

Run Line 43, which enters three geographic coordinate pairs (as tuples) in a list. So, there are three tuple elements in this list. Each tuple itself contains two elements.

Run lines 45 and 46 to initiate empty lists, where we'll eventually store the longitude (x) and latitude (y) values from all the tuples we encounter.

Run Lines 47-53 together, and now let's review what happened. The definite loop initiates and steps through each element in the latlongpairs list, first printing the element, then appending the first subelement encountered in the long (x) list, then the second subelement in the lat (y) list.

At last, though, lines 52 and 53 show that we can prompt the program to provide a mean value in a more straightforward manner than we have been. To do so, though, line 40 is needed. This imports the **numpy** module (and you can see we import it as **np** in the namespace at the end of Line 40). Calling np.mean twice (once on Line 52 and again on Line 53) reports the mean values of the two lists that we populated by cycling through the latlongpairs list and placing the x and y values from each tuple in these lists.

Then, **run Lines 56 to 58** together to undo this separation and place the tuples back together again, by making use of the **zip function**, which returns an iterator of tuples, specified as a list (as it initially was).

6) Working with Dictionaries

In contrast to lists and tuples that store (and access) information based on a sequential order, dictionaries instead make use of a key:value relationship. Instead of using an index value, then, to access values, a look-up tag (or key) is used.

Dictionaries can be identified by the curly brackets: {}

You can call them either as on Line 74 or Line 81.

Run Lines 74-78 together and see the output. Now there are two keys and two values in this dictionary. Lines 75 and 76 specifically show how to add such values to a dictionary. Values are also not immutable – you can change them. Line 78 updates the original value associated with 'prominence' on Line 76.

If you want to search for the value associated with the key 'prominence' in this dictionary, then, you can do so without needing to remember (or look up) its index value (**Run Line 79**).

Another dictionary is initiated on Line 84, with three key:value entries. **Run this line**, and then **run lines 87 and 88**, which are examples of searching operations that return values of 'true' or 'false' regarding whether or not the keys are encountered.

Lines 92 and 93 use the **.items() method** that is associated with dictionaries, but note the use of two iterator variables here in the definite loop. We need two, since the items method reports both the key *and* value for all elements in the dictionary. **Run them** and review the output.

As with tuples, you can convert dictionaries to lists if you wish, but if you do so (**run Line 96**), notice what comes over in the output and what does not when you run this line.

Running Lines 98 and 99 together shows you what the two operations .keys() and .values() – associated with dictionaries – provide. **Run line 101** to see what .items() provides, and how that data is reported.

7) Dictionaries Example

The code between lines 105 and 109 is probably starting to look far too familiar at this point. We are once again creating separate lists of names and population values. **Run lines 105-109** and then let's move on.

This time, though, let's take advantage of the key:value relationship in dictionaries to associate names with those population values. Since they're both lists and – importantly – at the moment sorted in the some sequential order, we can use the zip function (Line 112) and then place this iterator output in a new initiated dictionary (dictNVPop), done on Line 113. Run **lines 112 and 113** together.

Lines 117 is an example of generating a dictionary in a more compact way, using the values in the NV list (sorted, which is accomplished after Line 109), and using their index values (+1, to avoid starting at 0) to generate associated keys. **Run line 117 to review**.

Finally, view lines 122 through 125, which return to the dictionary we created after running Lines 112-113. This time, though, we sort according to the values.

First, though, we sort according to the keys (the county names, alphabetically). So run Line 122 first.

Then, run Lines 123-125 together. Here, we sort the items in reverse order (and store this in a tuple called CountyTuples). Next, we use the built-in **lambda** function that, in this case, is looking for an element in an object (x) in index position 1 (x[1]), since we know that in a tuple, that's the second value encountered. In our case, this will be the *population value*.

Now we set a simple definite loop to iterate through the tuples and report the name and then the population. That's what we see in the output of the console after you **run lines 122-125 together**

```
In [102]: sorted(dictNVPop.keys())
     ...: CountyTuples = sorted(dictNVPop.items() , reverse=True, key=lambda x: x[1])
     ...: for county in CountyTuples:
     ...:     print(county[0],county[1])
Clark 2204079
Washoe 460587
Carson City 54745
Lyon 54122
Elko 52649
Douglas 48309
Nye 44202
Churchill 24230
Humboldt 16826
White Pine 9592
Pershing 6508
Lander 5693
Lincoln 5223
Mineral 4457
Storey 4006
Eureka 1961
Esmeralda 850
```

This completes the structured part of Module 9. Feel free to experiment some more with setting variables and values, working with print statements, or anything else you are curious about.