Lab week 6

We have seen lists and `numpy` arrays. However, we can best describe things like graphs by using other types of structure.

Script

We have seen two types of *container*, a thing that holds multiple objects. The `list` behaves a bit like a mathematical set; a `numpy` `array` behaves like a linear algebra vector, or matrix.

There are other containers that Python includes that are useful in particular contexts.

Tuples

A tuple is *technically* a list which can't be changed. It is defined using round brackets:

```
tuple_1 = (1, 2, 3, 4, 5)
print(tuple_1)
```

```
(1, 2, 3, 4, 5)
```

It is accessed the same way as a list:

```
print(tuple_1[1])
print(tuple_1[2:4])
```

```
2
(3, 4)
```

But none of its values can be modified, and its size cannot be changed:

```
tuple_1[0] = 0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
last)

<ipython-input-3-187e83546d31> in <module>()
----> 1 tuple_1[0] = 0


TypeError: 'tuple' object does not support item assignment
```

The *cultural* difference is more subtle (sidenote for demonstrators or people with more expertise: see this description by Batchelder). A tuple should be used when you know

- exactly how many entries you need, and
- what the difference in *meaning* between the entries is.

For example, if you had a polynomial of order 3 with coefficients a_n, you would store the coefficients in a tuple.

We have already seen tuples. When returning multiple values from a function Python uses a tuple:

```
def swap(a, b):
    return b, a

values = swap(1, 2)
print(values[0])
values[0] = 3
```

```
2



---------------------------------------------------------------------------
----

TypeError                                 Traceback (most recent call
last)

<ipython-input-4-0cdd07c720d2> in <module>()
      4 values = swap(1, 2)
      5 print(values[0])
----> 6 values[0] = 3


TypeError: 'tuple' object does not support item assignment
```

Tuples should not be used just to stop somebody changing something, as it's easy to convert between lists and tuples:

```
list_1 = list(tuple_1)
list_1[0] = 3
tuple_2 = tuple(list_1)
print(tuple_2)
```

```
(3, 2, 3, 4, 5)
```

Exercise

Write a function that takes two tuples of the same length and adds the entries together, as if they were `numpy` `array`s, returning a tuple. You may want to use `numpy` `array`s for this.

Dictionaries

All our containers so far have been *ordered*: we access them using a number starting from zero. However, it doesn't always make sense for the container to be ordered. Instead we may want the entries, for example, to be named.

This is where the dictionary, or `dict`, comes in. It contains many objects (usually referred to as *values*), but links each with a unique *key*. The key can be anything that doesn't change, but is usually a string, or a number.

Consider the following group of functions:

```
import numpy

dict_1 = {"sin" : numpy.sin,
          "cos" : numpy.cos,
          "log" : numpy.log,
          "exp" : numpy.exp}

print(dict_1)
```

```
{'log': <ufunc 'log'>, 'exp': <ufunc 'exp'>, 'cos': <ufunc 'cos'>,
'sin': <ufunc 'sin'>}
```

The key is the string. The value is the function. We have used this example in the lab where we introduced lists, but there was no logic to the order. In this case the dictionary is not ordered: when you print the dictionary the order may be different each time, and different on different machines.

To access a dictionary we use square brackets, but index using the key:

```
print(dict_1["sin"])
```

```
<ufunc 'sin'>
```

To loop over a dictionary, we typically want to know both the key and the value. The `items()` method gives us both, and we use multiple unpacking:

```
for key, value in dict_1.items():
    print("Key:", key, ". Value:", value)
```

```
Key: log . Value: <ufunc 'log'>
Key: exp . Value: <ufunc 'exp'>
Key: cos . Value: <ufunc 'cos'>
Key: sin . Value: <ufunc 'sin'>
```

We can then use this more descriptively:

```
for name, f in dict_1.items():
    print("The", name, "function evaluated at 1 is", f(1))
```

```
The log function evaluated at 1 is 0.0
The exp function evaluated at 1 is 2.71828182846
The cos function evaluated at 1 is 0.540302305868
The sin function evaluated at 1 is 0.841470984808
```

Dictionaries are extremely useful when ascribing meaning or structure to your code. You can directly associate names or meanings to values instead of keeping two lists. Consider the use of dictionaries when describing a network, or a graph.

Exercise

A graph can be represented by the nodes, indexed by integers $0, \ldots$, and the weight (or cost) of the edge. In code we will represent this by a dictionary whose keys are the nodes connected by the edge and are values are the weights of the edge. So

```
g1 = {(0, 1) : 1, (1, 0) : 2}
```

is a graph with two nodes, `0` and `1`, with weight `1` going from `0` to `1` and weight `2` going from `1` to `0`.

Write a function that takes the dictionary graph `graph` and a path `path`, a list of nodes visited in order, and returns the total weight or cost of the path.

Sets

Python has another useful container: the *set*. This is closer to the mathematical ideal of a set than a `list` is, as

- it is unordered, like a dictionary;
- all its entries are *unique*.

A set is defined using curly brackets like a dictionary, but no key is given.

```
set_1 = {1, 2, 3, 1, 2, 4}
print(set_1)
```

```
{1, 2, 3, 4}
```

You can rapidly convert a list or a tuple to a set:

```
set_2 = set(tuple_2)
print(set_2)
```

```
{2, 3, 4, 5}
```

This removes all the duplicate entries, allowing you to rapidly count the number of unique entries of a container (using `len(set(...))`).

The other advantage of sets is that it is *extremely* fast to check if an entry appears in the list. We check if an entry appears in a container using `in`:

```
print(1 in list_1)
print(1 in tuple_1)
print(1 in set_1)
```

```
False
True
True
```

Whilst all work, this is much faster in the case of a `set` than for a `list` or a `tuple`.

Further looping with lists

We have seen two ways of looping over containers. If we just want the values in the container, we use `in` to pull those values out:

```
for number in list_1:
    print(number)
```

```
3
2
3
4
5
```

If we needed to use the ordering of the container, we can use the index. For this we've seen the `range` function:

```
for i in range(len(tuple_1)):
    print(tuple_1[i])
```

```
1
2
3
4
5
```

We can combine the safety and simplicity of getting the values with the ability to use the ordering via the index. This is particularly important if we have two lists (or similar) with the same length, and we want to set one using the other. To do this use `enumerate`:

```
for i, number in enumerate(tuple_1):
    list_1[i] = number
print(list_1)
```

```
[1, 2, 3, 4, 5]
```

The `enumerate` function returns the index into the container, *and also* the value of the element.

Alternatively, if you have multiple lists (or other containers) which you want to loop through, but don't explicitly need the index, you can get the values from multiple lists using `zip`:

```
for number1, number2 in zip(tuple_1, tuple_2):
    print(number1, number2)
```

```
1 3
2 2
3 3
4 4
5 5
```

When looping over lists using more complex data structures, the `enumerate` command is particularly useful.

Exercise

Take two lists `list_1` and `list_2`. Using both `zip` and `enumerate` together, print `(index, result)` where `index` should count the entry into `list_1` and `result = list_1[index] + index * list_2[index]`.