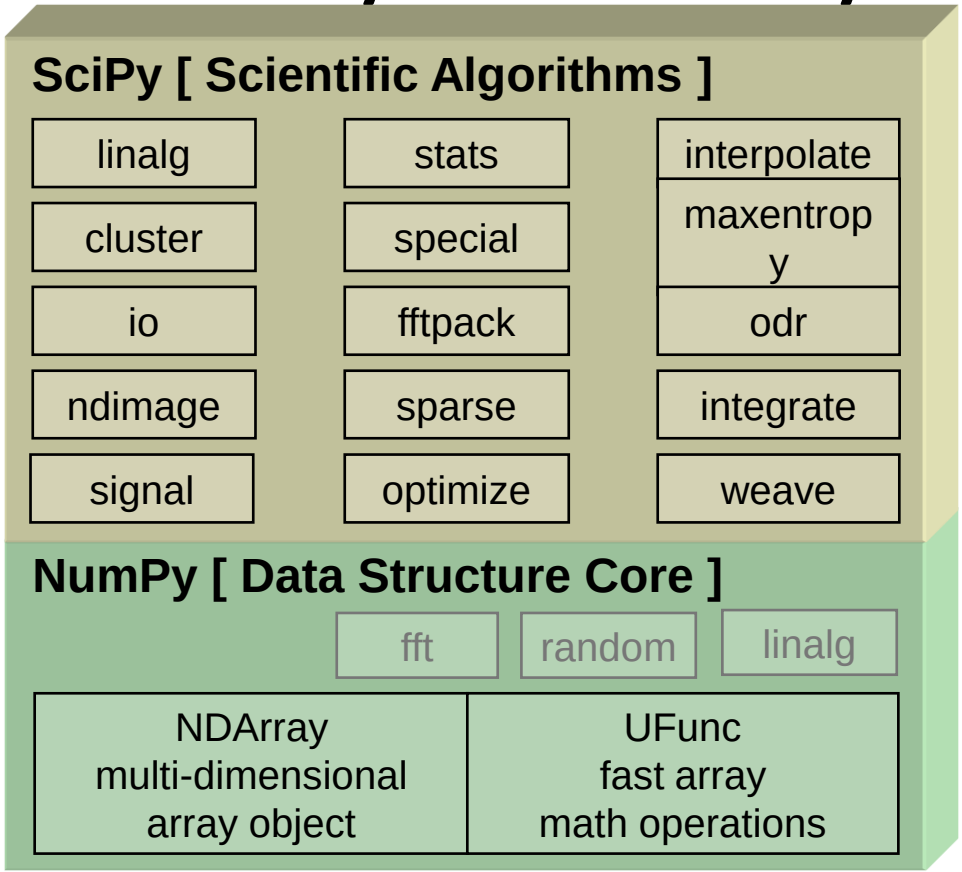
# PFE 3

# NumPy and SciPy



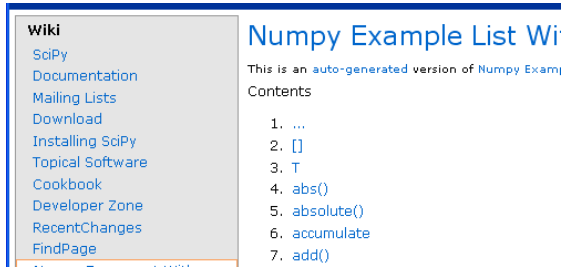
# Online Documentation

## SCIPY DOCUMENTATION PAGE

http://www.scipy.org/Documentation


## NUMPY EXAMPLES

http://www.scipy.org/Numpy_Example_List_With_Doc




### apply_along_axis( )

numpy.apply_along_axis(func1d, axis, arr, *args)

```
Execute func1d(arr[i],*args) where func1d takes 1-D arrays
and arr is an N-d array.  i varies so as to apply the functior
along the given axis for each 1-d subarray in arr.
```

**Example:**

```
>>> from numpy import *
>>> def myfunc(a):                                    # function
...   return (a[0]+a[-1])/2
...
>>> b = array([[1,2,3],[4,5,6],[7,8,9]])
>>> apply_along_axis(myfunc,0,b)                    # apply myfu
array([4, 5, 6])
>>> apply_along_axis(myfunc,1,b)                   # apply myfun
array([2, 5, 8])
```

# ndarray

```
In [191]: data = np.array(data)
```

# ndarray

```
In [191]: data = np.array(data)

In [192]: data
Out[192]:
array([[ 0.9526, -0.246 , -0.8856],
       [ 0.5639,  0.2379,  0.9104]])
```

# ndarray

```
In [191]: data = np.array(data)

In [192]: data
Out[192]:
array([[ 0.9526, -0.246 , -0.8856],
       [ 0.5639,  0.2379,  0.9104]])

In [193]: data*10
Out[193]:
array([[ 9.526, -2.46 , -8.856],
       [ 5.639,  2.379,  9.104]])
```

# ndarray

```
In [191]: data = np.array(data)

In [192]: data
Out[192]:
array([[ 0.9526, -0.246 , -0.8856],
       [ 0.5639,  0.2379,  0.9104]])

In [193]: data*10
Out[193]:
array([[ 9.526, -2.46 , -8.856],
       [ 5.639,  2.379,  9.104]])

In [194]: data + data
Out[194]:
array([[ 1.9052, -0.492 , -1.7712],
       [ 1.1278,  0.4758,  1.8208]])
```

# Types

```
In [216]: tab1 = np.array([1, 2, 3], dtype=np.float64)
```

# Types

```
In [216]: tab1 = np.array([1, 2, 3], dtype=np.float64)

In [217]: tab2 = np.array([1, 2, 3], dtype=np.int32)
```

# Types

```
In [216]: tab1 = np.array([1, 2, 3], dtype=np.float64)

In [217]: tab2 = np.array([1, 2, 3], dtype=np.int32)

In [218]: tab1.dtype
Out[218]: dtype('float64')

In [219]: tab2.dtype
Out[219]: dtype('int32')
```

# Types

```
In [216]: tab1 = np.array([1, 2, 3], dtype=np.float64)

In [217]: tab2 = np.array([1, 2, 3], dtype=np.int32)

In [218]: tab1.dtype
Out[218]: dtype('float64')

In [219]: tab2.dtype
Out[219]: dtype('int32')
```

| Data type | Description |
|---|---|
| bool_ | Boolean (True or False) stored as a byte |
| int_ | Default integer type (same as C `long`; normally either `int64` or `int32`) |
| intc | Identical to C `int` (normally `int32` or `int64`) |
| intp | Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`) |
| int8 | Byte (-128 to 127) |
| int16 | Integer (-32768 to 32767) |
| int32 | Integer (-2147483648 to 2147483647) |
| int64 | Integer (-9223372036854775808 to 9223372036854775807) |
| uint8 | Unsigned integer (0 to 255) |
| uint16 | Unsigned integer (0 to 65535) |
| uint32 | Unsigned integer (0 to 4294967295) |
| uint64 | Unsigned integer (0 to 18446744073709551615) |
| float_ | Shorthand for `float64`. |
| float16 | Half precision float: sign bit, 5 bits exponent, 10 bits mantissa |
| float32 | Single precision float: sign bit, 8 bits exponent, 23 bits mantissa |
| float64 | Double precision float: sign bit, 11 bits exponent, 52 bits mantissa |
| complex_ | Shorthand for `complex128`. |
| complex64 | Complex number, represented by two 32-bit floats (real and imaginary components) |
| complex128 | Complex number, represented by two 64-bit floats (real and imaginary components) |

**ndarrays** are similar to **lists** but all elements have to be of the same type

http://docs.scipy.org/doc/numpy-1.10.1/user/basics.types.html

# Basic initializations

```
In [201]: np.zeros(10)
Out[201]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

# Basic initializations

```
In [201]: np.zeros(10)
Out[201]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [202]: np.arange(10)
Out[202]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

# Basic initializations

```
In [201]: np.zeros(10)
Out[201]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [202]: np.arange(10)
Out[202]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [61]: np.arange(10.)
Out[61]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

# Basic initializations

```
In [201]: np.zeros(10)
Out[201]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [202]: np.arange(10)
Out[202]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [61]: np.arange(10.)
Out[61]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])

In [62]: np.arange(0.0, 1.0, 0.2)
Out[62]: array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

# Basic initializations

```
In [201]: np.zeros(10)
Out[201]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [202]: np.arange(10)
Out[202]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [61]: np.arange(10.)
Out[61]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])

In [62]: np.arange(0.0, 1.0, 0.2)
Out[62]: array([ 0. ,  0.2,  0.4,  0.6,  0.8])

In [33]: np.linspace(0., 1.0, 5.0)
Out[33]: array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

# Type casting

```
In [228]: tab1 = np.array([1, 2, 3], dtype=np.int32)

In [229]: tab1.dtype
Out[229]: dtype('int32')
```

# Type casting

```
In [228]: tab1 = np.array([1, 2, 3], dtype=np.int32)

In [229]: tab1.dtype
Out[229]: dtype('int32')

In [230]: float_tab1 = tab1.astype(np.float64)

In [231]: float_tab1.dtype
Out[231]: dtype('float64')
```

# Type casting

```
In [228]: tab1 = np.array([1, 2, 3], dtype=np.int32)

In [229]: tab1.dtype
Out[229]: dtype('int32')

In [230]: float_tab1 = tab1.astype(np.float64)

In [231]: float_tab1.dtype
Out[231]: dtype('float64')

In [232]: tab = np.array(['1.24', '-9.7', '43'], dtype=np.string_)

In [233]: tab.astype(float)
Out[233]: array([  1.24,  -9.7 ,  43.  ])
```

# numpy.load

numpy.load(*file*, *mmap_mode=None*, *allow_pickle=True*, *fix_imports=True*, *encoding='ASCII'*)

Load arrays or pickled objects from `.npy`, `.npz` or pickled files.

## Examples

Store data to disk, and load it again:

```
>>> np.save('/tmp/123', np.array([[1, 2, 3], [4, 5, 6]]))
>>> np.load('/tmp/123.npy')
array([[1, 2, 3],
       [4, 5, 6]])
```

# numpy.load

numpy.load(*file*, *mmap_mode=None*, *allow_pickle=True*, *fix_imports=True*, *encoding='ASCII'*)

Load arrays or pickled objects from `.npy`, `.npz` or pickled files.

```
In [95]: l = np.load("inter2d.npz")

In [96]: l
Out[96]: <numpy.lib.npyio.NpzFile at 0x3869b898>

In [97]: l.
l.close    l.fid      l.items    l.iterkeys  l.zip
l.f        l.files    l.iteritems l.keys

In [97]: l.files
Out[97]: ['y', 'x', 'z']

In [98]: l['y']
Out[98]:
array([  0.        ,   1.11111111,   2.22222222,   3.33333333,
         4.44444444,   5.55555556,   6.66666667,   7.77777778,
         8.88888889,  10.        ])
```

# numpy.loadtxt

numpy.**loadtxt**(*fname*, *dtype=<type 'float'>*, *comments='#'*, *delimiter=None*, *converters=None*, *skiprows=0*, *usecols=None*, *unpack=False*, *ndmin=0*)

Load data from a text file.

Each row in the text file must have the same number of values.

## Examples

```
>>> from io import StringIO   # StringIO behaves like a file object
>>> c = StringIO("0 1\n2 3")
>>> np.loadtxt(c)
array([[ 0.,  1.],
       [ 2.,  3.]])
```

```
>>> d = StringIO("M 21 72\nF 35 58")
>>> np.loadtxt(d, dtype={'names': ('gender', 'age', 'weight'),
...                      'formats': ('S1', 'i4', 'f4')})
array([('M', 21, 72.0), ('F', 35, 58.0)],
      dtype=[('gender', '|S1'), ('age', '<i4'), ('weight', '<f4')])
```

```
>>> c = StringIO("1,0,2\n3,0,4")
>>> x, y = np.loadtxt(c, delimiter=',', usecols=(0, 2), unpack=True)
>>> x
array([ 1.,  3.])
>>> y
array([ 2.,  4.])
```

# Indexing, Slicing and Broadcasting

indexing

```
In [234]: tab = np.arange(10)

In [235]: tab
Out[235]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [236]: tab[7]
Out[236]: 7
```

# Indexing, Slicing and Broadcasting

**`var[lower:upper:step]`**

indexing

slicing

```
In [234]: tab = np.arange(10)

In [235]: tab
Out[235]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [236]: tab[7]
Out[236]: 7

In [237]: tab[4:7]
Out[237]: array([4, 5, 6])
```

# Indexing, Slicing and Broadcasting

indexing

slicing

broadcasting

```
In [234]: tab = np.arange(10)

In [235]: tab
Out[235]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [236]: tab[7]
Out[236]: 7

In [237]: tab[4:7]
Out[237]: array([4, 5, 6])

In [238]: tab[4:7] = 7

In [239]: tab
Out[239]: array([0, 1, 2, 3, 7, 7, 7, 7, 8, 9])
```

# Multi-Dimensional Arrays

**DEFINITION**

```
>>> a = array([[ 0, 1, 2, 3],
               [10,11,12,13]])
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13]])
```

# Multi-Dimensional Arrays

## DEFINITION

```
>>> a = array([[ 0, 1, 2, 3],
               [10,11,12,13]])
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13]])
```

## (ROWS, COLUMNS)

```
>>> a.shape
(2, 4)
>>> shape(a)
(2, 4)
```

# Multi-Dimensional Arrays

## DEFINITION

```
>>> a = array([[ 0, 1, 2, 3],
               [10,11,12,13]])
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13]])
```

## (ROWS, COLUMNS)

```
>>> a.shape
(2, 4)
>>> shape(a)
(2, 4)
```

## NUMBER OF ELEMENTS

```
>>> a.size
8
>>> size(a)
8
```

# Multi-Dimensional Arrays

## DEFINITION

```
>>> a = array([[ 0, 1, 2, 3],
               [10,11,12,13]])
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13]])
```

## (ROWS, COLUMNS)

```
>>> a.shape
(2, 4)
>>> shape(a)
(2, 4)
```

## NUMBER OF ELEMENTS

```
>>> a.size
8
>>> size(a)
8
```

## NUMBER OF DIMENSIONS

```
>>> a.ndims
2
```

# Multi-Dimensional Arrays

## DEFINITION

```
>>> a = array([[ 0, 1, 2, 3],
               [10,11,12,13]])
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13]])
```

## (ROWS, COLUMNS)

```
>>> a.shape
(2, 4)
>>> shape(a)
(2, 4)
```

## NUMBER OF ELEMENTS

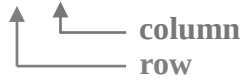
```
>>> a.size
8
>>> size(a)
8
```

## NUMBER OF DIMENSIONS

```
>>> a.ndims
2
```

## INDEXING ELEMENTS

```
>>> a[1,3]
13
```




```
>>> a[1,3] = -1
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,-1]])


>>> a[1]
array([10, 11, 12, -1])
```

# Reshape

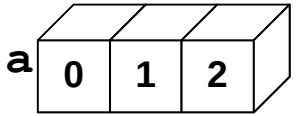
```
a = np.arange(10).reshape((2,5))
a.ndim
a.shape
a.size
a.T
a.dtype
```

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

# Indexing with None

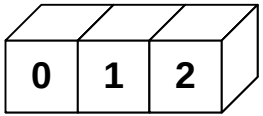
`None` is a special index that inserts a new axis in the array at the specified location.

Each `None` increases the array's dimensionality by 1.



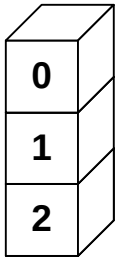

**1 X 3**

```
>>> y = a[None,:]
>>> shape(y)
(1, 3)
```



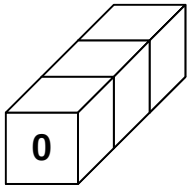

**3 X 1**

```
>>> y = a[:,None]
>>> shape(y)
(3, 1)
```




**3 X 1 X 1**

```
>>> y = a[:,None, None]
>>> shape(y)
(3, 1, 1)
```



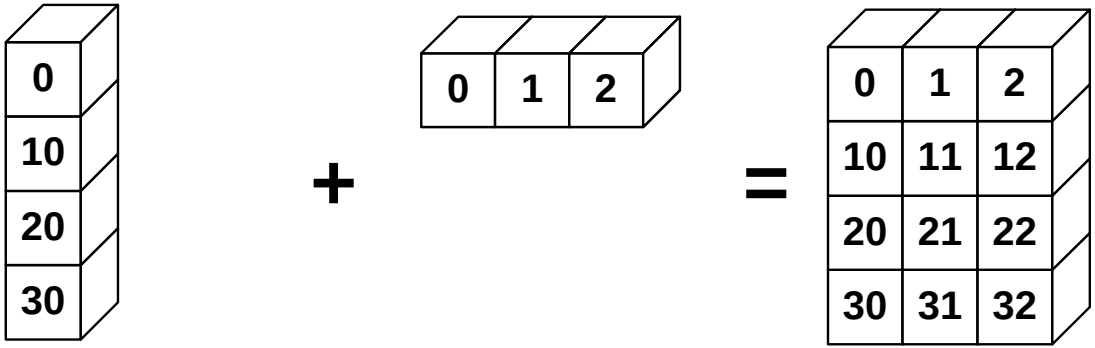
# Newaxis

```
>>> m
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> m[np.newaxis]
array([[0, 1, 2, 3, 4, 5, 6, 7]])
>>> m[:, np.newaxis]
array([[0],
       [1],
       [2],
       [3],
       [4],
       [5],
       [6],
       [7]])
```

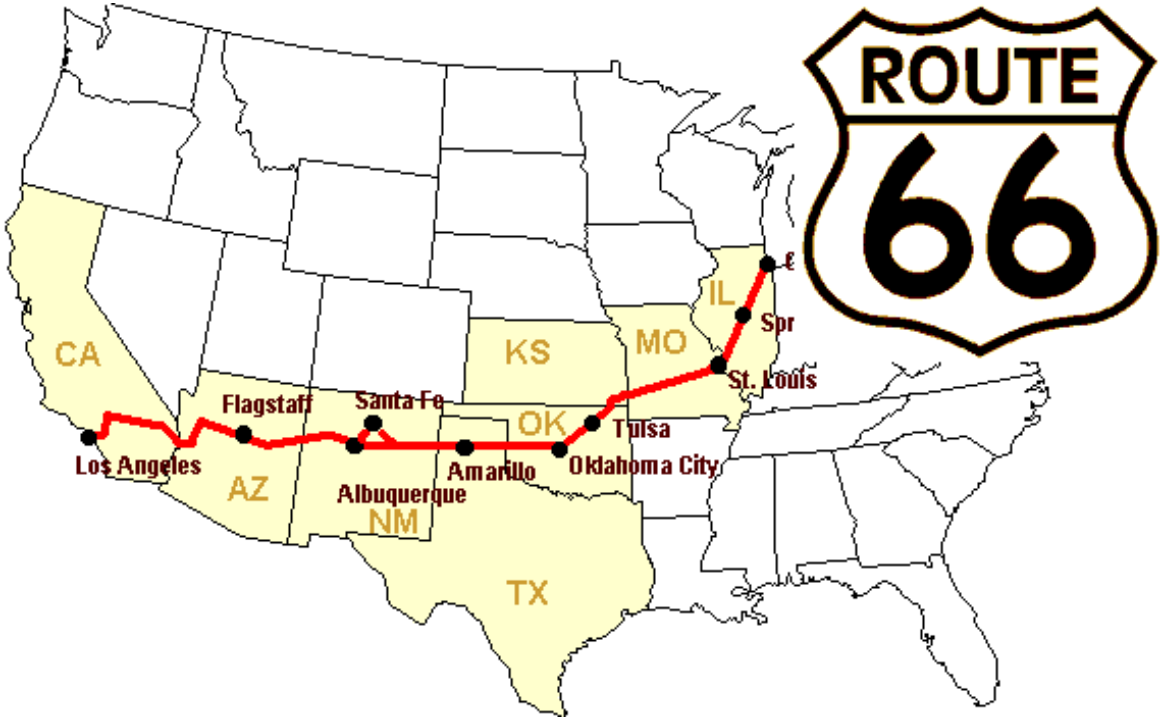
```
>>> m[:, np.newaxis].ndim
2
>>> m[:, np.newaxis].shape
(8, 1)
>>> m[np.newaxis].shape
(1, 8)
>>> m[np.newaxis].ndim
2
```

# Multi-dimensional broadcasting

```
>>> a = array((0,10,20,30))
>>> b = array((0,1,2))
>>> y = a[:, None] + b
```



# 2D Broadcasting Example

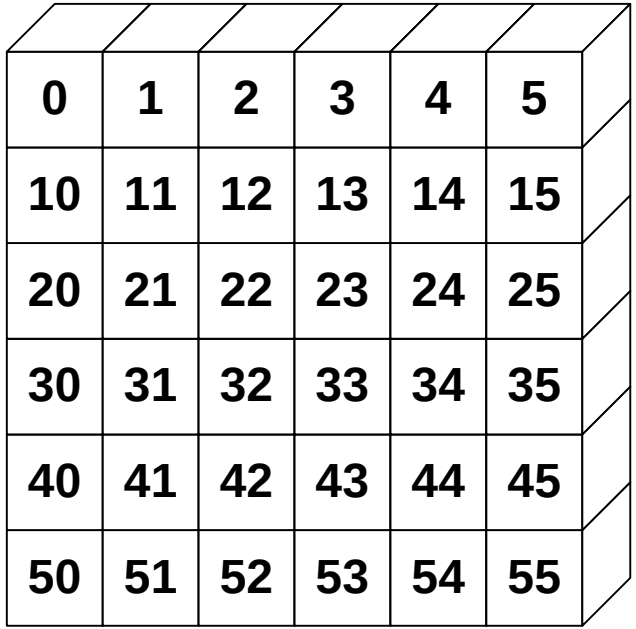


# 2D Broadcasting Example

```
>>> mileposts = np.array([0, 198, 303, 736, 871, 1175, 1475, 1544,
...        1913, 2448])
>>> distance_array = np.abs(mileposts - mileposts[:, np.newaxis])
>>> distance_array
array([[   0,  198,  303,  736,  871, 1175, 1475, 1544, 1913, 2448],
       [ 198,    0,  105,  538,  673,  977, 1277, 1346, 1715, 2250],
       [ 303,  105,    0,  433,  568,  872, 1172, 1241, 1610, 2145],
       [ 736,  538,  433,    0,  135,  439,  739,  808, 1177, 1712],
       [ 871,  673,  568,  135,    0,  304,  604,  673, 1042, 1577],
       [1175,  977,  872,  439,  304,    0,  300,  369,  738, 1273],
       [1475, 1277, 1172,  739,  604,  300,    0,   69,  438,  973],
       [1544, 1346, 1241,  808,  673,  369,   69,    0,  369,  904],
       [1913, 1715, 1610, 1177, 1042,  738,  438,  369,    0,  535],
       [2448, 2250, 2145, 1712, 1577, 1273,  973,  904,  535,    0]])
```

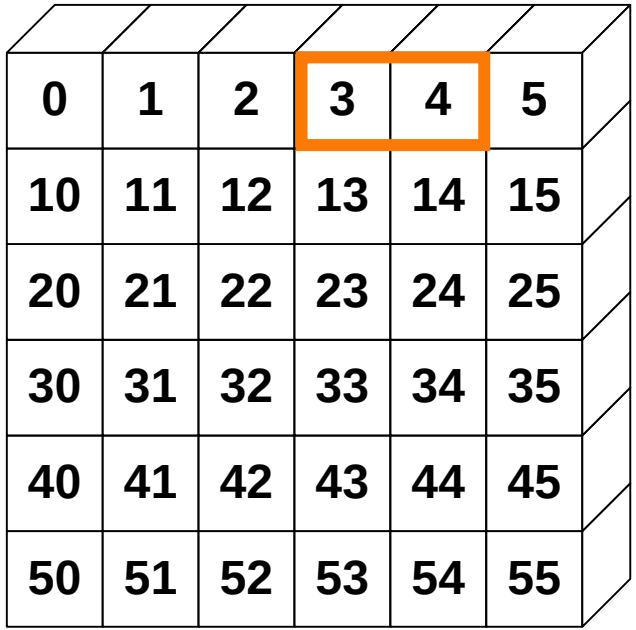
# 2D Array Slicing

```
>>> a[0,3:5]
```



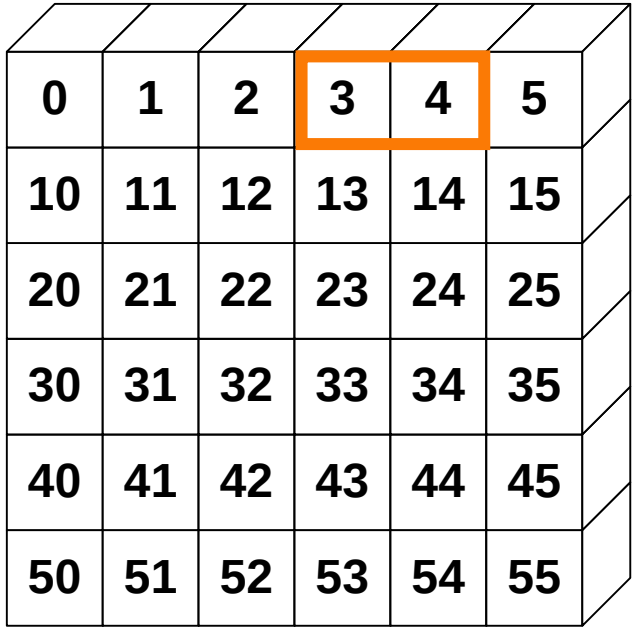
# 2D Array Slicing

```
>>> a[0,3:5]
array([3, 4])
```



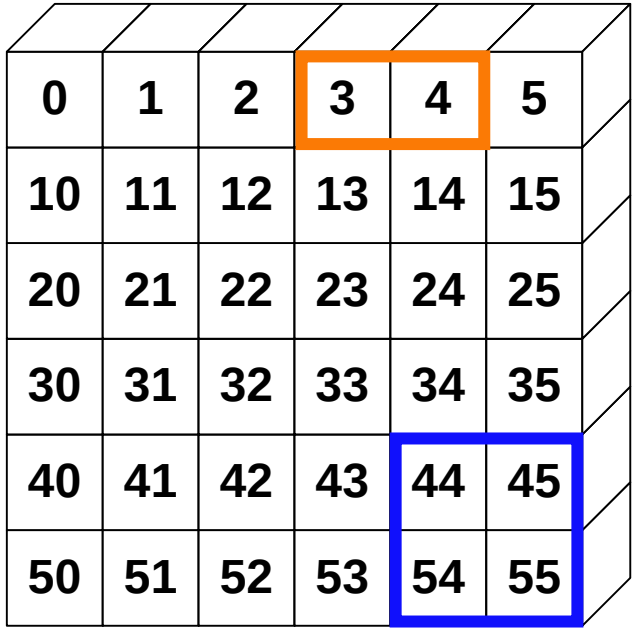
# 2D Array Slicing

```
>>> a[0,3:5]
array([3, 4])

>>> a[4:,4:]
```



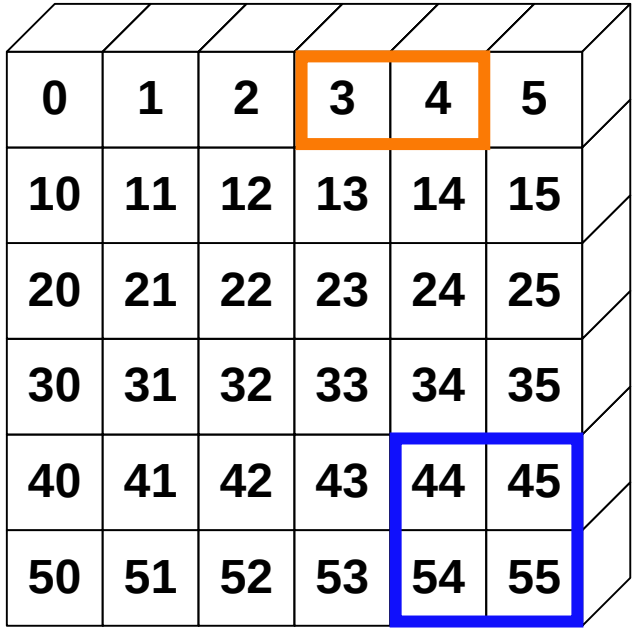
# 2D Array Slicing

```
>>> a[0,3:5]
array([3, 4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])
```



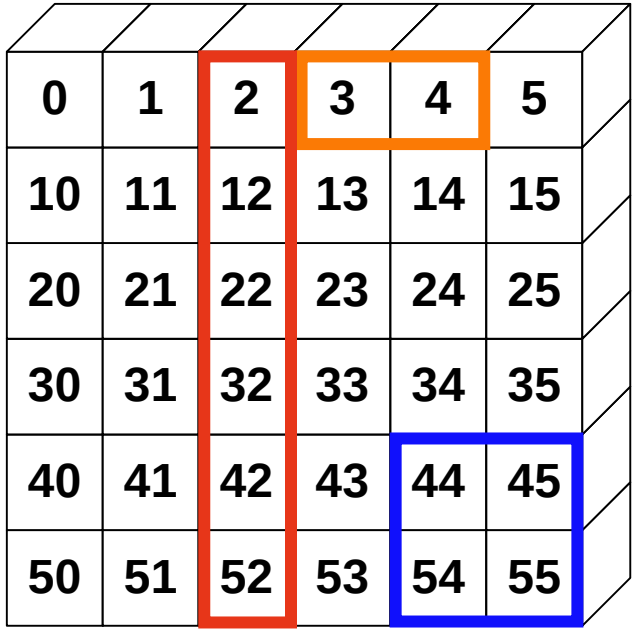
# 2D Array Slicing

```
>>> a[0,3:5]
array([3, 4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
```



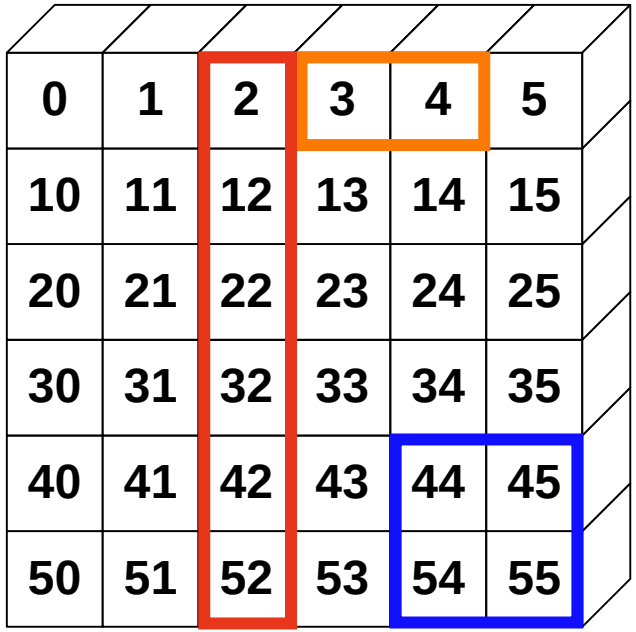
# 2D Array Slicing

```
>>> a[0,3:5]
array([3, 4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])
```



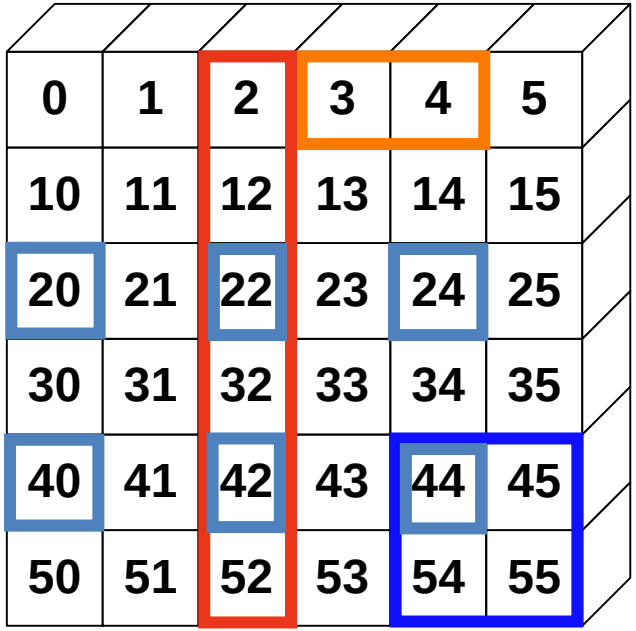
# 2D Array Slicing

```
>>> a[0,3:5]
array([3, 4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])


>>> a[2::2,::2]
```



# 2D Array Slicing

```
>>> a[0,3:5]
array([3, 4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])


>>> a[2::2,::2]
array([[20, 22, 24],
       [40, 42, 44]])
```



# Slices Are References

Slices are references to memory in the original array.

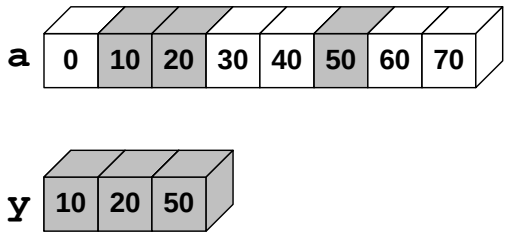
Changing values in a slice also changes the original array.

```
>>> a = array((0,1,2,3,4))

# create a slice containing only the
# last element of a
>>> b = a[2:4]
>>> b
array([2, 3])
>>> b[0] = 10

# changing b changed a!
>>> a
array([ 1,  2, 10, 3, 4])
```

# Fancy Indexing

## Positions

```
>>> a = arange(0,80,10)


# fancy indexing
>>> y = a[[1, 2, -3]]
>>> print y
[10 20 50]

# function take
>>> y = take(a,[1,2,-3])
>>> print y
[10 20 50]
```



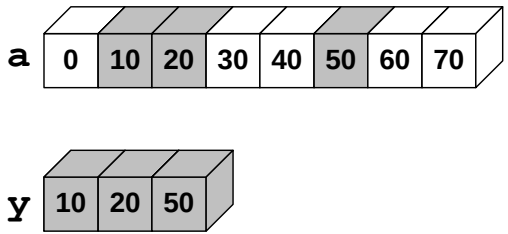
# Fancy Indexing

## Positions

```
>>> a = arange(0,80,10)


# fancy indexing
>>> y = a[[1, 2, -3]]
>>> print y
[10 20 50]

# function take
>>> y = take(a,[1,2,-3])
>>> print y
[10 20 50]
```

## Boolean expressions

```
>>> mask = array([0,1,1,0,0,1,0,0],
...              dtype=bool)

# fancy indexing
>>> y = a[mask]
>>> print y
[10,20,50]

# function compress
>>> y = compress(mask, a)
>>> print y
[10,20,50]
```



# Fancy Indexing in 2D

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
```

```
>>> a[3:,[0, 2, 5]]
```

```
>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
```

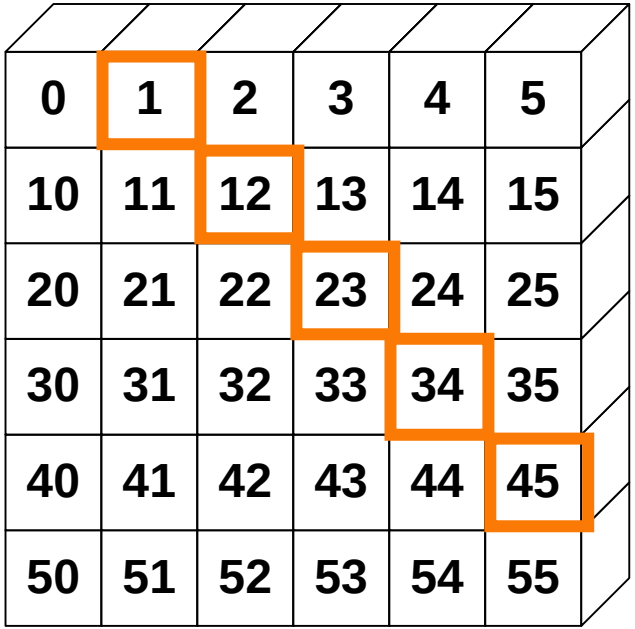
| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 10 | 11 | 12 | 13 | 14 | 15 |
| 20 | 21 | 22 | 23 | 24 | 25 |
| 30 | 31 | 32 | 33 | 34 | 35 |
| 40 | 41 | 42 | 43 | 44 | 45 |
| 50 | 51 | 52 | 53 | 54 | 55 |

Unlike slicing, fancy indexing creates copies instead of views into original arrays.

# Fancy Indexing in 2D

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
```

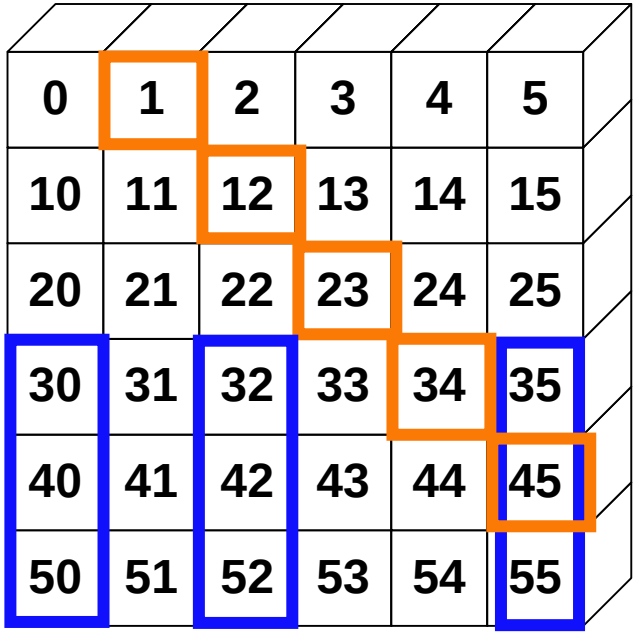
```
>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
```




Unlike slicing, fancy indexing creates copies instead of views into original arrays.

# Fancy Indexing in 2D

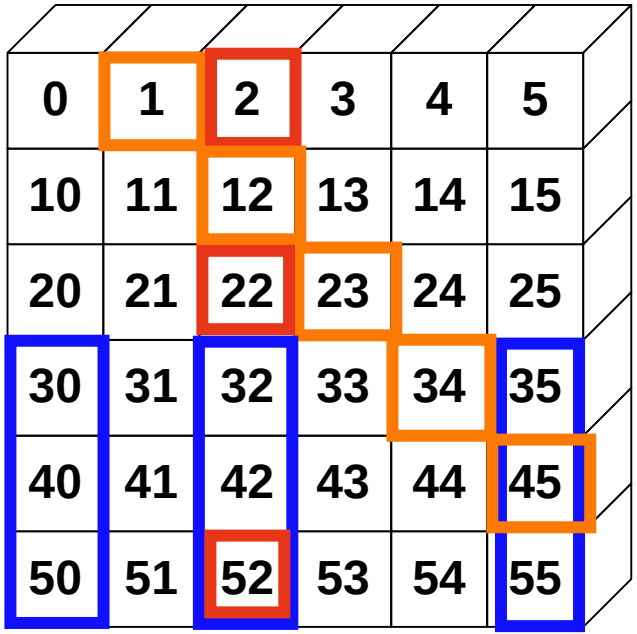
```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45]])
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
```




Unlike slicing, fancy indexing creates copies instead of views into original arrays.

# Fancy Indexing in 2D

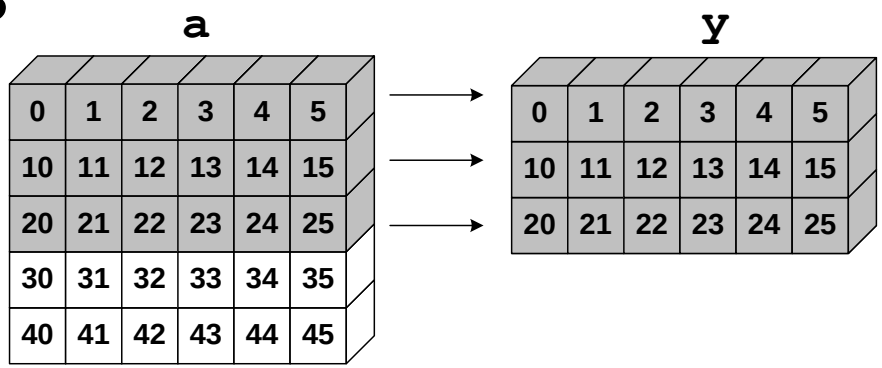
```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45]])
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
array([2,22,52])
```



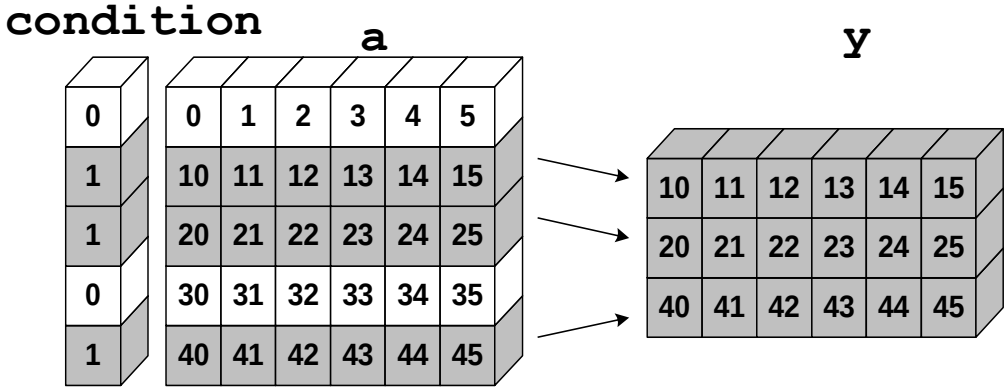

Unlike slicing, fancy indexing creates copies instead of views into original arrays.

# “Incomplete” Indexing

```
>>> y = a[:3]
```

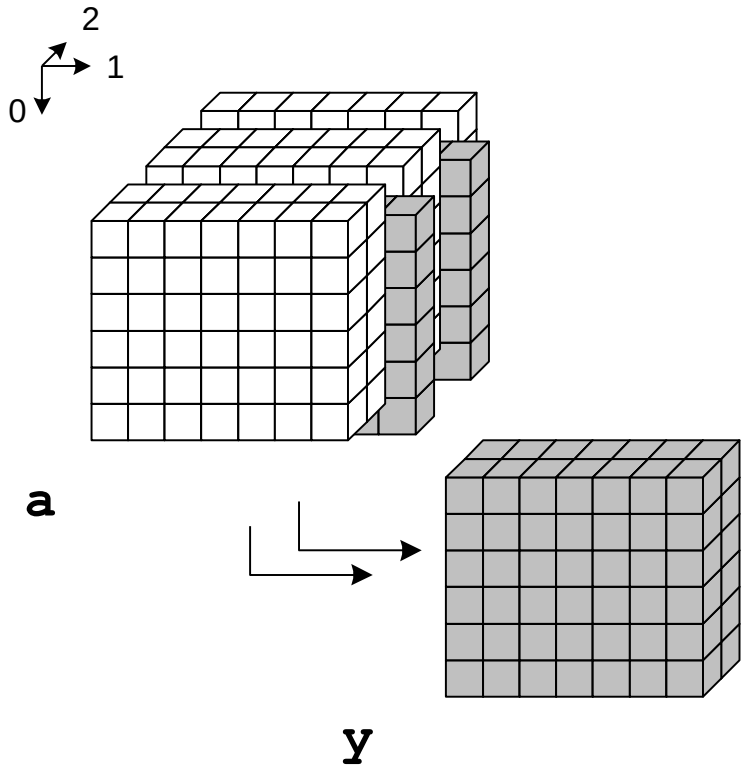



```
>>> y = a[condition]
```

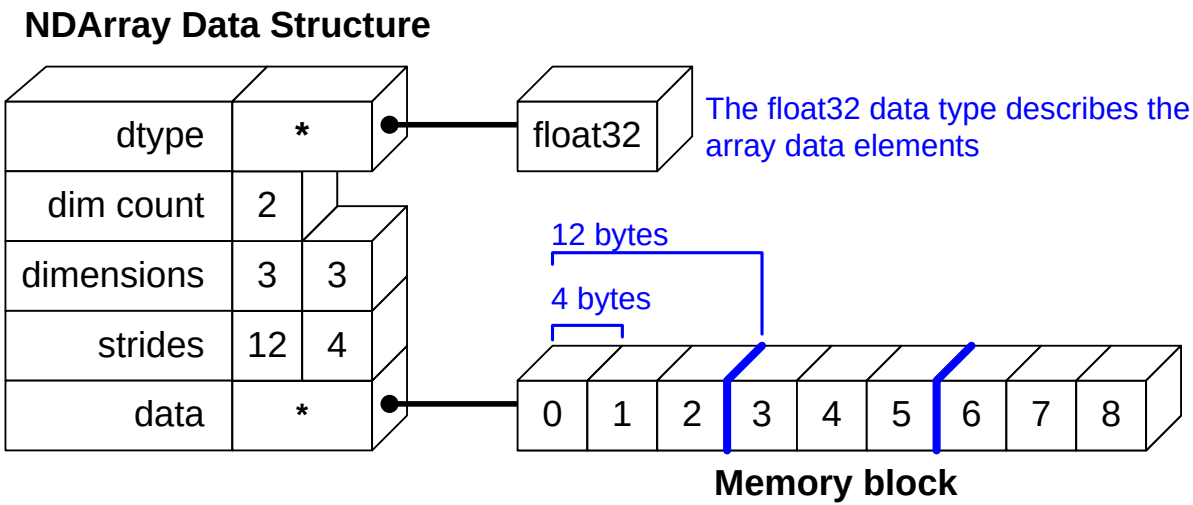


# 3D Example

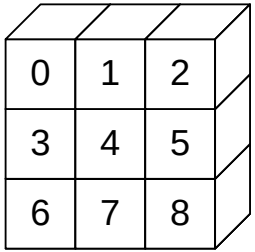
```
# retrieve two slices from a
# 3D cube via indexing
>>> y = a[:,:,[2,-2]]

# the take() function also works
>>> y = take(a,[2,-2], axis=2)
```



# Array Data Structure




**Python View:**



# Conditions

```
In [250]: mac == 3
Out[250]:
array([[False, False,  True],
       [False, False, False],
       [False, False, False]], dtype=bool)
```

```
In [76]: mac
Out[76]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

# Conditions

```
In [76]: mac
Out[76]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
In [250]: mac == 3
Out[250]:
array([[False, False,  True],
       [False, False, False],
       [False, False, False]], dtype=bool)

In [251]: mac < 5
Out[251]:
array([[ True,  True,  True],
       [ True, False, False],
       [False, False, False]], dtype=bool)
```

# Conditions

```
In [76]: mac
Out[76]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
In [250]: mac == 3
Out[250]:
array([[False, False,  True],
       [False, False, False],
       [False, False, False]], dtype=bool)

In [251]: mac < 5
Out[251]:
array([[ True,  True,  True],
       [ True, False, False],
       [False, False, False]], dtype=bool)

In [252]: mac[mac < 5]
Out[252]: array([1, 2, 3, 4])
```

# Conditions

```
In [76]: mac
Out[76]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
In [250]: mac == 3
Out[250]:
array([[False, False,  True],
       [False, False, False],
       [False, False, False]], dtype=bool)

In [251]: mac < 5
Out[251]:
array([[ True,  True,  True],
       [ True, False, False],
       [False, False, False]], dtype=bool)

In [252]: mac[mac < 5]
Out[252]: array([1, 2, 3, 4])

In [253]: mac[mac < 5] = 0

In [254]: mac
Out[254]:
array([[0, 0, 0],
       [0, 5, 6],
       [7, 8, 9]])
```

# Sorting

Sorting along an axis:

```
>>> a = np.array([[4, 3, 5], [1, 2, 1]])
>>> b = np.sort(a, axis=1)
>>> b
array([[3, 4, 5],
       [1, 1, 2]])
```

Sorting with fancy indexing:

```
>>> a = np.array([4, 3, 1, 2])
>>> j = np.argsort(a)
>>> j
array([2, 3, 1, 0])
>>> a[j]
array([1, 2, 3, 4])
```

# Sorting

Sorting along an axis:

```
>>> a = np.array([[4, 3, 5], [1, 2, 1]])
>>> b = np.sort(a, axis=1)
>>> b
array([[3, 4, 5],
       [1, 1, 2]])
```

Sorting with fancy indexing:

```
>>> a = np.array([4, 3, 1, 2])
>>> j = np.argsort(a)
>>> j
array([2, 3, 1, 0])
>>> a[j]
array([1, 2, 3, 4])
```

Finding minima and maxima:

```
>>> a = np.array([4, 3, 1, 2])
>>> j_max = np.argmax(a)
>>> j_min = np.argmin(a)
>>> j_max, j_min
(0, 2)
```

# Reductions

**Extrema:**

```
>>> x = np.array([1, 3, 2])
>>> x.min()
1
>>> x.max()
3

>>> x.argmin()  # index of minimum
0
>>> x.argmax()  # index of maximum
1
```

**Logical operations:**

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

```
>>> a = np.zeros((100, 100))
>>> np.any(a != 0)
False
>>> np.all(a == a)
True

>>> 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
```

# Iterating over arrays

- Over all elements
  - `for element in A.flat:`
- In multi-dimensional arrays over rows
  - `for row in A:`
- Over individual elements
  - `for i in range(A.shape[0]):`
    - `for j in range(A.shape[1]):`

# Universal Functions: *ufunc*

- ufunc is a function that is applied to all elements of an array

```
In [280]: tab = np.arange(10)

In [281]: np.sqrt(tab)
Out[281]:
array([ 0.        ,  1.        ,  1.41421356,  1.73205081,  2.        ,
        2.23606798,  2.44948974,  2.64575131,  2.82842712,  3.        ])

In [282]: np.exp(tab)
Out[282]:
array([  1.00000000e+00,   2.71828183e+00,   7.38905610e+00,
         2.00855369e+01,   5.45981500e+01,   1.48413159e+02,
         4.03428793e+02,   1.09663316e+03,   2.98095799e+03,
         8.10308393e+03])
```

# Mathematical functions

## Mathematical functions

### Trigonometric functions

| | |
|---|---|
| sin(x[, out]) | Trigonometric sine, element-wise. |
| cos(x[, out]) | Cosine element-wise. |
| tan(x[, out]) | Compute tangent element-wise. |
| arcsin(x[, out]) | Inverse sine, element-wise. |
| arccos(x[, out]) | Trigonometric inverse cosine, element-wise. |
| arctan(x[, out]) | Trigonometric inverse tangent, element-wise. |
| hypot(x1, x2[, out]) | Given the "legs" of a right triangle, return its hypotenuse. |
| arctan2(x1, x2[, out]) | Element-wise arc tangent of `x1/x2` choosing the quadrant correctly. |
| degrees(x[, out]) | Convert angles from radians to degrees. |
| radians(x[, out]) | Convert angles from degrees to radians. |
| unwrap(p[, discont, axis]) | Unwrap by changing deltas between values to 2*pi complement. |
| deg2rad(x[, out]) | Convert angles from degrees to radians. |
| rad2deg(x[, out]) | Convert angles from radians to degrees. |

### Hyperbolic functions

| | |
|---|---|
| sinh(x[, out]) | Hyperbolic sine, element-wise. |
| cosh(x[, out]) | Hyperbolic cosine, element-wise. |
| tanh(x[, out]) | Compute hyperbolic tangent element-wise. |
| arcsinh(x[, out]) | Inverse hyperbolic sine element-wise. |
| arccosh(x[, out]) | Inverse hyperbolic cosine, element-wise. |
| arctanh(x[, out]) | Inverse hyperbolic tangent element-wise. |

### Rounding

| | |
|---|---|
| around(a[, decimals, out]) | Evenly round to the given number of decimals. |
| round_(a[, decimals, out]) | Round an array to the given number of decimals. |
| rint(x[, out]) | Round elements of the array to the nearest integer. |
| fix(x[, y]) | Round to nearest integer towards zero. |
| floor(x[, out]) | Return the floor of the input, element-wise. |
| ceil(x[, out]) | Return the ceiling of the input, element-wise. |
| trunc(x[, out]) | Return the truncated value of the input, element-wise. |

### Sums, products, differences

| | |
|---|---|
| prod(a[, axis, dtype, out, keepdims]) | Return the product of array elements over a given axis. |
| sum(a[, axis, dtype, out, keepdims]) | Sum of array elements over a given axis. |
| nansum(a[, axis, dtype, out, keepdims]) | Return the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero. |
| cumprod(a[, axis, dtype, out]) | Return the cumulative product of elements along a given axis. |
| cumsum(a[, axis, dtype, out]) | Return the cumulative sum of the elements along a given axis. |
| diff(a[, n, axis]) | Calculate the n-th discrete difference along given axis. |
| ediff1d(ary[, to_end, to_begin]) | The differences between consecutive elements of an array. |
| gradient(f, *varargs, **kwargs) | Return the gradient of an N-dimensional array. |
| cross(a, b[, axisa, axisb, axisc, axis]) | Return the cross product of two (arrays of) vectors. |
| trapz(y[, x, dx, axis]) | Integrate along the given axis using the composite trapezoidal rule. |

### Exponents and logarithms

| | |
|---|---|
| exp(x[, out]) | Calculate the exponential of all elements in the input array. |
| expm1(x[, out]) | Calculate `exp(x) - 1` for all elements in the array. |
| exp2(x[, out]) | Calculate *2**p* for all *p* in the input array. |
| log(x[, out]) | Natural logarithm, element-wise. |
| log10(x[, out]) | Return the base 10 logarithm of the input array, element-wise. |
| log2(x[, out]) | Base-2 logarithm of *x*. |
| log1p(x[, out]) | Return the natural logarithm of one plus the input array, element-wise. |
| logaddexp(x1, x2[, out]) | Logarithm of the sum of exponentiations of the inputs. |
| logaddexp2(x1, x2[, out]) | Logarithm of the sum of exponentiations of the inputs in base-2. |

### Other special functions

| | |
|---|---|
| i0(x) | Modified Bessel function of the first kind, order 0. |
| sinc(x) | Return the sinc function. |

### Floating point routines

| | |
|---|---|
| signbit(x[, out]) | Returns element-wise True where signbit is set (less than zero). |
| copysign(x1, x2[, out]) | Change the sign of x1 to that of x2, element-wise. |
| frexp(x[, out1, out2]) | Decompose the elements of x into mantissa and twos exponent. |
| ldexp(x1, x2[, out]) | Returns x1 * 2**x2, element-wise. |

### Arithmetic operations

| | |
|---|---|
| add(x1, x2[, out]) | Add arguments element-wise. |
| reciprocal(x[, out]) | Return the reciprocal of the argument, element-wise. |
| negative(x[, out]) | Numerical negative, element-wise. |
| multiply(x1, x2[, out]) | Multiply arguments element-wise. |
| divide(x1, x2[, out]) | Divide arguments element-wise. |
| power(x1, x2[, out]) | First array elements raised to powers from second array, element-wise. |
| subtract(x1, x2[, out]) | Subtract arguments, element-wise. |
| true_divide(x1, x2[, out]) | Returns a true division of the inputs, element-wise. |
| floor_divide(x1, x2[, out]) | Return the largest integer smaller or equal to the division of the inputs. |
| fmod(x1, x2[, out]) | Return the element-wise remainder of division. |
| mod(x1, x2[, out]) | Return element-wise remainder of division. |
| modf(x[, out1, out2]) | Return the fractional and integral parts of an array, element-wise. |
| remainder(x1, x2[, out]) | Return element-wise remainder of division. |

### Handling complex numbers

| | |
|---|---|
| angle(z[, deg]) | Return the angle of the complex argument. |
| real(val) | Return the real part of the elements of the array. |
| imag(val) | Return the imaginary part of the elements of the array. |
| conj(x[, out]) | Return the complex conjugate, element-wise. |

### Miscellaneous

| | |
|---|---|
| convolve(a, v[, mode]) | Returns the discrete, linear convolution of two one-dimensional sequences. |
| clip(a, a_min, a_max[, out]) | Clip (limit) the values in an array. |
| sqrt(x[, out]) | Return the positive square-root of an array, element-wise. |
| square(x[, out]) | Return the element-wise square of the input. |
| absolute(x[, out]) | Calculate the absolute value element-wise. |
| fabs(x[, out]) | Compute the absolute values element-wise. |
| sign(x[, out]) | Returns an element-wise indication of the sign of a number. |
| maximum(x1, x2[, out]) | Element-wise maximum of array elements. |
| minimum(x1, x2[, out]) | Element-wise minimum of array elements. |
| fmax(x1, x2[, out]) | Element-wise maximum of array elements. |

# Vectorizing Functions

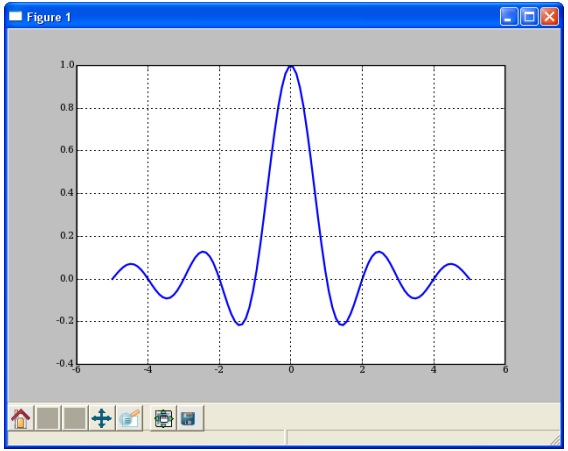
## SCALAR SINC FUNCTION

```
def sinc(x):
    if x == 0.0:
        return 1.0
    else:
        w = pi*x
        return sin(w) / w
```

```
# attempt
>>> x = array((1.3, 1.5))
>>> sinc(x)
ValueError: The truth value of an array
with more than one element is ambiguous.
Use a.any() or a.all()
>>> x = r_[-5:5:100j]
>>> y = vsinc(x)
>>> plot(x, y)
```

## SOLUTION

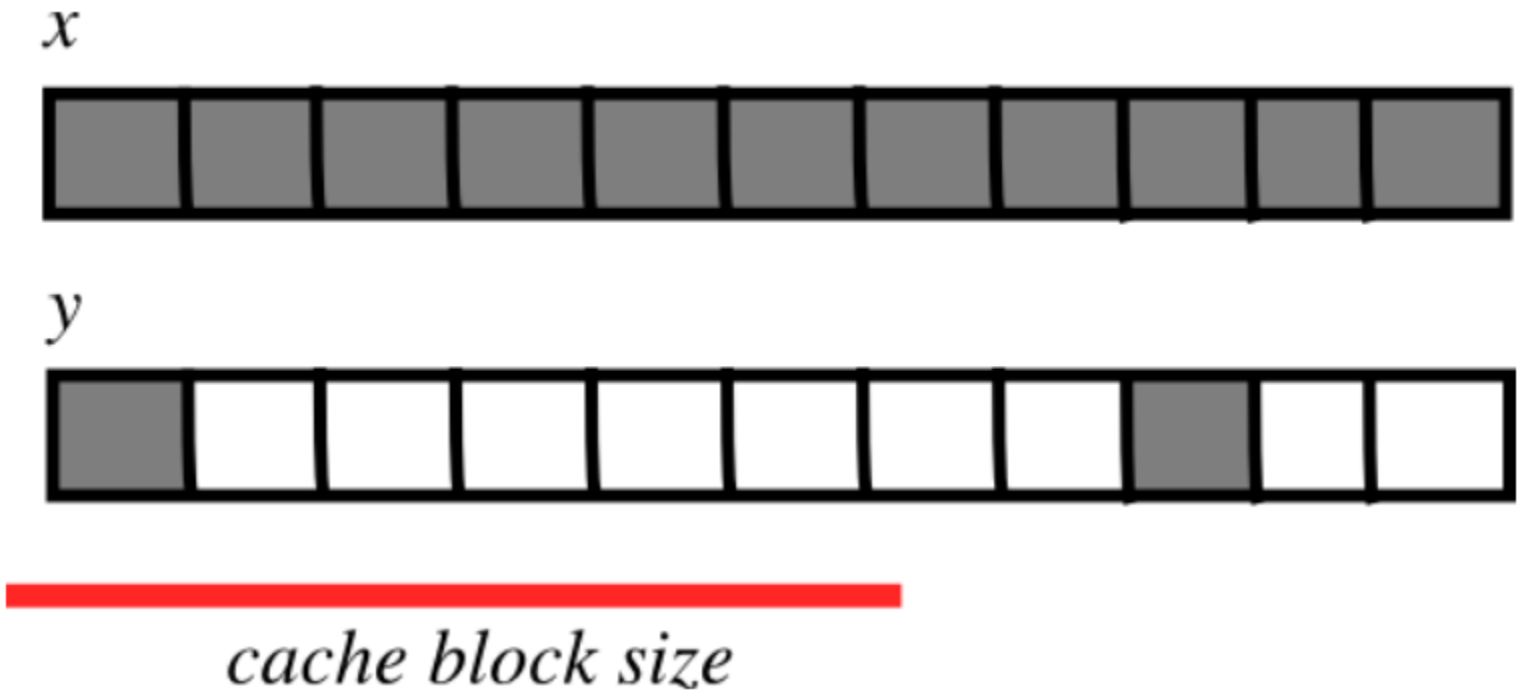
```
>>> from numpy import vectorize
>>> vsinc = vectorize(sinc)
>>> vsinc(x)
array([-0.1981, -0.2122])

>>> x2 = linspace(-5, 5, 101)
>>> plot(x2, sinc(x2))
```



# Memory Layout / Performance

```
In [1]: x = np.zeros((20000,))

In [2]: y = np.zeros((20000*67,))[::67]

In [3]: x.shape, y.shape
((20000,), (20000,))

In [4]: %timeit x.sum()
100000 loops, best of 3: 0.180 ms per loop

In [5]: %timeit y.sum()
100000 loops, best of 3: 2.34 ms per loop

In [6]: x.strides, y.strides
((8,), (536,))
```

# Cache



# Stochastics

```
In [279]: import numpy.random as random

In [280]: random.
random.Lock                 random.logistic             random.random_integers
random.RandomState          random.lognormal            random.random_sample
random.Tester               random.logseries            random.ranf
random.absolute_import      random.mtrand               random.rayleigh
random.bench                random.multinomial          random.sample
random.beta                 random.multivariate_normal  random.seed
random.binomial             random.negative_binomial    random.set_state
random.bytes                random.noncentral_chisquare random.shuffle
random.chisquare            random.noncentral_f         random.standard_cauchy
random.choice               random.normal               random.standard_exponential
random.dirichlet            random.np                   random.standard_gamma
random.division             random.operator             random.standard_normal
random.exponential          random.pareto               random.standard_t
random.f                    random.permutation          random.test
random.gamma                random.poisson              random.triangular
random.geometric            random.power                random.uniform
random.get_state            random.print_function       random.vonmises
random.gumbel               random.rand                 random.wald
random.hypergeometric       random.randint              random.warnings
random.info                 random.randn                random.weibull
random.laplace              random.random               random.zipf
```

# numpy.random.rand¶

**numpy.random.rand(***d0, d1, ..., dn***)**

Random values in a given shape.

Create an array of the given shape and populate it with random samples from a uniform distribution over `[0, 1)`.

**Parameters:** **d0, d1, ..., dn** : *int, optional*
The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

**Returns:** **out** : *ndarray, shape* `(d0, d1, ..., dn)`
Random values.

**See also:**
random

## Notes

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random_sample .

## Examples

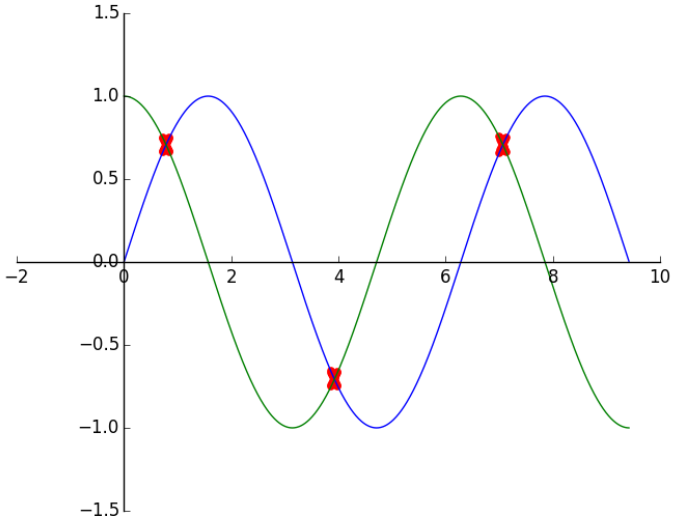
```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

# Exercises (no loops)

- Create an array of 25 random floating-point numbers in the range [0, 100] and set the highest element equal to 200 and all numbers smaller than 50 equal to 0.
- Create a matrix 9x9 of random numbers (int32) in the range of -100 to 100. Using this matrix create an array that only contains the even elements sorted in ascending order.

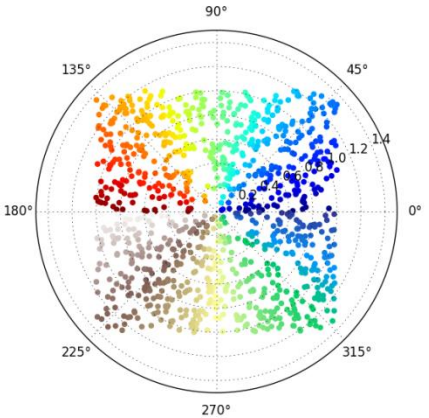
# Exercise (no loops)

- Create a graph of the sine and cosine functions in the range from 0 do 3*PI. Evaluate where the function graphs are close to each other (distance < 0.1) and indicate this using points.



# Exercise (no loops)

- Create a 500x2 matrix containing 500 2D points in the range $[-1,1]^2$. Transform the points (x, y) to points (t, r) in the polar coordinate system. Visualize the points as a scatter plot, such that the points are colored with the color map *jet* if the angle is t < 180 - and with the color map *terrain* in the converse case.



# Exercise

- Download the file populacje.txt which contains population numbers of rabbits, foxes and carrots for a given time period at an undisclosed location. Report the year in which each species had the greatest population number; for each year which species had the greatest population; the years where the total population of all species was greater than 100,000.