Numpy-II Notes

Content

- **Universal Functions (ufunc)**
  - Aggregate Function/ Reduction functions - `sum()`, `mean()`, `min()`, `max()`
  - Logical functions - `any()` , `all()`
  - Sorting function - `sort()`, `argsort()`
- **Use Case: Fitness Data analysis**
  - Loading data set and EDA using numpy
  - `np.unique()`
  - `argmin()`, `argmax()`
- **Operations on Numpy Arrays**
  - Algebric operations on np arrays with single numbers
  - Operations on two or more np arrays
  - Matrix Multiplication - `matmul()`, `@`, `dot()`
- **Vectorization**
  - `np.vectorize()`
- **Broadcasting**
  - `np.tile()` , `np.newaxis`

Aggregate / Universal Functions (`ufunc`)

Numpy universal functions are objects that belongs to `numpy.ufunc` class.

- Some ufuncs are **called automatically when the corresponding "arithmetic operator" is used on arrays**.

For example:

- When **addition of two array** is performed **element-wise** using `+` operator, then **np.add() is called internally.**

```
a = np.array([1,2,3,4])
b = np.array([5,6,7,8])
a+b  # ufunc `np.add()` called automatically
```

```
array([ 6,  8, 10, 12])
```

```
np.add(a,b)
```

```
array([ 6,  8, 10, 12])
```

Aggregate Functions/ Reduction functions

np.sum()

```
a = np.arange(12).reshape(3, 4)
a
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
np.sum(a)  # sums all the values present in array
```

```
66
```

We can do row-wise and column-wise sum by setting `axis` parameter

- **`axis = 0` ---> Changes will happen along the vertical axis**
- Summing of values happen **in the vertical direction**

```
np.sum(a, axis=0)
```

```
array([12, 15, 18, 21])
```

- **`axis = 1` ---> Changes will happen along the horizontal axis**
- Summing of values happen **in the horizontal direction**

```
np.sum(a, axis=1)
```

```
array([ 6, 22, 38])
```

---

np.mean()

```
np.mean(a)
```

```
5.5
```

```
np.mean(a, axis=0)
```

```
array([4., 5., 6., 7.])
```

```
np.mean(a, axis=1)
```

```
array([1.5, 5.5, 9.5])
```

np.min()

```
a
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
np.min(a)
```

```
0
```

```
np.min(a, axis = 1 )
```

```
array([0, 4, 8])
```

np.max()

```
a
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
np.max(a) # maximum value
```

```
11
```

```
np.max(a, axis = 0) # column wise max
```

```
array([ 8,  9, 10, 11])
```

Logical functions

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

```
array([1, 2, 3, 4])
```

np.any()

- `any()` returns `True` if **any of the elements** in the argument array is **non-zero**.

```
np.any([True, True, False])
```

```
True
```

```
a = np.array([1,2,3,4]) # atleast 1 element is non-zero
np.any(a)
```

```
True
```

```
a = np.array([1,0,0,0]) # atleast 1 element is non-zero
np.any(a)
```

```
True
```

```
a = np.zeros(4) # all elements are zero
np.any(a)
```

```
False
```

- `any()` returns `True` if **any of the corresponding elements** in the argument arrays follow the **provided condition**.

```
a = np.array([1,2,3,4])
b = np.array([4,3,2,1])
np.any(a<b) # Atleast 1 element in a < corresponding element in b
```

```
True
```

```
a = np.array([4,5,6,7])
b = np.array([4,3,2,1])
np.any(a<b) # All elements in a >= corresponding elements in b
```

```
False
```

np.all()

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

```
(array([1, 2, 3, 4]), array([4, 3, 2, 1]))
```

```
np.all(a<b) # Not all elements in a < corresponding elements in b
```

```
False
```

```
a = np.array([1,0,0,0])
b = np.array([4,3,2,1])
np.all(a<b) # All elements in a < corresponding elements in b
```

```
True
```

Multiple conditions for `.all()` function

```
a = np.array([1, 2, 3, 2])
b = np.array([2, 2, 3, 2])
c = np.array([6, 4, 4, 5])
((a <= b) & (b <= c)).all()
```

```
True
```

Sorting Arrays

- Default axis for sorting is the last axis of the array.

```
[ ] ↳ 10 cells hidden
```

Use Case: Fitness data analysis

```
!gdown 1kXqcJo4YzmwF1G2BPoA17CI49TZVHANF
```

```
data = np.loadtxt('fitness.txt', dtype='str')
```

```
data[:5]
```

```
array([['06-10-2017', '5464', '200', '181', '5', '0', '66'],
       ['07-10-2017', '6041', '100', '197', '8', '0', '66'],
       ['08-10-2017', '25', '100', '0', '5', '0', '66'],
       ['09-10-2017', '5461', '100', '174', '4', '0', '66'],
       ['10-10-2017', '6915', '200', '223', '5', '500', '66']],
      dtype='<U10')
```

What's the shape of the data?

```
data.shape
```

```
(96, 7)
```

There are 96 records and each datapoint has 7 features. These features are:

- Date
- Step count
- Mood
- Calories Burned
- Hours of sleep
- activity status
- weight

```
data[0]
```

```
array(['06-10-2017', '5464', '200', '181', '5', '0', '66'], dtype='<U10')
```

Whats the way to change columns to rows and rows to columns?

Transpose

```
data.T[0]
```

```
array(['06-10-2017', '07-10-2017', '08-10-2017', '09-10-2017',
       '10-10-2017', '11-10-2017', '12-10-2017', '13-10-2017',
       '14-10-2017', '15-10-2017', '16-10-2017', '17-10-2017',
       '18-10-2017', '19-10-2017', '20-10-2017', '21-10-2017',
       '22-10-2017', '23-10-2017', '24-10-2017', '25-10-2017',
       '26-10-2017', '27-10-2017', '28-10-2017', '29-10-2017',
       '30-10-2017', '31-10-2017', '01-11-2017', '02-11-2017',
       '03-11-2017', '04-11-2017', '05-11-2017', '06-11-2017',
       '07-11-2017', '08-11-2017', '09-11-2017', '10-11-2017',
       '11-11-2017', '12-11-2017', '13-11-2017', '14-11-2017',
       '15-11-2017', '16-11-2017', '17-11-2017', '18-11-2017',
       '19-11-2017', '20-11-2017', '21-11-2017', '22-11-2017',
       '23-11-2017', '24-11-2017', '25-11-2017', '26-11-2017',
       '27-11-2017', '28-11-2017', '29-11-2017', '30-11-2017',
       '01-12-2017', '02-12-2017', '03-12-2017', '04-12-2017',
       '05-12-2017', '06-12-2017', '07-12-2017', '08-12-2017',
       '09-12-2017', '10-12-2017', '11-12-2017', '12-12-2017',
       '13-12-2017', '14-12-2017', '15-12-2017', '16-12-2017',
       '17-12-2017', '18-12-2017', '19-12-2017', '20-12-2017',
       '21-12-2017', '22-12-2017', '23-12-2017', '24-12-2017',
       '25-12-2017', '26-12-2017', '27-12-2017', '28-12-2017',
       '29-12-2017', '30-12-2017', '31-12-2017', '01-01-2018',
       '02-01-2018', '03-01-2018', '04-01-2018', '05-01-2018',
       '06-01-2018', '07-01-2018', '08-01-2018', '09-01-2018'],
      dtype='<U10')
```

```
date, step_count, mood, calories_burned, hours_of_sleep, activity_status, weight = data.T
```

```
step_count
```

```
array(['5464', '6041', '25', '5461', '6915', '4545', '4340', '1230', '61',
       '1258', '3148', '4687', '4732', '3519', '1580', '2822', '181',
       '3158', '4383', '3881', '4037', '202', '292', '330', '2209',
       '4550', '4435', '4779', '1831', '2255', '539', '5464', '6041',
       '4068', '4683', '4033', '6314', '614', '3149', '4005', '4880',
       '4136', '705', '570', '269', '4275', '5999', '4421', '6930',
       '5195', '546', '493', '995', '1163', '6676', '3608', '774', '1421',
       '4064', '2725', '5934', '1867', '3721', '2374', '2909', '1648',
       '799', '7102', '3941', '7422', '437', '1231', '1696', '4921',
       '221', '6500', '3575', '4061', '651', '753', '518', '5537', '4108',
       '5376', '3066', '177', '36', '299', '1447', '2599', '702', '133',
       '153', '500', '2127', '2203'], dtype='<U10')
```

```
step_count.dtype
```

```
dtype('<U10')
```

Because Numpy type-casted all the data to strings. It's a string type where **U** means Unicode String. and 10 means 10 bytes.

Step Count

```
step_count = np.array(step_count, dtype = 'int')
step_count.dtype
```

```
dtype('int64')
```

```
step_count
```

```
array([5464, 6041,   25, 5461, 6915, 4545, 4340, 1230,   61, 1258, 3148,
       4687, 4732, 3519, 1580, 2822,  181, 3158, 4383, 3881, 4037,  202,
        292,  330, 2209, 4550, 4435, 4779, 1831, 2255,  539, 5464, 6041,
       4068, 4683, 4033, 6314,  614, 3149, 4005, 4880, 4136,  705,  570,
        269, 4275, 5999, 4421, 6930, 5195,  546,  493,  995, 1163, 6676,
       3608,  774, 1421, 4064, 2725, 5934, 1867, 3721, 2374, 2909, 1648,
        799, 7102, 3941, 7422,  437, 1231, 1696, 4921,  221, 6500, 3575,
       4061,  651,  753,  518, 5537, 4108, 5376, 3066,  177,   36,  299,
       1447, 2599,  702,  133,  153,  500, 2127, 2203])
```

Calories Burned

```
calories_burned = np.array(calories_burned, dtype = 'int')
calories_burned.dtype
```

```
dtype('int64')
```

Hours of Sleep

```
hours_of_sleep = np.array(hours_of_sleep, dtype = 'int')
```

```
hours_of_sleep.dtype
```

```
dtype('int64')
```

Weight

```
weight = np.array(weight, dtype = 'int')
weight.dtype
```

```
dtype('int64')
```

Mood

`Mood` is a categorical data type

```
mood
```

```
array(['200', '100', '100', '100', '200', '100', '100', '100', '100',
       '100', '100', '100', '300', '100', '100', '100', '100', '200',
       '200', '200', '200', '200', '200', '300', '200', '300', '300',
       '300', '300', '300', '300', '300', '200', '300', '300', '300',
       '300', '300', '300', '300', '300', '300', '300', '200', '300',
       '300', '300', '300', '300', '300', '300', '300', '300', '200',
       '100', '300', '300', '300', '300', '300', '300', '300', '100',
       '200', '200', '100', '100', '200', '200', '300', '200', '200',
       '100', '200', '100', '200', '200', '100', '100', '100', '100',
       '300', '200', '300', '200', '100', '100', '100', '200', '200',
       '100', '100', '300', '200', '200', '300'], dtype='<U10')
```

```
np.unique(mood)
```

```
array(['100', '200', '300'], dtype='<U10')
```

```
mood[mood == '300'] = 'Happy'
```

```
mood[mood == '200'] = 'Neutral'
```

```
mood[mood == '100'] = 'Sad'
```

```
mood
```

```
array(['Neutral', 'Sad', 'Sad', 'Sad', 'Neutral', 'Sad', 'Sad', 'Sad',
       'Sad', 'Sad', 'Sad', 'Sad', 'Happy', 'Sad', 'Sad', 'Sad', 'Sad',
       'Neutral', 'Neutral', 'Neutral', 'Neutral', 'Neutral', 'Neutral',
       'Happy', 'Neutral', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy',
       'Happy', 'Happy', 'Neutral', 'Happy', 'Happy', 'Happy', 'Happy',
       'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Neutral',
       'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy',
       'Happy', 'Happy', 'Neutral', 'Sad', 'Happy', 'Happy', 'Happy',
       'Happy', 'Happy', 'Happy', 'Happy', 'Sad', 'Neutral', 'Neutral',
       'Sad', 'Sad', 'Neutral', 'Neutral', 'Happy', 'Neutral', 'Neutral',
```

```
       'Sad', 'Neutral', 'Sad', 'Neutral', 'Neutral', 'Sad', 'Sad', 'Sad',
       'Sad', 'Happy', 'Neutral', 'Happy', 'Neutral', 'Sad', 'Sad', 'Sad',
       'Neutral', 'Neutral', 'Sad', 'Sad', 'Happy', 'Neutral', 'Neutral',
       'Happy'], dtype='<U7')
```

Activity Status

Here 0 means Feeling of inactiveness

500 means Feeling of activeness

```
activity_status
```

```
array(['0', '0', '0', '0', '500', '0', '0', '0', '0', '0', '0', '0',
       '500', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '500', '0', '0', '0', '0', '500', '0', '0', '0', '0', '0', '500',
       '500', '500', '500', '500', '500', '500', '500', '500', '0', '0',
       '0', '0', '0', '0', '500', '500', '500', '500', '500', '500',
       '500', '500', '500', '500', '500', '500', '0', '500', '500', '0',
       '500', '500', '500', '500', '500', '0', '500', '500', '500', '500',
       '0', '0', '0', '0', '500', '500', '500', '500', '0', '0', '0', '0',
       '0', '0', '0', '0', '500', '0', '500'], dtype='<U10')
```

```
activity_status[activity_status == '500'] = 'Active'
activity_status[activity_status == '0'] = 'Inactive'
```

```
activity_status
```

```
array(['Inactive', 'Inactive', 'Inactive', 'Inactive', 'Active',
       'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Inactive',
       'Inactive', 'Inactive', 'Active', 'Inactive', 'Inactive',
       'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Inactive',
       'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Inactive',
       'Active', 'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Active',
       'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Inactive',
       'Active', 'Active', 'Active', 'Active', 'Active', 'Active',
       'Active', 'Active', 'Active', 'Inactive', 'Inactive', 'Inactive',
       'Inactive', 'Inactive', 'Inactive', 'Active', 'Active', 'Active',
       'Active', 'Active', 'Active', 'Active', 'Active', 'Active',
       'Active', 'Active', 'Active', 'Inactive', 'Active', 'Active',
       'Inactive', 'Active', 'Active', 'Active', 'Active', 'Active',
       'Inactive', 'Active', 'Active', 'Active', 'Active', 'Inactive',
       'Inactive', 'Inactive', 'Inactive', 'Active', 'Active', 'Active',
       'Active', 'Inactive', 'Inactive', 'Inactive', 'Inactive',
       'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Active',
       'Inactive', 'Active'], dtype='<U8')
```

EDA: Insights from the data.

What's the average step count?

```
step_count.mean()
```

```
2935.9375
```

On which day the step count was highest?

```
step_count.argmax()
```

```
69
```

```
date[step_count.argmax()]
```

```
'14-12-2017'
```

Let's check the calorie burnt on the day

```
calories_burned[step_count.argmax()]
```

```
243
```

Let's try to get the number of steps on that day as well

```
step_count.max()
```

```
7422
```

What's the most frequent mood ?

One approach is for each of the category we get count of record and see which one is the highest

```
mood[mood == 'Sad'].shape
```

```
(29,)
```

```
mood[mood == 'Neutral'].shape
```

```
(27,)
```

```
mood[mood == 'Happy'].shape
```

```
(40,)
```

Another approach:

```
np.unique(mood)
```

```
array(['Happy', 'Neutral', 'Sad'], dtype='<U7')
```

We can get the count by passing in the parameter `return_counts = True`

```
np.unique(mood, return_counts = True)
```

```
(array(['Happy', 'Neutral', 'Sad'], dtype='<U7'), array([40, 27, 29]))
```

The most frequent mood is Happy :)

Comparing step counts on bad mood days and good mood days

Average step count on Sad mood days

```
np.mean(step_count[mood == 'Sad'])
```

```
2103.0689655172414
```

```
np.sort(step_count[mood == 'Sad'])
```

```
array([  25,   36,   61,  133,  177,  181,  221,  299,  518,  651,  702,
        753,  799, 1230, 1258, 1580, 1648, 1696, 2822, 3148, 3519, 3721,
       4061, 4340, 4545, 4687, 5461, 6041, 6676])
```

```
np.std(step_count[mood == 'Sad'])
```

```
2021.2355035376254
```

Average step count on happy days

```
np.mean(step_count[mood == 'Happy'])
```

```
3392.725
```

```
np.sort(step_count[mood == 'Happy'])
```

```
array([ 153,  269,  330,  493,  539,  546,  614,  705,  774,  995, 1421,
       1831, 1867, 2203, 2255, 2725, 3149, 3608, 4005, 4033, 4064, 4068,
       4136, 4275, 4421, 4435, 4550, 4683, 4732, 4779, 4880, 5195, 5376,
       5464, 5537, 5934, 5999, 6314, 6930, 7422])
```

Average step count on sad days - 2103.

Average step count on happy days - 3392

There may be relation between mood and step count

Let's try to check inverse. Mood when step count was greater/lesser

Mood when step count > 4000

```
np.unique(mood[step_count > 4000], return_counts = True)
```

```
(array(['Happy', 'Neutral', 'Sad'], dtype='<U7'), array([22,  9,  7]))
```

Out of 38 days when step count was more than 4000, user was feeling happy on 22 days.

Mood when step count <= 2000

```
np.unique(mood[step_count < 2000], return_counts = True)
```

```
(array(['Happy', 'Neutral', 'Sad'], dtype='<U7'), array([13,  8, 18]))
```

Out of 39 days, when step count was less than 2000, user was feeling sad on 18 days.

There may be a correlation between Mood and step count

Operations on Numpy Arrays

Algebric operations on np arrays with single numbers

```
m1 = np.arange(12).reshape(3, 4)
m1 + 2
```

```
array([[ 2,  3,  4,  5],
       [ 6,  7,  8,  9],
       [10, 11, 12, 13]])
```

```
m1 = np.arange(12).reshape(3, 4)
m1 * 2
```

```
array([[ 0,  2,  4,  6],
       [ 8, 10, 12, 14],
       [16, 18, 20, 22]])
```

Algebric operations on two np arrays

```
# Corresponding elements of arrays get added
a = np.array([1, 2, 3])
b = np.array([2, 2, 2])
a + b
```

```
array([3, 4, 5])
```

```
# Corresponding elements of arrays get multiplied
a * b
```

```
array([2, 4, 6])
```

Now what if we use multiplication operator on matrices created using numpy?

```
A = np.arange(12).reshape(3, 4)
A
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
B = np.arange(12).reshape(3, 4)
B
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
A * B
```

```
array([[  0,   1,   4,   9],
       [ 16,  25,  36,  49],
       [ 64,  81, 100, 121]])
```

- it ** did element-wise multiplication**

What is the requirement of dimensions of 2 matrices for Matrix Multiplication?

- **Columns of A = Rows of B**
- **If A is 3×4, B can be** 4×3... or $4 \times x$

```
B = B.reshape(4, 3)
B
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

```
A * B
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-10-a4cedde81ed0> in <module>()
----> 1 A * B

ValueError: operands could not be broadcast together with shapes (3,4) (4,3)
```

SEARCH STACK OVERFLOW

`*` operator only does Element-Wise Multiplication of 2 Matrices

- For actual Matrix Multiplication, We have a different method/operator

```
np.matmul()
```

```
np.matmul(A, B)
```

```
array([[ 42,  48,  54],
       [114, 136, 158],
       [186, 224, 262]])
```

- There's also a direct operator as well for Matrix Multiplication

```
@
```

```
A @ B
```

```
array([[ 42,  48,  54],
       [114, 136, 158],
       [186, 224, 262]])
```

- There is another method in np for doing Matrix Multiplication
  - `np.dot()`

```
np.dot(A, B)
```

```
array([[ 42,  48,  54],
       [114, 136, 158],
       [186, 224, 262]])
```

Other cases of `np.dot()`

- It performs inner product of vectors when both inputs are 1D array
- It performs multiplication when both input are scalers.

```
a= np.array([1,2,3])
b = np.array([1,1,1])
```

```
np.dot(a,b) # 1*1 + 2*1 + 3*1 = 6
```

```
6
```

```
np.dot(4,5)
```

```
20
```

Multiplication of a mix of matrices and vectors

```
A = np.arange(12).reshape(3, 4)  # A is a 3x4 Matrix
A
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
a = np.array([1, 2, 3])  # a is a 1x3 Vector
print(a)
print(a.shape)
```

```
[1 2 3]
(3,)
```

Will `A * a` work?

```
A * a
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-19-920aa4e58700> in <module>()
----> 1 A * a

ValueError: operands could not be broadcast together with shapes (3,4) (3,)
```

SEARCH STACK OVERFLOW

Will `a * A` work?

```
a * A
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-20-d157f8e14faf> in <module>()
----> 1 a * A

ValueError: operands could not be broadcast together with shapes (3,) (3,4)
```

Why does it not work for either cases?

- Because `*` **operator just performs element-wise multiplication**
- For this, **both `A` and `a` should have same shape**

However, `A * A` and `a * a` work

```
A * A # Multiplied with itself
```

```
array([[  0,   1,   4,   9],
       [ 16,  25,  36,  49],
       [ 64,  81, 100, 121]])
```

```
a * a
```

```
array([1, 4, 9])
```

Now Let's experiment with `np.matmul()`

```
np.matmul(A, a)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-23-76efef6bd8e9> in <module>()
----> 1 np.matmul(A, a)

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with
gufunc signature (n?,k),(k,m?)->(n?,m?) (size 3 is different from 4)
```

SEARCH STACK OVERFLOW

- **Columns of `A` (4) $\neq$ Rows of `a` (1)**

Will this work?

```
np.matmul(a, A)
```

```
array([32, 38, 44, 50])
```

- **Columns of `a` (3) = Rows of `A` (3)**

Vectorization

`np.vectorize()`

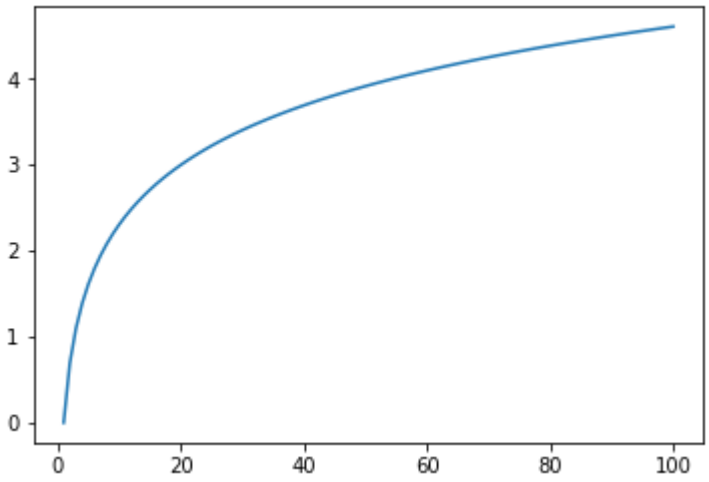
- The vectorized function **evaluates element by element of the input arrays** like the python `map` function

```
import math
import matplotlib.pyplot as plt

x = np.arange(1, 101)

y = np.vectorize(math.log)(x)

plt.plot(x, y)
plt.show()
```

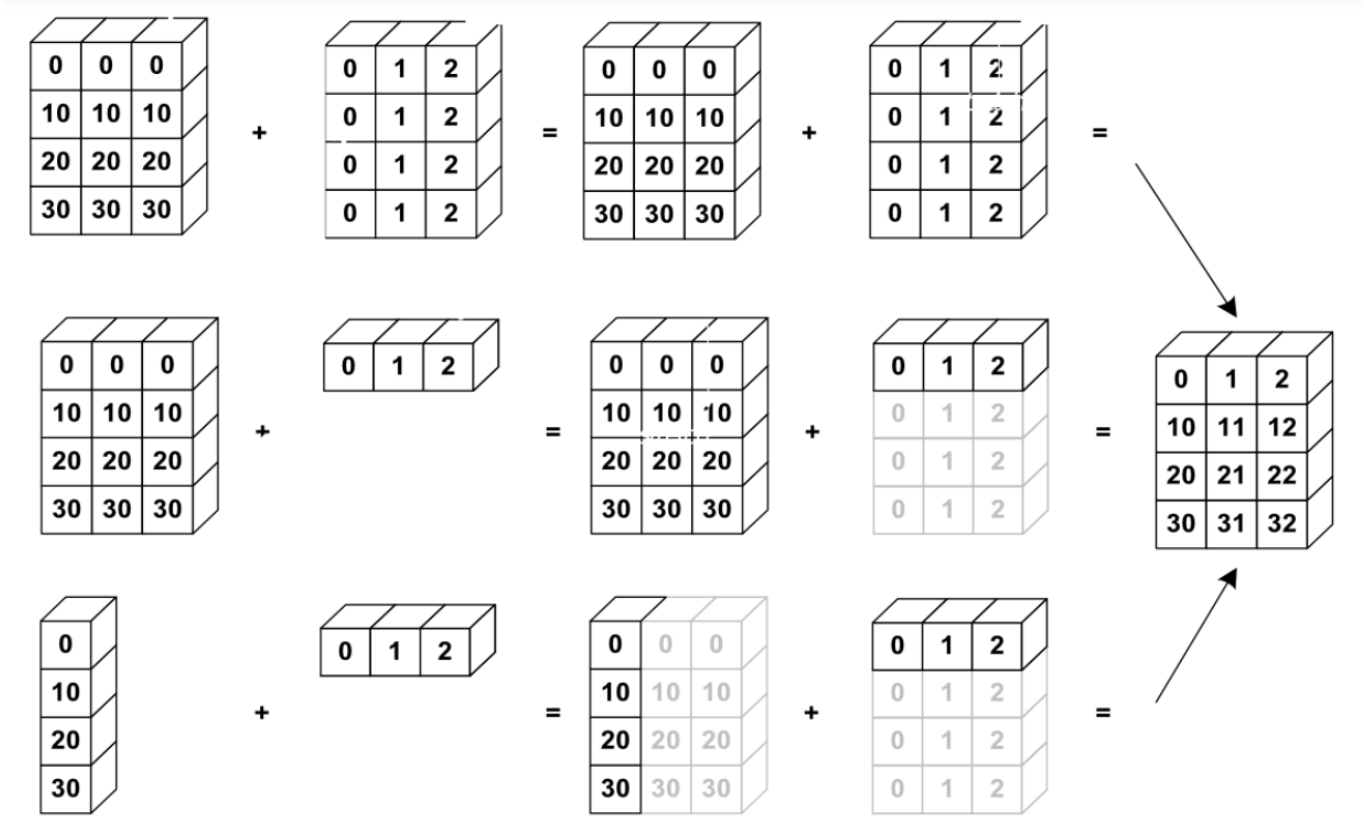



```
y
```

```
array([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, 2.99573227,
       3.04452244, 3.09104245, 3.13549422, 3.17805383, 3.21887582,
       3.25809654, 3.29583687, 3.33220451, 3.36729583, 3.40119738,
       3.4339872 , 3.4657359 , 3.49650756, 3.52636052, 3.55534806,
       3.58351894, 3.61091791, 3.63758616, 3.66356165, 3.68887945,
       3.71357207, 3.73766962, 3.76120012, 3.78418963, 3.80666249,
       3.8286414 , 3.8501476 , 3.87120101, 3.8918203 , 3.91202301,
       3.93182563, 3.95124372, 3.97029191, 3.98898405, 4.00733319,
       4.02535169, 4.04305127, 4.06044301, 4.07753744, 4.09434456,
       4.11087386, 4.12713439, 4.14313473, 4.15888308, 4.17438727,
       4.18965474, 4.20469262, 4.21950771, 4.2341065 , 4.24849524,
       4.26267988, 4.27666612, 4.29045944, 4.30406509, 4.31748811,
       4.33073334, 4.34380542, 4.35670883, 4.36944785, 4.38202663,
       4.39444915, 4.40671925, 4.41884061, 4.4308168 , 4.44265126,
       4.4543473 , 4.46590812, 4.47733681, 4.48863637, 4.49980967,
```

```
4.51085951, 4.52178858, 4.53259949, 4.54329478, 4.55387689,
4.56434819, 4.57471098, 4.58496748, 4.59511985, 4.60517019])
```

Broadcasting



Case1:

given two 2D array

```
[[0,   0,  0],      [[0, 1, 2],
 [10, 10, 10], and   [0, 1, 2],
 [20, 20, 20],       [0, 1, 2],
 [30, 30, 30]]       [0, 1, 2]]
```

Shape of **first array** is **4x3**

Shape of **second array** is **4x3**.

```
a = np.tile(np.arange(0,40,10), (3,1))
a
```

```
array([[ 0, 10, 20, 30],
       [ 0, 10, 20, 30],
       [ 0, 10, 20, 30]])
```

`np.tile` function is used to repeat the given array multiple times

```
np.tile(np.arange(0,40,10), (3,2))
```

```
array([[ 0, 10, 20, 30,  0, 10, 20, 30],
       [ 0, 10, 20, 30,  0, 10, 20, 30],
       [ 0, 10, 20, 30,  0, 10, 20, 30]])
```

```
a
```

```
array([[ 0, 10, 20, 30],
       [ 0, 10, 20, 30],
       [ 0, 10, 20, 30]])
```

```
a = a.T
```

```
a
```

```
array([[ 0,  0,  0],
       [10, 10, 10],
       [20, 20, 20],
       [30, 30, 30]])
```

```
b = np.tile(np.arange(0,3), (4,1))
```

```
b
```

```
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
```

```
a + b
```

```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

Case2 :

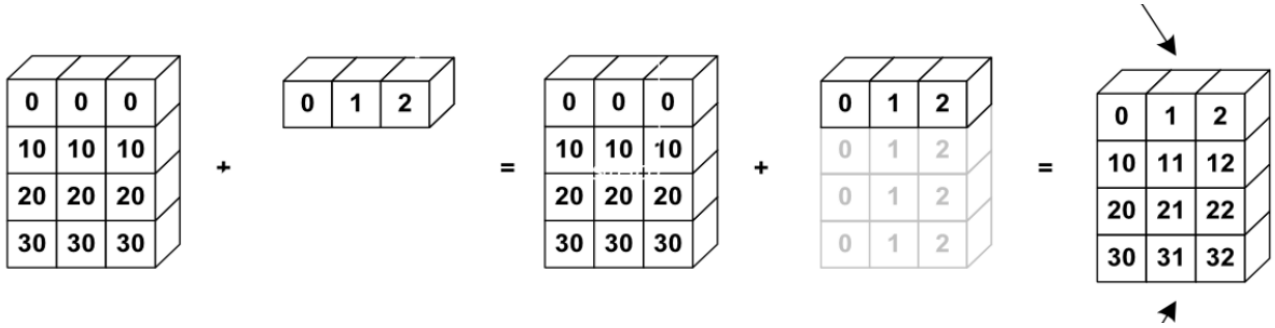
Imagine an array like this:

```
[[0,  0,  0],
 [10, 10, 10],
 [20, 20, 20],
 [30, 30, 30]]
```

We want to add the following array to it:

```
[[0, 1, 2]]
```

What broadcasting does is replicate the second array row wise 4 times to fit the size of first array.




```
a
```

```
array([[ 0,  0,  0],
       [10, 10, 10],
       [20, 20, 20],
       [30, 30, 30]])
```

```
b = np.arange(0,3)
b
```

```
array([0, 1, 2])
```

```
a + b
```

```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

Case 3:

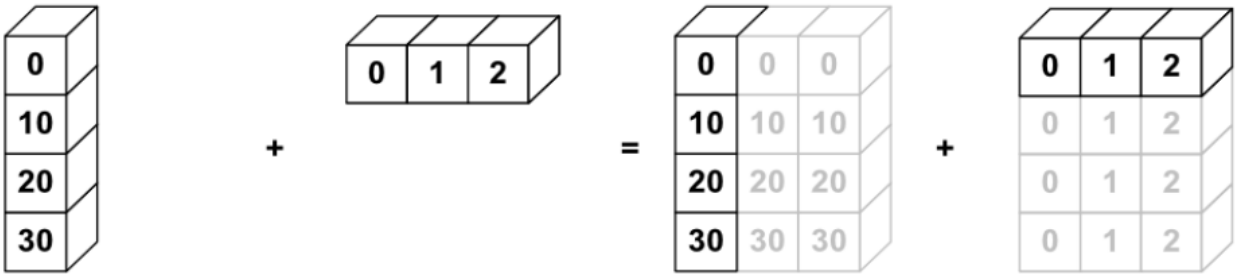
Imagine two array like this:

```
[[0],
[10],
[20],
[30]]
```

and

```
[[0, 1, 2]]
```

Broadcasting will replicate first array column wise 3 time and secord array row wise 4 times to match up the shape.




```
a = np.arange(0,40,10)
a
```

```
array([ 0, 10, 20, 30])
```

This is a 1D row wise array, But we want this array colum wise.

```
a = a.reshape(4,1)
a
```

```
array([[ 0],
       [10],
       [20],
       [30]])
```

There's another way of doing it: `np.newaxis`

`np.newaxis` adds an axis to the array

```
a = np.arange(0, 40, 10)
a
```

```
array([ 0, 10, 20, 30])
```

```
a = a[:, np.newaxis]
a
```

```
array([[ 0],
       [10],
       [20],
       [30]])
```

```
b = np.arange(0,3)
b
```

```
array([0, 1, 2])
```

```
a + b
```

```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

Example:

```
A = np.arange(1,10).reshape(3,3)
A
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
B = np.array([-1, 0, 1])
B
```

```
array([-1,  0,  1])
```

```
A * B
```

```
array([[-1,  0,  3],
       [-4,  0,  6],
       [-7,  0,  9]])
```

Why did `A * B` work in this case?

- `A` has 3 rows and 3 columns
- `B` is a 1-D vector with 3 elements
- So, **`B` gets broadcasted over `A` for each row of `A`**

```
A = np.arange(1,10).reshape(3,3)
A
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
B = np.arange(3, 10, 3).reshape(3,1)
B
```

```
array([[3],
       [6],
       [9]])
```

```
C = A + B
np.round(C, 1)
```

```
array([[ 4,  5,  6],
       [10, 11, 12],
       [16, 17, 18]])
```

How did this `A + B` work?

- `A` has 3 rows and 3 columns
- `B` has 3 rows and 1 column
- So, `B` **gets broadcasted on every column of** `A`
