WEEK 3

Explore Numpy module

Array Aggregation Functions

In the Python numpy module, we have many aggregate functions to work with a single-dimensional or multi-dimensional array. The Python numpy aggregate functions are sum, min, max, mean, average, product, median, standard deviation, variance to name few

First, we have to import Numpy as **import numpy as np.** To make a **Numpy** array, you can just use the **np.array()** function. The aggregate functions are given below

1. **np.sum(m)**: Used to find out the **sum** of the given array.
2. **np.prod(m)**: Used to find out the **product(multiplication)** of the values of m.
3. **np.mean(m)**: It returns the **mean** of the input array m.
4. **np.std(m)**: It returns the **standard deviation** of the given input array m.
5. **np.var(m)**: Used to find out the **variance** of the data given in the form of array m.
6. **np.min(m)**: It returns the **minimum value** among the elements of the given array m.
7. **np.max(m)**: It returns the **maximum value** among the elements of the given array m.
8. **np.argmin(m)**: It returns the **index of the minimum value** among the elements of the array m.
9. **np.argmax(m)**: It returns the **index of the maximum value** among the elements of the array m
10. **np.median(m)**: It returns the **median** of the elements of the array m
11. **np.prod(m):** It returns the **mean** of the input array m.

```
import numpy as np
a=np.array([1,2,3,4,5])
print("Array a :",a)
sum=np.sum(a)
print("sum :",sum)
product=np.prod(a)
print("product :",product)
mean=np.mean(a)
print("mean :",mean)
standard_deviation=np.std(a)
print("standard_deviation :",standard_deviation)
variance=np.var(a)
print("variance :",variance)
minimum=np.min(a)
print("minimum value :",minimum)
maximum=np.max(a)
print("maximum value :",maximum)
minimum_index=np.argmin(a)
print("minimum index :",minimum_index)
maximum_index=np.argmax(a)
print("maximum-index :",maximum_index)
median=np.median(a)
print("median :",median)
prod=np.prod(a)
print("product :",prod)
```

OUTPUT

```
Array a : [1 2 3 4 5]
sum : 15
product : 120
mean : 3.0
standard_deviation : 1.4142135623730951
variance : 2.0
minimum value : 1
maximum value : 5
minimum index : 0
maximum-index : 4
median : 3.0
product : 120
```

Vectorized Operations

Processing such a large amount of data in python can be slow as compared to other languages like C/C++. This is where vectorization comes into play.

Vectorization is a technique of implementing array operations without using for loops. Instead, we use functions defined by various modules which are highly optimized that reduces the running and execution time of code

Vectorized array operations will be faster than their pure Python equivalents

Vectorized sum

```
# importing the modules
import numpy as np
import timeit

# vectorized sum
print(np.sum(np.arange(4)))

print("Time taken by vectorized sum : ",end= "")
%timeit np.sum(np.arange(4))

# iterative sum
total = 0
for item in range(0, 4):
    total += item
a = total
print("\n" + str(a))

print("Time taken by iterative sum : ",end= "")
%timeit a
```

OUTPUT

```
6
Time taken by vectorized sum : 6.63 µs ± 193 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops e
ach)

6
Time taken by iterative sum : 30.8 ns ± 8.87 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loop
s each)
```

Vectorized multiplication

```
# importing the modules
import numpy as np
import timeit

# vectorized sum
np.a=[4,5,1]
```

```
print(np.prod(np.a))

print("Time taken by vectorized product : ",end= "")
%timeit np.prod(np.a)

# iterative sum
total = 1
for item in np.a:
    total =total*item
t = total
print(t)

print("Time taken by iterative multiplication : ",end= "")
%timeit t
```

OUTPUT

```
20
Time taken by vectorized product : 9.85 µs ± 1.36 µs per loop (mean ± std. dev. of 7 runs, 100,000 lo
ops each)
20
Time taken by iterative multiplication : 43.2 ns ± 4.69 ns per loop (mean ± std. dev. of 7 runs, 10,0
00,000 loops each)
```

Use Map, Filter, Reduce and Lambda Functions with NumPy

Map

map() function returns a map object (which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)

Filter

The filter() function returns an iterator were the items are filtered through a function to test if the item is accepted or not

Reduce

The **reduce (fun, seq)** function is used to **apply a particular function passed in its argument to all of the list elements** mentioned in the sequence passed along. This function is defined in "**functools**" module.

Lambda

What is the syntax of a lambda function (or lambda operator)?

lambda arguments: expression

Think of lambdas as one-line methods without a name. They work practically the same as any other method in Python, for example:

```
def add(x,y):
        return x + y
```

Can be translated to:

lambda x, y: x + y

Lambdas differ from normal Python methods because they can have only one expression, can't contain any statements and their return type is a function object

map

The map() function iterates through all items in the given iterable and executes the function we passed as an argument on each of them.

The syntax is:

```
map(function, iterable(s))
```

We can pass as many iterable objects as we want after passing the function we want to use:

```
# Without using lambdas
def test(s):
   return s[0] == "A"

fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]
ob = map(test, fruit)

print(list(ob))
```

output is
[True, False, False, True, False]

Filter

filter() Function takes a function object and an iterable and creates a new list.
The syntax is:

```
filter(function, iterable(s))
```

As the name suggests, filter() forms a new list that contains only elements that satisfy a certain condition, i.e. the function we passed returns True.

```
def starts_B(s):
   return s[0] == "B"

fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]
```

```
fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]
filter_ob = filter(lambda s: s[0] == "B", fruit)

print(list(filter_ob))
```

reduce() Function

reduce() works differently than map() and filter(). It does not return a new list based on the function and iterable we've passed. Instead, it returns a single value.

reduce() isn't a built-in function anymore, and it can be found in the functools module.

The syntax is:

reduce(function, sequence[, initial])

reduce() works by calling the function we passed for the first two items in the sequence. The result returned by the function is used in another call to function alongside with the next (third in this case), element.

This process repeats until we've gone through all the elements in the sequence.

The optional argument initial is used, when present, at the beginning of this "loop" with the first element in the first call to function. In a way, the initial element is the 0th element, before the first one, when provided

We start with a list [2, 4, 7, 3] and pass the add(x, y) function to reduce() alongside this list, without an initial value

reduce() calls add(2, 4), and add() returns 6

reduce() calls add(6, 7) (result of the previous call to add() and the next element in the list as parameters), and add() returns 13

reduce() calls add(13, 3), and add() returns 16

Since no more elements are left in the sequence, reduce() returns 16

```
from functools import reduce

def add(x, y):
    return x + y

list = [2, 4, 7, 3]
print(reduce(add, list))
```

Running this code would yield:

OUTPUT :16

Again, this could be written using lambdas:

```
from functools import reduce

list = [2, 4, 7, 3]
print(reduce(lambda x, y: x + y, list))
```

OUTPUT :16

Example to run all the functions together

```
lst=[4,2,0,5,1,6,3]
from functools import reduce
print(list(map(lambda num: num**2,lst)))
print(list(filter(lambda num:num>2,lst)))
print(reduce(lambda x,y: x+y,lst))
```

OUTPUT

```
[16, 4, 0, 25, 1, 36, 9]
[4, 5, 6, 3]
21
```