MATH6005 Python Lab 3

January 24, 2019

3 Introduction to NumPy

This week we are going to learn about **NumPy** and manipulating 1D NumPy arrays. There is also a *debug* challenge and a *'do something we haven't taught you how to do'* challenge at the end of the lab.

The lab assumes you are using `Spyder 3` and Python 3.x

You should have attended Lecture 3 in the Python series and read the lecture notes before attempting this lab.

Remember for any code where you wish to use numpy you need to import it:

```
In [1]: import numpy as np
```

3.1 NumPy Basics

3.1.1 Example 1: Create NumPy arrays and accessing data

The fundermental building block in NumPy is the `numpy.ndarry`

As we discovered in the last lecture arrays are quite different from a Python `List`. However, creation and accessing individual array items and slicing array is very similar to a list. A big difference is that a `numpy.ndarray` requires all data values to be of the **same type**.

Suppose that we want a numpy array containing the integers 4, 3, 1, 5 and 6.

A simple way to create such an array and access its data is to use the following syntax.

```
In [2]: arr = np.array([4, 3, 1, 5, 6])
        print('the array contains {0}'.format(arr))
        print('a numpy array has has type {0}'.format(type(arr)))
        print('the array has a shape of {0}'.format(arr.shape))
        print('The item at index 0 in the array is {0}'.format(arr[0]))
        print('The item at index 2 in the array is {0}'.format(arr[2]))
        print('If we slice the array between item 0 and 2 we get {0}'.format(arr[:2]))
        print('If we slice the array between item 3 and 5 we get {0}'.format(arr[3:5]))

the array contains [4 3 1 5 6]
a numpy array has has type <class 'numpy.ndarray'>
the array has a shape of (5,)
The item at index 0 in the array is 4
The item at index 2 in the array is 1
If we slice the array between item 0 and 2 we get [4 3]
```

```
If we slice the array between item 3 and 5 we get [5 6]
```

3.1.2 Example 2: Creating empty arrays

- The size of a NumPy array needs to be defined upfront.

- You cannot dynamically append to a `numpy.ndarray` like you can with a `List`
- One option is to create an empty numpy array or a numpy array containing all zeros (if using numeric data).
- When you create an 'empty' array you are allocated a space in the computer's memory.
- There may be some strange data in inside (note your output might look different to the below depending on what is in your computer's memory)

```
In [3]: empty_arr = np.empty(10)
        print(empty_arr)
        print(type(empty_arr))
        print(empty_arr.shape)
```

```
[6.94870485e-310 4.67787424e-310 6.94867951e-310 6.94870505e-310
 6.94870234e-310 6.94870505e-310 6.94870502e-310 6.94867951e-310
 6.94867951e-310 6.94867951e-310]
<class 'numpy.ndarray'>
(10,)
```

- Alternatively you could create an array that contains all zeros

```
In [4]: zeros_arr = np.zeros(10)
        print(zeros_arr)
        print(type(zeros_arr))
        print(zeros_arr.shape)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
<class 'numpy.ndarray'>
(10,)
```

3.1.3 Example 3: Manipulating array data

- It is trivial to update individual elements in an array

```
In [5]: data = np.zeros(10)
        print('original data {0}'.format(data))

        data[5] = 111
        data[9] = 222
        print('updated data {0}'.format(data))
```

```
original data [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
updated data [  0.   0.   0.   0.   0. 111.   0.   0.   0. 222.]
```

- When you slice an array you effectively create a **view** of the data.
- This means that when you update the slice you update the original array as well.
- Numpy keeps the data in the same place in memory for efficiency.

```
In [6]: data = np.zeros(10)
        print('original data {0}'.format(data))

        slice_of_data = data[3:6] # slice from index 3 to index 5
        slice_of_data += 999 #incremement each value in the slice by 999
        print('slice of data {0}'.format(slice_of_data))
        print('the original data is also updated {0}'.format(data))
```

```
original data [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
slice of data [999. 999. 999.]
the original data is also updated [  0.   0.   0. 999. 999. 999.   0.   0.   0.   0.]
```

- NumPy makes basic matrix alegbra simple and efficient
- Here we are going to work with 2 arrays of the same size.
- We are going to create a sequence of numbers in each using the `np.arange()` function

```
In [7]: data1 = np.arange(10)  #create a sequence of integers 0-9
        data2 = np.arange(10,20) #create a sequence of integers 10-19

        print('data1: {0}'.format(data1))
        print('data2: {0}'.format(data2))

        print('The square of each value in data1 {0}'.format(data1**2))
        print('The summation of the the two arrays {0}'.format(data1 + data2))
        print('The difference between the the two arrays {0}'.format(data1 - data2))

        data1 += 10
        print('If we add 10 to each element in data1 we get {0}'.format(data1))
```

```
data1: [0 1 2 3 4 5 6 7 8 9]
data2: [10 11 12 13 14 15 16 17 18 19]
The square of each value in data1 [ 0  1  4  9 16 25 36 49 64 81]
The summation of the the two arrays [10 12 14 16 18 20 22 24 26 28]
The difference between the the two arrays [-10 -10 -10 -10 -10 -10 -10 -10 -10 -10]
If we add 10 to each element in data1 we get [10 11 12 13 14 15 16 17 18 19]
```

3.1.4 Exercise 1: Create and manipulate your own numpy array

- Create two numpy arrays of size 10.

- The first array should be called `array_1` have all zero values
- The second array `array_2` should be a sequence from 90 to 99
- Create a slice of `array_1` to access the last 5 elements of the array.
- Add the value 10 to each of these slices
- Now multiply the two arrays together and print out the result

The expected result is:

```
[0, 0, 0, 0, 0, 950, 960, 970, 980, 990]
```

3.2 NumPy for data analysis

3.2.1 Example 1: Basic statistical analysis of the Standard Normal

We are going to explore the basic analysis capabilities of numpy.

The first thing we will do is generate some synthetic data. We are going to take 10000 random samples from the standard normal distribution. Numpy has a function to do this called `numpy.random.randn`

```
In [8]: data = np.random.randn(10000)
        print(data.shape)
        print(type(data))
```

```
(10000,)
<class 'numpy.ndarray'>
```

We are then going to create a function to conduct and report a descriptive analysis of our data.

```
In [9]: def descriptives(data):
            """
            Returns mean, stdev, and 1st and 99th
            percentile of a 1D numpy.array

            Keyword arguments:
            data -- 1d numpy.ndarray containing data to analyse
            """
            mean = data.mean()
            std = data.std()
            per_1st = np.percentile(data, 1)
            per_99th = np.percentile(data, 99)

            return mean, std, per_1st, per_99th


        results = descriptives(data)
        print(results)
```

```
(-0.004860055017635998, 0.9873939059870946, -2.314439717295787, 2.2606790556874214)
```

So we can see that the mean of the distribution is approx zero with a standard deviation of 1.

We can also see that 99% of our data lie between -2.3 and +2.3 (which we would expect from the standard normal)

It is very simple to work with NumPy arrays containing numeric data. For example if we wanted to find all of our samples that are greater than or equal to +2.3 we use:

```
In [10]: result = data >= 2.3

         print(result.shape)
         print(type(result))
         print(result)

(10000,)
<class 'numpy.ndarray'>
[False False False ... False False False]
```

The code returns a new `numpy.ndarray` that contains boolean (True/False) values. The value at array index `i` is `True` if the corresponding value at index `i` in array `data` is >= 2.3 and `False` otherwise. If we had used a python `List` we would have needed to loop through all of list items and perform the check ourselves.

Let's create some generalised functions to return the probabilities that a value is greater or less than a user specified value in our data set.

To do that we need to know that

```
False == 0
True == 1
```

Therefore we can simply the sum of our boolean array to find out how many array elements are greater or less than a user specified values e.g.

```
(data >= 2.3).sum()
```

```
In [11]: def prob_great_than_or_equal_to(data, x):
             '''
             Return the proportion of the dataset that
             is greater than or equal to x

             Keyword arguments
             data -- a numpy.ndarray containing numeric data
             x -- a numeric value. Function returns proportion where data >=x
             '''
             return (data >= x).sum()/data.shape[0]


         def prob_less_than_or_equal_to(data, x):
             '''
             Return the proportion of the dataset that
             is less than or equal to x
```

```
        Keyword arguments
        data -- a numpy.ndarray containing numeric data
        x -- a numeric value. Function returns proportion where data <=x
        '''
        return (data <= x).sum()/data.shape[0]

    x1 = prob_great_than_or_equal_to(data, 1.96)
    x2 = prob_less_than_or_equal_to(data, -1.96)

    print(x1, x2)
```

```
0.0242 0.0238
```

Our test of these functions shows use that around 95% of data lie between points -1.96 and +1.96 (which again we would expect with the standard normal).

3.2.2 Example 2: Simple linear regression using data in numpy arrays

NumPy arrays are the fundermental building block of the Python SciPy stack. Scientific computing in Python nearly always makes use of `numpy.ndarrays` at some level.

In this example we will load two NumPy arrays from file and conduct a simple linear regression. The method of Ordinary Least Squares is used to fit a linear model to some data stored in numpy array (think $y = \beta\_1x + \beta\_0 + \epsilon$).

We have two datasets.

- `breach.csv`: monthly totals of people waiting 4 or more hours in English emergency departments
- `dtocs.csv`: monthly total of the number of people waiting to be discharged (leave) hospial and go home or transfer to another form of healthcare.

We are going to (naively) assess the relationship between these two variables. For the purposes of this example we are going to ignore that these two datasets are time-series data.

The library we will use to conduct linear regression is `statsmodels.api` We will use the function OLS which accepts two keyword arguments: y and x

Once we have conducted the analysis we will print the results summary.

```
In [12]: import statsmodels.api as sm  #this contains the analysis function we will use

    def load_dtoc_dataset():
        '''
        Loads the breach and dtoc data sets into memory
        Returns a tuple of numpy.ndarrays representing
        breach and dtoc dataset respectively.
        '''
        #note we use skip_header because the dataset has column descriptors
        dtoc = np.genfromtxt('dtocs.csv', skip_header=1)
```

```
    breach = np.genfromtxt('breach.csv', skip_header=1)
    return breach, dtoc


breach, dtoc = load_dtoc_dataset()

#regression code
dtoc = sm.add_constant(dtoc) # an intercept term to the model
model = sm.OLS(breach, dtoc)
results = model.fit()

print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.714
Model:                            OLS   Adj. R-squared:                  0.710
Method:                 Least Squares   F-statistic:                     194.6
Date:                Thu, 24 Jan 2019   Prob (F-statistic):           6.80e-23
Time:                        21:22:00   Log-Likelihood:                -945.02
No. Observations:                  80   AIC:                             1894.
Df Residuals:                      78   BIC:                             1899.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const      -1.633e+05      2e+04     -8.178      0.000   -2.03e+05   -1.24e+05
x1            57.7282      4.138     13.950      0.000      49.489      65.967
==============================================================================
Omnibus:                       11.472   Durbin-Watson:                   0.888
Prob(Omnibus):                  0.003   Jarque-Bera (JB):               16.361
Skew:                           0.591   Prob(JB):                     0.000280
Kurtosis:                       4.874   Cond. No.                     2.61e+04
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 2.61e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

- The results of regression analysis gives an adjusted R-squared of 0.71
- Both the intercept (cont) and dtocs (x1) have confidence intervals that do not include zero
- Our analysis suggests that (on the face of it) there is an association between the two variables

3.2.3 Exercise 1: Descriptive analysis of hourly banks customers

The dataset `bank_arrivals.csv` contains 1000 hourly observations of customers arriving at a bank cashiers queue.

- Load the bank arrivals from the file into a numpy array
- Check that you have successfully loaded all 1000 observations
- Create a function `descriptives` that accepts an numpy array as an argument.
- The function should calculate and return the mean, standard deviation, median and inter-quartile range
- **Tip**: The inter-quartile range is the difference between the 75th and 25th percentile; the median is the 50th percentile.
- Calculate the mean, standard deviation, median and inter-quartile range for bank arrivals data
- Print the summary statistics to the console.

```
In [13]: def descriptives(data):
             """
             Returns mean, var, median and IQR of a 1D numpy.array

             Keyword arguments:
             data -- 1d numpy.ndarray containing data to analyse
             """
             mean = data.mean()
             var = data.var()
             median = np.percentile(data, 50)
             iqr = np.percentile(data, 75) - np.percentile(data, 25)

             return mean, var, median, iqr

         arrival_data = np.genfromtxt('bank_arrivals.csv', delimiter=',')
         print(arrival_data.shape)

         summary_stats = descriptives(arrival_data)
         print(summary_stats)
```

```
(1000,)
(9.987, 9.848831, 10.0, 4.0)
```

3.2.4 Exercise 2: Before and After Analysis of Stroke Treatment Rates

The file `lysis.csv` contains a monthly proportion of a hospital's stroke patients that have been treated with a potentially life saving drug to remove a blood clot from their brain.

There are a total of 54 months in the dataset.

In month 42 the hospital introduced a new process for fast tracking patients to treatment. Your task is to quantify the difference in treatment rates before and after the fast track intervention and to determine if the result is statistically significant.

To do this you need to:

- import the OLS function from `statsmodels.api`
- Read the `lysis.csv` file into a `numpy.ndarray` variable (hint: watch out for headers in the file)
- Create a numpy array that is the same size as the lysis array. It should contain zero when it the month (index) is less than when the intervention was introduced (42) and 1 for all months after the intervention was introduced. Your array should look like

```
dummy == [0,0,0, ... ,1,1,1]

# Hints:
# dummy.shape[0] == 54;  (i.e. its length is 54)
# where index < 42 dummy[index] = 0; where index >=42 dummy[index] == 1
```

- Conduct a regression analysis of the before and after period and display the results.
- Tip: remember that the regression analysis should include a constant term.
- The independent variables in your regression is the dummy variable
- The dependent variables in your regression is the data read in from `lysis.csv`

3.3 Looping over NumPy Arrays

3.3.1 Example 1: Basic Iteration over an array

- For a 1D `numpy.ndarray` you can iterate (loop) over each element much in the same way you would iterate over items in a `List`
- Here we create an array with 10 items - the numbers 0 to 9.
- We then iterate over each item using a `for` loop

```
In [14]: data = np.arange(10)

         for x in data:
             print(x, end= ' ')

0 1 2 3 4 5 6 7 8 9
```

- Alternatively a `for` loop and array index approach could be used

```
In [15]: data = np.arange(10)

         for i in range(data.shape[0]):
             print(data[i], end = ' ')

0 1 2 3 4 5 6 7 8 9
```

3.3.2 Example 2: Finding the maximum value in an array

A simple example of a loop is finding the maximum value in an array. This works in a similar fashion to when using a python `List`.

```
In [16]: def max_value(data):
             '''
             Loop over each element in array
             and return the maximum value

             Keyword arguments:
             data -- numpy.array containing numeric data
             '''
             max_value = 0

             for x in data:
                 max_value = max(x, max_value)
             return max_value

         data = np.array([100, 10, 10, 1000, 999, 1])
         print(max_value(data))
```

```
1000
```

Note that in practice you would use numpy's build-it function for max as opposed to writing your own code.

```
data = np.array([100, 10, 10, 1000, 999, 1])
print(data.max())
```

3.3.3 Example 3: Vectorizing a function

- NumPy allows you to vectorize a function.
- This means that you can apply a function to every item in an array without requiring an explicit `for` loop in your code.

As an example consider the following array:

```
data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

If we wanted to limit the values in this array to a maximum value of 5 then one way to do this is to use an explicit `for` loop to iterate over all of the elements and impose a ceiling on the value. I.e.

```
In [17]: def ceiling(arr, upper_limit):
             '''
             Loop through all elements of a np.ndarray
             and impose a max_value ceiling on the data.

             Keyword arguments:
             arry - numeric np.ndarray to iterate
             upper_limit - the numeric upper limit on values in arr
```

```
            '''
            for i in range(arr.shape[0]):
                arr[i] = min(arr[i], upper_limit)

        data = np.arange(10)

        print('original data: {0}'.format(data))

        ceiling(data, 5)

        print('data with ceiling {0}'.format(data))
```

```
original data: [0 1 2 3 4 5 6 7 8 9]
data with ceiling [0 1 2 3 4 5 5 5 5 5]
```

- However, with NumPy we could instead make use of the `np.vectorize()` function
- To make it work we need to update the `ceiling` function so that it works on individual array elements
- This means that we no longer have a `for` loop in our code!

```
In [18]: def ceiling(to_test, upper_limit):
            '''
            Returing the minimum value by
            comparing to_test to max_value

            Keyword arguments:
            to_test - numeric value to test if breaches ceiling
            upper_limit - the numeric upper limit on to_test
            '''
            return min(to_test, upper_limit)

        # v_ceiling is a wrapper function that we call instead of ceiling
        v_ceiling = np.vectorize(ceiling)

        data = np.arange(10)
        c_data = v_ceiling(data, 5)

        print('original data: {0}'.format(data))
        print('data with ceiling {0}'.format(c_data))
```

```
original data: [0 1 2 3 4 5 6 7 8 9]
data with ceiling [0 1 2 3 4 5 5 5 5 5]
```

- In this simple example the function `ceiling` is just a wrapper for the the built in function `min`
- Therefore we could vectorize `min` instead.
- This means we need even less code again!

- In reality you are more likely to vectorize your own custom functions.

```
In [19]: # here we vectorize min instead of our custom function
         v_ceiling = np.vectorize(min)

         data = np.arange(10)
         c_data = v_ceiling(data, 5)

         print('original data: {0}'.format(data))
         print('data with ceiling {0}'.format(c_data))
```

```
original data: [0 1 2 3 4 5 6 7 8 9]
data with ceiling [0 1 2 3 4 5 5 5 5 5]
```

3.3.4 Example 4: Using np.where() and fancy indexing

Another efficient alternative to using a loop in NumPy is to use `np.where()`

- `np.where()` is like asking "tell me where in this array, values satisfy a given condition".

```
In [20]: data = np.array([0, 1, 2, 500, 700])
         results = np.where(data > 2)

         print(results)
```

```
(array([3, 4]),)
```

- The `results` above tell us that the array `data` contains values that are > 3 in indexes 3 and 4
- This might be a useful result on its own.

- To go one step further and access and manipulate the values in indexes 3 and 4 we need to use fancy indexing

```
In [21]: data = np.array([0, 1, 2, 500, 700])
         indexes = [1, 3, 4]
         sliced_data = data[indexes]

         print(sliced_data)
```

```
[  1 500 700]
```

- The code above demonstrates fancy indexing.

- We defined an `List` of indexes i.e. `indexes = [1, 3, 4]` (this could also be another `np.ndarray`)
- The code `data[indexes]` 'looks up' the values contained in indexes 1, 3, and 4 of the array `data`

- If we combine using `np.where()` and 'fancy indexing' we have very powerful code

```
In [22]: data = np.array([0, 1, 2, 500, 700])
         sliced_data = data[np.where(data > 2)]

         print(sliced_data)

         data[np.where(data > 2)] = 999
         print(data)
```

```
[500 700]
[  0   1   2 999 999]
```

- We can implement the `ceiling` function from the previous example using np.where()
- It is neat solution that requires minimal code

```
In [23]: def ceiling(data, upper_limit):
             '''
             Returing the minimum value by
             comparing to_test to max_value

             Keyword arguments:
             to_test - numeric value to test if breaches ceiling
             upper_limit - the numeric upper limit on to_test
             '''
             data[np.where(data > upper_limit)] = upper_limit

         data = np.arange(10)

         print('original data: {0}'.format(data))
         ceiling(data, 5)
         print('data with ceiling {0}'.format(data))
```

```
original data: [0 1 2 3 4 5 6 7 8 9]
data with ceiling [0 1 2 3 4 5 5 5 5 5]
```

3.3.5 Exercise 4: Calculate the winsorized mean

- Assume you have the data

```
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

- The 20% Winsorized mean of `data` is calculated on a modified data set where the top and bottom 10% of values are replaced by the 10th and 90th percentiles.

- In this case the 10th percentile = 2 and the 90th = 10. Therefore the winsorized dataset is:

```
win_data = [2, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10]

win_mean = win_data.mean()
```

- Write a function `winsorise(data, cut_off = 0.1)`
- The function must modify the numeric np.ndarray `data` so that is it is winsorised.

- A cut_off = 0.1 specifies that the function uses the 10th and 90th percentiles as cut-offs

Hints:

- There are multiple ways to solve this problem
  - You could use a `for` loop
  - You could create a function that is vectorized using `np.vectorize()`
  - You could use `np.where()` and fancy indexing

3.3.6 Week 3: Debug Challenge

Each laboratory will have a debug challenge. You will be given a pre-existing script containing Python code. The catch is that the code doesn't run!

Your challenge is to find and correct the errors so that the script correctly executes.

The challenges are based around common problems students have when writing code. If you do the exercises it will help you debug your own code and maybe even avoid the mistakes in the first place!

Challenge 1: This weeks debug challenge is fixing a monte-carlo simulation model.

We return to our stroke care example from earlier in the lab. This time rather than conducting an empricial analysis we are going to model each stage of the emeregency stroke care pathway and examine the distribution of performance that might be expected.

The model is a simplification of a stroke treatment pathway at a hospital.
Patients must be treated within 180 minutes of the onset of their symptoms.

- They must be brought to hospital
- be seen in the ED
- undego a CT scan
- be clinically assessed by a stroke medic
- not have any additional illnesses that prevent treatment

Instructions: * open `wk3_debug_challenge.py` in `spyder 3`.
* Attempt to run the code. The code will raise errors. * Fix the bugs!

Hints: * Read the Python interpreter output.
* The errors reported can look confusing at first, but read them carefully and they will point you to the lines of code with problems. * The `Spyder` IDE may give you some hints about formatting errors * It can be useful to use `print()` to display intermediate calculations and variable values.
* Remember that `Spyder` has a variable viewer where you can look at the value of all variables created.
* There might be multiple bugs! When you fix one and try to run the code you might find another!

3.3.7 Week 2: Do something we haven't taught you how to do challenge!

Challenge 1: Preprocessing data to detrend a timeseries In example 2, we conducted a simple linear regression to assess the relationship between two variables. Our initial analysis is problematic because both variables are time series and contain autocorrelation and trend. This means that we violate some of the assumptions of ordinary least squares regression and our estimates of the relationship are likely to be incorrect.

In practice, we would pre-process the data in the numpy arrays before conducting the regression. A simple way to do this is to take the first difference of the time series. If Y_t represents the value of the time series at time t then the first difference is equal to:

$$Y_t = Y_{t+1} - Y_t$$

Tip: If an array a has length n then an array of the first differences of a is $n - 1$
Task:

- Copy the code from Example 2.
- Modify the code to take the first difference of the breach and dtoc numpy arrays
- Conduct the regression analysis using the first differences (detrended data)
- Look at the new R^2. Is there still strong evidence of a relationship?