In [14]:
```
import numpy as np
```

Elementwise Operations

1. Basic Operations

with scalars

In [5]:
```
a = np.array([1, 2, 3, 4]) #create an array

a + 1
```

Out[5]:
```
array([2, 3, 4, 5])
```

In [6]:
```
a ** 2
```

Out[6]:
```
array([ 1,  4,  9, 16], dtype=int32)
```

All arithmetic operates elementwise

In [8]:
```
b = np.ones(4) + 1  #1,1,1,1
print(b)
print(a)
a - b
```

```
[2. 2. 2. 2.]
[1 2 3 4]
```

Out[8]:
```
array([-1.,  0.,  1.,  2.])
```

In [ ]:
```
a * b  #[1,2,3,4] [2,2,2,2]
```

Out[5]:
```
array([ 2.,  4.,  6.,  8.])
```

In [11]:
```
# Matrix multiplication

c = np.diag([1, 2, 3, 4])

print(c * c)  #element by element
print("*****************")
print(c.dot(c)) #matrix mul
```

```
[[ 1  0  0  0]
 [ 0  4  0  0]
 [ 0  0  9  0]
 [ 0  0  0 16]]
*****************
[[ 1  0  0  0]
 [ 0  4  0  0]
 [ 0  0  9  0]
 [ 0  0  0 16]]
```

In [12]:
```
import numpy as np
np.matmul(c,c)  #matrix mutiplication
```

Out[12]:
```
array([[ 1,  0,  0,  0],
       [ 0,  4,  0,  0],
       [ 0,  0,  9,  0],
       [ 0,  0,  0, 16]])
```

comparisions

In [13]:
```
a = np.array([1, 2, 3, 4])
b = np.array([5, 2, 2, 4])
a == b
```

Out[13]:
```
array([False,  True, False,  True])
```

In [ ]:
```
a > b
```

Out[11]:
```
array([False, False,  True, False], dtype=bool)
```

In [ ]:
```
#array-wise comparisions
a = np.array([1, 2, 3, 4])
b = np.array([5, 2, 2, 4])
c = np.array([1, 2, 3, 4])

np.array_equal(a, b)
```

Out[12]:
```
False
```

In [ ]:
```
np.array_equal(a, c)
```

Out[13]:
```
True
```

Logical Operations

In [ ]:
```
a = np.array([1, 1, 0, 0], dtype=bool) #1=true,0=False
b = np.array([1, 0, 1, 0], dtype=bool)

np.logical_or(a, b)
```

Out[14]:
```
array([ True,  True,  True, False], dtype=bool)
```

In [ ]:
```
np.logical_and(a, b)
```

Out[15]:
```
array([ True, False, False, False], dtype=bool)
```

Transcendental functions:

In [7]:
```
a = np.arange(5)

np.sin(a)
```

Out[7]:
```
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ])
```

In [8]:
```
np.log(a)
```

```
C:\Users\91920\anaconda3\lib\site-packages\ipykernel_launcher.py:1: RuntimeWarn
ing: divide by zero encountered in log
  """Entry point for launching an IPython kernel.
```

Out[8]:
```
array([      -inf, 0.        , 0.69314718, 1.09861229, 1.38629436])
```

In [ ]:
```
np.exp(a)   #evaluates e^x for each element in a given input
```

Out[18]:
```
array([  1.        ,   2.71828183,   7.3890561 ,  20.08553692,  54.59815003])
```

Shape Mismatch

In [9]:
```
a = np.arange(4)
print(a.shape)

a + np.array([1, 2])
```

```
(4,)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-9-cfac243e1786> in <module>
      2 print(a.shape)
      3
----> 4 a + np.array([1, 2])

ValueError: operands could not be broadcast together with shapes (4,) (2,)
```

Basic Reductions

computing sums

In [ ]:
```
x = np.array([1, 2, 3, 4])
np.sum(x)
```

Out[4]:
```
10
```

In [9]:
```
#sum by rows and by columns

x = np.array([[1, 1], [2, 2]])
x
```

Out[9]:
```
array([[1, 1],
       [2, 2]])
```

In [10]:
```
x.sum()
```

Out[10]:
```
6
```

In [21]:
```
x.sum(axis=0)   #columns first dimension
```

Out[21]:
```
array([3, 3])
```

In [ ]:
```
x.sum(axis=1)  #rows (second dimension)
```

Out[7]:
```
array([2, 4])
```

Other reductions

In [3]:
```
x = np.array([1, 3, 2])
x.min()
```

Out[3]:
```
1
```

In [4]:
```
x.max()
```

Out[4]:
```
3
```

In [5]:
```
x.argmin()# index of minimum element
```

Out[5]:
```
0
```

In [13]:
```
y = np.array([[5, 2], [1, 4]])
print(y.argmin())
```

```
2
```

In [6]:
```
x.argmax()# index of maximum element
```

Out[6]:
```
1
```

Logical Operations

In [4]:
```
np.all([True, True, False]) #0, False
```

Out[4]:
```
False
```

In [5]:
```
np.any([True, False, False]) #True
```

Out[5]:
```
True
```

In [23]:
```
#Note: can be used for array comparisions
a = np.zeros((10, 10))#0 a==0
print(a)
np.any(a != 0)
```

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Out[23]:
```
False
```

In [24]:
```
np.all(a == a)
```

Out[24]:
```
True
```

In [26]:
```
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()
```

Out[26]:
```
True
```

Statistics

In [27]:
```
x = np.array([1, 2, 3, 1]) #1,1,2,3
y = np.array([[1, 2, 3], [5, 6, 1]])
x.mean()
y.mean()
```

Out[27]:
```
3.0
```

In [29]:
```
np.median(x) #middle element
```

Out[29]:
```
1.5
```

In [ ]:
```
np.median(y, axis= 1) # last axis
```

Out[25]:
```
array([ 2.,  5.])
```

In [ ]:
```
x.std()          # full population standard dev.
```

Out[26]:
```
0.82915619758884995
```

Example:

Data in populations.txt describes the populations of hares and lynxes (and carrots) in northern Canada during 20 years.

In [15]:
```
#load data into numpy array object
data = np.loadtxt('population.txt') #C:/Users/91920/Python/population.txt
```

In [5]:
```
'''file=open("population.txt","r")
print(file.read())
file.close()'''
```

Out[5]:
```
'file=open("population.txt","r") \nprint(file.read())\nfile.close()'
```

In [6]:
```
data
```

Out[6]:
```
array([[ 1900., 30000.,  4000., 48300.],
       [ 1901., 47200.,  6100., 48200.],
       [ 1902., 70200.,  9800., 41500.],
       [ 1903., 77400., 35200., 38200.],
       [ 1904., 36300., 59400., 40600.],
       [ 1905., 20600., 41700., 39800.],
       [ 1906., 18100., 19000., 38600.],
       [ 1907., 21400., 13000., 42300.],
       [ 1908., 22000.,  8300., 44500.],
       [ 1909., 25400.,  9100., 42100.],
       [ 1910., 27100.,  7400., 46000.],
       [ 1911., 40300.,  8000., 46800.],
       [ 1912., 57000., 12300., 43800.],
       [ 1913., 76600., 19500., 40900.],
       [ 1914., 52300., 45700., 39400.],
       [ 1915., 19500., 51100., 39000.],
       [ 1916., 11200., 29700., 36700.],
       [ 1917.,  7600., 15800., 41800.],
       [ 1918., 14600.,  9700., 43300.],
       [ 1919., 16200., 10100., 41300.],
       [ 1920., 24700.,  8600., 47300.]])
```

In [20]:
```
x = data[ : ,0:3]
y = data[ : , -1:]
y
```

Out[20]:
```
array([[48300.],
       [48200.],
       [41500.],
       [38200.],
       [40600.],
       [39800.],
       [38600.],
       [42300.],
       [44500.],
       [42100.],
       [46000.],
       [46800.],
       [43800.],
       [40900.],
       [39400.],
       [39000.],
       [36700.],
       [41800.],
       [43300.],
       [41300.],
       [47300.]])
```

In [24]:
```
y.max()
b=y.argmax()
```

In [25]:
```
x[b]
```

Out[25]:
```
array([ 1900., 30000.,  4000.])
```

In [9]:
```
year, hares, lynxes, carrots = data.T #columns to variables
data.T
```

Out[9]:
```
array([[ 1900.,  1901.,  1902.,  1903.,  1904.,  1905.,  1906.,  1907.,
         1908.,  1909.,  1910.,  1911.,  1912.,  1913.,  1914.,  1915.,
         1916.,  1917.,  1918.,  1919.,  1920.],
       [30000., 47200., 70200., 77400., 36300., 20600., 18100., 21400.,
        22000., 25400., 27100., 40300., 57000., 76600., 52300., 19500.,
        11200.,  7600., 14600., 16200., 24700.],
       [ 4000.,  6100.,  9800., 35200., 59400., 41700., 19000., 13000.,
         8300.,  9100.,  7400.,  8000., 12300., 19500., 45700., 51100.,
        29700., 15800.,  9700., 10100.,  8600.],
       [48300., 48200., 41500., 38200., 40600., 39800., 38600., 42300.,
        44500., 42100., 46000., 46800., 43800., 40900., 39400., 39000.,
        36700., 41800., 43300., 41300., 47300.]])
```

In [ ]:
```
year, hares, lynxes, carrots = data.T #columns to variables
print(year)
```

```
[ 1900.  1901.  1902.  1903.  1904.  1905.  1906.  1907.  1908.  1909.
  1910.  1911.  1912.  1913.  1914.  1915.  1916.  1917.  1918.  1919.
  1920.]
```

In [ ]:
```
#The mean population over time
populations = data[:, 1:]
populations
```

Out[21]:
```
array([[ 30000.,   4000.,  48300.],
       [ 47200.,   6100.,  48200.],
       [ 70200.,   9800.,  41500.],
       [ 77400.,  35200.,  38200.],
       [ 36300.,  59400.,  40600.],
       [ 20600.,  41700.,  39800.],
       [ 18100.,  19000.,  38600.],
       [ 21400.,  13000.,  42300.],
       [ 22000.,   8300.,  44500.],
       [ 25400.,   9100.,  42100.],
       [ 27100.,   7400.,  46000.],
       [ 40300.,   8000.,  46800.],
       [ 57000.,  12300.,  43800.],
       [ 76600.,  19500.,  40900.],
       [ 52300.,  45700.,  39400.],
       [ 19500.,  51100.,  39000.],
       [ 11200.,  29700.,  36700.],
       [  7600.,  15800.,  41800.],
       [ 14600.,   9700.,  43300.],
       [ 16200.,  10100.,  41300.],
       [ 24700.,   8600.,  47300.]])
```

In [ ]:
```
#sample standard deviations
populations.std(axis=0)
```

Out[22]:
```
array([ 20897.90645809,  16254.59153691,   3322.50622558])
```

In [ ]:
```
#which species has the highest population each year?

np.argmax(populations, axis=1)
```

Out[23]:
```
array([2, 2, 0, 0, 1, 1, 2, 2, 2, 2, 2, 2, 0, 0, 0, 1, 2, 2, 2, 2, 2])
```

Broadcasting

Basic operations on numpy arrays (addition, etc.) are elementwise

This works on arrays of the same size. Nevertheless, It's also possible to do operations on arrays of different sizes if NumPy can transform these arrays so that they all have the same size: this conversion is called broadcasting.

The image below gives an example of broadcasting:

In [7]:
```
np.arange(0, 40, 10)
```

Out[7]:
```
array([ 0, 10, 20, 30])
```

In [8]:
```
np.tile(np.arange(0, 40, 10), (3,1))
```

Out[8]:
```
array([[ 0, 10, 20, 30],
       [ 0, 10, 20, 30],
       [ 0, 10, 20, 30]])
```

In [ ]:
```
a = np.tile(np.arange(0, 40, 10), (3,1))
print(a)

print("**************")
a=a.T
print(a)
```

```
[[ 0 10 20 30]
 [ 0 10 20 30]
 [ 0 10 20 30]]
**************
[[ 0  0  0]
 [10 10 10]
 [20 20 20]
 [30 30 30]]
```

In [ ]:
```

b = np.array([0, 1, 2])
b
```

Out[35]:
```
array([0, 1, 2])
```

In [ ]:
```

a + b
```

Out[36]:
```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

In [ ]:
```
a = np.arange(0, 40, 10)
a.shape
```

Out[45]:
```
(4,)
```

In [ ]:
```
a = a[:, np.newaxis]  # adds a new axis -> 2D array
a.shape
```

Out[42]:
```
(4, 1)
```

In [ ]:
```
a
```

Out[43]:
```
array([[ 0],
       [10],
       [20],
       [30]])
```

In [ ]:
```
a + b
```

Out[58]:
```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

Array Shape Manipulation

Flattening

In [10]:
```
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a)
a.ravel() #Return a contiguous flattened array. A 1-D array, containing the eleme
```

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

Out[10]:
```
array([1, 2, 3, 4, 5, 6])
```

In [11]:
```
a.T #Transpose
```

Out[11]:
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

In [12]:
```
a.T.ravel()
```

Out[12]:
```
array([1, 4, 2, 5, 3, 6])
```

Reshaping

The inverse operation to flattening:

In [17]:
```
print(a.shape)
print(a)   #4,3 =12 ,,,, 1*12,,,,2*6 ,,3*4
b=a.reshape((6,1))
print(b)
c=a.reshape((3,2))
print(c)
d=a.reshape((4,4))
print(d)
```

```
(2, 3)
[[1 2 3]
 [4 5 6]]
[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]
[[1 2]
 [3 4]
 [5 6]]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-17-33c50e170589> in <module>
      5 c=a.reshape((3,2))
      6 print(c)
----> 7 d=a.reshape((4,4))
      8 print(d)

ValueError: cannot reshape array of size 6 into shape (4,4)
```

In [ ]:
```
b = a.ravel()
print(b)
```

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

In [ ]:
```
b = b.reshape((2, 3))
b
```

Out[57]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [ ]:
```
b[0, 0] = 100
a
```

Out[59]:
```
array([[100,   2,   3],
       [  4,   5,   6]])
```

Note and Beware: reshape may also return a copy!:

In [ ]:
```
a = np.zeros((3, 2))
b = a.T.reshape(3*2)
b[0] = 50
a
```

Out[60]:
```
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
```

Adding a Dimension

Indexing with the np.newaxis object allows us to add an axis to an array

newaxis is used to increase the dimension of the existing array by one more dimension, when used once. Thus,

1D array will become 2D array

2D array will become 3D array

3D array will become 4D array and so on

In [ ]:
```
z = np.array([1, 2, 3])
z
```

Out[61]:
```
array([1, 2, 3])
```

In [ ]:
```
z[:, np.newaxis]
```

Out[62]:
```
array([[1],
       [2],
       [3]])
```

Dimension Shuffling

In [ ]:
```
a = np.arange(4*3*2).reshape(4, 3, 2)
a.shape
```

Out[63]:
```
(4, 3, 2)
```

In [ ]:
```
a
```

Out[77]:
```
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],

       [[ 6,  7],
        [ 8,  9],
        [10, 11]],

       [[12, 13],
        [14, 15],
        [16, 17]],

       [[18, 19],
        [20, 21],
        [22, 23]]])
```

In [ ]:
```
a[0, 2, 1]
```

Out[64]:
```
5
```

Resizing

In [3]:
```
a = np.arange(2)
a.resize((8,))
a
```

Out[3]:
```
array([0, 1, 0, 0, 0, 0, 0, 0])
```

However, it must not be referred to somewhere else:

In [ ]:
```
b = a
a.resize((4,))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-68-702766c88583> in <module>()
      1 b = a
----> 2 a.resize((4,))

ValueError: cannot resize an array that references or is referenced
by another array in this way.  Use the resize function
```

Sorting Data

In [ ]:
```
#Sorting along an axis:
a = np.array([[5, 4, 6], [2, 3, 2]])
b = np.sort(a, axis=1)
b
```

Out[69]:
```
array([[4, 5, 6],
       [2, 2, 3]])
```

In [ ]:
```
#in-place sort
a.sort(axis=1)
a
```

Out[84]:
```
array([[4, 5, 6],
       [2, 2, 3]])
```

In [ ]:
```
#sorting with fancy indexing
a = np.array([4, 3, 1, 2])
j = np.argsort(a)
j
```

Out[85]:
```
array([2, 3, 1, 0])
```

In [ ]:
```
a[j]
```

Out[86]:
```
array([1, 2, 3, 4])
```