In this project you going to implement some basics of numpy that you learnt till now.

Numpy

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays. If you are already familiar with MATLAB, you might find this tutorial useful to get started with Numpy.

1. Numpy Array

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

In [ ]:
```
import numpy as np

# Q1. Create a rank 1 array with elements 1,2,3 (1 marks)

## start code here
a =

# Q2. print its type and shape (1 marks)

## Start code here

# Q3. Print each of its element individually (1 marks)

## Start code here

# Q4. Change element of the array at zero index with 5  (1 marks)

## Start code here

# print new array

# Q5. Create a rank 2 array like [[1,2,3],[4,5,6]]  (1 marks)

## Start code here
b =

# Q6. print its shape (1 marks)

print(b[0, 0], b[0, 1], b[1, 0])    # Prints "1 2 4"
```

Numpy also provides many functions to create arrays:

In [ ]:
```
# Q1. Create an array of all zeros with shape (2,2) (1 marks)
## Start code here
a =
print(a)

# Q2. Create an array of all ones with shape (1,2) (1 marks)
## Start code here
b =
print(b)

# Q3. Create a constant array with shape (2,2) and value 7 (1 marks)
## Start code here
c =
print(c)

# Q4. Create a 2x2 identity matrix (1 marks)
## Start code here
d =
print(d)

# Q5. Create an (2,2) array filled with random values (1 marks)
## Start code here
e =
print(e)
```

You can read about other methods of array creation in the documentation in the link below.

http://docs.scipy.org/doc/numpy/user/basics.creation.html#arrays-creation

2. Array indexing

Numpy offers several ways to index into arrays.

a) Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

In [ ]:
```
# Q1. Create the following rank 2 array with shape (3, 4) and values as given below (1 mark)
# [[ 1  2   3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]

# Start code here
a =

# Q2. Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2): (1 mark)

# Start code here
b =

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77    # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array. Note that this is quite different from the way that MATLAB handles array slicing:

In [ ]:
```
# Q1. Create the following rank 2 array with shape (3, 4) value as given below (1 marks)
# [[ 1  2   3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]

# Start code here
a =

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the original array:

# Q2. Create Rank 1 view of the second row of a (1 marks)
# Start code here
row_r1 =

# Q3. # Create Rank 2 view of the second row of a (1 marks)
# Start code here
row_r2 =

print(row_r1, row_r1.shape)
print(row_r2, row_r2.shape)
# We can make the same distinction when accessing columns of an array:

# Q4. Create Rank 1 view of the second column of a (1 marks)
# Start code here
col_r1 =

# Q5. Create Rank 2 view of the second column of a (1 marks)
# Start code here
col_r2 =

print(col_r1, col_r1.shape)
print(col_r2, col_r2.shape)
```

b) Integer array indexing: When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

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

# An example of integer array indexing.
# Using the array "a" created above and integer indexing

# Q1. print an array  that should have shape (3,) and should print "[1 4 5]" when printed (1 marks)
## Start code


# The method of integer array indexing that you implemented is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:

# Q2. Print the array that prints "[2 2]" when printed (1 marks)

## Start code

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

In [ ]:
```
# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)


# Create an array of indices
b = np.array([0, 2, 0, 1])

# Q1. Select one element from each row of a using the indices in b and print it (1 marks)
## Start code


# Q2. Mutate one element from each row of a using the indices in b by adding 10 to it and print it (1 marks)
## Start code

print(a)
```

c) Boolean array indexing: Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

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

# Q1. Find the elements of a that are bigger than 2 using boolean indexing (1 marks)
## Start code
bool_idx =

print(bool_idx)

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])

# We can do all of the above in a single concise statement:
print(a[a > 2])
```

For brevity we have left out a lot of details about numpy array indexing; if you want to know more you should read the documentation from the given link. http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html

3. Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

In [ ]:
```
x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

You can read all about numpy datatypes in the documentation from the given link.
http://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html

4. Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

In [ ]:
```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Q1. print Elementwise sum of x and y (1 marks)
## Start code here

# Q2. print Elementwise difference of x and y (1 marks)
## Start code here

# Q3. print Elementwise product of x and y (1 marks)
## Start code here

# Q4. print Elementwise divison of x and y (1 marks)
## Start code here

# Q5. print Elementwise square root of x (1 marks)
## Start code here
```

Note that unlike MATLAB, * is elementwise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

In [ ]:
```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Q1. print Inner product of vectors v and w; (1 marks)
## Start code here

# Q2. print Matrix / vector product of x and v; (1 marks)
## Start code here

# Q3. print Matrix / matrix product of x and y; (1 marks)
## Start code here
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is sum:

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

# Q1. Compute sum of all elements and print it (1 marks)
## Start code here

# Q2. Compute sum of each column and print it (1 marks)
## Start code here

# Q3. Compute sum of each row and print it (1 marks)
## Start code here
```

You can find the full list of mathematical functions provided by numpy in the documentation on the provided link.
http://docs.scipy.org/doc/numpy/reference/routines.math.html

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the T attribute of an array object:

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

# Q1. print x and its transpose (2 marks)
## Start code here

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])

# Q2. print v and its transpose (2 marks)
## Start code here
```