Lab Task 2 - Numpy fundamentals

CSIT375 AI for Cybersecurity
SCIT, University of Wollongong

1 Lab Objectives

This lab aims to quickly walk you through the most fundamental parts of NumPy, including:

1. how to create/initiate 1D arrays and 2D matrices,
2. how to get/set the shape of numpy arrays,
3. and basic operations such as how to calculate the dot product of two NumPy arrays, and take the matrix multiplications,
4. how to visualise the data using pandas dataframe and matplotlib libraries.

Please try out the following cells, run the Python code in your notebook, and understand how they work.

This is not an assignment and you do not need to submit it

2 Array Creation

2.1 `ndarray` in NumPy

NumPy's main object is the homogeneous multidimensional array (`ndarray`). It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.

A list of elements can be expressed as an `ndarray` of rank 1, i.e. a 1D array; a matrix can be expressed as an `ndarray` of rank 2.

Here lists some important attributes of `ndarray`:

- **`ndarray.ndim`**: the rank of the `ndarray`. For instance, a matrix has a rank of 2.
- **`ndarray.shape`**: the dimensions of the `ndarray` as a tuple of integers. For a matrix having 20 rows and 30 columns, the shape is `(20, 30)`.
- **`ndarray.size`**: the total number of elements in the `ndarray`, which is equal to the product of the dimensions.
- **`ndarray.dtype`**: an object describing the type of the elements in the array.

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

[2]:
```
arr = np.array(range(1, 10))
print('arr\t\t', arr)
print('arr.ndim\t', arr.ndim)
```

```
print('arr.shape\t', arr.shape)
print('arr.size\t', arr.size)
print('arr.dtype\t', arr.dtype)
```

```
arr             [1 2 3 4 5 6 7 8 9]
arr.ndim        1
arr.shape       (9,)
arr.size        9
arr.dtype       int64
```

There are several ways to create `ndarrays`.

2.2 Using the **array** function

You can create a 1D `ndarray` from an existing list/array easily using the `array` function.

[3]:
```
np.array([1, 2, 3, 4])
```

[3]:
```
array([1, 2, 3, 4])
```

Note that there is only one argument. So never do this:

[4]:
```
# Do not do this
#np.array(1, 2, 3, 4)
```

To create a matrix, call `array` on a sequence of sequence.

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

[5]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

[6]:
```
x.shape # 3 rows, 2 columns
```

[6]:
```
(3, 2)
```

2.3 Using **zeros**, **ones**, **empty**

When the contents of the array to be created are unknown, but its dimensions are known, use one of `zeros`, `ones`, `empty`.

[7]:
```
np.zeros((2, 3))  # the elements are explicitly initialized to zeros
```

[7]:
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

[8]:
```
np.ones((2, 3))  # the elements are explicitly initialized to ones
```

[8]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

[9]:
```
np.empty((2, 3))  # the elements are not explicitly initialized; expect random␣
 ↪values
```

[9]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

The default `dtype` is `numpy.float64` for these functions.

2.4 Using `arange`, `linspace`

Similar to `range` in Python, `arange` in NumPy returns a sequence of numbers in a `ndarray`. Use the `dtype` parameter to change the type, or use `astype()` function to cast into another type.

[10]:
```
np.arange(1, 3, 0.2)
```

[10]:
```
array([1. , 1.2, 1.4, 1.6, 1.8, 2. , 2.2, 2.4, 2.6, 2.8])
```

Instead of specifying the `step` as above, we can use `linspace` when we are trying to create a sequence of floating point numbers, and specify how many elements we want.

[11]:
```
np.linspace(1, 3, 7)
```

[11]:
```
array([1.        , 1.33333333, 1.66666667, 2.        , 2.33333333,
       2.66666667, 3.        ])
```

3 Playing with the Shapes of `ndarrays`

[12]:
```
arr = np.arange(1, 10)
arr
```

[12]:
```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

3.1 How to Get the Shape of an `ndarray`

[13]:
```
arr.shape
```

[13]:
```
(9,)
```

3.2 How to Reshape the `ndarray`:

[14]:
```
arr.reshape((3, 3))
```

[14]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

The `reshape` function returns a new `ndarray` with the shape changed without modifying the original one.

[15]:
```
arr
```

[15]:
```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[16]:
```
arr.reshape((9,1))
```

[16]:
```
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6],
       [7],
       [8],
       [9]])
```

To directly modify the shape of an `ndarray`:

[17]:
```
arr.shape = (3, 3)
arr
```

[17]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Note that 1d arrays and 2d arrays are different.

[18]:
```
xx = arr.reshape((1,9))
xx
```

[18]:
```
array([[1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

[19]:
```
xx.shape
```

[19]:
```
(1, 9)
```

[20]:
```
arr
```

[20]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

[21]:
```
arr.shape
```

[21]:
```
(3, 3)
```

4 Basic Operations

4.1 Indexing and slicing

The items of an array can be accessed and assigned to the same way as other Python sequences (e.g. lists):

[22]:
```
a = np.arange(10)
print(a)
a[0], a[2], a[-1]
```

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

[22]:
```
(0, 2, 9)
```

The usual python idiom for reversing a sequence is supported:

[23]:
```
a[::-1]
```

[23]:
```
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

For multidimensional arrays, indices are tuples of integers:

[24]:
```
a = np.diag(np.arange(3))
print(a)
print(a[1, 1])

a[2, 1] = 10 # third line, second column
print(a)
print(a[1])
```

```
[[0 0 0]
 [0 1 0]
 [0 0 2]]
1
[[ 0  0  0]
 [ 0  1  0]
 [ 0 10  2]]
[0 1 0]
```

Slicing: Arrays, like other Python sequences can also be sliced:

[25]:
```
a = np.arange(10)
print(a)
print(a[2:9:3]) # [start:end:step]
```

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

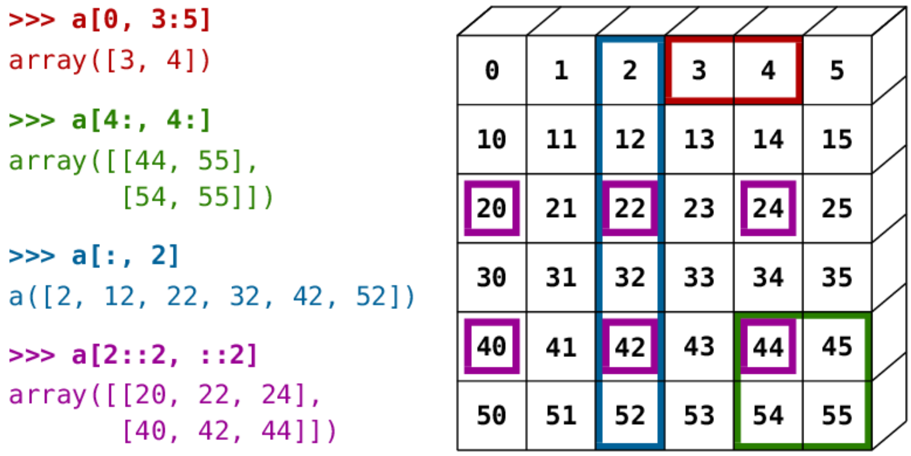
Note that the last index is not included:

[26]:
```
a[:4]
```

[26]: array([0, 1, 2, 3])

A small illustrated summary of NumPy indexing and slicing:

[27]:
```
# this is for illustration only, no need to run the cell by yourself
from IPython.display import display, Image
display(Image(filename="numpy_indexing.png", height=400, width=400))
```



4.2 Arithmetic Operators

Arithmetic operators on `ndarrays` apply elementwise (so the operation is *vectorized*). A new `ndarray` will be created to hold the result.

[28]:
```
arr = np.array([1, 2])
arr
```

[28]: array([1, 2])

[29]:
```
arr + 1
```

[29]: array([2, 3])

[30]:
```
arr * 2
```

[30]: array([2, 4])

[31]:
```
arr ** 2
```

[31]: array([1, 4])

4.3 Dot Products

Given two `ndarrays` with proper shapes:

[32]:
```
a = np.array([1, 2])
a
```

[32]:
```
array([1, 2])
```

[33]:
```
b = np.array([[1], [2]])
b
```

[33]:
```
array([[1],
       [2]])
```

To calculate the dot product, the sentence in the following cell is intuitive but **WRONG**:

[34]:
```
a * b  # calculating elementwise product!
```

[34]:
```
array([[1, 2],
       [2, 4]])
```

To correctly calculate the dot products of two `ndarrays`, use `numpy.dot` or the `dot` function on the `ndarray` object.

[35]:
```
a.dot(b)
```

[35]:
```
array([5])
```

[36]:
```
np.dot(a, b)
```

[36]:
```
array([5])
```

Both ways create a new `ndarray` to hold the results without modifying the original ones.

4.4 Matrix multiplication

Below is an example showing how to multiply two matrices and get the result.

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}\begin{bmatrix}-1 & 1\\2 & 1\end{bmatrix} = \left[\left(-1\cdot\begin{bmatrix}1\\3\end{bmatrix} + 2\cdot\begin{bmatrix}2\\4\end{bmatrix}\right) \quad \left(1\cdot\begin{bmatrix}1\\3\end{bmatrix} + 1\cdot\begin{bmatrix}2\\4\end{bmatrix}\right)\right] = \begin{bmatrix}3 & 3\\5 & 7\end{bmatrix}$$

Since Python 3.5, we can use the operator `*` for element-wise multiplication, and the operator `@` for matrix multiplication.

[37]:
```
A = np.array([[1,2],[3,4]])
B = np.array([[-1,1],[2,1]])
print(np.matmul(A, B)) # preferred over np.dot()
print(A @ B) # Matrix multiplication. A and B must be numpy arrays
print(A * B) # Element-wise multiplaction

# Unlike ordinary multiplication, matrix multiplication is not symmetric,
```

```python
# so that, in general A@B does not equal B@A:
print(B @ A)
```

```
[[3 3]
 [5 7]]
[[3 3]
 [5 7]]
[[-1  2]
 [ 6  4]]
[[2 2]
 [5 8]]
```

[38]:
```python
x = np.array([1, 2]).reshape((2,1))
print(A @ x)
```

```
[[ 5]
 [11]]
```

To solve $\mathbf{Az} = \mathbf{B}$, we have $\mathbf{z} = \mathbf{A}^{-1}\mathbf{B}$

[39]:
```python
z = np.linalg.inv(A) @ B
z
```

[39]:
```
array([[ 4. , -1. ],
       [-2.5,  1. ]])
```

[40]:
```python
A @ z
```

[40]:
```
array([[-1.,  1.],
       [ 2.,  1.]])
```

4.5 Working with mathematical formulas

The ease of implementing mathematical formulas that work on arrays is one of the things that make NumPy so widely used in the scientific Python community.

For example, this is the mean square error formula: $MeanSquareError = \frac{1}{2n}\sum_{i=1}^{n}(y_i' - y_i)^2$

Implementing this formula is simple and straightforward in NumPy using `np.sum` and `np.square`.

Now put what you've learned into practice by trying out the following MSE function.

[41]:
```python
def mean_squared_error(y_true, y_pred):
    """
    Compute the Mean Squared Error (MSE) between true and predicted values.

    Parameters:
        y_true (numpy.ndarray): Array of true values.
        y_pred (numpy.ndarray): Array of predicted values.

    Returns:
```

```
        float: Mean Squared Error (MSE).
    """
    # Ensure inputs are NumPy arrays
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)

    # Compute squared differences
    squared_errors = np.square(y_true - y_pred)

    # Compute sum of squared differences
    sum_squared_errors = np.sum(squared_errors)

    # Compute mean squared error
    mse = sum_squared_errors / (2 * len(y_true))

    return mse
```

[42]:
```
# Example:
y_true = np.array([3, -0.5, 2, 7])  # Example true values
y_pred = np.array([2.5, 0.0, 2, 8])  # Example predicted values
mse = mean_squared_error(y_true, y_pred)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 0.1875
```

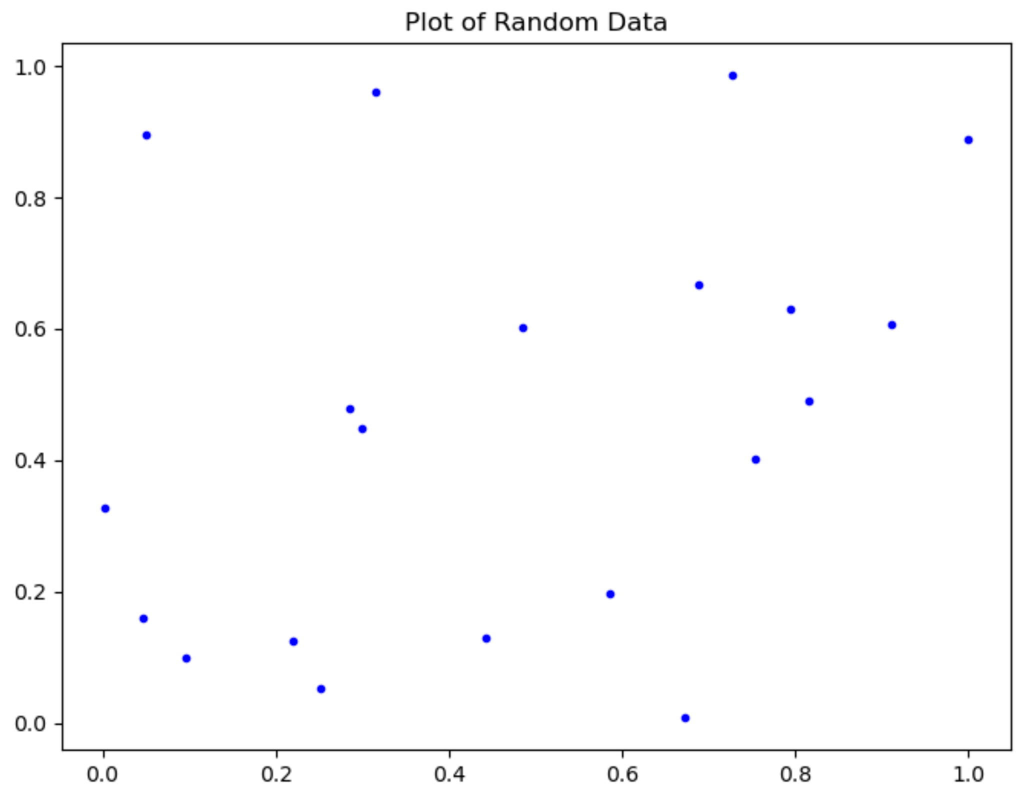
4.6 Working with pandas dataframe and matplotlib

Pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables.

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy.

[43]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Generate 20 random data points
x = np.random.rand(20)  # Generating random x values
y = np.random.rand(20)  # Generating random y values

# Creating a pandas DataFrame
data = pd.DataFrame({'X': x, 'Y': y})

# Creating a scatter plot using Matplotlib with plt.plot
# 'b.' indicates blue dots
plt.figure(figsize=(8, 6))
```

```python
plt.plot(data['X'], data['Y'], "b.", label='Random Data')

# Adding title
plt.title('Plot of Random Data')
# Displaying the plot
plt.show()
```



5 References

- NumPy documents: https://numpy.org/doc/stable/user/index.html#user
- Pandas documents: https://pandas.pydata.org/docs/user_guide/index.html
- Matplotlib documents: https://matplotlib.org/stable/users/index.html