NumPy Arrays II

Activity 1: Array Length

Let's begin the lesson with array length. The length of an array (or array length) is:

- The number of items in a one-dimensional array.
- The number of rows in a two-dimensional array.
- The number of blocks in a three-dimensional array.

To calculate the length of a NumPy array, you can use the `len()` function.

Let's create an array for all the three different dimensions and calculate their lengths using the `len()` function.

```
# Create a NumPy array for all the three different dimensions and calculate their lengt
import numpy as np
arr1=np.ones((5,2,6),dtype=int)
arr1
```

```
len(arr1)
```

```
len(arr2)
```

```
arr2=np.zeros((20))
arr2
```

Note: You do not have to create the arrays with the exact same items. You can choose to have different items in the arrays.

As you can see, the `len()` function gives different outputs for multi-dimensional NumPy arrays.

Activity 2: The `size` Keyword

To find the number of items in a NumPy array, you can use the `size` keyword. Regardless of dimensions, the `size` keyword will always return the number of items in a NumPy array.

```
# Compute the number of items in the 'one_dim_ar', 'two_dim_ar' and 'three_dim_ar' Num
arr1.size
```

```
arr1=np.ones((5,2,6,6),dtype=int)
arr1
arr1.size
```

```
arr1
```

Note: The `size` keyword is not available for a Python list.

Activity 3: Descriptive Statistics

Using NumPy arrays, we can easily do some statistical calculations.

Consider that you are a smartphone retailer and you have few smartphones in your inventory.

Smartphone Model	Price (INR)	# Units Available
Samsung Galaxy M30S	13999	9
Realme C2	6298	8
Xiaomi Redmi Note 7 Pro	10999	9
Xiaomi Redmi Note 8 Pro	14999	9

Smartphone Model	Price (INR)	# Units Available
Realme C2 32GB	7298	8
Realme C2 2GB RAM	6385	8
Realme 5	8999	9
Xiaomi Redmi Note 7S 64GB	9999	6
Xiaomi Redmi Note 8	9999	5
Vivo Z1 Pro	13868	7

Suppose you decided to do some analysis of your inventory. In the process, you want to find answers to the following questions:

1. What is the total monetary value of the inventory?
2. What is the average (or mean) price of a smartphone ?
3. What is the price of the cheapest smartphone in the inventory?
4. What is the price of the most expensive smartphone in the inventory?
5. What is the median price of a smartphone?
6. What is the most commonly occurring price of a smartphone?

You can answer all these questions in a few seconds by creating NumPy arrays and by applying the `sum(), mean(), median(), min()` and `max()` functions.

Let's first create a NumPy array for the phone data given above, then find the answers to these questions one-by-one.

```
# Create two NumPy arrays: one for the smartphone prices and another for the number of
import numpy as np
arr_p=np.array([13999,6298,10999,14999,7298,6385,8999,9999,9999,13868])
arr_u=np.array([9,8,9,9,8,8,9,6,5,7])
```

```
type(arr_p)
```

```
arr_p=np.array(arr_p)
```

Now, let's answer the first question. To find the total monetary value of the inventory, you have to multiply each smartphone price with its corresponding number of units available and then add all the products of the multiplications.

Let the total monetary value be M , price of a smartphone be p , the number of units available be u and the varieties of smartphones be n . Then $M = p_1 \times u_1 + p_2 \times u_2 + p_3 \times u_3 + \cdots + p_n \times u_n$

Therefore, we have to multiply the `prices` array values with the `units_available` array values to get a new array containing total prices for each smartphone. Then using the `sum()` function, we will add all the values of the new array.

```
# Compute the total monetary value of the inventory.

# Create an array containing total price for each smartphone.
arr_m=arr_p*arr_u
print(arr_m)
# Calculate the total monetary value.
value=np.sum(arr_m)
print(value)
```

```
[125991  50384  98991 134991  58384  51080  80991  59994  49995  97076]
807877
```

Note: We cannot add, subtract, multiply and divide two Python lists like NumPy arrays.

Now, using the `mean()` function, compute the average price of a smartphone.

```
#  Compute the average price of a smartphone.
np.mean(arr_p)
```

Now, using the `min()` function, compute the price of the cheapest smartphone.

```
#  Using the 'min()' function, compute the lowest price of a smartphone.
np.min(arr_p)
```

```
np.int64(6298)
```

Now, using the `max()` function, compute the price of the expensive smartphone.

```
#  Using the 'max()' function, compute the highest price of a smartphone.
np.max(arr_p)
```

Now, using the `median()` function, compute the median price of a smartphone.

```
#  Using the 'median()' function, compute the median price of a smartphone.
np.median(arr_p)
```

Now, let's compute the most commonly occurring price of a smartphone. If you look at the dataset, the most commonly occurring price is `9999` because it occurs twice. Rest of the prices occur only once.

The value which occurs the most number of times is called the **modal** value or simply **mode**.

Unfortunately, the `numpy` module does not have a function to calculate the modal value. So, either we can create our own function which is a very complicated process or we can use the `mode()` function from the `scipy` library.

For the time being we will choose the second option. At the end of the class we will create our own version of the `mode()` function.

In the `scipy` library, there is a module called `stats` which contains the `mode()` function. So we have to import the `stats` module from the `scipy` library.

```
#  Compute the modal value using the 'mode()' function from the 'scipy.stats' module.
from scipy import stats
stats.mode(arr_u)
```

In the output, you can see that `9999` is the modal value and it occurs twice in the `prices` array.

Note: `from library_name import module_name` is another way of importing a module. It is also a standard practice.

Activity 4: Few More Operations On A NumPy Array^^

Performing mathematical operations on a NumPy array is easier compared to a Python list.

Let's say you have a NumPy array with radii of 20 circles and want to compute the area of every circle. Then you can simply use the double-asterisk (`**`) operator on the NumPy array to square the values. Then multiply the NumPy array with `pi`.

Note: Area of a circle with the radius r is πr^2.

```
# Square the values in a numpy array.

# 1. First create a Python list having radii of 20 circles where each radii is a random
import random

list_radii=[random.randint(1,10) for i in range(1,21)]
#print(list_radii)


# 2. Convert the list into a NumPy array using the 'array()' function.
arr_radii=np.array(list_radii)
print(arr_radii)
# 3. Square the elements of NumPy array using the exponent (**) operator. Use can use t
area=np.pi*arr_radii**2 *arr_radii*list_radii
print(area)
```

```
[ 1  2  6  2 10  2  6  9  4  2  5  7  1  1  9 10  8  5  3 10]
[3.14159265e+00 5.02654825e+01 4.07150408e+03 5.02654825e+01
 3.14159265e+04 5.02654825e+01 4.07150408e+03 2.06119894e+04
 8.04247719e+02 5.02654825e+01 1.96349541e+03 7.54296396e+03
 3.14159265e+00 3.14159265e+00 2.06119894e+04 3.14159265e+04
 1.28679635e+04 1.96349541e+03 2.54469005e+02 3.14159265e+04]
```

```
print(list_radii)
```

```
[1, 2, 6, 2, 10, 2, 6, 9, 4, 2, 5, 7, 1, 1, 9, 10, 8, 5, 3, 10]
```

```
Start coding or generate with AI.
```

```
import numpy as np
```

```
np.pi
```

```
3.141592653589793
```

```
np.pi*list_radii
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-3199672004.py in <cell line: 0>()
----> 1 np.pi*list_radii

TypeError: can't multiply sequence by non-int of type 'float'
```

```
2*list_radii
```

```
[1,
 2,
 6,
 2,
 10,
 2,
 6,
 9,
 4,
 2,
 5,
 7,
 1,
 1,
 9,
 10,
 8,
 5,
 3,
 10,
 1,
 2,
 6,
 2,
 10,
 2,
 6,
 9,
 4,
 2,
 5,
 7,
 1,
 1,
 9,
 10,
 8,
 5,
 3,
 10]
```

```
a=np.array(list_radii)
```

```
a**20*np.pi
```

```
array([ 3.14159265e+00,  3.29419866e+06,  1.14861605e+16,  3.29419866e+06,
        2.43984870e+19,  3.29419866e+06,  1.14861605e+16, -1.97577232e+19,
        3.45421765e+12,  3.29419866e+06,  2.99605623e+14,  2.50674798e+17,
        3.14159265e+00,  3.14159265e+00, -1.97577232e+19,  2.43984870e+19,
        3.62200973e+18,  2.99605623e+14,  1.09540563e+10,  2.43984870e+19])
```

```
list_radii*list_radii
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-1098251414.py in <cell line: 0>()
----> 1 list_radii*list_radii

TypeError: can't multiply sequence by non-int of type 'list'
```

Notice that when you print the values of a NumPy array (in this case `np_radii`) using the `print()` function, the items of the NumPy array are not separated by comma in the output. For all practical purposes, this is just a different behaviour of a NumPy array. Do not worry about it.

If you try to square the radii values stored in a Python list using the same process, then Python will throw the `TypeError` error.

```
#  Directly apply the exponent (**) operator on a Python list.
list_radii**2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-3873202343.py in <cell line: 0>()
      1 #  Directly apply the exponent (**) operator on a Python list.
----> 2 list_radii**2

TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

```
list_radii+list_radii
```

```
[1,
 2,
 6,
 2,
 10,
 2,
 6,
 9,
 4,
 2,
 5,
 7,
 1,
 1,
 9,
 10,
 8,
 5,
 3,
 10,
 1,
 2,
 6,
 2,
 10,
 2,
 6,
 9,
 4,
 2,
 5,
 7,
 1,
 1,
 9,
 10,
 8,
 5,
 3,
 10]
```

Even if you simply multiply a list containing numeric values with a floating-point number, then also Python will throw the `TypeError` error

```
#  Directly multiply a Python list with a float number.
```

```
#  Directly multiply a Python list with a int number.
```

To find the area of the circles whose radii are stored in a Python list, you will have to use a loop.

```
# Square all the items in a Python list.
for i in list_radii:
  a=np.pi*i**2
  print(a)
```

Now, using the same approach, you create two NumPy arrays: one having radii (numbers from `1` to `10`) of `10` cylinders and another having their corresponding heights (numbers from `11` to `20`).

Note: The volume of a cylinder is $\pi r^2 h$, where h is height of the cylinder and r is the radius of the cylinder.

```
#  Create two NumPy arrays. One having a radii of 10 cylinders and another having thei
# Compute the volume of the 10 cylinders by multiplying the NumPy arrays and store the
arr_r=np.arange(1,11)
arr_h=np.arange(11,21)
print(arr_r)
print(arr_h)
```

```
vol=np.pi*arr_r**2*arr_h
print(vol)
```

So, here we got an array containing the volumes of the corresponding cylinders.

Activity 5: Python List And NumPy Array Performance Comparison^

As we discussed earlier, the execution time for a NumPy array is lesser as compared to a Python list. The difference is most significant when the sizes of lists and arrays are in thousands and above.

Let's first create a Python list and a NumPy array both having 100 thousand (or 1 lakh) items. Then let's compute how much time (in seconds) is taken to create the list and the array.

```
# Run the code shown below to see that NumPy arrays are faster than Python lists.
# 1. Import the 'numpy' and 'time' modules
import numpy as np
import time

# 2. Record the time before creating a Python list.
py_t0 = time.time()

# 3. Create a Python list containing 100,000 items from 1 to 100,000.
py_list = [i for i in range(1, 10000001)]

# 4. Record the time after creating the list.
py_t1 = time.time()

# 5. Calculate the time taken to create a Python list by computing the time difference
py_time_diff = (py_t1 - py_t0)
print("Time taken to create a Python list is", py_time_diff, "seconds.")
```

```
Time taken to create a Python list is 0.3925609588623047 seconds.
```

```
# 6. Record the time before creating a NumPy array.
np_t0 = time.time()

# 7. Create a NumPy array containing just one row and 100,000 items from 1 to 100,000.
np_ones = np.arange(1, 10000001)
```

```
# 8. Record the time after creating the NumPy array.
np_t1 = time.time()

# 9. Calculate the time taken to create a NumPy array by computing the time difference
np_time_diff = (np_t1 - np_t0)
print("Time taken to create a NumPy array is", np_time_diff, "seconds.")
```

```
Time taken to create a NumPy array is 0.05287766456604004 seconds.
```

```
# 10. Calculate the factor by which a NumPy array creation is faster than a Python lis
print("A NumPy array is", py_time_diff // np_time_diff, "times faster than a Python li
# Recall that // means floor division.
```

```
A NumPy array is 7.0 times faster than a Python list for the same size.
```

```
py_time_diff // np_time_diff
```

```
py_time_diff / np_time_diff
```

```
round((py_time_diff / np_time_diff),0)
```

```
py_time_diff // np_time_diff
```

If you run the above code several times, you will see that almost always NumPy arrays are faster than Python lists by a huge margin.
