Chapter 2 - Numpy

In [1]:
```
2 ** 1000
```

Out[1]:
```
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376
```

In [2]:
```
import numpy as np
```

In [3]:
```
np.int64(2) ** 1000
```

Out[3]:
```
0
```

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

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

In [5]:
```
len(arr)
```

Out[5]:
```
3
```

In [6]:
```
arr[1]
```

Out[6]:
```
2
```

In [7]:
```
type(arr[1])
```

Out[7]:
```
numpy.int32
```

In [8]:
```
arr.dtype
```

Out[8]:
```
dtype('int32')
```

In [9]:
```
arr64 = np.array([1, 2, 3], dtype=np.int64)
arr64
```

Out[9]:
```
array([1, 2, 3], dtype=int64)
```

In [10]:
```
arr * arr
```

Out[10]:
```
array([1, 4, 9])
```

In [11]:
```
v1 = np.random.rand(1000000)
v2 = np.random.rand(1000000)
```

In [12]:
```
%time v1 * v2
```

```
Wall time: 41 ms
```

Out[12]:
```
array([0.41682525, 0.3373227 , 0.11560451, ..., 0.5219113 , 0.09658863,
       0.10501364])
```

In [13]:
```
np.dot(arr, arr)   #1*1 + 2*2 + 3*3
```

Out[13]:
```
14
```

In [14]:
```
arr @ arr
```

Out[14]:
```
14
```

In [15]:
```
mat = np.array([[1,2,3],[4,5,6],[7,8,9]])
mat
```

Out[15]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [17]:
```
v = np.arange(12)
v
```

Out[17]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

In [18]:
```
v.reshape((4,3))
```

Out[18]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [20]:
```
mat = np.arange(12).reshape((4,3))
mat
```

Out[20]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [22]:
```
mat2 = mat.reshape((3,4))
mat2
```

Out[22]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [23]:
```
mat[1,2] = 17     # I don't understand here.
```

In [24]:
```
mat2
```

Out[24]:
```
array([[ 0,  1,  2,  3],
       [ 4, 17,  6,  7],
       [ 8,  9, 10, 11]])
```

In [25]:
```
mat.T    # We can transpose a matrix by calling the dot T attributes
```

Out[25]:
```
array([[ 0,  3,  6,  9],
       [ 1,  4,  7, 10],
       [ 2, 17,  8, 11]])
```

In [26]:
```
nums = [1,2,3,4,5]
nums[2:4] # This is using slice, slice is useful for getting a subset of larger c
```

Out[26]:
```
[3, 4]
```

In [27]:
```
v = np.arange(1,6)
v[2:4]
```

Out[27]:
```
array([3, 4])
```

In [28]:
```
arr = np.arange(12).reshape((3,4))
arr
```

Out[28]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [29]:
```
arr[0]
```

Out[29]:
```
array([0, 1, 2, 3])
```

In [30]:
```
arr[1,1]
```

Out[30]:
```
5
```

In [31]:
```
arr[1, 1] # This will get the element of the second row, the second column.
```

Out[31]:
```
5
```

In [32]:
```
arr[:, 1]  # This will get the element of every row, the second column.
```

Out[32]:
```
array([1, 5, 9])
```

In [33]:
```
arr[:,1].reshape((3, 1))
```

Out[33]:
```
array([[1],
       [5],
       [9]])
```

In [35]:
```
arr[1:, 2:]  # We can use slicint on both axis. array one column, and then two co
```

Out[35]:
```
array([[ 6,  7],
       [10, 11]])
```

In [38]:
```
arr[1:, 2:] = 7
arr          # We can use slicing to set values.
```

Out[38]:
```
array([[0, 1, 2, 3],
       [4, 5, 7, 7],
       [8, 9, 7, 7]])
```

In [41]:
```
import numpy as np
```

In [42]:
```
arr = np.arange(3)
```

In [43]:
```
np.array([True, False, True])]  # We will get only 0 and 2 which corresponds to t
```

Out[43]:
```
array([0, 2])
```

In [44]:
```
arr >=1 # arr[0] is 0,is not >= 1. All the rest is True.
```

Out[44]:
```
array([False,  True,  True])
```

In [45]:
```
arr[arr>=1]
```

Out[45]:
```
array([1, 2])
```

In [46]:
```
arr = np.arange(10)
```

In [48]:
```
arr[(arr>2) & (arr<7)]
```

Out[48]:
```
array([3, 4, 5, 6])
```

In [50]:
```
arr[(arr>7) | (arr<2)]     # either bigger than 7 or smaller than 2
```

Out[50]:
```
array([0, 1, 8, 9])
```

In [51]:
```
arr[~(arr>7)]   # use the tilde sign to negate a condition, array but not array b
```

Out[51]:
```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

In [53]:
```
mat = np.random.rand(5, 5)
mat
```

Out[53]:
```
array([[0.19408057, 0.77747249, 0.40596808, 0.79930594, 0.89325112],
       [0.24545891, 0.74570744, 0.2340754 , 0.8595367 , 0.0068745 ],
       [0.94506875, 0.22559032, 0.45579212, 0.30096307, 0.09812604],
       [0.16499075, 0.07228821, 0.97628327, 0.21242384, 0.11636345],
       [0.43531605, 0.45974984, 0.9359389 , 0.62038562, 0.895476  ]])
```

In [54]:
```
np.abs(mat - mat.mean())
```

Out[54]:
```
array([[0.28897893, 0.294413  , 0.07709141, 0.31624644, 0.41019162],
       [0.23760059, 0.26264794, 0.2489841 , 0.37647721, 0.47618499],
       [0.46200926, 0.25746917, 0.02726737, 0.18209643, 0.38493346],
       [0.31806875, 0.41077128, 0.49322377, 0.27063565, 0.36669605],
       [0.04774344, 0.02330965, 0.4528794 , 0.13732612, 0.4124165 ]])
```

In [55]:
```
np.abs(mat - mat.mean()) > 1.5*mat.std()
```

Out[55]:
```
array([[False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False,  True, False, False],
       [False, False, False, False, False]])
```

In [56]:
```
mat[np.abs(mat - mat.mean()) > 1.5*mat.std()]
```

Out[56]:
```
array([0.97628327])
```

In [58]:
```
mat[np.abs(mat - mat.mean()) > 1.5*mat.std()] = mat.mean()
```

In [59]:
```
arr = np.arange(3)
arr + 4
```

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

In [60]:
```
arr / 7
```

Out[60]:
```
array([0.        , 0.14285714, 0.28571429])
```

In [61]:
```
arr ** 2
```

Out[61]:
```
array([0, 1, 4], dtype=int32)
```

In [63]:
```
mat = np.arange(9).reshape((3, 3))  # matrix three by three
vec = np.arange(3)
```

In [64]:
```
mat + vec  # Numpy is matching the shapes and extending vector to be the same siz
```

Out[64]:
```
array([[ 0,  2,  4],
       [ 3,  5,  7],
       [ 6,  8, 10]])
```

In [65]:
```
v1 = np.arange(3)
v2 = np.arange(3).reshape((3, 1))
```

In [66]:
```
v2
```

Out[66]:
```
array([[0],
       [1],
       [2]])
```

In [68]:
```
v2.shape   # (3, 1) means 3 rows and 1 column
```

Out[68]:
```
(3, 1)
```

In [69]:
```
v1 + v2
```

Out[69]:
```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

In [70]:
```
v = np.arange(12).reshape((4, 3))
```

In [71]:
```
dir(v)
```

```
 'reshape',
 'resize',
 'round',
 'searchsorted',
 'setfield',
 'setflags',
 'shape',
 'size',
 'sort',
 'squeeze',
 'std',
 'strides',
 'sum',
 'swapaxes',
 'take',
 'tobytes',
 'tofile',
 'tolist',
 'tostring',
 'trace',
```

In [72]:
```
v.T
```

Out[72]:
```
array([[ 0,  3,  6,  9],
       [ 1,  4,  7, 10],
       [ 2,  5,  8, 11]])
```

In [73]:
```
v.any()
```

Out[73]:
```
True
```

In [74]:
```
v.all()  # The truth value of zero is false.
```

Out[74]:
```
False
```

In [75]:
```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

In [76]:
```
v.prod()
```

Out[76]:
```
0
```

In [77]:
```
v.sum(axis=1)   # It will work row-wise, you will get a new array with the sum of
```

Out[77]:
```
array([ 3, 12, 21, 30])
```

In [79]:
```
v.sum(axis=0)   # It will get the sum of each column.
```

Out[79]:
```
array([18, 22, 26])
```

In [80]:
```
v1 = v.copy()
v
```

Out[80]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [82]:
```
data = v.dumps()
data       # The result is sequence of bytes
```

Out[82]:
```
b'\x80\x02cnumpy.core.multiarray\n_reconstruct\nq\x00cnumpy\nndarray\nq\x01K\x0
0\x85q\x02c_codecs\nencode\nq\x03X\x01\x00\x00\x00bq\x04X\x06\x00\x00\x00latin1
q\x05\x86q\x06Rq\x07\x87q\x08Rq\t(K\x01K\x04K\x03\x86q\ncnumpy\ndtype\nq\x0bX\x
02\x00\x00\x00i4q\x0cK\x00K\x01\x87q\rRq\x0e(K\x03X\x01\x00\x00\x00<q\x0fNNNJ\x
ff\xff\xff\xffJ\xff\xff\xff\xffK\x00tq\x10b\x89h\x03X0\x00\x00\x00\x00\x00\x00
\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00\x04\x00\x00\x00\x05\x00\x0
0\x00\x06\x00\x00\x00\x07\x00\x00\x00\x08\x00\x00\x00t\x00\x00\x00n\x00\x00\x
00\x0b\x00\x00\x00q\x11h\x05\x86q\x12Rq\x13tq\x14b.'
```

In [87]:
```
v2 = np.loads(data)
v2
```

```
C:\Users\danal\anaconda3\lib\site-packages\ipykernel_launcher.py:1: Deprecation
Warning: np.loads is deprecated, use pickle.loads instead
  """Entry point for launching an IPython kernel.
```

Out[87]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [88]:
```
np.prod(v)
```

Out[88]:
```
0
```

In [89]:
```
np.sin(np.pi/2)
```

Out[89]:
```
1.0
```

In [90]:
```
v = np.arange(-3,3)
np.sin(v)
```

Out[90]:
```
array([-0.14112001, -0.90929743, -0.84147098,  0.        ,  0.84147098,
        0.90929743])
```

In [91]:
```
def noneg(n):
    if n <0:
        return 0
    return n
```

In [92]:
```
noneg(7)
```

Out[92]:
```
7
```

In [94]:
```
noneg(-3)
```

Out[94]:
```
0
```

In [95]:
```
@np.vectorize
def noneg(n):
    if n <0:
        return 0
    return n
```

In [96]:
```
def noneg(n):
    if n <0:
        return 0
    return n
noneg = np.vectorize(noneg)
```

In [97]:
```
noneg(3)
```

Out[97]:
```
array(3)
```

In [98]:
```
noneg(3).shape   # The result is empty. This means we can use this value as scala
```

Out[98]:
```
()
```

In [99]:
```
noneg(v)
```

Out[99]:
```
array([0, 0, 0, 0, 1, 2])
```

In [100]:
```
nv = np.array([-1, np.nan, 1])
np.sin(nv)
```

Out[100]:
```
array([-0.84147098,         nan,  0.84147098])
```

In [102]:
```
noneg(nv)    # The reason for that is nan is a very negative type.
```

```
C:\Users\danal\anaconda3\lib\site-packages\numpy\lib\function_base.py:2167: Run
timeWarning: invalid value encountered in ? (vectorized)
  outputs = ufunc(*inputs)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-102-d5558f1c62df> in <module>
----> 1 noneg(nv)

~\anaconda3\lib\site-packages\numpy\lib\function_base.py in __call__(self, *arg
s, **kwargs)
   2089             vargs.extend([kwargs[_n] for _n in names])
   2090
-> 2091         return self._vectorize_call(func=func, args=vargs)
   2092
   2093     def _get_ufunc_and_otypes(self, func, args):

~\anaconda3\lib\site-packages\numpy\lib\function_base.py in _vectorize_call(sel
f, func, args)
   2168
   2169             if ufunc.nout == 1:
-> 2170                 res = array(outputs, copy=False, subok=True, dtype=otyp
es[0])
   2171             else:
   2172                 res = tuple([array(x, copy=False, subok=True, dtype=t)

ValueError: cannot convert float NaN to integer
```

In [103]:
```
np.nan > 0
```

Out[103]:
```
False
```

In [104]:
```
np.nan < 0
```

Out[104]:
```
False
```

In [105]:
```
np.nan == np.nan
```

Out[105]:
```
False
```

In [106]:
```
@np.vectorize
def noneg(n):
    if not np.isnan(n) and n < 0:
        return n.__class__(0)
    return n
```

In [107]:
```
noneg(nv)
```

Out[107]:
```
array([ 0., nan,  1.])
```

In [108]:
```
@np.vectorize
def isneg(n):
    return not np.isnan(n) and n < 0
```

In [109]:
```
nv[isneg(nv)] = 0
```

In [110]:
```
nv
```

Out[110]:
```
array([ 0., nan,  1.])
```