numpy-ufunc

August 22, 2024

[ ]:
```
"Numpy  Universal Functions"
```

[ ]:
```
what is ufunc?
```

ufuncs stands for “universal Functions” and they are Numpy functions that operate on the ‘ndarray’ object

[ ]:
```
Use of ufuncs:
```

ufuncs are used to implement “vectorization” in NumPy which is way faster then iterating over elements

They also provide broadcasting and additional methodslike reduce,accumulate etc, that are very helpful for computation.

Vectorisation:process of performing opeartions on entire arrays without using any loops in python.

this is one of the key feature that makes NumPy fast and effiecient.

[ ]:
```
key Points about vectorization:

1.Avoid explicit loops:instead of using for loops for iterate entire array. we␣
 ↪can prform opearations directly on the array.
```

[24]:
```
import numpy as np

# Without vectorization (using a loop)
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
c = np.zeros(4)
for i in range(len(a)):
    c[i] = a[i] + b[i]

print(c)
# With vectorization
c = a + b

print(c)
```

```
[ 6.  8. 10. 12.]
[ 6  8 10 12]
```

In the vectorized version, the addition is done on the entire array at once, which is not only more concise but also faster.

2.leverages Low-Level-Optimisations: Numpy is implemented in C,when you we use vectorised operations,you're essentially calling optimised C code .This results in significant speedups compared to executing Python loops.

3.Memory Efficiency:Vectorised opearations often reduce the memory overhead because they avoid the creation of temporary variables and additional memory allocations that might be needed when looping in Python

4.Broadcasting:NumPy's broadcasting allows us to perform operations on arrays of different shapes in vectorised manner.

[32]:
```
a=np.array([1,2,3,4])
b=np.array([[1],[2],[3],[4]])
c=a+b
print(c)
```

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

Here, a is a 1D array, and b is a 2D array, but they can be added together through broadcasting.

[ ]:
```
speedup:
```

[71]:
```
import numpy as np
import time

#create a large arrays
size=100000
a=np.random.rand(size)
b=np.random.rand(size)

#loop-based addition

start_time=time.time()
result=np.zeros(size)
for i in range(size):
    result[i]=a[i]+b[i]
print(f"Loop took {time.time() - start_time} seconds")

#vectorised addition
start_time=time.time()
result=a+b
```

```
print(f"vectorised opearation took {time.time() - start_time} seconds")
```

```
Loop took 0.06978631019592285 seconds
vectorised opearation took 0.008102178573608398 seconds
```

[ ]:

ufuncs also takes additional arguments ,like:

'where' - boolean array or condition defining where the opearations should take place

'dtype' defining the return type of elements

'out'- output array where the return value should be copied

[ ]:
```
What is Vectorization?


Converting iterative statements into a vector based operation is called␣
 ↪vectorization.

It is faster as modern CPUs are optimized for such operations.
```

[ ]:
```
Add elements of two lists:
Without ufunc, we can use Python's built-in zip() method:
```

[74]:
```
x=[1,2,3,4]
y=[3,4,5,6]
z=[]

for i ,j in zip(x,y):
    z.append(i+j)
print(z)
```

```
[4, 6, 8, 10]
```

[ ]:
```
NumPy has ufunc for ythis ,called 'add(x,y)' that will produce the same result
```

[76]:
```
import numpy as np
x=[1,2,3,4]
y=[2,3,4,5]
z=np.add(x,y)
print(z)
```

```
[3 5 7 9]
```

[ ]:
```
"create our own ufunc"
```

[ ]:
```
How to create our own ufunc?
```

to craete our own ufunc ,we have to define a function ,like you do with normal functions in python ,then add it to your NumPy ufunc library with the 'frompyfunc()' method.

[ ]:
```
The frompyfunc() method takes the following arguements:
1.function()-the name of the function
2.inputs-the number of input  arguements(arrays)
3.outputs-the number of output arrays.
```

[82]:
```
import numpy as np
def myadd(x,y):
    return x+y
myadd=np.frompyfunc(myadd,2,1) # here myadd is function and 2 is number of␣
 ↪inputs and 1 is number of outputs
print(myadd([1,2,4,5],[4,5,6,7]))
```

```
[5 7 10 12]
```

[86]:
```
def myadd(x,y,z):
    return x+y+z
myadd=np.frompyfunc(myadd,3,1)
print(myadd([1,2,3,5],[2,4,5,6],[4,5,6,6]))
```

```
[7 11 14 17]
```

[ ]:
```
"check if a function is ufunc"
```

check the type of function to check if it is a ufunc or not

S ufunc should return <class 'numpy.ufunc'>.

[89]:
```
print(type(myadd))
```

```
<class 'numpy.ufunc'>
```

If it is not a ufunc, it will return another type, like this built-in NumPy function for joining two or more arrays:

Example

[92]:
```
print(type(np.concatenate))
```

```
<class 'numpy._ArrayFunctionDispatcher'>
```

If the function is not recognized at all, it will return an error:

[97]:
```
import numpy as np

print(type(np.vasuhblah))
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
```

```
Cell In[97], line 3
      1 import numpy as np
----> 3 print(type(np.vasuhblah))

File ~\anaconda3\Lib\site-packages\numpy\__init__.py:333, in __getattr__(attr)
    330     "Removed in NumPy 1.25.0"
    331     raise RuntimeError("Tester was removed in NumPy 1.25.")
--> 333 raise AttributeError("module {!r} has no attribute "
    334                      "{!r}".format(__name__, attr))

AttributeError: module 'numpy' has no attribute 'vasuhblah'
```

To test if function is a ufunc is in an if statement ,use the 'numpy.ufunc' value(np.ufunc if you use np as an alias for numpy)

[103]:
```
#using of statement

if type(myadd)==np.ufunc:
    print("yes it is")
else:
    print("It is not")
```

```
yes it is
```

[ ]:
```
"Simple Arithemetic"
```

You could use arithmetic operators + - * / directly between NumPy arrays, but this section discusses an extension of the same where we have functions that can take any array-like objects e.g. lists, tuples etc. and perform 'arithmetic conditionally.'

Arithematic Conditionally:means we can define conditions where the arithematic opearation should happen.

All of the discussed arithmetic functions take a 'where' parameter in which we can specify that condition.

[ ]:
```
"Addition"
```

the 'add()' function sums the content of two arrays and return the results in a new array

[110]:
```
arr1=np.array([1,2,3,5,6,6])
arr2=np.array([3,4,5,6,7,8])
newarr=np.add(arr1,arr2)
print(newarr)
```

```
[ 4  6  8 11 13 14]
```

[ ]:
```
"Subtraction"
```

The 'subtract()' function subtracts the values from one array with the values from another array, and return the results in a new array.

[113]:
```
arr1=np.array([1,2,3,5,6,6])
arr2=np.array([3,4,5,6,7,8])
newarr=np.subtract(arr1,arr2)
print(newarr)
```

```
[-2 -2 -2 -1 -1 -2]
```

[ ]:
```
'Multiplication'
```

The multiply() function multiplies the values from one array with the values from another array, and return the results in a new array.

[116]:
```
arr1=np.array([1,2,3,5,6,6])
arr2=np.array([3,4,5,6,7,8])
newarr=np.multiply(arr1,arr2)
print(newarr)
```

```
[ 3  8 15 30 42 48]
```

[ ]:
```
'Division'
```

The divide() function divides the values from one array with the values from another array, and return the results in a new array.

[119]:
```
arr1=np.array([1,2,3,5,6,6])
arr2=np.array([3,4,5,6,7,8])
newarr=np.divide(arr1,arr2)
print(newarr)
```

```
[0.33333333 0.5        0.6        0.83333333 0.85714286 0.75      ]
```

[121]:
```
arr1=np.array([1,2,3,5,6,6])
arr2=np.array([[3],[4],[5],[6],[7],[8]])
newarr=np.divide(arr1,arr2)
print(newarr)

#we can do any shape or any dimention
```

```
[[0.33333333 0.66666667 1.         1.66666667 2.         2.        ]
 [0.25       0.5        0.75       1.25       1.5        1.5       ]
 [0.2        0.4        0.6        1.         1.2        1.2       ]
 [0.16666667 0.33333333 0.5        0.83333333 1.         1.        ]
 [0.14285714 0.28571429 0.42857143 0.71428571 0.85714286 0.85714286]
 [0.125      0.25       0.375      0.625      0.75       0.75      ]]
```

[ ]:
```
'power'
```

The power() function rises the values from the first array to the power of the values of the second array, and return the results in a new array.

[124]:
```
arr1=np.array([1,2,3,5,6,6])
arr2=np.array([3,4,5,6,7,8])
newarr=np.power(arr1,arr2)
print(newarr) # it will takes first array as elements= and second array as␣
 ↪power of that element
```

```
[      1      16     243   15625  279936 1679616]
```

[ ]:
```
"Remainder"
```

Both the mod() and the remainder() functions return the remainder of the values in the first array corresponding to the values in the second array, and return the results in a new array.

[129]:
```
import numpy as np

arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 7, 9, 8, 2, 33])

newarr = np.mod(arr1, arr2)

print(newarr)

#here first array as numerator and second as denomenator and gets remainder only
```

```
[ 1  6  3  0  0 27]
```

[131]:
```
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 7, 9, 8, 2, 33])

newarr = np.remainder(arr1, arr2)

print(newarr)
```

```
[ 1  6  3  0  0 27]
```

[ ]:
```
'Quotient and Mod'
```

The divmod() function return both the quotient and the mod. The return value is two arrays, the first array contains the quotient and second array contains the mod.

[134]:
```
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 7, 9, 8, 2, 33])

newarr = np.divmod(arr1, arr2)
```

```
print(newarr) #here we get first array as quotients amd second array as␣
  ↪remainders
```

```
(array([ 3,  2,  3,  5, 25,  1]), array([ 1,  6,  3,  0,  0, 27]))
```

[ ]:
```
'Absolute Values'
```

Both the absolute() and the abs() functions do the same absolute operation element-wise but we should use absolute() to avoid confusion with python's inbuilt math.abs()

[141]:
```
arr=np.array([-2,-3,3,4,5,-3])
newarr=np.absolute(arr)
print(newarr)
```

```
[2 3 3 4 5 3]
```

[ ]:
```
"Rounding Decimals"
```

[ ]:
```
there are primarly five ways of rounding off decimals in Numpy:

1.truncation
2.fix
3.rounding
4.floor
5.ceil
```

[ ]:
```
'truncation'
```

Remove the decimals, and return the float number closest to zero. Use the trunc() and fix() functions.

[144]:
```
arr=np.trunc([-23.4234,3.2324])
print(arr)
```

```
[-23.   3.]
```

[146]:
```
arr=np.fix([-234.442435,5.35353])
print(arr)
```

```
[-234.    5.]
```

[ ]:
```
'rounding'
```

The around() function increments preceding digit or decimal by 1 if >=5 else do nothing.

E.g. round off to 1 decimal point, 3.16666 is 3.2

[154]:
```
arr=np.around(3.32463,3)#Round off 3.1666 to 2 decimal places:
print(arr)
```

```
3.325
```

[ ]:
```
'floor'
```

The floor() function rounds off decimal to nearest lower integer.

E.g. floor of 3.166 is 3.

[157]:
```
arr=np.floor([23.3424,4242.424])
print(arr)
```

```
[  23. 4242.]
```

[ ]:
```
'Ceil'
```

The ceil() function rounds off decimal to nearest upper integer.

E.g. ceil of 3.166 is 4.

Example

[160]:
```
arr=np.ceil([132.4424,35.35])
print(arr)
```

```
[133.  36.]
```

[ ]:
```
"NumPy Logs"
```

numPy provides a functions to perform log at the base 2, e and 10.

we will also explore how we can take log for any base by creating a custom ufunc.

all the log functions will place -inf or inf in the elements if the log can not be computed.

[ ]:
```
"Log at Base 2"
```

use the log2() function to perform log at the base 2.

[164]:
```
arr=np.arange(1,10)
print(np.log2(arr))
```

```
[0.         1.         1.5849625  2.         2.32192809 2.5849625
 2.80735492 3.         3.169925  ]
```

Note: The arange(1, 10) function returns an array with integers starting from 1 (included) to 10 (not included).

[ ]:
```
"Log at Base 10"
```

Use the log10() function to perform log at the base 10.

[170]:
```
arr=np.arange(1,10)
print(np.log10(arr))
```

```
[0.         0.30103    0.47712125 0.60205999 0.69897    0.77815125
 0.84509804 0.90308999 0.95424251]
```

[ ]:
```
"Natural Log, or Log at Base e"
```

Use the log() function to perform log at the base e.

[173]:
```
arr=np.arange(1,20)
print(np.log(arr))
```

```
[0.         0.69314718 1.09861229 1.38629436 1.60943791 1.79175947
 1.94591015 2.07944154 2.19722458 2.30258509 2.39789527 2.48490665
 2.56494936 2.63905733 2.7080502  2.77258872 2.83321334 2.89037176
 2.94443898]
```

[ ]:
```
"Log at Any Base"
```

NumPy does not provide any function to take log at any base ,so we use the frompyfunc() function along with inbuilt function math.log() with two input parameters and one output parameters

[212]:
```
from math import log
import numpy as np
nplog=np.frompyfunc(log,2,1) # log means opeartoin to perform and2 means two␣
 ↪inputs and 1 means output
print(nplog(100,15)) #here 100 with base 15
```

```
1.7005483074552052
```

[218]:
```
nplog=np.frompyfunc(log,2,1) # log means opeartoin to perform and2 means two␣
 ↪inputs and 1 means output
print(nplog(10,2))
```

```
3.3219280948873626
```

[ ]:
```
"NumPy Summations"
```

the main difference between summation and addition is

adition will perform with in two arguements whereas summation happens over n elements.

[223]:
```
import numpy as np

arr1=np.array([1,2,3,4,5,6,6])
arr2=np.array([23,4,5,6,7,88,2])
newarr=np.add(arr1,arr2)
print(newarr)
```

```
[24  6  8 10 12 94  8]
```

[ ]:
```
'sum'
```

[229]:
```
import numpy as np

arr1=np.array([1,2,3,4,5,6,6])
arr2=np.array([23,4,5,6,7,88,2])
newarr=np.sum([arr1,arr2])
print(newarr)
```

```
162
```

[ ]:
```
"Summation Over an Axis"
```

if you specify 'axis=1', NumPy will sum the numbers in each in each array.

In NumPy, the concept of an "axis" is used to specify the direction along which operations are performed on arrays. NumPy arrays can have multiple dimensions, and the axis parameter helps define which dimension you want to apply a function or operation to.

Here's how the axis works:

Axis 0: Refers to the first dimension (rows in a 2D array). Operations along axis 0 are applied column-wise. Axis 1: Refers to the second dimension (columns in a 2D array). Operations along axis 1 are applied row-wise. Axis 2, 3, ...: For higher-dimensional arrays, these refer to the subsequent dimensions.

[242]:
```
arr1=np.array([12,2,3])
arr2=np.array([2,3,4])
newarr=np.sum([arr1,arr2],axis=1) #here axis=0 means rows thorugh rpws of␣
 ↪values should be sum
print(newarr)
```

```
[17  9]
```

[ ]:
```
"Cummulative Sum"
```

cummulative sum means partially adding the elememts in an array

E.g. The partial sum of [1, 2, 3, 4] would be [1, 1+2, 1+2+3, 1+2+3+4] = [1, 3, 6, 10].

perform partial sum with cumsum() function

[247]:
```
arr=np.array([1,2,3,4])
print(np.cumsum(arr))
```

```
[ 1  3  6 10]
```

[ ]:
```
"NumPy products"
```

to product of the elements in an array ,use the prod() function.

[250]:
```
arr=np.array([2,3,4,56,6])#2*3*4*56*6
print(np.prod(arr))
```

```
8064
```

[252]:
```
#with two array

arr1=np.array([1,2,3,4]) #1*2*3*4*2*4*6*8
arr2=np.array([2,4,6,8])
newarr=np.prod([arr1,arr2])
print(newarr)
```

```
9216
```

[ ]:
```
"Product over an Axis"
```

if you specify axis=1 ,NumPy return the product o each array.

[255]:
```
arr1=np.array([1,2,3,4]) #1*2,2*4,3*6,4*8
arr2=np.array([2,4,6,8])
newarr=np.prod([arr1,arr2],axis=0) #irt will product cloumns
print(newarr)
```

```
[ 2  8 18 32]
```

[263]:
```
arr1=np.array([1,2,3,4]) #1*2*3*4,2*4*6*8
arr2=np.array([2,4,6,8])
#rows thorugh product
newarr=np.prod([arr1,arr2],axis=1)
print(newarr)
```

```
[ 24 384 384]
```

[267]:
```
arr1=np.array([1,2,3,4]) #1*2*3*4*2*4*6*8
arr2=np.array([2,4,6,8])
newarr=np.prod([arr1,arr2],axis=1)
print(newarr)
```

```
[ 24 384]
```

[ ]:
```
"cummulative product"
```

Cummulative Product Cummulative product means taking the product partially.

E.g. The partial product of [1, 2, 3, 4] is [1, 1*2*, *123*, *1234*] = [1, 2, 6, 24]

Perfom partial sum with the cumprod() function.

[270]:
```
arr=np.array([1,2,4,5])
print(np.cumprod(arr))
```

```
[ 1  2  8 40]
```

[ ]:
```
"NumPy Differences"
```

A discrete difference means substracting two successive elements.

```
 A discrete difference means subtracting two successive elements.
```

E.g. for [1, 2, 3, 4], the discrete difference would be [2-1, 3-2, 4-3] = [1, 1, 1]

To find the discrete difference, use the diff() function.

[281]:
```
arr1=np.array([4,2,3,4])#2-4,3-2,4-3
newarr=np.diff(arr1)
print(newarr)
```

```
[-2  1  1]
```

We can perform this operation repeatedly by giving parameter n.

E.g. for [1, 2, 3, 4], the discrete difference with n = 2 would be [2-1, 3-2, 4-3] = [1, 1, 1] , then, since n=2, we will do it once more, with the new result: [1-1, 1-1] = [0, 0]

[286]:
```
arr=np.array([2,3,4,7])#(1,1,3)->(0,2)
newarr=np.diff(arr,n=2) #it will perform 3 times substrct
print(newarr)
```

```
[0 2]
```

[ ]:
```
"Finding LCM (Lowest Common Multiple)"
```

the lowest common Multiple is the smallest number that is a common multiple of two numbers.

[3]:
```
import numpy as np
# Find the LCM of of the following two numbers:
num1=4
num2=8
x=np.lcm(num1,num2)
print(x)
```

```
8
```

[ ]:
```
"Finding LCM Arrays"
```

to Find the lowest common multiple of all values in an array,you can use the reduce() method.

[ ]:
```
The reduce() function is a ufunc,in this case the lcm() function,on each␣
 ↪element and reduce the array by one dimention.
```

[8]:
```
arr=np.array([3,6,9,12])
x=np.lcm.reduce(arr)
print(x)
```

36

[14]:
```
# Find the LCM of all values of an array where the array contains all integers␣
↪from 1 to 10:

arr=np.arange(1,11)
x=np.lcm.reduce(arr)
print(x)
```

2520

[ ]:
```
"NumPy GCD Greatest Common Denominator"
```

the greatest common denominator is also known as highest common factor(HCF) is the biggest number that is common factor of both of the numbers.

[17]:
```
import numpy as np

n1=12
n2=16
x=np.gcd(n1,n2)
print(x)
```

4

[ ]:
```
"Finding GCD in Arrays"
```

To find the Highest Common Factor of all values in an array, you can use the reduce() method.

The reduce() method will use the ufunc, in this case the gcd() function, on each element, and reduce the array by one dimension.

[20]:
```
arr=np.array([2,4,6,8,10])
x=np.gcd.reduce(arr)
print(x)
```

2

[ ]:
```
"NumPy Trignometric Functions"
```

NumPy provides the ufunc sin().cos(),tan() that takes values in radians and produce the corresponding sin,cos,tan values

[24]:
```
# Find the sine value of PI/2
x=np.sin(np.pi/2)
```

```
print(x)
```

```
1.0
```

[26]:
```
# Find sine values for all of the values in arr:
arr=np.array([np.pi/2,np.pi/3,np.pi/4])
x=np.sin(arr)
print(x)
```

```
[1.         0.8660254  0.70710678]
```

[ ]:
```
"Convert Degrees Into Radians"
```

By default all trignometric functions takes radians as parametrs but we can convert radains to degrees and vice versa as well in numpy.

Note: radians values are pi/180 * degree_values.

[32]:
```
arr=np.array([90,180,270,360])
x=np.deg2rad(arr)
print(x)
```

```
[1.57079633 3.14159265 4.71238898 6.28318531]
```

[ ]:
```
"Radians to Degrees"
```

[36]:
```
arr=np.array([np.pi/2,np.pi/3,np.pi/4])
x=np.rad2deg(arr)
print(x)
```

```
[90. 60. 45.]
```

[ ]:
```
"Finding Angles"
```

Finding angles from values of sine ,cos,tan E.g sin,cos,tan inverse (arcsin,arccos,arctan)

NumPy provides ufunc arcsin(),arccos(),arctan() that produce radian values for corresponding sin,cos and tan values given.

[39]:
```
# Find the angle of 1.0:
x=np.arcsin(1.0)
print(x)
```

```
1.5707963267948966
```

[ ]:
```
"Angles of Each Value in Arrays"
```

[41]:
```
arr=np.array([-1,1,0.1])
x=np.arcsin(arr)
```

```
print(x)
```

```
[-1.57079633  1.57079633  0.10016742]
```

[ ]:
```
"Hypotenues"
```

finding hypotenues using Pythagoras theorem in NumPy.

NumPu provides that hypot() function that takes the base and perpendicular values and produce hypotenues based on pythagoras theorem

a$^{2=b}$2+c^2 pythogoras theorem

[44]:
```
# Find the hypotenues for 4 base and 3 perpendicular:
base=3
perp=4
x=np.hypot(base,perp)
print(x)
```

```
5.0
```

[ ]:
```
"NumPy hyperbolic Functions"
```

NumPy provides the ufunc sinh(),cosh() and tanh() that takes values in radians and produce the corresponding sinh,cosh,tanh values.

[47]:
```
import numpy as np

x = np.sinh(np.pi/2)

print(x)
```

```
2.3012989023072947
```

[49]:
```
import numpy as np

arr = np.array([np.pi/2, np.pi/3, np.pi/4, np.pi/5])

x = np.cosh(arr)

print(x)
```

```
[2.50917848 1.60028686 1.32460909 1.20397209]
```

“Finding Angles”

Finding angles from values of hyperbolic sine, cos, tan. E.g. sinh, cosh and tanh inverse (arcsinh, arccosh, arctanh).

Numpy provides ufuncs arcsinh(), arccosh() and arctanh() that produce radian values for corresponding sinh, cosh and tanh values given.

[52]:
```
import numpy as np

x = np.arcsinh(1.0)

print(x)
```

```
0.881373587019543
```

[ ]:
```
"Angles of Each Value in Arrays
Example"
```

[54]:
```
import numpy as np

arr = np.array([0.1, 0.2, 0.5])

x = np.arctanh(arr)

print(x)
```

```
[0.10033535 0.20273255 0.54930614]
```

[ ]:
```

```

[ ]:
```
"NumPy Set Operations"
```

Set: –> A set in mathematics is collection of unique elements.

`sets are used for operations invloving frequent intersection,union and difference opeartions.`

[ ]:
```
"Create Sets in NumPy"
```

we can use NumPy's unique() method to find the unique elements from any array,E.G create a set array but remember that the set arrays should be 1-D arrays

[60]:
```
arr=np.array([1,2,3,4,5,6,7,1,3,4,4,6])
x=np.unique(arr)
print(x)
```

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

[ ]:
```
"Finding Union"
```

To find the unique values of two arrays, use the union1d() method.

[65]:
```
arr1=np.array([2,4,3,2,4])
arr2=np.array([2,3,4,5,2,6,7])
x=np.union1d(arr1,arr2)
print(x) #merge unique from both tables
```

```
[2 3 4 5 6 7]
```

[ ]:
```
"Finding Intersection"
```

To find only the values that are present in both arrays, use the intersect1d() method.

[68]:
```
arr1=np.array([2,4,3,2,4])
arr2=np.array([2,3,4,5,2,6,7])
x=np.intersect1d(arr1,arr2)
print(x) #common
```

```
[2 3 4]
```

Note: the intersect1d() method takes an optional argument assume_unique, which if set to True can speed up computation. It should always be set to True when dealing with sets.

[ ]:
```
"Finding Difference"
```

To find only the values in the first set that is NOT present in the seconds set, use the setdiff1d() method.

[79]:
```
arr1=np.array([2,4,3,2,4])
arr2=np.array([2,3,4,5,2,6,7])
x=np.setdiff1d(arr2,arr1,assume_unique=True)
print(x)
```

```
[5 6 7]
```

[ ]:
```
"Finding Symmetric Difference"
```

To find only the values that are NOT present in BOTH sets, use the setxor1d() method.

[82]:
```
set1 = np.array([1, 2, 3, 4])
set2 = np.array([3, 4, 5, 6])

newarr = np.setxor1d(set1, set2, assume_unique=True)

print(newarr)
```

```
[1 2 5 6]
```

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]: