Synoptic Meteorology I: Assignment #3

Due: 2 October 2018

Learning Objectives: to gain an understanding of the built-in and module-based Python structures for storing multiple data elements, including the differences between them; to understand how to conduct meteorological computations across an array.

So far, we have only considered variables containing a single value. Often, however, the data that we work with will contain multiple elements or dimensions. Fortunately, Python offers three built-in methods for representing such data, while the numpy module provides for data arrays.

The first built-in method for representing multiple data elements is known as a *dictionary*. Python dictionaries contain two values for each data element: a *key*, used to refer to the data, and the *value* associated with the key. For example, imagine that you wanted to use a dictionary to keep track of your grades in this class. After the first assignment is returned, you might initialize the dictionary:

```
grades = {'Assignment 1':100}
```

where the dictionary named grades is initialized using squiggly brackets. In this example, the key name is a string and the grade is an integer; however, as we will soon see, keys do not have to be strings, and the values do not have the be integers.

If you want to refer to your grade on Assignment 1,

```
print(grades['Assignment 1'])
```

where the key is what is used to refer to the element of grades that you wish to obtain.

If you want to add your grade on Assignment 2, you don't have to recreate the entire dictionary. Instead, you can assign it directly:

```
grades['Assignment 2'] = 94
```

You can print the entire dictionary as well:

```
print(grades)
```

If you do this after adding your Assignment 2 grade, you'll confirm that it was added.

Let's say you had the chance to revise an assignment for extra points (don't get any ideas!). You can even change your grade in the gradebook:

```
grades['Assignment 2'] = 98
```

This, of course, is not the only useful example of a dictionary. For instance, a dictionary could be used to store METAR station identifiers as keys with the full station name as the values, i.e.,

```
metarlocations = {'MKE':'Milwaukee', 'GRB':'Green Bay'}
```

The second built-in method for representing multiple data elements is known as a *tuple*. Unlike a dictionary, the values of a tuple cannot be changed after they have been assigned (are immutable). They also do not represent paired values.

The EMS building is located at 43.08°N, 87.89°W. We can represent this information as a tuple:

```
location = (43.08, -87.89)
```

Note how there is no explicit denotation of the values as representing latitude or longitude. A tuple is defined by parentheses around its elements, in contrast to the squiggly brackets for dictionaries.

Since there are no user-defined keys with a tuple, we refer to the elements of a tuple using numbers. In this regard, Python is a bit confusing to introductory programmers: it uses 0 to refer to the first (or in Python, 0^{th}) element, 1 to refer to the second (or in Python, 1^{th}) element, and so on. Thus, if we wanted to extract out the latitude in the location tuple above, we would have to know that the latitude is the 0^{th} element of the tuple, after which we could print the desired value:

```
print(location[0])
```

It is possible to combine data structures. For instance, a dictionary of the Wisconsin NWS office locations could use the three-letter station identifier as the key and a two-element tuple as the value for their locations in latitude and longitude:

```
nwslocations = {'MKX':(42.97,-88.55), 'GRB':(44.50,-88.11),
                'ARX':(43.82,-91.19)}
```

from which we could print the latitude and longitude for the Milwaukee/Sullivan office:

```
print(nwslocations['MKX'])
```

Tuples are most often used to store data that should not change, such as fixed locations, or to pass fixed values to Python functions and routines.

The final built-in method for representing multiple data elements is a *list*. Python lists are far more powerful than we will describe here, but in general lists are the closest things that Python has (in built-in form, at least) to arrays. Like dictionaries, the values in a list can change after assignment; like tuples, there are no user-defined keys within lists.

A simple list might include the three-letter station identifiers of the Wisconsin NWS offices:

```
nwsoffices = ['MKX', 'GRB', 'ARX']
```

Lists are defined by square brackets around its elements. Each is referred to in a similar fashion to tuples, as described above.

Python has many built-in methods that can be used to work with lists. For example, if the NWS added an office in Madison, we could add it to the list using the list's `append` method:

```
nwsoffices.append('MSN')
```

If we wanted to count the number of elements in the list, the Python `len()` function can be used:

```
len(nwsoffices)
```

If we want to alphabetically sort the elements of the list, the list's `sort` method can be used:

```
nwsoffices.sort()
```

If we want to print out all elements of the list, one-by-one, we can use Python's `for` loop construct to do so:

```
for office in nwsoffices:
    print(office)
```

Note that Python syntax is to use four spaces to indent code within loops, functions, and the like. The for loop loops over the `nwsoffices` list, storing each individual element to the `office` variable as it does so, with the only activity inside the loop being to print the value of `office`. If you type the above code into the Python command window, you'd do so one line at a time, hitting Return twice after the last line is entered.

Despite the presence of the dictionary, tuple, and lists built-in data structures, the most commonly used structure for handling multiple data elements is the numpy array construct. Beyond providing a powerful array method, the numpy (short for numerical Python) module provides a wide range of functions, methods, and structures useful for working with data.

To access the numpy array structure, first load the numpy module:

```
import numpy as np
```

The `np` alias is the *de facto* Python standard for numpy.

Let us assume that we have three temperature observations that we wish to represent as an array: 82.2, 83.8, and 64.6. We can define an array using the `np.array()` function:

```
temps = np.array([82.2, 83.8, 64.6])
```

In this instance, we have passed the `np.array()` function a list of the temperature observations. In more advanced applications, we would likely read in the data from a file using either Python's built-in routines, the pandas module (for text or spreadsheet data), or the xarray or netcdf4-python modules (for gridded data).

As with Python lists, there are far more things that we can do with a numpy array than described here. Nevertheless, one particularly powerful characteristic of numpy arrays is that math functions or Python methods distribute to all array elements without having to refer to them individually.

For example, if we want to manually convert all elements of the `temps` array to °C, we can do so with just one line of code:

```
tempsc = (temps-32)/1.8
```

Printing the tempsc array confirms that the conversion has been completed on all array elements.

Alternatively, we can load the metpy module, assign units of °F to all array elements, and convert to °C in a few short lines:

```
import metpy
temps = temps * metpy.units.units('degF')
tempsc = temps.to('degC')
```

Printing the tempsc array confirms that the conversion has been completed on all array elements.

Individual elements of numpy arrays can be accessed in the same fashion as individual elements of Python lists and tuples: just keep in mind that Python refers to elements starting with 0!

It is also possible to create and work with two-, three-, or even higher-dimensional arrays. It is not hard to think of the meteorological applications of such arrays, particularly for model data (which typically contain all three spatial dimensions as well as the temporal dimension).

For example, let us assume that we have temperature observations at three cities and four times. We can define a two-dimensional array to store these data:

```
temps = np.array([[82.2, 83.8, 64.6],[71.4, 81.1, 63.4],[69.9,
                  79.4, 58.2],[80.0, 88.6, 70.7]])
```

The two-dimensional array is formed from a two-dimensional list. Each of the four three-element lists represents temperature observations at three cities at one time. Each time, then, is represented by a different list. Combined together, the city varies across the columns, whereas the time varies across the rows of the array. Printing the resulting temps array to the screen helps to illustrate this.

Let's say that we want to extract the observations for all cities at a given time. We can do this:

```
print(temps[1, :])
```

This statement prints the 1th row (second time) for all columns (all cities). Likewise, we can extract the observations for all times for a given city:

```
print(temps[:, 1])
```

This statement prints all rows (all times) for the 1th column (second city).

In both examples, the : operator tells Python to consider all rows or columns. It can also be used with numbers before and/or after the : to subset the results.

More information about numpy arrays, including a selection of the functions and methods that may be used with them, is available from:

https://jakevdp.github.io/PythonDataScienceHandbook/02.02-the-basics-of-numpy-arrays.html

For this assignment, download a full day's worth of temperature and dew point temperature from Weather Underground. For example, the data from KMKE for 1 August 2018 can be obtained at:

https://www.wunderground.com/history/daily/KMKE/date/2018-8-1

where you would change the identifier and date string for your desired location and date. Create a two-dimensional numpy array to store temperature (first row) and dew point temperature (second row); time will vary across columns. Only the times at the end of each hour are necessary. Use the metpy units function to assign the correct units to these data, then use the metpy conversion fuction to convert the data to °C. Print the final data to the screen. When done, use Python to write out a string with your name and the then-current date and time. Take a screenshot and print it or submit it via e-mail before the start of the next class.