Introduction to Numpy / Matplotlib

Starting to investigate data

Assignment Contents

- `numpy`
  - The numpy Array
  - Indexing and Slicing
  - Array Comparison and Arithmetic
  - Creating Arrays
  - np.where
- 'pandas'

EXPECTED TIME 1.5 HRS

Overview

`Numpy` is one of the fundamental math libraries in Python, enabling efficient large-scale calculations. Similarly, `Pandas` is a central data-processing library. `Pandas` is built on top of `numpy`, providing much of `numpy`'s speed in a more user friendly wrapper.

This assignment will focus on the basic function of `numpy` and `pandas`: accessing data. This includes accessing data with slices and indexing.

Next week will extend this access practice with starting to further manipulate data, and create visualization leveraging `matplotlib` and `pandas`.

The content in Lectures 8-6 through 8-11 will not be covered here. Regression models are covered in Lessons 11 and 12. Useful `pandas` functionality (such as `.describe()` from Lecture 8-9) will be used next week.

Activities in this Assignment

- Reshaping arrays
- Arithmetic with arrays
- Aggregating array values
- Understanding `matplotlib` syntax

```
import numpy as np ### numpy is conventionally imported as `np`
import pandas as pd ### pandas is conventionally imported as `pd`
import matplotlib.pyplot as plt ### Import matplotlib for plotting
```

numpy

The numpy Array

Below, let's take a look at the differences between a list and a numpy array:

```
a1 = np.array([1,2,3,4,5]) # Create array
l1 = [1,2,3,4,5] # Create synonomous list

# Index, and slice
print("Length:", len(a1), len(l1))
print("index 2:", a1[2], l1[2])
print("slicing:", a1[::2], l1[::2])
```

That is all pretty similar

```
# Create 2-d array and 2-d list
a2 = np.array([[1,2],[3,4]])
l2 = [[1,2],[3,4]]

# Index and slice
print("Length:", len(a2), len(l2))
print("index 1:", a2[1], l2[1])
print("val at (0,0):", a2[0,0], l2[0][0])
```

Above there are some differences.

As the np.array turns multi-dimensional, it remains a single object that can be referred to as such. `a2` knows that it has two dimensions and allows indexing with `[0,0]`, where as the nested list must be indexed with `[0][0]` instead.

This is indicitive of the fact that `Numpy` has implemented many functions that use the structure of arrays and matricies in calculations.

Indexing and Slicing

The next few questions will reinforce the manners in which slicing and indexing may occur with `numpy` arrays; more specifically, 2-dimensional numpy arrays.

These techniques may also be found in Lecture 8-2

They will all be carried out on ex_arr, created below.

```
ex_arr = np.arange(36).reshape(6,6) **2
### `.arange()` found in Lec. 8-3.
### `.reshape() found in Lec. 8-2/3
### Element-wise manipulation found Lec 8-3

print(ex_arr)
```

First, returning a few specific elements. Remember, indexing on 2-d arrays is accomplished as: `<array>[<row>, <column>]`

```
### Element in row 0 and column 0
print("ex_arr[0,0]: ", ex_arr[0,0])

### Element in row 3, and column 5
print("ex_arr[3,5]: ", ex_arr[3,5])

### Element in row 10 and column 10 (does not exist)
print("ex_arr[10,10]: ", ex_arr[10,10])
```

Question 1

```
### GRADED
### How would the element <64> be returned from ex_arr?
### Specifically: assign integers to "row" and "column" such that
### ex_arr[row,column] returns 64

### YOUR ANSWER BELOW

row = 1
column = 2

### For testing answer: (64 should print out)
print(ex_arr[row,column])
```

Question 2

```
### GRADED
### How would the element <729> be returned from ex_arr?
### Specifically: assign integers to "row" and "column" such that
### ex_arr[row,column] returns 729

### YOUR ANSWER BELOW

row = 4
column = 3

### For testing answer: (729 should print out)
print(ex_arr[row,column])
```

Arrays may also be sliced. One again, slices are within square-brackets, and are sliced according to `[<rows>, <columns>]`. Of note, when slicing an array, an array is also returned. As with other python slices, slices go up-to but do not include the final index.

Notice what happens in the last example where only one slice is given.

```
print("Full Array:  \n", ex_arr)

### Select 2nd and 3rd row; 4th and 5th column
print("ex_arr[1:3, 3:5]: \n",ex_arr[1:3, 3:5])

### Select 3rd row and beyond; 4th column and beyond
print("\n\nex_arr[2:, 3:]: \n",ex_arr[2:, 3:])

### Only passing in one slice; only rows are sliced

print("\n\nex_arr[3:]:  \n", ex_arr[3:])
```

Question 3

```
### GRADED

### Define the variables row_s, row_e, col_s, and col_e such that `ex_arr[row_s:row_e, col_s, col_e]`
### returns the first two columns of the first two rows of ex_arr.

### e.g. Return should be:
###      [[0,1],
###       [36,49]]

### YOUR ANSWER BELOW

row_s = 0
row_e = 2

col_s = 0
col_e = 2

### For testing. The following should print:

### [[ 0  1]
### [36 49]]

print(ex_arr[row_s:row_e, col_s:col_e])
```

Question 4

```
### GRADED

### Define the variables row_s, row_e, col_s, and col_e such that `ex_arr[row_s:row_e, col_s, col_e]`
### returns the 3rd, 4th, and 5th columns of the 2nd, and 3rd rows of ex_arr.

### e.g. Return should be:
###      [[ 64  81 100]
###       [196 225 256]]
### YOUR ANSWER BELOW

row_s = 1
row_e = 3

col_s = 2
col_e = 5

### For testing. The following should print:

### [[ 64  81 100]
###  [196 225 256]]

print(ex_arr[row_s:row_e, col_s:col_e])
```

Reshaping

Reshaping will not be tested directly (though it may be used effectively in some of the following exercises). However, see below for a couple more examples of how reshaping arrays does (and does not) work.

```
test_arr = np.arange(8)
print("Default Array: \n", test_arr)

print("\n\n.reshape(2,4): \n", test_arr.reshape(2,4))
print("\n\n.reshape(4,2):  \n", test_arr.reshape(4,2))
print("\n\n.reshape(2,2,2) (3-d array):  \n", test_arr.reshape(2,2,2))


### Below will not work correctly
print("\n\n.reshape(2,5):  \n", test_arr.reshape(2,5))
```

Array Comparison and Arithmetic

The array, element-wise arethmetic shown in Lecutre 8-3 is demonstrated below. However it will not be tested.

One important note - highlighted already in Lecture 8-3 - multiplying two arrays with the asterisk `<*>` is NOT the same as performing matrix multiplication.

```
a = np.array([[1,2],[3,4]])
b = np.array([[1,0],[2,3]])

print("a: \n", a)
print("\nb:  \n", b)
print("\n########################\n")

### Aritmetic
print("\na+b: \n", a+b)
print("\na-b: \n", a-b)
print("\na*b: \n", a*b)
print("\na/b: \n", a/b)

### Comparators
print("\na<b: \n", a<b)
print("\na>b:  \n", a>b)
print("\na==b:  \n", a==b)
```

Creating Arrays

Unlike lists, every element must be of the same data type. Below, although the array constructor was passed a string and an int, numpy decided that the array was all strings.Functionally, `numpy` decided that it could not change "hello" into a type of number, so to find the lowest common denominator, it changed the 4 to a string.

```
a3 = np.array(["hello", 4])
print(type(a3[1]))
a3
```

Below are a couple ways `numpy` can create arrays. -- Also covered in Lecture 8-3

```
### reshape a 1-d array to 2-d
a1 = np.array([0,1,2,3,4,5,6,7,8])
print("original: ", a1)
print("reshaped: \n", a1.reshape((3,3)))

print()
### create an nxp array of 0s
print("np.zeros: \n", np.zeros((3,3)))

print()
### create an nxp array of 1s
print("np.ones: \n", np.ones((3,3)))

print()
### create array with np.arange()
```

```
print("np.arange(4,20).reshape(4,4): \n", np.arange(4,20).reshape(4,4))

print()
### create array with np.arange()
print("np.arange(4,21,4): \n", np.arange(4,21,4))
```

Question 5

```
### GRADED

### Create a numpy array containing the integers 0 through 199 in order.
### e.g. np.array([0,1,2,3,...,197,198,199])

### Assign array to variable `arr_200`

### HINT: It is recommended you use one of the functions demonstrated two cells above.
### YOUR ANSWER BELOW

arr_200 = np.arange(200)
```

Question 6

```
### GRADED

### Create a numpy array with 5 columns and 25 rows, where all values are 1.

### Assign array to variable `arr_ones_125`

### HINT: It is recommended you use one of the functions demonstrated under the "Creating Arrays" header

### YOUR ANSWER BELOW

arr_ones_125 = np.ones((25,5))
```

np.where

`np.where` is a useful function not only when using numpy, but also when starting to use pandas.

The function (demonstrated in Lecture 8-4) allows you to create an array from another array, in which the new array is created from a conditional on the orignial array.

See the example below. The original array is simply the numbers 0 through 24. np.where creates an array of `booleans` where every number that is divisible by 3 is a "`True`" and all others are "`False`".

For the second example. those divisble by 3's are set to 3, and not divisible by 3 are set to 0.

Finally, an array is return of only the elements that are divisible by 3

```
arr = np.arange(25)
print(arr)
print("\nTrues and Falses: \n", np.where(arr % 3 == 0, True, False))
print("\n3's and 0's:  \n",  np.where(arr % 3 == 0, 3, 0))
print("\nNo conditions: \n", np.where(arr % 3 == 0))
```

As you can see from the example above, the first argument is a conditional. The second argument describes what should be inserted if the conditional is true. The third argument describes what should be inserted if the conditional is False.

Question 7

```
### GRADED

### Below the array `arr_250` is defined which inludes the integers from 1 to 250.

### Return an array of the same length (250) where all integers evenly divisible by 5 (n %5 ==0)
### are replaced with the string "five", and all other integers are replaced with "not".

### Save new array to five_not

arr_250 = np.arange(1,251)

### YOUR ANSWER BELOW

div_by_five = "five"
not_div_by_five = "not"

five_not = np.where(arr_250%5 == 0, "five", "not")
```

np.random

Below gives a few more examples of using `np.random` demonstrated in Lecutre 8-4

```
unif = np.random.uniform(5,10,size= 10000) # 10000 random uniforms from the interval 5 to 10
norm = np.random.normal(7.5, 1, size = 10000) # 10000 random normals from distribution of mean = 7.5 and standard deviation 1

fig, (ax1, ax2) = plt.subplots(1,2)
ax1.hist(unif, bins = 50)
ax1.set_title("Uniform Randoms")
ax2.hist(norm, bins = 50)
ax2.set_title("Normal Randoms");
```

Pandas

At this point the assignment will move on review the `Pandas` actions demonstrated in lecture.

`[Of note, the "` pandas_datareader `" package is not available in Vocareum. Thus it will not be reviewed or tested here.]`

While `numpy` was tested using "toy" data. For working with `pandas`, a real dataset will be brought in.

The dataset is selected from Kaggle's lending club data. While the full dataset is enormous, what below there are 1,000 records selected from the "accepted" data, which is to say all of the records represent a bid for a loan that was accepted.

See the first 5 rows printed below.

```
df = pd.read_csv('../resource/asnlib/publicdata/part_acc.csv', index_col= 0)
df.head()
```

The `.head()` function is very useful for looking at the top of a dataframe.

While the lecture looked at accessing mostly "toy" dataframes, all of the same principles apply with this larger DataFrame. --- Acceccing via index and column occurs in the same way.

Notice how there as an unmarked column on the left of the DataFrame -- That is the index. Below the first two rows are accessed with both the `.loc[]` and `.iloc[]` methods previewed in lecture. -- Remember the `.loc[]` method uses index names for access, and `.iloc[]` uses ordinal position

```
df.loc[0:1]
```

```
df.iloc[0:2]
```

Note above how the slices passed to `.iloc[]` and `.loc[]` were different. When using names in `.loc[]`, the returned data goes *up to and includes* the final row. When using numeric indicies with `.iloc[]`, the conventional `Python` slicing occurs where the slice goes up to but does **NOT** include the final index.

This can be confusing and is worth paying close attention to, especially when the index is the range 0 through n.

Below the indicies 100 through 105 are accessed.

```
df.iloc[100:106]
```

```
df.loc[100:105]
```

Question 8

```
### GRADED

### Suppose you want to access the rows with indicies 50 and 51,.
### Assign numbers to `start` and `end` such that df.loc[start:end]
### returns the rows with indicies 50 and 51

### YOUR ANSWER BELOW

start = 50
end = 51

### To test your answer

df.loc[start:end]
```

Question 9

```
### GRADED

### Suppose you want to access the rows with indicies 50 and 51,.
### Assign numbers to `start` and `end` such that df.iloc[start:end]
### returns the rows with indicies 50 and 51

### YOUR ANSWER BELOW

start = 50
end = 52

### To test your answer

df.iloc[start:end]
```

The names of the coulumns are significantly more descriptive than the index names.

Because the DataFrame is so large, not all of the columns are displayed by `Jupyter` by default. Below all of the column names are printed out.

```
print(df.columns)
```

For the purposes of this exercise only the first few columns will be selected - as a way to demonstrate that columns may be indexed either by name or number.

See below, the second and third columns (`loan_amnt` and `funded_amnt`) are selected, using `.loc` , `.iloc` , and bare square brackets.

Each command ends with a `.head()` -- This is merely for display convenience. `.head()` will return only the first 5 items. **In the questions you will need to return the entirety of the columns for credit. `.head()` is used here only for demonstration**

```
df.loc[:,'loan_amnt':'funded_amnt'].head()
```

```
df.iloc[:,1:3].head()
```

```
### NOTE: The column names are collected within a list
df[['loan_amnt','funded_amnt']].head()
```

Question 10

```
### GRADED
### Using one of the methods demonstrated above, assign a DataFrame with the columns
### "id", and "loan_amnt" to the variable df_first_2

### NOTE: DO NOT APPEND THE `.head()` METHOD TO YOUR SELECTION

### YOUR ANSWER BELOW

df_first_2 = df[['id','loan_amnt']]
```

```
### For reference for Question 11
df.head(5)
```

Question 11

```
### GRADED

### Using the methods demonstrated above, (or refer to Lecture 8-6)
### Assign values to the variables `row` and `col` such that
### 17.14 (under 'int_rate'; 4th row) is returned from the call to df.loc[row,col]

### YOUR ANSWER BELOW

row = 3
col = 'int_rate'

### Testing answer
print(df.loc[row,col])
```

Question 12

```
### GRADED

### Using the methods demonstrated above, (or refer to Lecture 8-6)
### Assign values to the variables `row` and `col` such that
### 17.14 (under 'int_rate'; 4th row) is returned from the call to df.iloc[row,col]

### YOUR ANSWER BELOW

row = 3
col = 5

### Testing answer
print(df.iloc[row,col])
```