Library Usage Python Concepts to Solve Exercise Problems (P_VTP4)

11/23/2020

Contents

- Basic Slicing
  - P_VT4: 7
- Basic (Vectorized) Operations
  - P_VTP4: 2
- Bitwise Operations
  - P_VTP4: 12
- Arithmetic Functions
  - P_VTP4: 14
- Mathematical Functions
  - P_VTP4: 13, 15, 22, 23, 24
- String Operations
  - P_VTP4: 16, 17, 18

- Shuffle Usage
  - P_VTP4: 19, 20, 21
- Iterating Over Array
  - P_VTP4: 9, 10, 11
- Statistical Functions
  - P_VTP4: 25, 26, 27
- Sorting Functions
  - P_VTP4: 30, 31, 32
- Correspondence Between Each Topic and Related VTPs
- Conclusion

Correspondence Index

Numpy Arrays

- Numpy is **a library** for the Python, adding support for large, **multi-dimensional arrays** and **matrices**, along with a large collection of high-level **mathematical functions** to operate on these arrays.

- **Importing** a numpy can be seen as follow:

import numpy	a = numpy.array([4, 6, 2, 9])
import numpy as np	a = np.array([4, 6, 2, 9])
from numpy import *	a = array([4, 6, 2, 9])

Creating Numpy Arrays

Example-1: To create an array of **int** datatype
a = array([10, 20, 30, 40, 50], int) print(a) >> [10 20 30 40 50]

Example-2: To create an array of **float** datatype
a = array([10.1, 20.2, 30.3, 40.4, 50.5], float) print(a) >> [10.1 20.2 30.3 40.4 50.5]

Example-3: To create an array of **char** datatype
a = array([‘a’, ‘b’, ‘c’, ‘d’]) print(a) >> ['a' 'b' 'c' 'd']
Note: No need to specify explicitly the char datatype

Example-4: To create an array of **str** datatype

```
a = array (['abc', 'bcd', 'cde', 'def'], dtype=str)
print(a) >>  ['abc' 'bcd' 'cde' 'def']
```

Example-5: To create an array from tuple

```
a = array ((1, 3, 2))
print(a) >>  [1 3 2]
```

Example-6: To create an 2D array with 2 rows and 3 cols

```
a = array ([[1, 2, 3],
            [4, 5, 6]])
print(a) >>  [[1 2 3]
              [4 5 6]]
```

Creating an array with numpy-arange()

Syntax	arange(start, stop, stepsize)	
Example -1	arange(10)	Produces items from 0 – 9 >>[0 1 2 3 4 5 6 7 8 9]
Example -2	arange(5, 10)	Produces items from 5 - 9) >> [5 6 7 8 9]
Example -3	arange(1, 10, 3)	Produces items from 1, 4, 7 >> [1 4 7]
Example -4	arange(10, 1, -1)	Produces items >> [10 9 8 7 6 5 4 3 2]
Example -5	arange(0, 10, 1.5)	Produces [0. 1.5 3. 4.5 6. 7.5 9.]

Creating an array with numpy-zeros()

Syntax	zeros(n, datatype)	
Example -1	zeros(5)	Produces items [0. 0. 0. 0. 0.] Default datatype is float.
Example -2	zeros(5, int)	Produces items [0 0 0 0 0]
Example -3	zeros(1, 2)	Creating a 1*2 array with all zeros and produces items [0. 0.]

Attributes of an Array

- The **‘ndim’** attribute represents the number of dimensions or axes of an array.
- The number of dimensions are also called as ‘**rank**’.

Example

```
a = array ([1, 2, 3]) # one dimensional array
print (a.ndim) >> 1
b = numpy.array([[1, 2, 3],
                 [4, 5, 6]]) # two dimensional array
print(b.ndim) >> 2
```

- The **‘size’** attribute gives the total number of items in an array.

Example

```
a = array ([1, 2, 3])
print (a.size) >> 3
b = numpy.array([[1, 2, 3],
                 [4, 5, 6]])
print(b.size) >> 6
```

- The ‘**shape**’ attribute gives the shape of the array with corresponding rows and columns.

Example

```
a = array ([1, 2, 3])
print (a.shape) >> (1, 3)
b = numpy.array([[1, 2, 3],
                 [4, 5, 6]])
print(b.shape) >> (2, 3)
```

- The ‘**dtype**’ attribute gives the data type of the elements in the array.

Example

```
a = array ([1, 2, 3])
print (a.dtype) >> int32
b = array ([1.3, 2.1, 3.9])
print(b.dtype) >> float64
```

Methods of an Array

- The '**reshape**' method is useful to **change the shape** of an array.

Example-1:	

a = arange(10)
#Change the shape as 2 Rows, 5 Cols
a = a.reshape(2, 5)
print(a)
	Outputs:
	[[0 1 2 3 4]
	[5 6 7 8 9]]

Example-2:	

#Change the shape to 5 rows, 2 cols
a = a.reshape(5, 2)
print(a)
	Outputs:
	[[0 1]
	[2 3]
	[4 5]
	[6 7]
	[8 9]]

- The '**flatten**' method is useful to **return copy** of an array collapsed into **one dimension**.

Example-1:	

#flatten() method
a = array([[1, 2], [3, 4]])
print(a)
#Change to 1D array
a = a.flatten()
print(a)
	Outputs:
	[1 2 3 4]

- The **append()** method appends values along the mentioned axis at the end of the array

Example-1: Working on 1D	

import numpy as np
 	
arr1 = np.arange(3)
print("1D arr1 : ", arr1)
 	
arr2 = np.arange(3, 6)
print("\n1D arr2 : ", arr2)
 	
# appending the arrays 	
print("\nAppended arr3 : ", **np.append**(arr1, arr2))
	Outputs:
	1D arr1 : [0 1 2]
	
	1D arr2 : [3 4 5]
	
	Appended arr3 :
	[0 1 2 3 4 5]

Example-2: Working on 2D	

import numpy as np
 	
arr1 = np.arange(4).reshape(2, 2)
print("2D arr1 : \n", arr1)
 	
arr2 = np.arange(8, 12).reshape(2, 2)
print("\n2D arr2 : \n", arr2)
 	
# appending the arrays 	
arr3 = **np.append**(arr1, arr2)
print("\nAppended arr3 by flattened : ", arr3)
	Outputs:
	2D arr1 :
	[[0 1]
	[2 3]]
	
	2D arr2 :
	[[8 9]
	[10 11]]
	
	Appended arr3 by flattened : [0 1 2 3 8 9 10 11]

- The **vstack()** function is used to stack the sequence of input arrays vertically to make a single array.

Example:	

import numpy as np
 	
in_arr1 = np.array([8, 1, 3])
print ("1st Input array : \n", in_arr1)
 	
in_arr2 = np.array([2, 5, 4])
print ("2nd Input array : \n", in_arr2)
 	
# Stacking the two arrays vertically 	
out_arr = np.vstack((in_arr1, in_arr2))
print ("Output vertically stacked array:\n ", out_arr)
	Outputs:
	1st Input array :
	[8 1 3]
	2nd Input array :
	[2 5 4]
	Output vertically stacked array:
	[[8 1 3]
	[2 5 4]]

Indexing of an Array

Example	

from numpy import *
#Create an 2D array with 3 rows, 3 cols
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
#Display only rows
for i in range(len(a)):
    print(a[i])
#display item by item
for i in range(len(a)):
    for j in range(len(a[i])):
        print(a[i][j], end=' ')
x = array([1, 2, 3, 4, 5])
arr = x[array([1, 3, -1])]
print("\n Elements are : \n",arr)
	Output :
	[1, 2, 3]
	[4, 5, 6]
	[7, 8, 9]
	1 2 3 4 5 6 7 8 9
	Elements are :
	[2 4 5]

- In **Boolean Array Indexing**, elements are returned which **satisfy Boolean expression**.
- It is used **for filtering** the desired element values.

Example-1:	

a = array([10, 40, 80, 50, 100])
# Select numbers greater than 40 	
print(**a[a>40]**)
a = array([10, 40, 80, 50, 100])
# print the squaring to the multiples of 50 	
print(**a[a%50==0]**2**)
	Outputs:
	[50 80 100]
	[2500 10000]

Basic Slicing

Example

import numpy as np

Arrange elements from 0 to 10

a = np.arange(10)

print(a) >> [0 1 2 3 4 5 6 7 8 9]

a[start:stop:step]

print(a[-4:8:1]) >> [6 7]

The : operator means all elements till the end

print(a[7:]) >> [7 8 9]

A 3 dimensional array.

b = np.array([[[1, 2, 3],[7, 1, 6]],
[[4, 3, 1],[1, 2, 2]]])

Equivalent to b[: ,: ,1] and it will print values from every row and column in index 1 using basic slicing with ellipsis

print(b[...,1]) >> [[2 1]
[3 2]]

Basic (Vectorized) Operations

Importance of vectorized operations
1. Operations are faster - Adding two arrays in the form a + b is faster than taking corresponding items of both arrays and then adding them.
2. Syntactically clearer - Writing a + b is clearer than using the loops
3. Provides compact code

Example-1	a = array([1, 2, 3, 4]) #Adds 5 to each item of an array print(a + 5) >> [6, 7, 8, 9] #Subtracts 1 from each item of an array print(a - 1) >> [0, 1, 2, 3] #Multiply 2 to each item of an array print(a * 2) >> [2, 4, 6, 8] #Divide 1 to each item of an array print(a / 1) >> [1, 2, 3, 4] #Sum of array elements performing unary operation print(a.sum()) >> 10 #Squaring each element print(a ** 2) >> [1, 4, 9, 16]
Example-2	a1 = array([0, 2, 3, 1]) a2 = array([1, 2, 3, 4]) #Adds each item of a1 and a2 Print(a1 + a2) >> [1, 4, 6, 5]

Index Page

Bitwise Operations

- bitwise_and() function is used to compute the bit-wise AND of two array element-wise.
- bitwise_or() function is used to compute the bit-wise OR of two array element-wise.
- bitwise_xor() function is used to compute the bit-wise XOR of two array element-wise.

Example

a = 11

b = 10

print(**bitwise_and**(a, b)) >> 10

print(**bitwise_or**(a, b)) >> 11

print(**bitwise_xor**(a, b)) >> 1

Arithmetic Functions

add()	Add arguments element-wise.
subtract()	Subtract arguments element-wise.
multiply()	Multiply arguments element-wise.
divide()	Array element from first array is divided by elements from second element.
negative()	Numerical negative, element-wise.
power()	First array elements raised to powers from second array, element-wise.

- **Example**

```
import numpy as np
arr1 = [2, 4, 6, 2]
arr2 = [2, 2, 3, 1]
print (np.divide(arr1, arr2)) >> [ 1.  2.  2.  2.]
print (np.multiply(arr1, arr2)) >> [ 4  8 18  2]
in_num1 = 3
in_num2 = 1
print (np.negative(in_num1)) >> -3
print (np.add(in_num1, in_num2)) >> 4
print (np.subtract(in_num1, in_num2))  >> 2
arr1 = [2, 2, 2, 2, 2]
arr2 = [2, 3, 4, 5, 6]
print (np.power(arr1, arr2) ) >> [ 4  8 16 32 64]
```

Mathematical Functions

- **min(a)** returns the **min value** in the array a.
- **max(a)** returns the **max value** in the array a.
- **around()** helps user to **evenly round** array elements to the given number of decimals.
- **dot()** returns the dot **product value** of elements.
- **isreal()** tests element-wise whether it is **a real number or not** and return the result as a **boolean array**.
- **conj()** helps the user to conjugate any complex number. The conjugate of a complex number is obtained by changing **the sign of its imaginary part.**

- **Example**

(for max() and min())

a = [1, 4, 5]

b = [7, 3, 1]

print(**np.maximum(a,b)**) >> [7 4 5]

print(**np.minimum(a,b)**) >> [1 3 1]

(for around())

in_array = [.4, 2.2, 1.1, 8.6]

print (**around(in_array)**) >> [0. 2. 1. 9.]

in_array = [.43, 3.53, .11]

print (**around(in_array)**) >> [0. 4. 0.]

in_array = [.3338, 1.55454, .73415]

print (around(in_array, **decimals = 3**)) >> [0.334 1.555 0.734]

- **Example**

(for dot())

```
import numpy as np
print("Dot Product of scalar values :", np.dot(3, 2))
>> Dot Product of scalar values : 6
vector_a = 3 + 4j
vector_b = 2 + 5j
print("Dot Product of vector values :", np.dot(vector_a, vector_b))
 >> Dot Product of vector values : (-14+23j)
arr_a = np.array([[1, 1], [5, 3]])
arr_b = np.array([[2, 1], [3, 2]])
print("Dot Product in 2D array  : \n", np.dot(arr_a, arr_b))
>> Dot Product in 2D array  :  [[ 5  3]
                               [19 11]]
```

- **Example**

(for isreal(), conj())

```
import numpy as np
print(np.isreal([2+1j,  0j]), "\n") >> [False  True]
print(np.isreal([3,  0]), "\n") >> [ True  True]

in_complx1  = 1+3j
print (np.conj(in_complx1))  >> (1-3j)

in_complx2  =8-5j
print (np.conj(in_complx2))  >> (8+5j)
```

String Operations

- **lower( )** returns the **lowercase** string from the given string.

- **upper( )** returns the **uppercase** string from the given string.

- **split()** returns a **list of strings** after breaking the given string by the **specified separator**.

- **join()** returns a string in which the elements of sequence have been **joined by str separator**.

- **count()** returns the number of **occurrences of a substring** in the given string.

- **rfind()** returns the **highest index** of the substring if found in given string. If not found then it returns -1.
- **isnumeric()** returns **True** if all characters in the string are numeric characters, Otherwise, it returns **False**.

- **Example**

\# converting to lowercase

print(**np.char.lower**('WELCOME')) >> welcome

\# converting to uppercase

print(**np.char.upper**('hi John')) >> HI JOHN

splitting a string

print(**np.char.split**('Today is holiday')) >> ['Today', 'is', 'holiday']

splitting a string with another format

print(**np.char.split**('Today, is, holiday', sep = ','))

>>['Today', ' is', ' holiday']

Joining a string with str separator

print(**np.char.join**('-', 'welcome')) >> w-e-l-c-o-m-e

Joining a string with another format by str separator

print(**np.char.join**(['-', ':'], ['geeks', 'for'])) >> ['g-e-e-k-s' 'f:o:r'] a=np.array(['Welcome', 'from', 'this place'])

counting a substring and the output will be printed like [0 0 1]

print(**np.char.count**(a,'this')) >> [0 0 1]

counting a substring and the output will be printed like [0 0 1]

print(**np.char.count**(a, 'om')) >> [1 1 0]

Finding a substring and the output will be printed like [0 -1 0]

print(**np.char.rfind**(a,'from')) >> [-1 0 -1]

Checking numeric or not and the ouput will be printed as True or False.

print(**np.char.isnumeric**('Welcome')) >> False

print(**np.char.isnumeric**('12')) >> True

String Comparision

- **equal()** checks for string1 == string2 element wise.
- **not_equal()** checks whether two string is **unequal or not**.
- **greater()** checks whether string1 **is greater** than string2 or not.
- **greater_equal()** checks whether string1 **>=** string2 or not.
- **less_equal()** checks whether string1 is <= string2 or not.
- **less()** checks whether string1 is **lesser than** string2 or not.

Example

import numpy as np

print(**np.char.equal**('Welcome','hi')) >> False

print(**np.char.not_equal**('welcome','hi')) >> True

print(**np.char.greater**('welcome','hi')) >> True

print(**np.greater_equal**([2., 3.], [1., 2.])) >> [True True]

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

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

print(a **>=** b) >> [False True]

print(a < b) >> [True False]

print(**np.less_equal**([4., 2.], [3., 3.])) >> [False True]

print(**np.less**([4., 2.], [3., 3.])) >> [False True]

Shuffle Usage

- **random.shuffle()** is used to shuffle the list in place. i.e., it randomizes the order of items in a list.
- **Example**

import random

number_list = [7, 4, 1, 8]

Assume the output result after shuffle be [1, 8, 7, 4] print(**random.shuffle**(number_list)) **>>** [1, 8, 7, 4]

To Shuffle **two List at once** with the same order

list1_names = ['Jack', 'Emma', 'Smitt']

list2_id = [70, 50, 90]

```
mapIndexPosition = list(zip(list1_names, list2_id))
random.shuffle(mapIndexPosition)
list1_names, list2_id = zip(*mapIndexPosition)

print(" \nLists after Shuffling")
>>  Lists after Shuffling
print("Employee Names: ", list1_names)
>> Employee Names:  ('Emma', 'Smitt', 'Jack')
print("Employee ID: ", list2_id)
>> Employee ID:  (50, 90, 70)
```

- **random.shuffle()** does **not work with string** and so, shuffling string can be done by following step by step.

  - **Convert** String to list
  - **Shuffle** the list randomly
  - **Convert** the shuffled list into String

- **Example**

```
import random
sampleStr = "Welcome"
char_list = list(sampleStr) # convert string into list
random.shuffle(char_list) # shuffle list
finalStr = ''.join(char_list) # convert list to string
# Assume the resulted shuffled string is wemeocl
print(finalStr) >> wemeoc
```

Iterating Over Array

- NumPy package contains an iterator object **numpy.nditer**.
- It is an efficient **multidimensional iterator object** using which it is possible to iterate over an array.
- **Example**

```
import numpy as np
a = np.arange(8) # creating an array using arrange method
a = a.reshape(2,4) # shape array with 2 rows and  4 columns
print(a) >> [[0 1 2 3]
             [4 5 6 7]]
print('Iterating an array is:')
for x in np.nditer(a):
   print(x, end = ",")    >> 0,1,2,3,4,5,6,7,
print()
```

Creating second array using array method

b = np.array([5, 6, 7, 8], dtype = int)

print(b) >> [5 6 7 8]

If two arrays are broadcastable, a combined nditer object is able to iterate upon them concurrently.

```
print('Modified array is:')
for x,y in np.nditer([a,b]):
    print("%d:%d" % (x,y), end = ",")
```

output >> 0:5,1:6,2:7,3:8,4:5,5:6,6:7,7:8,

- Array values can also **be modified** by using **op_flags** using the iterator nditer.
- Its **default** value is **read-only**, but can be set to read-write or write-only mode.
- **Example**

```
 import numpy as np
 a = np.arange(4)
 a = a.reshape(2,2) # shape array with 2 rows and 2 columns
print(a) >> [[0 1]
             [2 3]]
# modifying array values
for x in np.nditer(a, op_flags = ['readwrite']):
   x[...] = 3*x
 print('Modified array is:‘, a) >> [[0 3]
                                    [6 9]]
```

Statistical functions

- **mean(arr, axis = None)** computes the **arithmetic mean** (average) of the given data (array elements) along the specified axis along which we want to calculate the arithmetic mean.

- **var(arr, axis = None)** computes the **variance** of the given data (array elements) along the specified axis.

- **std(arr, axis = None)** computes the **standard deviation** of the given data (array elements) along the specified axis.

- axis = **0** means along **the column** and axis = **1** means working along **the row**.

- To **calculate mean**, if arr = [2, 3, 4, 5],then (2+3+4+5)/4 = 3.5 and the output will be printed like 3.5.

- To **calculate variance**, x = 1 1 1 1 1 Standard Deviation = 0 . Variance = 0, y = 9, 2, 5, 4, 12, 7, 8, 11, 9, 3, 7, 4, 12, 5, 4, 10, 9, 6, 9, 4

Step 1 : Mean of distribution 4 = 7

Step 2 : Summation of (x - x.mean())**2 = 178

Step 3 : Finding Mean = 178 /20 = 8.9

This Result is Variance.

- To **calculate standard Deviation**, x = 1 1 1 1 1 Standard Deviation = 0 . Variance = 0, y = 9, 2, 5, 4, 12, 7, 8, 11, 9, 3, 7, 4, 12, 5, 4, 10, 9, 6, 9, 4

Step 1 : Mean of distribution 4 = 7

Step 2 : Summation of (x - x.mean())**2 = 178

Step 3 : Finding Mean = 178 /20 = 8.9

This Result is Variance.

Step 4 : Standard Deviation = sqrt(Variance) = sqrt(8.9) = 2.983.

- **Example**

```
arr = [2, 3, 4, 5] # 1D array
print("mean of arr : ", np.mean(arr)) >> 3.5
print("Variance of arr : ", np.var(arr)) >> 1.25
print("std of arr : ", np.std(arr)) >> 1.11803398875
# 2D array
arr = [[4, 1, 0],
       [5, 6, 2],
       [3, 2, 4]]
```

mean of the flattened array, calculate the sum of all values and then divided by 9

```
print("\nmean of arr, axis = None : ", np.mean(arr)) >> 3.0
# var of the flattened array
print("\nvar of arr, axis = None : ", np.var(arr)) >> 3.33333333333
print("\nstd of arr, axis = None : ", np.std(arr)) >> 1.82574185835
```

mean along the axis = 0 that calculates mean value along each column

print("\nmean of arr, axis = 0 : ", np.mean(arr, axis = 0))

>>[4. 3. 2.]

var along the axis = 0

print("\nvar of arr, axis = 0 : ", np.var(arr, axis = 0))

>>[0.66666667 4.66666667 2.66666667]

std along the axis = 0

print("\nstd of arr, axis = 0 : ", np.std(arr, axis = 0))

[0.81649658 2.1602469   1.63299316]

mean along the axis = 1 that calculates mean value along each row

```
print("\nmean of arr, axis = 1 : ", np.mean(arr, axis = 1))
>> [ 1.66666667  4.33333333  3.        ]
```

var along the axis = 1

```
print("\nvar of arr, axis = 1 : ", np.var(arr, axis = 1))
>> [ 2.88888889  2.88888889  0.66666667]
```

std along the axis = 1

```
print("\nstd of arr, axis = 1 : ", np.std(arr, axis = 1))
>> [ 1.69967317  1.69967317  0.81649658]
```

Sorting functions

- numpy.sort() : This function returns a **sorted copy** of an array.

- numpy.argsort():This function returns **the indices** that would sort an array.

- numpy.lexsort():This function returns **an indirect stable sort** using a sequence of keys.

- **Example** (for sort())

```
import numpy as np
# sort along the first axis
 a = np.array([[1, 5], [7, 3]])
 print("original array is:\n",a) >> [[1 5]
                                      [7 3]]
# sorted values for each column with axis = 0
print ("Along first axis = 0 : \n", np.sort(a, axis = 0))
>>[[1 3]
   [7 5]]
print ("\nAlong none axis : \n", np.sort(a, axis = None) )
>> [1 3 5 7]
```

- **Example** (for argsort())

```
import numpy as np
a = np.array([6, 3, 1, 2, 3])
# unsorted array print
print('Original array:\n', a) >> [6 3 1 2 3]
# Sort array indices
b = np.argsort(a)
print('Sorted indices of original array->', b) >> [2 3 1 4 0]
# To get sorted array using sorted indices, c is temp array created of same len as of b
c = np.zeros(len(b), dtype = int)
for i in range(0, len(b)):
    c[i]= a[b[i]]
print('Sorted array->', c) >> [1 2 3 3 6]
```

Example (lexsort())

```
import numpy as np
# Numpy array created  First column
a = np.array([3, 1, 3, 6])

# Second column
b = np.array([7, 1, 3, 7])
print('column a, column b')

for (i, j) in zip(a, b):
    print(i, ' ', j)
# Sort by a then by b
ind = np.lexsort((b, a))
print('Sorted indices->', ind)
```

Output :

```
column a, column b
3   7
1   1
3   3
6   7
Sorted indices-> [1 2 0 3]
```

Correspondence Between Each Topic and Related VTPs

Library Usage Python Concept Topic	Related Problem Number
Creating Numpy Arrays	1, 4
Attributes of an Array	3
Methods of an Array	5, 28, 29
Indexing of an Array	6, 8
Basic Slicing	7
Basic (Vectorized) Operations	2
Bitwise Operations	12

Library Usage Python Concept Topic	Related Problem Number
Arithmetic Functions	14
Mathematical Functions	13, 15, 22, 23, 24
String Operations	16, 17, 18
Shuffle Usage	19, 20, 21
Iterating Over Array	9, 10, 11
Statistical Functions	25, 26, 27
Sorting Functions	30, 31, 32

Conclusion

- This slide introduces numpy library usage concepts for Python Programming.