Vectorization

Vectorization is one of the core feature of NumPy
Vectorization allows to apply an action to every element of an array with a single line of code, avoiding loops and making your code more readable and efficient.

- Array Operations
- Vectorized functions or ufunc
- Boolean array and indexing

Array Operations

Any arithmetic operations between equal-size arrays applies the operation elementwise

```
A+B      +    element by element addition
A-B      -    element by element subtraction
A*B      *    element by element multiplication
A/B      /    element by element division
A**B     **   element by element power

If is B a scalar it will be broadcasted to all elements of A
```

```
A=np.ones((2,3))
[[1. 1. 1.]
[1. 1. 1.]]

B=np.random.randint(10,size=(2,3))
[[5 4 1]
[4 1 6]]

B*2
[[10 8 2]
[ 8 2 12]]

A+B
[[6. 5. 2.]
[5. 2. 7.]]
```

Vectorized built-in functions

Universal Functions: Fast Element-wise Array Functions
A universal function, or *ufunc*, is a function that performs elementwise operations on data in ndarrays, thus removing the need for flow control loops.

Example of ufuncs are **sqrt, exp, sin, cos, log, log10**.

Available ufunc for math operations can be found here
https://numpy.org/doc/stable/reference/ufuncs.html#math-operations

```
a = np.arange(1,4)
[1 2 3]
```

Calculate the square root of each element in the array a

```
np.sqrt(a)
[1.          1.41421356  1.73205081]
```

```
np.power(a,2)
[1 4 9])
```

Boolean array and indexing

In NumPy you can perform element-wise comparison

```
>
>=
<
<=
==
!=
```

```
logical_and, logical_or
equivalent to operators & |
```

Comparison operators applied to an array, return a Boolean array of the same size.

```
arr1=np.arange(1,11)
 [ 1  2  3  4  5  6  7  8  9 10]

bool1=arr1 >= 6
[False False False False False  True  True  True  True  True]
```

1	2	3	4	5	6	7	8	9	10
False	False	False	False	False	True	True	True	True	True

```
arr1[bool1]  #extract True elements
[ 6 7  8 9 10 ]

arr1[bool1]=arr1[bool1]*10 #replace True elements
[ 1 2 3 4 5 60 70 80 90 100 ]
```

Boolean array and indexing


```
bool2 = np.logical_and( arr1 < 8 , arr1 >= 5 )
[False False False False  True  True  True False False]

arr1[bool2]
[5 6 7]
```

```
bool3 = np.logical_or( arr1 <=3 , arr1 > 9 )

arr1[bool3]
[ 1  2  3 10]
```

How would you count the elements satisfying condition?

Also explore the function np.count_nonzero()
https://numpy.org/doc/stable/reference/generated/numpy.count_nonzero.html

Boolean array and indexing


```
arr2=np.array([[1,2,3,4],[5,6,7,8]])
[[1 2 3 4]
[5 6 7 8]]

bool1=np.logical_or(arr2 <=2, arr2 > 6)
[[ True  True False False]
[False False  True  True]]


arr2[bool1]  #extract True  elements
[1 2 7 8]



arr2[bool1]=arr2[bool1]**2 #replace True elements
[[ 1  4  3  4]
 [ 5  6 49 64]]

arr2[bool1]=0
[[0 0 3 4]
 [5 6 0 0]]
```

Finding maximum and minimum - max, min, argmax, argmin

```
M=np.max(A, axis=n)       return the maximum along a given axis.
m=np.min(A, axis=n)       return the minimum along a given axis.
ind=np.argmax(A, axis=n) return the indices of the maximum values along
an axis.
ind=np.argmix(A, axis=n) return the indices of the minimum values along
an axis.
```

```
M = np.array([[1, 2, 3,4], [8, 6, 0,9], [9, 2, 3,12]])
[[ 1  2  3  4]
[ 8  6  0  9]
[ 9  2  3 12]]
```

```
axis=1   row
axis=0   column
```

```
vm=np.min(M, axis=1)        #returns minimum of each row
[1 0 2]

indm=np.argmin(M, axis=1) #returns the indices of the minimum
[0, 2, 1]                  values of each row
```

In the case of a 1D array, no need to specify the axis

```
v1=np.random.randint(1,50, size=6)
[22, 38, 30, 40, 18,  1]

m=np.min(v1) #1  returns the minimum value
i=np.argvmin(v1) #5  returns the index of the min value
```

Sorting - sort, argsort

S=np.sort(A, axis=n) return a sorted copy of an array. Sort along an axis.
ind=np.argsort(A, axis=n) returns an array of indices of the same shape as *A* that index elements along the given axis in sorted order

```
Ms=np.sort(M, axis=0)  #sort each column
[[ 1  1  0  4]
[ 8  2  3  9]
[ 9  6  3 12]]

Mind=np.argsort(M, axis=0) #Returns indices that would sort M
[[0 2 1 0]
[1 0 0 1]
[2 1 2 2]]


vs=np.sort(v1)  #returns the sorted array
[ 1, 18, 22, 30, 38, 40]

ind=np.argsort(v1) #returns the indices that would sort an
array
[5, 4, 0, 2, 1, 3]
```

some statistical functions – mean, std

```
m=np.mean(A, axis=n)  compute and return the arithmetic mean along
the specified axis.

s=np.std(A, axis=n)   compute and return the standard deviation
along the specified axis.
```

```
M = np.array([[1, 2, 3,4], [8, 6, 0,9], [9, 2, 3,12]])
[[ 1  2  3  4]
[ 8  6  0  9]
[ 9  2  3 12]]

m=np.mean(M, axis=0) #mean of each column
[6.         3.33333333 2.         8.33333333]

s=np.std(M, axis=1) #std of each row
[1.11803399 3.49106001 4.15331193]
```

More statistical functions
https://numpy.org/doc/stable/reference/routines.statistics.html

Some math functions – sum, cumsum

```
s=sum(A, axis=n)    calculate and return the sum of array elements
over a given axis.

c=cumsum(A, axis=n) return the cumulative sum of the elements along a
given axis.
```

```
M = np.array([[1, 2, 3,4], [8, 6,
0,9], [9, 2, 3,12]])
[[ 1  2  3  4]
[ 8  6  0  9]
[ 9  2  3 12]]

s=np.sum(M, axis=0) #sum of each column
[18 10  6 25]

c=np.cumsum(M, axis=1) #cumulative sum of each row
[[ 1  3  6 10]
 [ 8 14 14 23]
 [ 9 11 14 26]]
```

More math functions
https://numpy.org/doc/stable/reference/routines.math.html

Importing data with loadtxt:

```
A=np.loadtxt(filename, delimiter='sep')  store data in a 2D array
```

- loadtxt will work on ASCII files that contain **numbers**
- If the field separator is not a space, you should specify a delimiter character

Example: file is on Canvas DATA

```
dat = np.loadtxt('ph.dat')
```

You can use slicing to store the first column in variable ph, and second column in variable p:

```
ph = dat[:,0]
p = dat[:,1]
```

You can also define the dtype of the resulting array, and skip comment lines
https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html

Exporting array with savetxt

`np.savetxt(filename, X, fmt='format', delimiter='sep')` save an array to a text file.

If you do not specify the fmt and delimiter, it will format as %.18e with a space as delimiter

```
M = np.array([[1, 2, 3,4], [8,6, 0,9], [9, 2, 3,12]])
[[ 1  2  3  4]
[ 8  6  0  9]
[ 9  2  3 12]]
```

```
np.savetxt('file1.txt', M, fmt='%d', delimiter=':') #save in file1.txt, format each number as integer, and set delimiter to colon.
```

```
np.savetxt('file1.csv', M, fmt='%.1f', delimiter=',') #save in file1.csv, format each number as float with 1 decimal precision, and set delimiter to comma.
```

You can also write the header, and comment lines
https://numpy.org/doc/stable/reference/generated/numpy.savetxt.html

Looping over ndarrays using the usual python syntax

Numpy arrays are iterables and so you can loop over its elements with the usual python syntax.

Iterate on the elements of a 1-D array:

```
arr = np.array([1, 2, 3])
for x in arr:
    print(x)
```

```
1
2
3
```

In a 2-D array the standard for loop will go through all the rows.

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

for x in arr:
  print(x)
```

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

To iterate on each scalar element of the 2-D array, see next slide.

Looping over elements of ndarrays – np.nditer()

To iterate through each scalar (element) of a ndarray we need to use n for loops, where n is the dimension of the array.

To avoid nested loops, you can use the **nditer()**
https://numpy.org/doc/stable/reference/arrays.nditer.html

nditer() iterates through each scalar element of a ndarray (one by-one).

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

for x in np.nditer(arr):
  print(x)
```

```
1
2
3
4
5
6
```

Summary of built-in functions useful for data analysis

```
B=np.function(A, axis=n)

axis=0: apply operation column-wise, across all rows for each column.
axis=1: apply operation row-wise, across all columns for each row
```

`M=np.max(A, axis=n)`	maximum along a given axis.
`ind=np.argmax(A, axis=n)`	indices of the maximum values along an axis.
`m=np.min(A, axis=n)`	minimum along a given axis.
`ind=np.argmix(A, axis=n)`	indices of the minimum values along an axis
`S=np.sort(A, axis=n)`	sorted copy of an array. Sort along an axis.
`ind=np.argsort(A, axis=n)`	indices in sorted order
`m=np.mean(A, axis=n)`	mean along the specified axis.
`s=np.std(A, axis=n)`	standard deviation along the specified axis.
`s=sum(A, axis=n)`	sum of elements over a given axis.
`c=cumsum(A, axis=n)`	cumulative sum of elements along a given axis.