Numpy Cheat Sheet

In [1]:
```
import numpy as np
```

In [2]:
```
%precision 2
ipython_plain = get_ipython().display_formatter.formatters['text/plain']
ipython_plain.for_type(np.float64, ipython_plain.lookup_by_type(float));
```

Create

In [3]:
```
x = np.arange(3)   ; print(x, x.dtype)
x = np.arange(3.0) ; print(x, x.dtype)
print( np.arange(1, 3, 0.5) )
```

```
[0 1 2] int64
[0. 1. 2.] float64
[1.  1.5 2.  2.5]
```

In [4]:
```
v3 = np.array([1, 2, 3])
v3
```

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

In [5]:
```
m33 = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
m33
```

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

In [6]:
```
from scipy.sparse import csr_matrix
sparse = csr_matrix(np.array([[0, 0, 1]]))
(sparse + sparse)[0, 2]
```

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

In [7]:
```
v2 = np.array([1, 2])
v2a = v2
v2a[0] = 3
v2
# NOTE: Direct assignment results in a reference!
```

Out[7]:
```
array([3, 2])
```

In [8]:
```
v2b = v2 + 1
v2b[0] = 5
v2
# NOTE: Arithmetic operations results in a copy!
```

Out[8]:
```
array([3, 2])
```

In [9]:
```
v2c = v2.copy()
v2c[0] = 0
v2
```
Out[9]:
```
array([3, 2])
```

Stacking

In [10]:
```
np.column_stack((v3, v3))
```
Out[10]:
```
array([[1, 1],
       [2, 2],
       [3, 3]])
```

In [11]:
```
np.hstack((v3, v3))
```
Out[11]:
```
array([1, 2, 3, 1, 2, 3])
```

In [12]:
```
np.hstack((m33, m33))
```
Out[12]:
```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6],
       [7, 8, 9, 7, 8, 9]])
```

In [13]:
```
np.vstack((v3, v3))
```
Out[13]:
```
array([[1, 2, 3],
       [1, 2, 3]])
```

Describe

In [14]:
```
m33.shape
```
Out[14]:
```
(3, 3)
```

In [15]:
```
m33.ndim
```
Out[15]:
```
2
```

In [16]:
```
m33.size
```
Out[16]:
```
9
```

Select

In [17]:
```
v3[2]
```
Out[17]:
```
3
```

In [18]:
```
m33[2, 2]
```
Out[18]:
```
9
```

In [19]:
```
print( m33[:, 2] )
print( m33[2, :] )
# NOTE: Both Slices Return 1-D *Vectors*
```

```
[3 6 9]
[7 8 9]
```

In [20]:
```
print( m33[:, 2:3] )
print( m33[2:3, :] )
# NOTE: Both Slices Return 2-D *Matrixes*
```

```
[[3]
 [6]
 [9]]
[[7 8 9]]
```

In [21]:
```
m33[1]
# NOTE: Returns 1-D *Vector*
```

Out[21]:
```
array([4, 5, 6])
```

In [22]:
```
m33[1:3]
# NOTE: Returns 2-D *Matrix*
```

Out[22]:
```
array([[4, 5, 6],
       [7, 8, 9]])
```

In [23]:
```
print( m33[1:2] )
print( m33[[1]] )
# NOTE: Returns 2-D *Matrix*
```

```
[[4 5 6]]
[[4 5 6]]
```

np.vectorize()

In [24]:
```
vect_fn = np.vectorize(lambda x: -x if x < 3 else 0)
vect_fn(m33)
```

Out[24]:
```
array([[-1, -2,  0],
       [ 0,  0,  0],
       [ 0,  0,  0]])
```

Aggregate

In [25]:
```
m33
```

Out[25]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [26]:
```
print( m33.min() )
print( m33.max() )
```

```
1
9
```

In [27]:
```
print( m33.max(axis=0) )
```

```
print( m33.max(axis=1) )
# NOTE: Both Return 1-D *Vectors*
```

```
[7 8 9]
[3 6 9]
```

In [28]:
```
m33.mean(), m33.std(), m33.var()
```

Out[28]:
```
(5.00, 2.58, 6.67)
```

Reshape

In [29]:
```
m23 = np.array([[1, 2, 3],
                [4, 5, 6]])
m23
```

Out[29]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [30]:
```
m23.reshape(3, 2)
# Scan Order: Left-to-Right, Top-to-Bottom
```

Out[30]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In [31]:
```
m23.reshape(1, -1)
```

Out[31]:
```
array([[1, 2, 3, 4, 5, 6]])
```

In [32]:
```
print( m23.ravel() )
print( m23.reshape(-1) )
print( m23.flatten() )
# NOTE: All Return 1-D *Vectors*
```

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

In [33]:
```
m23.T
```

Out[33]:
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

In [34]:
```
v3, v3.T
# NOTE: Vectors *CANNOT* be Transposed!
```

Out[34]:
```
(array([1, 2, 3]), array([1, 2, 3]))
```

In [35]:
```
v3[None]
# wrap/encapsulate inside an additional outer dimension
```

Out[35]:
```
array([[1, 2, 3]])
```

In [36]:
```
v3.reshape(-1, 1)
# v3[None].T
# convert vector to" column vector"
```

Out[36]:
```
array([[1],
       [2],
       [3]])
```

In [37]:
```
x = np.array([[[0], [1], [2]]])
print( x.shape )
print( np.squeeze(x).shape )
print( np.squeeze(x, axis=0).shape )
print( np.squeeze(x, axis=2).shape )
print( np.array([[100]]).squeeze() )
# remove axis of length 1
```

```
(1, 3, 1)
(3,)
(3, 1)
(1, 3)
100
```

Linear Algebra

In [38]:
```
np.triu(m33), np.tril(m33)
```

Out[38]:
```
(array([[1, 2, 3],
        [0, 5, 6],
        [0, 0, 9]]),
 array([[1, 0, 0],
        [4, 5, 0],
        [7, 8, 9]]))
```

In [39]:
```
v3
```

Out[39]:
```
array([1, 2, 3])
```

In [40]:
```
v3 @ v3
# vector dot product (1*1 + 2*2 + 3*3)
```

Out[40]:
```
14
```

In [41]:
```
m33 @ m33
# matrix multiplication
```

Out[41]:
```
array([[ 30,  36,  42],
       [ 66,  81,  96],
       [102, 126, 150]])
```

In [42]:
```
m33 * m33
# element-wise multiplication
```

Out[42]:
```
array([[ 1,  4,  9],
       [16, 25, 36],
       [49, 64, 81]])
```

In [43]:
```
np.linalg.inv(np.array([[1,0], [0,2]]))
# matrix inverse
```

Out[43]:
```
array([[1. , 0. ],
       [0. , 0.5]])
```

In [44]:
```
np.linalg.pinv(np.array([[1,0], [0,2]]))
# matrix pseudo-inverse
```

Out[44]:
```
array([[1. , 0. ],
       [0. , 0.5]])
```

Random

In [45]:
```
np.random.seed(0)
```

In [46]:
```
np.random.random(3)
# uniform between 0.0 and 1.0
```

Out[46]:
```
array([0.55, 0.72, 0.6 ])
```

In [47]:
```
[np.random.randint(3) for _ in range(10)]
# np.random.randint(<max-excl>)
```

Out[47]:
```
[1, 1, 2, 0, 2, 0, 0, 0, 2, 1]
```

In [48]:
```
[np.random.randint(1, 4) for _ in range(10)]
# np.random.randint(<min-incl>, <max-excl>)
```

Out[48]:
```
[3, 3, 1, 2, 2, 2, 2, 1, 2, 1]
```

In [49]:
```
np.random.uniform(1, 5, 10)
# np.random.uniform(<min>, <max>, <size>)
```

Out[49]:
```
array([1.08, 4.33, 4.11, 4.48, 4.91, 4.2 , 2.85, 4.12, 1.47, 3.56])
```

In [50]:
```
np.random.normal(5, 1, (3, 4))
# np.random.normal(<mean>, <stddev>, <size>)
```

Out[50]:
```
array([[2.45, 5.65, 5.86, 4.26],
       [7.27, 3.55, 5.05, 4.81],
       [6.53, 6.47, 5.15, 5.38]])
```

In [51]:
```
np.random.choice(v3, size=5, replace=True)
# 'replace': with/without replacement
```

Out[51]:
```
array([3, 1, 2, 2, 2])
```

In [52]:
```
x = np.arange(5)
np.random.shuffle(x)
x
# shuffle *in-place*
```

Out[52]:
```
array([0, 4, 2, 3, 1])
```

Filter

In [53]:
```
m23
```

Out[53]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [54]:
```
m23 < 4
```

Out[54]:
```
array([[ True,  True,  True],
       [False, False, False]])
```

In [55]:
```
(m23 < 2) | (m23 > 4)
```

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

In [56]:
```
m23[(m23 < 4) & (m23 >= 1)]
# NOTE: Returns 1-D *Vector*
```

Out[56]:
```
array([1, 2, 3])
```

In [57]:
```
np.where(m23 < 4, 1, -1)
# np.where(<condition>, <val_if_true>, <val_if_false>)
```

Out[57]:
```
array([[ 1,  1,  1],
       [-1, -1, -1]])
```

Misc

In [58]:
```
v10 = np.linspace(0, 5, 6)
v10
# np.linspace(<start-incl>, <stop-incl>, <num_samples>)
```

Out[58]:
```
array([0., 1., 2., 3., 4., 5.])
```

In [59]:
```
np.percentile(v10, [25, 50, 100])
# Returns value(s) corresponding to given percentile(s)
```

Out[59]:
```
array([1.25, 2.5 , 5.  ])
```

In [60]:
```
x = np.array([[1, 2, 1], [1, 3, 1], [1, 2, 1]])
print('x:\n', x)
print('Unique Elements:\n', np.unique(x))
print('Unique Rows:\n', np.unique(x, axis=0))
print('Unique Cols:\n', np.unique(x, axis=1))
```

```
x:
 [[1 2 1]
 [1 3 1]
 [1 2 1]]
Unique Elements:
 [1 2 3]
Unique Rows:
 [[1 2 1]
 [1 3 1]]
Unique Cols:
 [[1 2]
 [1 3]
 [1 2]]
```

In [ ]:
```

```