# MATH6005 Introduction to Python. Lecture 3

January 23, 2019

## 3 Introduction to scientific programming using NumPy

### 3.1 Topics covered

- Python data science libraries
- Introduction to NumPy
- Numpy arrays (versus Lists)
- Creating 1D NumPy arrays
- Accessing data in numpy arrays
- NumPy dataTypes
- Data wrangling and analysis

### 3.2 Python data science libraries

There is a whole ecosystem of scientific python libraries built around manipulating, processing and visualising data

- **NumPy** – the fundamental package for numerical computation
- SciPy library – numerical algorithms and analysis toolboxes
- Matplotlib – high quality 2D plotting tools
- Pandas – high performance and easy to use data structures
- SciKit Learn – Machine learning

### 3.3 The `import` statement (revision)

- The data science modules need to be imported into your Python script before you can us them
- This is achieved using the `import` statement.

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

- Here we have imported the `numpy` package and aliased it as `np`
- The alias is a shorthand for accessing functions e.g.

```
In [5]: random_number = np.random.randint(0, 10)
        print(random_number)
```

```
1
```

## 3.4 Introduction to NumPy

- So far we have used a `List` for holding 'arrays' of data
- Lists are easy to use and very flexible

```
In [3]: my_list = ['spam', 3, 9.5, 'eggs', ['sub list', 3], int]
        my_list.append('foo')
        my_list[0] = 999
        print(my_list)
```

```
[999, 3, 9.5, 'eggs', ['sub list', 3], <class 'int'>, 'foo']
```

- The flexibility of a `List` means that they are not well suited to scientific computing
- NumPy provides optimised efficient code for managing data (typically quantitative data)
- NumPy is 'closer to the metal'
- For scientific computing you **should** use numpy instead of Python Lists

### 3.4.1 NumPy Arrays

- The fundermental building block of numpy is the `numpy.ndarray`

```
In [4]: my_arr = np.array([1, 2, 3, 4, 5, 6])
        print(my_arr)
        print(my_arr[3])
```

```
[1 2 3 4 5 6]
4
```

- It looks similar to a list.
- Let's see what happens if we try to treat it as a List...

```
In [5]: my_arr = np.array([1, 2, 3, 4, 5, 6])
        my_arr.append(7)
```

```
        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-5-1b4f0977d0f8> in <module>()
          1 my_arr = np.array([1, 2, 3, 4, 5, 6])
    ----> 2 my_arr.append(7)


        AttributeError: 'numpy.ndarray' object has no attribute 'append'
```

```
In [6]: my_arr[0] = 'foo'
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-6-bf8976ca4c67> in <module>()
----> 1 my_arr[0] = 'foo'


ValueError: invalid literal for int() with base 10: 'foo'
```

- NumPy array are **NOTHING** like a `List`.
- Array size and datatype are declared **upfront** and data are stored efficiently in memory.
- This improves performance - sometimes dramatically.
- The consequences are that arrays only allow variables of 1 data type and size must be known in advance

### 3.4.2 NumPy Arrays are N-Dimensional

- You can think of a `numpy.ndarray` as a **matrix**
- E.g. you could have a (2 x 2) matrix

```
In [7]: matrix_2x2 = np.array([[1, 0], [0, 1]])
        print(matrix_2x2)
```

```
[[1 0]
 [0 1]]
```

- Today we will focus on 1d

- We will cover > 1 dimension in the next lecture

## 3.5 Creating 1-Dimensional Arrays

- Pre-populated Arrays

```
In [3]: my_arr = np.array([1, 2, 3, 4, 5, 6])
        my_arr2 = np.array([1.5, 1.67, 8.99])
        my_arr3 = np.array(['spam', 'eggs', 'foo', 'bar'])
        print(my_arr2)
```

```
[1.5  1.67 8.99]
```

- Sequences of numbers using `np.arange`
- Useful when testing code

```
In [9]: my_arr = np.arange(10)
        print(my_arr)
```

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

- Create empty arrays
- NumPy fills it with random data from memory

```
In [10]: my_arr = np.empty(10, dtype=np.int64)
         print(my_arr)
         print(my_arr.dtype)
```

```
[139641206150632 139641206150656 139641332507904 139641332507848
 139641411581488 139641414146288 139641205983472 139641417985192
 139641205964568 139641205983728]
int64
```

- Create arrays of zeros

```
In [11]: my_arr = np.zeros(10, dtype=np.int64)
         print(my_arr)
         print(my_arr.dtype)
```

```
[0 0 0 0 0 0 0 0 0 0]
int64
```

- It is always a good idea to check the `shape` of your array
- This is useful for 1D and multi-dimentional arrays (covered in Lecture 4)

```
In [7]: my_arr = np.arange(5000)
        my_arr.shape
```

```
Out[7]: (5000,)
```

### 3.6 Useful properties of NumPy arrays for data science (1)

- Speed of computation

```
In [8]: python_list = list(range(1000000))
        numpy_array = np.arange(1000000)
```

```
In [9]: %timeit sum(python_list)
```

```
6.01 ms ± 75.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [11]: %timeit np.sum(numpy_array)
```

```
679 µs ± 6.64 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

### 3.7 Useful properties of NumPy arrays for data science (2)

- You can think of a `numpy.ndarray` as a **matrix**
- NumPy makes matrix algebra **very** easy (minimal code) and **fast**

```
In [16]: x = np.array([10, 20, 30, 40])
         y = np.array([10, 10, 10, 10])

         z = x - y

         print(z)
```

```
[ 0 10 20 30]
```

```
In [17]: x = np.array([10, 20, 30, 40], dtype=np.int64)
         y = x + 5
         z = y ** 2

         print(z)
```

```
[ 225  625 1225 2025]
```

### 3.8 Accessing and manipulating data in an NumPy array

```
In [13]: short_sequence = np.arange(10, dtype=np.int16)
         print(short_sequence)
         short_sequence[4] = 444
         print(short_sequence)
```

```
[0 1 2 3 4 5 6 7 8 9]
[  0   1   2   3 444   5   6   7   8   9]
```

- **Tip**: NumPy is strict about datatypes.
- If you put a float (decimal number) into an integer array then it will be **truncated**

```
In [19]: integer_matrix = np.arange(5, dtype=np.int64)
         floating_point_number = 22.667
         integer_matrix[0] = floating_point_number

         print(integer_matrix)
```

```
[22  1  2  3  4]
```

### 3.8.1 Accessing subsets using slicing

- Access subsets of arrays uses **slicing** notation
- `array[start:end:step]`
- `start` is included and `end` is excluded `[start, end)`
- **Tip**: if `start` or `end` are *ommitted* numpy uses the corresponding index for the start or end of the array
- **Tip**: Don't forget that arrays are **zero** indexed

**Slicing: Example 1**

- `matrix = [10, 11, 12, 13, 14, 15]`

- Select array elements 3 through 4

```
In [20]: complete_matrix = np.array([10, 11, 12, 13, 14, 15])
```

```
In [22]: slice_matrix = complete_matrix[3:5]

         print('complete matrix: {0}'.format(complete_matrix))
         print('slice of matrix: {0}'.format(slice_matrix))
```

```
complete matrix: [10 11 12 13 14 15]
slice of matrix: [13 14]
```

**Slicing: Example 2**

- `matrix = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
- Select the last four elements of the array
- We can do this by ommitting the `end` parameter

```
In [23]: complete_matrix = np.arange(10)
         print(complete_matrix)
```

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

```
In [24]: slice_matrix = complete_matrix[6:]

         print('original matrix: {0}'.format(complete_matrix))
         print('slice of matrix: {0}'.format(slice_matrix))
```

```
original matrix: [0 1 2 3 4 5 6 7 8 9]
slice of matrix: [6 7 8 9]
```

- An alterative way to slice is to use negative notation
- Negative notation e.g. `-x` is interpreted as `length of array - x`

```
In [25]: slice_matrix = complete_matrix[-4:]

         print('complete matrix: {0}'.format(complete_matrix))
         print('slice of matrix: {0}'.format(slice_matrix))
```

```
complete matrix: [0 1 2 3 4 5 6 7 8 9]
slice of matrix: [6 7 8 9]
```

**Slicing: Example 3**

- `matrix = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]`
- Select the first three elements of the array

```
In [13]: complete_matrix = np.arange(10, 20)

         slice_matrix = complete_matrix[0:3]  #could also omit the start of 0

         print('original matrix: {0}'.format(complete_matrix))
         print('slice of matrix: {0}'.format(slice_matrix))
```

```
original matrix: [10 11 12 13 14 15 16 17 18 19]
slice of matrix: [10 11 12]
```

- Alternative using negative notation

```
In [27]: slice_matrix = complete_matrix[0:-7]

         print('original matrix: {0}'.format(complete_matrix))
         print('slice of matrix: {0}'.format(slice_matrix))
```

```
original matrix: [10 11 12 13 14 15 16 17 18 19]
slice of matrix: [10 11 12]
```

### 3.8.2 Accessing data using fancy indexing

```
In [15]: indexes = [2, 4, 6]
         sub_matrix = complete_matrix[indexes]

         print('original matrix: {0}'.format(complete_matrix))
         print('sub matrix: {0}'.format(sub_matrix))
```

```
original matrix: [10 11 12 13 14 15 16 17 18 19]
sub matrix: [12 14 16]
```

**Watch out!**

- A slice is a **view** of the data. It is **not** a copy.

```
In [17]: complete_matrix = np.arange(10, 20)
         slice_matrix = complete_matrix[0:3]

         print(slice_matrix)
         slice_matrix[0] = 999
         print(slice_matrix)
```

```
[10 11 12]
[999  11  12]
```

```
In [31]: print(complete_matrix)
```

```
[999  11  12  13  14  15  16  17  18  19]
```

### 3.9 Practical Example of using NumPy for data analysis

**ED attendance data**

Data is held in the `minor_illness_ed_attends.csv`

Description: The number of patients registered at GP surgery who attend ED per week (standardised to 10k of registered patients).

We are going to:

1. Read in data from file into a numpy array (data is 1 dimensional)
2. Calculate some summary statistics (mean, stdev, percentiles)
3. Produce a frequency histogram
4. Analyse the mean attendences before week 10 and after and including week 10
5. Identify the top 5% of weeks of attendences for further investigation.

**Step 1: Read in the data**

```
In [22]: file_name = 'data/minor_illness_ed_attends.csv'
         ed_data = np.genfromtxt(file_name, skip_header=1, delimiter=',')
```

```
In [23]: print(ed_data.shape)
         print(ed_data.dtype)
```

```
(74,)
float64
```

**Step 2: Calculate summary statistics**

- NumPy makes it easy to calculate means, stdev and other summary statistics of ndarrays.

In [19]:
```
def ed_summary_statistics(data):
    """
    Returns mean, stdev and 5/95 percentiles of ed data

    Keyword arguments:
    data -- 1d numpy.ndarray containing data to analyse
    """
    mean = data.mean()
    std = data.std()
    min_attends = data.min()
    max_attends = data.max()
    per_95 = np.percentile(data, 95)

    return mean, std, min_attends, max_attends, per_95
```

In [20]:
```
def print_summary_stats(mean, std, minimum, maximum, per_95):
    """
    Prints summary statistics in formatted text.

    Keyword arguments:
    mean -- mean average
    std -- standard deviation
    minimum -- min value
    maximum -- max value
    """
    print('Mean:\t{0:0.2f}'.format(mean))
    print('Stdev:\t{0:0.2f}'.format(std))
    print('Min:\t{0:0.2f}'.format(minimum))
    print('Max:\t{0:0.2f}'.format(maximum))
    print('95th:\t{0:0.2f}'.format(per_95))
```

In [41]:
```
mean, std, min_attends, max_attends, per_95 = ed_summary_statistics(ed_data)

print_summary_stats(mean, std, min_attends, max_attends, per_95)
```

```
Mean:        2.92
Stdev:        0.71
Min:        1.62
Max:        5.11
95th:        3.99
```

In [26]:
```
stats = ed_summary_statistics(ed_data)
type(stats)
```

Out[26]:
```
tuple
```

**Step 4: Frequency histogram**

- NumPy has a `histogram` function.
- You need to specify **bins** (frequency ranges) and supply the data

```
In [37]: mybins = np.linspace(start=1.5, stop = 5.5, num=9)
         freq, bins = np.histogram(ed_data, bins=mybins, density=False)
         print(bins)
         print(freq)
```

```
[1.5 2.  2.5 3.  3.5 4.  4.5 5.  5.5]
[ 7 19 18 13 14  2  0  1]
```

**Step 5: Before and After analysis**

```
In [38]: def sub_group_analysis(data, group_name):
             mean, std, min_attends, max_attends, per_95 = ed_summary_statistics(data)
             print('***{0}'.format(group_name))
             print_summary_stats(mean, std, min_attends, max_attends, per_95)

         week = 10  # the week the intervention begins
         sub_group_analysis(ed_data[0:week-1], 'before')
         sub_group_analysis(ed_data[week:], 'after')
```

```
***before
Mean:         3.06
Stdev:         0.57
Min:        2.12
Max:        3.99
95th:         3.84
***after
Mean:         2.89
Stdev:         0.72
Min:        1.62
Max:        5.11
95th:         3.97
```

**Step 6: Find the top 5% of weeks**

- A typical solution is to use a loop.
- NumPy privides the `where` function to simplify and speed up the process

```
In [40]: stats = ed_summary_statistics(ed_data)

         #95th percentile is contained in index 4
         per_95 = stats[4]
```

```
extreme_week = np.where(ed_data >= per_95)

print(extreme_week)
```

```
Out[40]: (array([10, 41, 65]),)
```

### 3.10 Labs

- You will get chance to practice using basic NumPy in the Labs
- Please have a go before the labs and ask us questions!
- You will need to use NumPy in your final assignment - please learn how to use it now!
- Please come along to the correct lab!