Starting with an understanding of NumPy library for Python

NumPy's role in data analysis:

- array operations;
- multidimensional arrays called ndarray (1-D is a list, 2-D is like a spreadsheet, 3-D is like a Rubik's cube, which can be imagined as a list of lists or spreadsheets)
- descriptive statistics
- and a whole lot more...beyond this presentation's scope

Whearas pandas is known for:

- adding on to NumPy functionality
- time series functionality
- ways to manage missing data
- labeled axes which prevent errors in data alignment
- Series (1-D) and DataFrames (2-D)

This notebook is based on https://docs.scipy.org/doc/numpy/user/quickstart.html (https://docs.scipy.org/doc/numpy/user/quickstart.html)

- the link has great list of methods, clickable near the end

The main unit in NumPy is the multidimensional array

- values of all same type, usually numbers
- indexed by tuple
- dimensions are called axes
- numpy.array is not the same as Python's built-in array.array which is 1-D and relatively basic

Examples of ndarrays (n-dimensional arrays)

Array with one axis with 3 elements; length is 3; shape is (1, 3)

```
[1, 2, 1]
```

Array with 2 axes, each with length 3; size is (2, 3)

```
[[ 1., 0., 0.],
 [ 0., 1., 2.]]
```

The NumPy website lists noteable methods to explore right at the start: ndarray.ndim, .shape, .dtype, .itemsize, .data

In [1]:
```python
import numpy as np
a = np.arange(15) # create array with one axis, from 0 up to but not including 15
a
```

Out[1]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

In [2]:
```
a.reshape(3, 5) # convert to 3 axes
```
Out[2]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

In [3]:
```
a # was a actually changed when .reshape was applied?... no
```
Out[3]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

In [4]:
```
a.shape # we expect it to be 1-D with length 15
```
Out[4]:
```
(15,)
```

Farm example - basic operations

- Farms are all in B.C., greater than 11 acres.
- Each data point is the number of farms in each category, ie. beef or grain
- The farms array is one column of data from a .csv found kinda randomly among openly-shared data online
- It's altered slightly to fit a length of 12 for some calculations later

In [5]:
```
farms = np.array( [63,47,127,13, 9, 56,34,33,21,32,170,0] ) # create 1-apex array, ha
farms
```
Out[5]:
```
array([ 63,  47, 127,  13,   9,  56,  34,  33,  21,  32, 170,   0])
```

In [6]:
```
# addition and subtraction require equal-sized arrays; multiplication & exponents, et

# let's see what the numbers look like if every category rose by 5 farms next year

farms + 5 # farms remains unchanged
```
Out[6]:
```
array([ 68,  52, 132,  18,  14,  61,  39,  38,  26,  37, 175,   5])
```

In [7]:
```
# are any values > 100?
farms > 100
```
Out[7]:
```
array([False, False,  True, False, False, False, False, False, False,
       False,  True, False], dtype=bool)
```

Operations on whole array

In [8]:
```
np.sum(farms) # total number of farms
```
Out[8]:
```
605
```

In [9]:
```
np.min(farms) # lowest count of farms in a category (ie. beef)
```
Out[9]:
```
0
```

In [10]:
```
np.max(farms) # largest value in the farms array
```
Out[10]:
```
170
```

In [11]:
```
farms.reshape(3,4) # see it as a multi-apex array
```
Out[11]:
```
array([[ 63,  47, 127,  13],
       [  9,  56,  34,  33],
       [ 21,  32, 170,   0]])
```

In [12]:
```
farms2 = farms.reshape(3,4)
```

In [13]:
```
farms2
```
Out[13]:
```
array([[ 63,  47, 127,  13],
       [  9,  56,  34,  33],
       [ 21,  32, 170,   0]])
```

In [14]:
```
farms2.shape # version 2 of farms has 3 rows of 4 elements
```
Out[14]:
```
(3, 4)
```

In [15]:
```
farms2.ndim # has 2 dimensions, like a spreasheet or matrix
```
Out[15]:
```
2
```

In [16]:
```
# use axis parameter to do operations along a row
farms2.sum(axis=0) # sum of each column
```
Out[16]:
```
array([ 93, 135, 331,  46])
```

In [17]:
```
farms2.min(axis=1) # minimum in each row
```
Out[17]:
```
array([13,  9,  0])
```

3-D array example

In [18]:
```
b = np.arange(24).reshape(2,3,4) # create array with 24 items, starting at 0
# looks like 2 collections of 3 rows with 4 elements each
b
```
Out[18]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

Indexes in arrays

In [19]:
```
farms # print farms again
```
Out[19]:
```
array([ 63,  47, 127,  13,   9,  56,  34,  33,  21,  32, 170,   0])
```

In [20]:
```
farms[2] # extract a value from 1D array; 0-indexed
```
Out[20]:
```
127
```

In [21]:
```
# want to extract slice from 127 (3rd element, index 2) to 9 (5th element, index 4)
farms[2:5] # slice from index 2 up to but not including index 5
```

Out[21]:
```
array([127,  13,   9])
```

In [22]:
```
# multidimensional arrays use tuples for index
farms2 # see array b again
```

Out[22]:
```
array([[ 63,  47, 127,  13],
       [  9,  56,  34,  33],
       [ 21,  32, 170,   0]])
```

In [23]:
```
farms2[1,3] # extract row with index 1, and element with index 3
# aka row 2, column 4
```

Out[23]:
```
33
```

In [24]:
```
# extract a column
farms2[0:3, 1] # row 1 to 3, at column 2
farms2[ : ,1] # same thing
```

Out[24]:
```
array([47, 56, 32])
```

In [25]:
```
# another way to display array
for row in farms2:
    print(row)
```

```
[ 63  47 127  13]
[ 9 56 34 33]
[ 21  32 170   0]
```

In [26]:
```
# .flat breaks apart a 2D array for display and operations
for element in farms2.flat:
    print(element)
```

```
63
47
127
13
9
56
34
33
21
32
170
0
```

In [27]:
```
# calculate number of farms if they rose 50%
for element in farms2.flat:
    print(element * 1.5)
```

```
94.5
70.5
190.5
19.5
13.5
84.0
51.0
49.5
31.5
48.0
255.0
0.0
```

Diabetes example - basic math and stats

In [28]:
```
# each row is one day's before-meal breakfast, lunch, and dinner measurement in mmol/
# typical goal is around 5.5 mmol/L
# too high or low means you had too much/little insulin at last meal

bgCan = [[ 5.6 , 7.8, 6.0 ], [ 12.2, 4.4, 6.7 ]] # create a list of lists
bgCan
```

Out[28]:
```
[[5.6, 7.8, 6.0], [12.2, 4.4, 6.7]]
```

In [29]:
```
for row in bgCan:
    print(row)
```

```
[5.6, 7.8, 6.0]
[12.2, 4.4, 6.7]
```

In [30]:
```
bgCan # data output retains 'list' look
```

Out[30]:
```
[[5.6, 7.8, 6.0], [12.2, 4.4, 6.7]]
```

In [31]:
```
# so find out its type
type(bgCan)
```

Out[31]:
```
list
```

In [32]:
```
bgCan = np.array(bgCan) # create array from list of 2 lists
bgCan # finally looks like a 2D array

# array has 2 days, 3 meals each
```

Out[32]:
```
array([[  5.6,   7.8,   6. ],
       [ 12.2,   4.4,   6.7]])
```

In [33]:
```
# convert Canadian diabetic blood sugar mmol/L to American mg/dL
# handy because much literature is published for Americans
# multiply mmol/L by 18 to get mg/dL

for row in bgCan:
    print(bgCan * 18)
```

```
[[ 100.8  140.4  108. ]
 [ 219.6   79.2  120.6]]
[[ 100.8  140.4  108. ]
 [ 219.6   79.2  120.6]]
```

In [34]:
```
# since American units are larger, re-do as integers
# make new array from (bgCan * 18) and set data type to integer
bgUS = np.array(bgCan * 18, dtype='int32')
bgUS
```

Out[34]:
```
array([[100, 140, 108],
       [219,  79, 120]], dtype=int32)
```

In [35]:
```
# transpose bgCan
bgCan.T
# now have 2 columns of 3 rows; each day is a column now, and each meal gets an apex
```

Out[35]:
```
array([[  5.6,  12.2],
       [  7.8,   4.4],
       [  6. ,   6.7]])
```

In [36]:
```
# back to the original couple of days of meal data
bgCan
```

Out[36]:
```
array([[  5.6,   7.8,   6. ],
       [ 12.2,   4.4,   6.7]])
```

In [37]:
```
# max and min are now interesting, and easy once the data is in an array
np.min(bgCan)
```

Out[37]:
```
4.4000000000000004
```

In [38]:
```
# better to use a variable so the output is formatted nicely
bg_min = np.min(bgCan) # get lowest value in whole array
print(bg_min) # show the value just as it appears in array
```

```
4.4
```

In [39]:
```
bg_max = np.max(bgCan) # get max value
print(bg_max)
```

```
12.2
```

Counting poses a problem

- Nice to also get counts of values below 4.5 or so
- Also nice to count values over 9 or so when fasting (several hours after meals)
- Can't find a count method built in to NumPy AND Python basics don't work with NumPy's ndarrays

Something like this doesn't work here:

```
count = 0
for item in bgCan:
    if item < 4.6:
        count += 1
```

In [40]:

```
# temporary solution
bg_low = bgCan < 4.5
bg_low # there is one value less than 4.6, at row 2, column 2
```

Out[40]:

```
array([[False, False, False],
       [False,  True, False]], dtype=bool)
```

In [41]:

```
# then count the True values
bg_low_count = np.count_nonzero(bg_low)
```

In [42]:

```
# or even more clearly build it into a human sentence
print("You had", bg_low_count, "low value(s) at", bgCan.size, "meals.")
```

```
You had 1 low value(s) at 6 meals.
```

We really need pandas now for counts and more