Basic exercises on Python Machine Learning Packages

There are four key/basic packages that are most widely used for data analysis and machine learning.

• NumPy
• Pandas
• Matplotlib
• SciPy

Pandas, NumPy, and Matplotlib play a major role and have the scope of usage in almost all data analysis tasks. Whereas SciPy supplements NumPy library and has a variety of key high-level science and engineering modules, the usage of these functions, however, largely depend on the use case to use case.

NumPy

NumPy is the core library for scientific computing in Python. It provides a high performance Multi-dimensional array object, and tools for working with these arrays. It's a successor of Numeric package. In 2005, Travis Oliphant created NumPy by incorporating features of the competing Numarray into Numeric, with extensive modifications. The concepts and the code examples to a great extent have been explained in the simplest form in his book Guide to NumPy. Here we'll only be looking at some of the key NumPy concepts that are a must or good to know in relevance to machine learning.

It is the fundamental package for scientific computing with Python. It contains various features including these important ones:

- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

Installation:

- **Mac** and **Linux** users can install NumPy via pip command:
  ```
  pip install numpy
  ```
- **Windows** does not have any package manager analogous to that in linux or mac.
  Please download the pre-built windows installer for NumPy (according to your system configuration and Python version).
  And then install the packages manually. Example: conda install numpy

Array

A NumPy array is a collection of similar data type values, and is indexed by a tuple of nonnegative numbers. The rank of the array is the number of dimensions, and the shape of an array is a tuple of numbers giving the size of the array along each dimension.

We can initialize NumPy arrays from nested Python lists, and access elements using square brackets.

Example 01: code for initializing NumPy array

```
import numpy as np

# Create a rank 1 array
a = np.array([0, 1, 2])
print(type(a))
# this will print the dimension of the array
print(a.shape)
print(a[0])
print(a[1])
print(a[2])
# Change an element of the array
a[0] = 5
print(a)

# Create a rank 2 array
b = np.array([[0,1,2],[3,4,5]])
print(b.shape)
print(b)
print(b[0, 0], b[0, 1], b[1, 0])
```

Creating NumPy Array

NumPy also provides many built-in functions to create arrays. The best way to learn this is through examples.

Example 02: Creating NumPy array

```
# Create a 3x3 array of all zeros
a = np.zeros((3,3))
print(a)

# Create a 2x2 array of all ones
b = np.ones((2,2))
print(b)

# Create a 3x3 constant array
```

```
c = np.full((3,3), 7)
print(c)

# Create a 3x3 array filled with random values
d = np.random.random((3,3))
print(d)

# Create a 3x3 identity matrix
e = np.eye(3)
print(e)

# convert list to array
f = np.array([2, 3, 1, 0])
print(f)

# arange() will create arrays with regularly incrementing values
g = np.arange(20)
print(g)

# note mix of tuple and lists
h = np.array([[0, 1,2.0],[0,0,0],(1+1j,3.,2.)])
print(h)

# create an array of range with float data type
i = np.arange(1, 8, dtype=np.float)
print(i)

# linspace() will create arrays with a specified number of items which are
# spaced equally between the specified beginning and end values j= np.linspace(start, stop, num)
j = np.linspace(2, 4, 5)
print(j)
```

Data Types

An array is a collection of items of the same data type and NumPy supports and provides built-in functions to construct arrays with optional arguments to explicitly specify required data types.

Example 3: NumPy data types

```
# Let numpy choose the datatype
x = np.array([0, 1])
y = np.array([2.0, 3.0])
# Force a particular datatype
z = np.array([5, 6], dtype=np.int64)
print x.dtype, y.dtype, z.dtype
```

Array Indexing

Knowing the basics of array indexing is important for analysing and manipulating the array object. NumPy offers many ways to do array indexing.

- **Slicing:** Just like lists in python, NumPy arrays can be sliced. As arrays can be multidimensional, you need to specify a slice for each dimension of the array.
- **Integer array indexing:** In this method, lists are passed for indexing for each dimension. One to one mapping of corresponding elements is done to construct a new arbitrary array.
- **Boolean array indexing:** This method is used when we want to pick elements from array which satisfy some condition.

Example 4: Python program to demonstrate indexing in numpy

```
import numpy as np

# An exemplar array
arr = np.array([[-1, 2, 0, 4],
                [4, -0.5, 6, 0],
                [2.6, 0, 7, 8],
                [3, -7, 4, 2.0]])

# Slicing array
temp = arr[:2, ::2]
print ("Array with first 2 rows and alternate columns(0 and 2):\n", temp)

# Integer array indexing example
temp = arr[[0, 1, 2, 3], [3, 2, 1, 0]]
print ("\nElements at indices (0, 3), (1, 2), (2, 1),(3, 0):\n", temp)

# boolean array indexing example
cond = arr > 0 # cond is a boolean array
temp = arr[cond]
print ("\nElements greater than 0:\n", temp)
```

```
Output:
Array with first 2 rows and alternatecolumns(0 and 2):
 [[-1.  0.]
 [ 4.  6.]]


Elements at indices (0, 3), (1, 2), (2, 1),(3, 0):
 [ 4.  6.  0.  3.]
```

```
Elements greater than 0:
 [ 2.   4.   4.   6.   2.6  7.   8.   3.   4.   2. ]
```

Basic array operations:

NumPy provides a rich set of built-in arithmetic functions those can be applied to perform operations on arrays.

- **Operations on single array:** We can use overloaded arithmetic operators to do element-wise operation on array to create a new array. In case of +=, -=, *= operators, the existing array is modified.

Example 5:

```
# Python program to demonstrate
# basic operations on single array
import numpy as np

a = np.array([1, 2, 5, 3])

# add 1 to every element
print ("Adding 1 to every element:", a+1)

# subtract 3 from each element
print ("Subtracting 3 from each element:", a-3)

# multiply each element by 10
print ("Multiplying each element by 10:", a*10)

# square each element
print ("Squaring each element:", a**2)

# modify existing array
a *= 2
print ("Doubled each element of original array:", a)

# transpose of array
a = np.array([[1, 2, 3], [3, 4, 5], [9, 6, 0]])

print ("\nOriginal array:\n", a)
print ("Transpose of array:\n", a.T)
```

```
Output:

Adding 1 to every element: [2 3 6 4]

Subtracting 3 from each element: [-2 -1  2  0]

Multiplying each element by 10: [10 20 50 30]
```

```
Squaring each element: [ 1  4 25  9]
Doubled each element of original array: [ 2  4 10  6]


Original array:
 [[1 2 3]
 [3 4 5]
 [9 6 0]]
Transpose of array:
 [[1 3 9]
 [2 4 6]
 [3 5 0]]
```

- **Unary operators:** Many unary operations are provided as a method of **ndarray** class. This includes sum, min, max, etc. These functions can also be applied row-wise or column-wise by setting an axis parameter

Example 6:

```
# Python program to demonstrate
# unary operators in numpy
import numpy as np

arr = np.array([[1, 5, 6],
                [4, 7, 2],
                [3, 1, 9]])

# maximum element of array
print ("Largest element is:", arr.max())
print ("Row-wise maximum elements:",arr.max(axis = 1))

# minimum element of array
print ("Column-wise minimum elements:",arr.min(axis = 0))

# sum of array elements
print ("Sum of all array elements:",arr.sum())

# cumulative sum along each row
print ("Cumulative sum along each row:\n",arr.cumsum(axis = 1))
```

```
Output:
Largest element is: 9
Row-wise maximum elements: [6 7 9]
Column-wise minimum elements: [1 1 2]
Sum of all array elements: 38
```

```
Cumulative sum along each row:
[[ 1  6 12]
 [ 4 11 13]
 [ 3  4 13]]
```

- **Binary operators:** These operations apply on array elementwise and a new array is created. You can use all basic arithmetic operators like +, -, /, , *etc. In case of* +=, -=, = operators, the exsisting array is modified.

Example 7:

```
# Python program to demonstrate
# binary operators in Numpy
import numpy as np

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

# add arrays
print ("Array sum:\n", a + b)

# multiply arrays (elementwise multiplication)
print ("Array multiplication:\n", a*b)

# matrix multiplication
print ("Matrix multiplication:\n", a.dot(b))
```

```
Output:
Array sum:
[[5 5]
 [5 5]]
Array multiplication:
[[4 6]
 [6 4]]
Matrix multiplication:
[[ 8  5]
 [20 13]]
```

- **Universal functions (ufunc):** NumPy provides familiar mathematical functions such as sin, cos, exp, etc. These functions also operate elementwise on an array, producing an array as output.

Example 8:

```
# Python program to demonstrate
# universal functions in numpy
import numpy as np

# create an array of sine values
a = np.array([0, np.pi/2, np.pi])
print ("Sine values of array elements:", np.sin(a))

# exponential values
a = np.array([0, 1, 2, 3])
print ("Exponent of array elements:", np.exp(a))

# square root of array values
print ("Square root of array elements:", np.sqrt(a))
```

```
Output:
Sine values of array elements: [  0.00000000e+00   1.00000000e+00   1.22464680e-16]
Exponent of array elements: [  1.           2.71828183   7.3890561   20.08553692]
Square root of array elements: [ 0.          1.          1.41421356  1.73205081]
```

Example 9: Matrix inverse

```
##simple python program to find the inverse of an array (without exception handling)
import numpy as np
arr = np.array([[-1, 2, 0, 4],
                [4, -0.5, 6, 0],
                [2.6, 0, 7, 8],
                [3, -7, 4, 2.0]])
inverse = np.linalg.inv(arr)
inverse
```

Linear Algebra using NumPy: (Few in-built functions): Try these out

Matrix and Vector Product

dot(a, b[, out])	Dot product of two arrays.
linalg.multi_dot(arrays)	Compute the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order.
vdot(a, b)	Return the dot product of two vectors.
inner(a, b)	Inner product of two arrays.
outer(a, b[, out])	Compute the outer product of two vectors.

matmul(x1, x2, /[, out, casting, order, …])	Matrix product of two arrays.
tensordot(a, b[, axes])	Compute tensor dot product along specified axes for arrays >= 1-D.
einsum(subscripts, *operands[, out, dtype, …])	Evaluates the Einstein summation convention on the operands.
einsum_path(subscripts, *operands[, optimize])	Evaluates the lowest cost contraction order for an einsum expression by considering the creation of intermediate arrays.
linalg.matrix_power(a, n)	Raise a square matrix to the (integer) power *n*.

Matrix Eigen Values

linalg.eig(a)	Compute the eigenvalues and right eigenvectors of a square array.
linalg.eigh(a[, UPLO])	Return the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix.
linalg.eigvals(a)	Compute the eigenvalues of a general matrix.