Python #4
Lists and Dictionaries

Location: https://people.sc.fsu.edu/∼jburkardt/classes/python_2022/python04/python04.pdf

Freely adapted from the Python lessons at https://software-carpentry.org/

Lists are not Arrays

- *A numpy array is designed for numerical work;*
- *A Python list is a simpler, more flexible way of grouping data;*
- *Lists are immediately available, without requiring an extra library;*
- *Lists can contain any type of data (strings, True/False, integers, reals);*
- *A list can contain a mixture of data types;*
- *Using a list, we will handle multiple medical data files;*

1 Lists are simpler than numpy arrays

We have already seen that, by importing `numpy`, we can create arrays that can store, manipulate, and plot numerical data as vectors and matrices. We have to create arrays by using the `numpy.array()` function, and most of the operations that we carry out require using a function from that library, such as `numpy.max()`.

However, the basic Python language includes a simpler way of grouping data, called a **list**. For some of our upcoming work, we will find that lists can help us in ways that arrays cannot.

- Create a list by enclosing elements in square brackets;
- Elements can be of any Python data type;

- A list can contain items of different data types;
- Elements are indexed just like arrays are, starting at 0;
- Elements can be added or removed by `list.append(value)` or `list.remove(value)`;

2 Let's make a list!

To create a Python list, we literally simply list our elements, separated by commas, and enclosed by square brackets:

```
colleges = [ 'Carlow', 'Chatham', 'CMU', 'Duquesne', 'University of Pittsburgh' ]
print ( colleges )
print ( colleges[1] )    #  Returns an element
print ( colleges[1:4] )  #  Returns a list!
print ( colleges[-1] )
print ( "  Number of colleges in list is ", len ( colleges ) )  # len(list) gives length.
```

We can modify a list by replacing or modifying an indexed value, or appending a new value (it goes at the end), or removing a particular value.

```
colleges[3] = 'Robert Morris'
print ( colleges )
colleges[3] = colleges[3] + ' University'
print ( colleges )
colleges.append ( 'Point Park' )
print ( colleges )
colleges.remove ( 'Chatham' )
print ( colleges )
```

We can also use the `list.pop()` command to extract an item by index.

```
print ( colleges )
value = colleges.pop ( 1 )
print ( '  value = colleges.pop(1) = ', value )
print ( colleges )
value = colleges.pop ( 0 )
print ( '  value = colleges.pop(0) = ', value )
print ( colleges )
```

The `len()` function gives us the length of a list.

```
print ( len ( colleges ) )
print ( len ( [ 10, 20, 30, 40 ] ) )
```

3 Lists of mixed types

The items in a list don't have to have a common type. A single list could include name (a string), weight (a real number), height in inches (an integer), and is-vacccinated (logical, with value True or False):

```
patient0 = [ 'Robert Baratheon', 235.4, 73, False ]
patient1 = [ 'Arya Stark', 134.7, 68, True ]
```

4 Concatenating two lists into one

We have already seen that we can "add" two strings together using the + sign. The same procedure works for lists. Notice that a list is not a set. The same value can occur more than once in a list. In concatenation, the elements of the two lists are simply displayed one after the other. In some cases, this would mean that the same value would show up twice.

```
odd = [ 1, 3, 5, 7, 9 ]
prime = [ 2, 3, 5, 7 ]
oddplusprime = odd + prime
```

If we want instead a single list in which each element shows up no more than once, we can used `and` operator instead:

```
oddandprime = odd and prime
```

5 Keeping lists separate, in a bigger list

Obviously, a doctor might want to keep a list of all the patients in a practice. Assume we have already defined `patient0` and `patient1` as above. The concatenation operator `bf` will simply give us a long list of all the information.

```
patientpluspatient = patient0 + patient1    # Using + sign
```

But it's probably much more useful to make a nested list, that is, a list of lists. In that way, each patient's information shows up as a separate row of a table, and we can more easily pull out a particular record by indexing:

```
patients = [ patient0, patient1]  # Making a list of lists (a table)
print ( 'patients=', patients )
print ( 'patients[1]=', patients[1] )
print ( 'patients[0][1]=', patients[0][1] )
```

Notice carefully in the last line how we asked for item 0 of `patients`, and then for item 1 of that item 0, the weight of `patients[0]`, so we write `patients[0][1]`. For a `numpy` array, we would instead have asked for `patients[0,1]`!

6 Appending one more list to a list of lists

Our doctor seems to have started out with just two patients. Their records are stored in the `patients` list. Each patient record is, in turn, a separate list. Now suppose a third patient wishes to be treated by the doctor.
The first think the receptionist does is create a new list containing the information for this patient:

```
patient3 = [ 'Tyrion Lannister', 140.5, 50, True ]
```

Now it's necessary to add this patient's records to the single object `patients`, which contains all the information. We do this using the `.append()` method:

```
patients.append ( patient3 )
```

To verify that this worked correctly, we can simply type

```
print ( patients )
```

7 Careful when copying!

In Python, the equals sign (which we think of as the assignment operator) doesn't always work the way you would expect. Python thinks of the name of a list as "pointing" to the list. So when we write `A = [ 1, 2, 3]`, Python creates an object with the appropriate values, and then says, essentially, "If you ever want to see these values again, ask for **A**." Now suppose we issue the Python command `B = A`. Python says, "I see you want to make another name to refer to the same data. OK, either **A** or **B** will work the same now." What happens next may not be what you expect!

```
odds = [ 1, 3, 5, 7, 9 ]
prime = odds              # This does NOT make a new set of data, just a new pointer
prime.remove ( 9 )
prime.remove ( 1 )
print ( 'prime = ', prime )
print ( 'odds = ', odds )
```

Luckily, we can avoid this problem by using the `list.copy()` function, which guarantees that Python creates a new pointer **and** a new set of data:

```
odds = [ 1, 3, 5, 7, 9 ]
prime = odds.copy()             # This makes a new pointer and new set of data
prime.remove ( 9 )
prime.remove ( 1 )
print ( 'prime = ', prime )
print ( 'odds = ', odds )
```

This feature of Python is there for a good reason, but it sometimes contradicts the way a programmer thinks, and so will occasionally give you a moment or two of confusion!

8 Why are we interested in lists?

Although lists are important in Python, one reason we are considering them for this project is so that we will be able to process our medical data in an efficient manner. The complete records for the medical study are stored in a number of separate files. To analyze them, we will need to determine all the filenames, put all these names into a list, and then use "pop()" or indexing to pull out the names one by one during processing. So we have to understand how lists work, so that we can implement that step in a natural way.

9 The dictionary type

Along with lists, Python includes another data type, called a `dict`, for "dictionary". A `dict` can be thought of as a set of values that can be retrieved by keys, instead of an index. Suppose, for instance, that we wanted to store a list of the atomic weights of the elements. Since the elements are numbered, a natural approach would be to create a list. Restricting ourselves to the first 10 elements for this example, we write:

```
atw_list = [ 1.008, 4.0026, 6.94, 9.0122, 10.81, 12.011, 14.007, 15.999, 18.998, 20.180 ]
```

If the user wants the atomic weight of Carbon, they have to remember that this is element 6, and hence stored in `atw_list[5]`. If we are interacting with users, it might be more convenient to be able to retrieve the value of the atomic weight given the key of the element name

A suitable Python `dict` can be initialized by the command

```
atw_dict = dict ( [ ("Hydrogen":1.008), ("Helium":4.0026), ("Lithium":6.94), \
("Beryllium":9.0122), ("Boron":10.81), ("Carbon":12.011), ("Nitrogen":14.007), \
("Oxygen":15.999), ("Fluorine":18.998), ("Neon":20.180) ] )
```

Now, instead of requesting `atw_list[5]`, the user can ask for `atw_dict['Carbon']`.

If we want to start a dictionary from scratch, we create an empty `dict` using a pair of curly brackets (not square brackets!):

```
capital_dict = {}
```

We can add pairs of keys and values

```
capital_dict["Pennsylvania"] = "Harrisburg"
capital_dict["Virginia"] = "Richmond"
atw_dict["Sodium"] = 22.990
```

and so on. To see the contents of a `dict`, we can, as usual, rely on the `print()` command:

```
print ( capital_dict )
```

10 A computational example

The Collatz transformation, applied to a positive integer n, is defined by

$$T(n) = \begin{cases} \frac{n}{2} & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

.

We might code this as:

```
  if ( ( n % 2 ) == 0 ):
    Tn = n // 2
  else:
    Tn = 3 * n + 1
```

A Collatz sequence starting from n is the sequence of values $n, T(n), T(T(n)), ...$ By convention, the sequence is terminated if it reaches the value 1. So far, every Collatz sequence examined has terminated in this way, but there is no proof that this must be so.

As an example, the Collatz sequence starting at 6 is:

```
6 3 10 5 16 8 4 2 1
```

and the Collatz sequence starting at 19 is

```
19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

Notice that the first sequence jumps from 3 to 10, the second from 20 to 10, and after that the sequence are (and must be) identical.

If we are going to compute many Collatz sequences, we can take advantage of this merging process. When we start computing a new sequence, we can check to see if we encounter a value already computed by another sequence. Since that sequence presumaby terminated at 1, so does this one, and we can stop.

To make this happen efficiently, we can use a Python `dict`. It starts out empty. Given any n, we check to see if it is already in the dict. If so, we stop. Otherwise, we compute T(n), add (n,T(n)) to the dict, and then let T(n) become our next value of n.

```
  if ( not collatz_dictionary ):
    collatz_dictionary = {}

  while ( not ( n in collatz_dictionary ) ):
    Tn = collatz ( n )
    collatz_dictionary[n] = Tn
    n = Tn
```

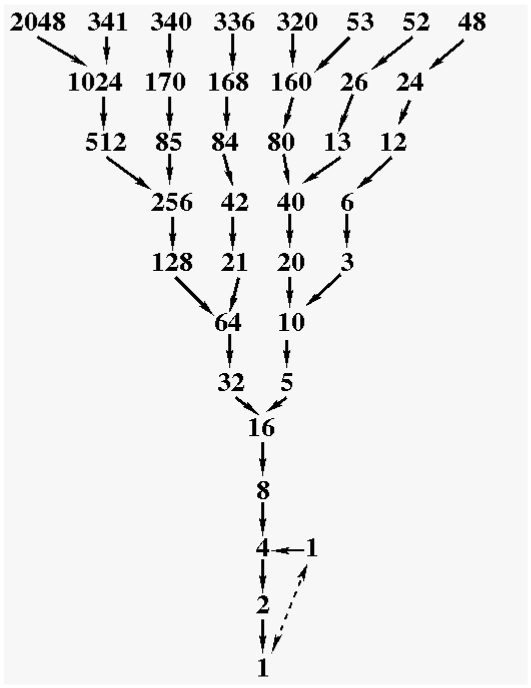
To see the efficiency of this approach, here is what the `dict` looks like after computing Collatz sequences starting at 1, 5 and 15:

```
[(1, 2), (2, 1), (4, 2), (5, 16), (8, 4), (10, 5), (15, 46), (16, 8), (20, 10), (23, 70),
(35, 106), (40, 20), (46, 23), (53, 160), (70, 35), (80, 40), (106, 53), (160, 80)]
```

and here is how the `dict` is updated, after we compute the additional sequences starting at 2, 3, 4, 6, 7, 8, 9 and 10:

```
[(1, 2), (2, 1), (3, 10), (4, 2), (5, 16), (6, 3), (7, 22), (8, 4), (9, 28), (10, 5),
(11, 34), (13, 40), (14, 7), (15, 46), (16, 8), (17, 52), (20, 10), (22, 11), (23, 70),
(26, 13), (28, 14), (34, 17), (35, 106), (40, 20), (46, 23), (52, 26), (53, 160), (70, 35),
(80, 40), (106, 53), (160, 80)]
```

If you continue this experiment by computing the sequence that starts with $n = 27$, you will see that the `dict` is able to compactly to incorporate new data for a long sequence whose values range into the thousands. If we had, instead, tried to use an array in some way, or a list, we would have wasted a lot of unused storage, and had to deal with slower access to a much larger data structure.



By the way, if you imagine the integers as nodes of a directed graph, then the Collatz transformation is telling you how to draw directed edges from n to $T(n)$, in such a way that we end up with a directed tree diagram rooted at 1, (assuming that every sequence does actually reach 1!)