listsvsarrays

January 24, 2022

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        %matplotlib inline
```

1 Lists vs. numpy arrays, dictionaries, functions, and lambda functions

We are going to introduce a few more python concepts. If you've been working through the NHANES example notebooks, you will have seen these in use already. There is a lot to say about these new concepts, but we will only be giving a brief introduction to each. For more information, follow the links provided or do your own search for the many great resources available on the web.

1.0.1 Lists vs numpy arrays

Lists can have multiple datatypes. For example one element can be a string and another can be and int and another a float. Lists are defined by using the square brackets: [ ], with elements separated by commas, ',' . Ex:

```
my_list = [1, 'Colorado', 4.7, 'rain']
```

Lists are indexed by position. Remember, in Python, the index starts at 0 and ends at length(list)-1. So to retrieve the first element of the list you call:

```
my_list[0]
```

Numpy arrays np.arrays differ from lists is that the contain only 1 datatype. For example all the elements might be ints or strings or floats or objects. It is defined by np.array(object), where the input 'object' can be for example a list or a tuple.
Ex:

```
my_array = np.array([1, 4, 5, 2])
```

or

```
my_array = np.array((1, 4, 5, 2))
```

Lists and numpy arrays differ in their speed and memory efficiency. An intuitive reason for this is that python lists have to store the value of each element and also the type of each element (since the types can differ). Whereas numpy arrays only need to store the type once because it is the same for all the elements in the array.

You can do calculations with numpy arrays that can't be done on lists.
Ex:

```
my_array/3
```

will return a numpy array, with each of the elements divided by 3. Whereas:

```
my_list/3
```

Will throw an error.

You can appened items to the end of lists and numpy arrays, though they have slightly different commands. It is almost of note that lists can append an item 'in place', but numpy arrays cannot.

```
my_list.append('new item')
np.append(my_array, 5) # new element must be of the same type as all other elements
```

Links to python docs:
Lists, arrays, more on arrays

```
In [2]: my_list = [1, 2, 3]
        my_array = np.array([1, 2, 3])

In [3]: # Both indexed by position
        my_list[0]

Out[3]: 1

In [4]: my_array[0]

Out[4]: 1

In [5]: my_array/3

Out[5]: array([0.33333333, 0.66666667, 1.        ])

In [6]: my_list/3


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

        TypeError                                 Traceback (most recent call last)

        <ipython-input-6-344e5631acd2> in <module>()
    ----> 1 my_list/3


        TypeError: unsupported operand type(s) for /: 'list' and 'int'
```

```
In [7]: my_list.append(5) # inplace
        my_list

Out[7]: [1, 2, 3, 5]

In [8]: my_array = np.append(my_array, 5) # cannot do inplace because not always contiguous me
        my_array

Out[8]: array([1, 2, 3, 5])
```

1.0.2 Dictionaries

Store key-values pairs and are indexed by the keys. denoted with {key1: value1, key2: value2}.
The keys must be unique, the values do not need to be unique.
Can be used for many tasks, for example, creating DataFrames and changing column names.

More about dictionaries in section 5.5

```
In [9]: dct = {'thing 1': 2, 'thing 2': 1}
        dct['thing 1']

Out[9]: 2

In [10]: # adding to a dictionary
         dct['new thing'] = 'woooo'

In [14]: dct['new thing']

Out[14]: 'woooo'

In [17]: dct['hello'] = 'world'
         dct

Out[17]: {'thing 1': 2, 'thing 2': 1, 'new thing': 'woooo', 'hello': 'world'}

In [21]: # Create DataFrame
         df = pd.DataFrame({'col1':range(3), 'col2':range(3,6)})
         df

Out[21]:    col1  col2
         0     0     3
         1     1     4
         2     2     5

In [28]: # Change column names
         df.rename(columns={'col1': 'apples', 'col2':'oranges'}, inplace = True)

In [29]: df.rename(columns={'apples' : 'green', 'oranges' : 'yellow'})

Out[29]:    green  yellow
         0      0       3
         1      1       4
         2      2       5
```