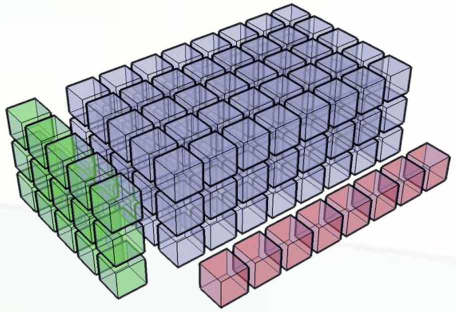
Python-Numpy

Dr. Sarwan Singh

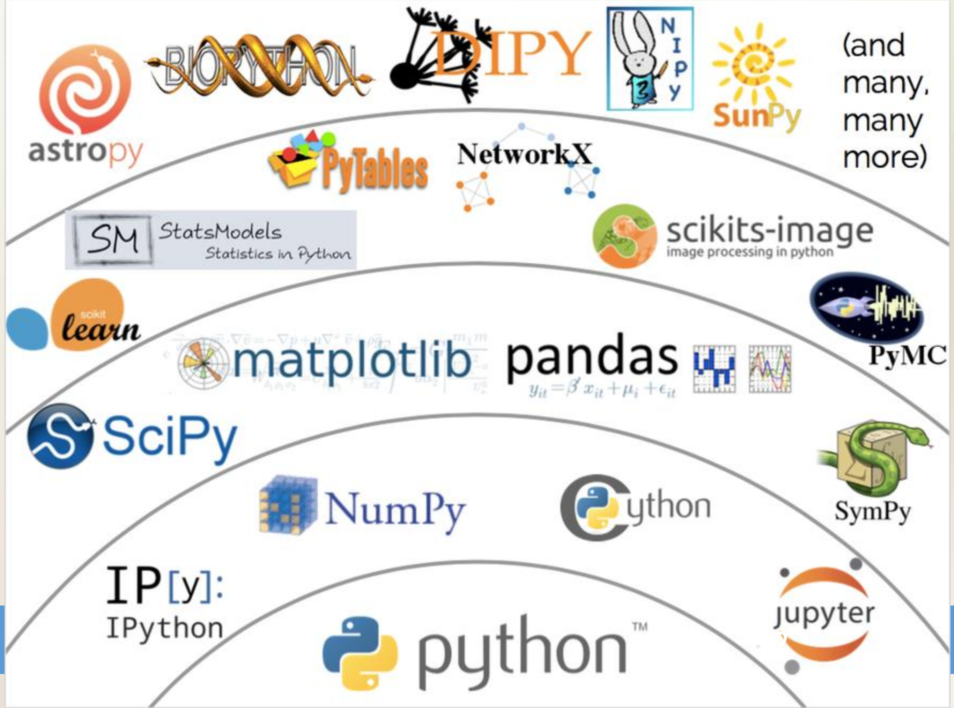
NumPy - Numerical **Python**,
library consisting of multidimensional array objects and a collection of routines for processing those arrays.

Agenda

NumPy
Numerical Python

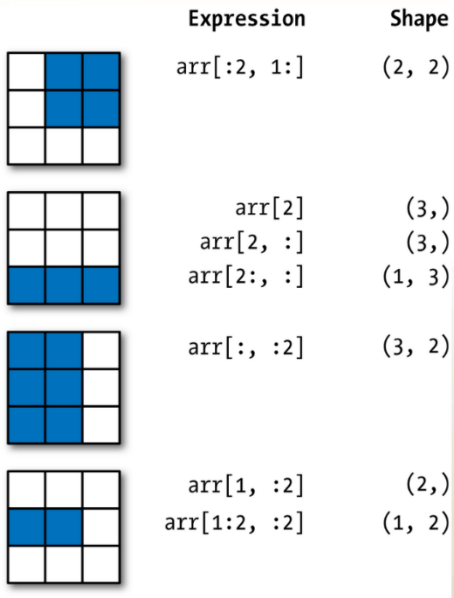


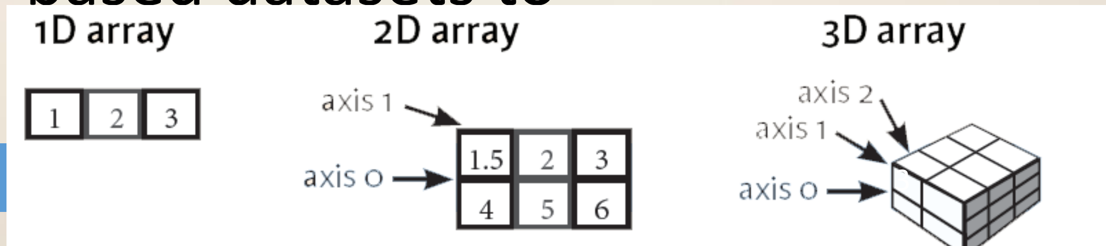
- Introduction
- History, usage
- Universal Functions
- Indexing, Slicing and Iterating
- Stacking -splitting arrays
- Broadcasting
- Reading from csv files



Introduction

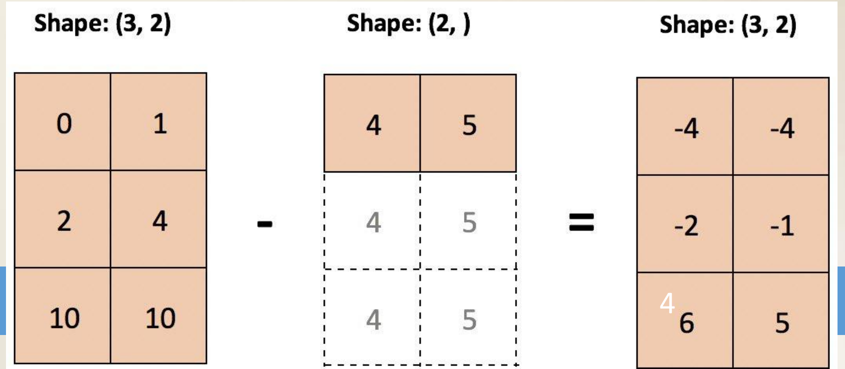
- Numerical Python (Numpy) has greater role for numerical computing in Python.
- It provides the data structures, algorithms, and library glue needed for most scientific applications involving numerical data in Python.
- It has fast and efficient multidimensional (N-dimensional) array object ndarray
- Functions for performing element-wise computations with arrays or mathematical operations between arrays
- It has tools for reading and writing array-based datasets to disk





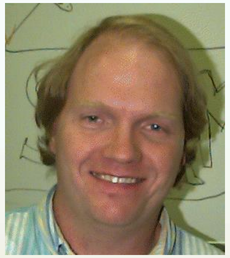
Introduction

- It is useful for Linear algebra operations, Fourier transform, and random number generation
- It has sophisticated (broadcasting) functions
- It has tools for integrating C/C++ and Fortran code
- It provides an efficient interface to store and operate on dense data buffers
- NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python
- Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.
- NumPy is licensed under the [BSD license]



History

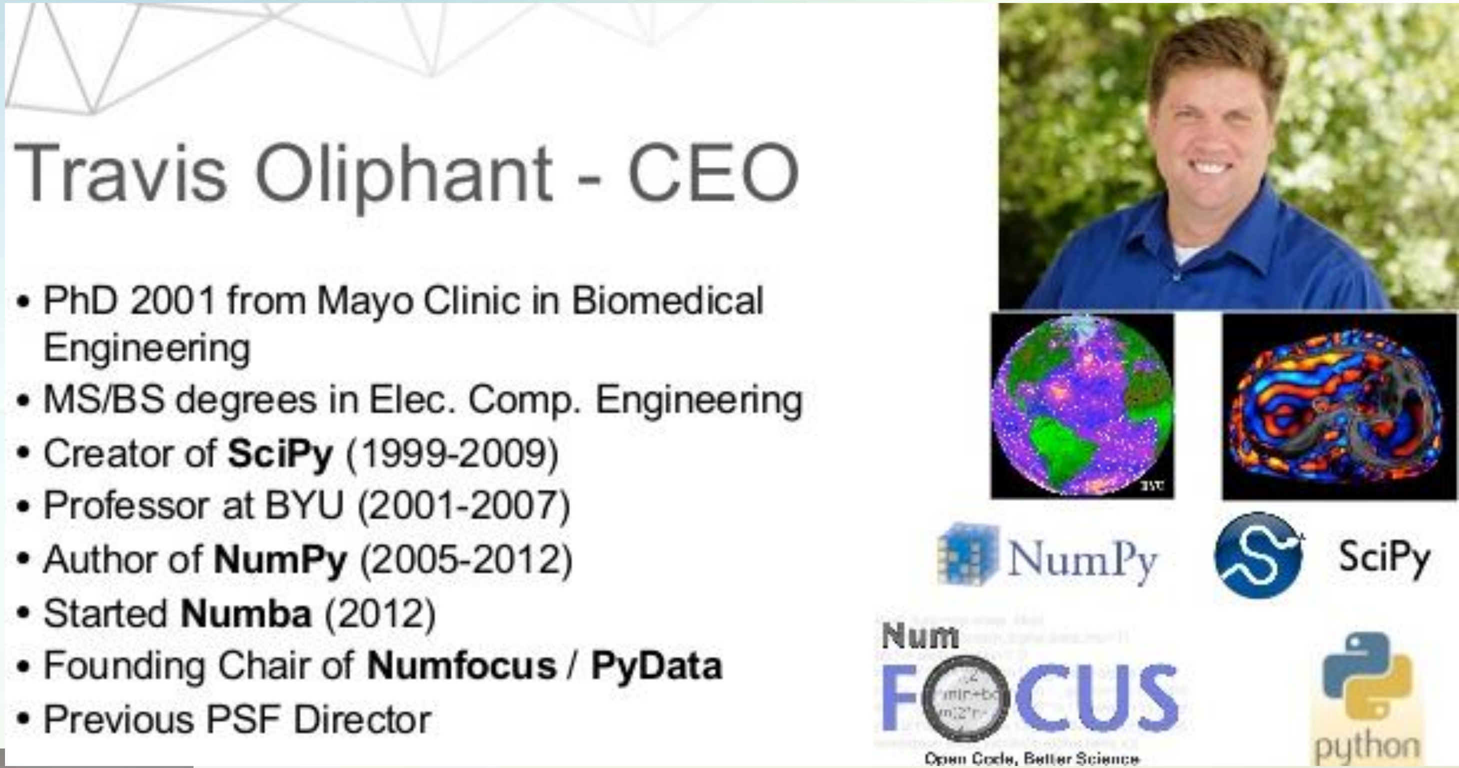
- **NumPy** derives from an old library called Numeric, which was the first array object built for Python. (written in 2005 launched in 2006)
- Numeric was quite successful and was used in a variety of applications before being phased out.

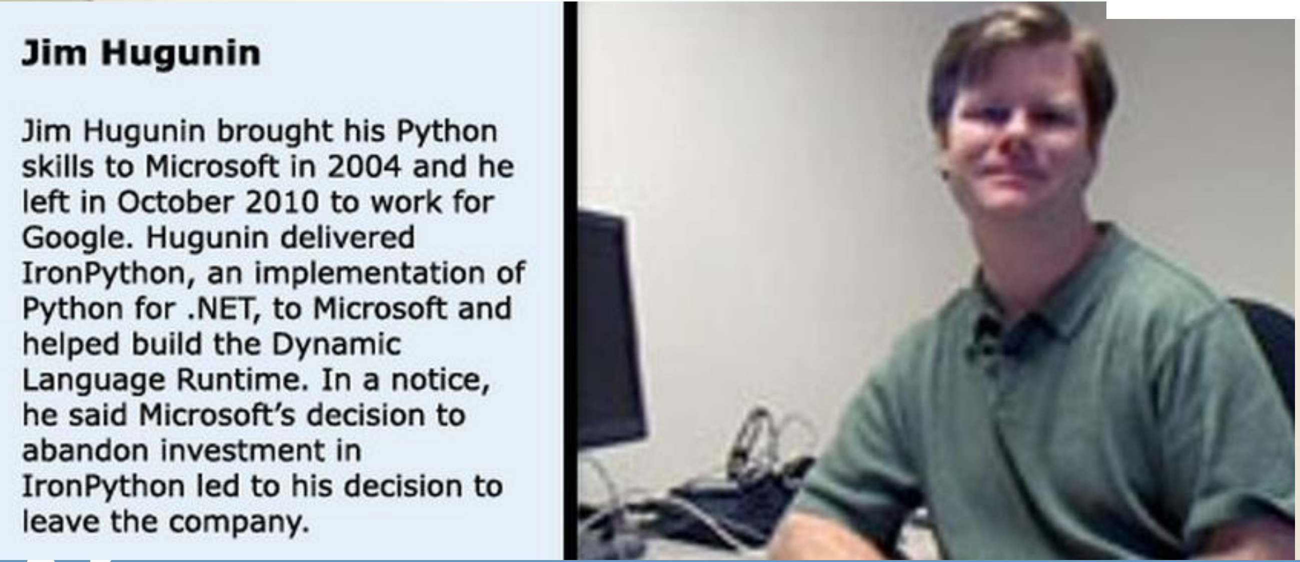


Jim Fulton

Jim Hugunin

Person	Package	Year
Jim Fulton	Matrix Object in Python	1994
Jim Hugunin	Numeric	1995
Perry greenfield, Rick white, Todd Miller	Numarray	2001
Travis Olipant	NumPy	2005


Travis Oliphant - CEO
• PhD 2001 from Mayo Clinic in Biomedical Engineering
• MS/BS degrees in Elec. Comp. Engineering
• Creator of SciPy (1999-2009)
• Professor at BYU (2001-2007)
• Author of NumPy (2005-2012)
• Started Numba (2012)
• Founding Chair of Numfocus / PyData
• Previous PSF Director
NumPy
SciPy
Num FOCUS
Open Code, Better Science
python


Jim Hugunin
Jim Hugunin brought his Python skills to Microsoft in 2004 and he left in October 2010 to work for Google. Hugunin delivered IronPython, an implementation of Python for .NET, to Microsoft and helped build the Dynamic Language Runtime. In a notice, he said Microsoft's decision to abandon investment in IronPython led to his decision to leave the company.

- NumPy’s main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.
- In NumPy dimensions are called *axes*.
- NumPy defines N-dimensional array type called ndarray(also known by the alias array). It describes the collection of items of the same type. Items in the collection can be accessed using a zero-based index.
- numpy.array is not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays.

Syntax

- numpy.array(
  object,
  dtype = None,
  copy = True,
  order = None,
  subok = False,
  ndmin = 0)

dtype : Desired data type of array, optional

copy : Optional. By default (true), the object is copied

order : C (row major) or F (column major) or A (any) (default)

subok : By default, returned array forced to be a base class array. If true, sub-classes passed through

ndmin : Specifies minimum dimensions of resultant array

```
import numpy as np
```

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

```
a
```

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

```
print("Shape    : " ,a.shape )
print("Size     : " ,a.size )
print("Datatype : " ,a.dtype )
print("ndim     : " ,a.ndim )
```

```
Shape    :  (3,)
Size     :  3
Datatype :  int32
ndim     :  1
```

```
aa=np.array( [ [1,2,3,4],[5,6,7,8] ])
```

```
aa
```

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

```
print("Shape    : " ,aa.shape )
print("Size     : " ,aa.size )
print("Datatype : " ,aa.dtype )
print("ndim     : " ,aa.ndim )
```

```
Shape    :  (2, 4)
Size     :  8
Datatype :  int32
ndim     :  2
```

a = np.array(1,2,3,4)   # WRONG

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

- The function zeros creates an array full of zeros,
- the function ones creates an array full of ones,
- the function empty creates an array whose initial content is random and depends on the state of the memory.
- By default, the dtype of the created array is float64.
- np.ones((2,3,4), dtype=np.int16)
- np.zeros((3,4))
- np.empty((2,3))

```
a2 = np.array([1, 2, 3, 4], ndmin = 2)
```

```
print("Array    : " ,a2 )
print("Shape    : " ,a2.shape )
print("Size     : " ,a2.size )
print("Datatype : " ,a2.dtype )
print("ndim     : " ,a2.ndim )
```

```
Array    :  [[1 2 3 4]]
Shape    :  (1, 4)
Size     :  4
Datatype :  int32
ndim     :  2
```

```
aa
```

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

```
aa[1,3]
```

```
8
```

```
aa[1]
```

```
array([5, 6, 7, 8])
```

```
aa[1,3]=20
```

```
aa
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7, 20]])
```

Printing Array

- When you print an array, NumPy displays it in a similar way to nested lists

```
a = np.arange(6) # 1d array
print(a)

[0 1 2 3 4 5]

b = np.arange(12).reshape(4,3) # 2d array
print(b)

[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]

c = np.arange(24).reshape(2,3,4) # 3d array
print(c)

[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Basic Operations

- Arithmetic operators on arrays apply *elementwise*. A new array is created and filled with the result
  - * , + , +=, *=
  - Dot

```
A-B
```

```
array([[-4, -4],
       [-4, -4]])
```

```
A **2
```

```
array([[0, 1],
       [4, 9]], dtype=int32)
```

```
B <6
```

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

```
A=np.arange(4).reshape(2,2)
B=np.array([[4,5],[6,7]])
print("A \n",A,"\n B\n",B, "\nA+B\n", A+B)
```

```
A
 [[0 1]
 [2 3]]
 B
 [[4 5]
 [6 7]]
A+B
 [[ 4  6]
 [ 8 10]]
```

```
# elementwise product
print("A \n",A,"\n B\n",B, "\nA*B\n", A*B)
```

```
A
 [[0 1]
 [2 3]]
 B
 [[4 5]
 [6 7]]
A*B
 [[ 0  5]
 [12 21]]
```

```
# matrix product
print("A \n",A,"\n B\n",B, "\nA*B\n", A.dot(B) )
```

```
A
 [[0 1]
 [2 3]]
 B
 [[4 5]
 [6 7]]
A*B
 [[ 6  7]
 [26 31]]
```

- numpy.arange(start, stop, step, dtype)
- numpy.linspace(start, stop, num, endpoint, retstep, dtype) - the number of evenly spaced values between the specified interval

```
print("A \n",A)
print("B \n",B)
print("A.max " , A.max())
print("A.min " , A.min())
print("A.sum " , B.sum())
```

```
A
 [[0 1]
 [2 3]]
B
 [[4 5]
 [6 7]]
A.max  3
A.min  0
A.sum  22
```

```
B.min(axis=1)
```

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

```
B.cumsum(axis=1)
```

```
array([[ 4,  9],
       [ 6, 13]], dtype=int32)
```

Universal Functions

- NumPy provides familiar mathematical functions such as sin, cos, and exp. In NumPy, these are called “universal functions”(ufunc).
- Within NumPy, these functions operate elementwise on an array, producing an array as output.

```
np.sqrt(A)
```

```
array([[ 0.        ,  1.        ],
       [ 1.41421356,  1.73205081]])
```

```
np.exp(A)
```

```
array([[  1.        ,   2.71828183],
       [  7.3890561 ,  20.08553692]])
```

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

```
array([[ 4,  6],
       [ 8, 10]])
```

Unary ufuncs

Universal functions

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating-point, or complex values
`sqrt`	Compute the square root of each element (equivalent to `arr ** 0.5`)
`square`	Compute the square of each element (equivalent to `arr ** 2`)
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or –1 (negative)
`ceil`	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
`floor`	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as a separate array
`isnan`	Return boolean array indicating whether each value is `NaN` (Not a Number)

Source:
Python for Data Analysis

Binary universal functions

Function	Description
add	Add corresponding elements in arrays
subtract	Subtract elements in second array from first array
multiply	Multiply array elements
divide, floor_divide	Divide or floor divide (truncating the remainder)
power	Raise elements in first array to powers indicated in second array
maximum, fmax	Element-wise maximum; fmax ignores NaN
minimum, fmin	Element-wise minimum; fmin ignores NaN
mod	Element-wise modulus (remainder of division)
copysign	Copy sign of values in second argument to values in first argument
greater, greater_equal, less, less_equal, equal, not_equal	Perform element-wise comparison, yielding boolean array (equivalent to infix operators >, >=, <, <=, ==, !=)
logical_and, logical_or, logical_xor	Compute element-wise truth value of logical operation (equivalent to infix operators & |, ^)

Source:
Python for Data Analysis

Indexing, Slicing and Iterating

- shape
- reshape () - resize an array.
- Itemsize - length of each element of array in bytes.

```
A = np.array([[1,2,3],[4,5,6]])
print(A)
```

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

```
B = A.reshape(3,2)
print(B)
```

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

```
A = np.arange(24)
print(A)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 1
```

```
B= A.reshape(2,3,4)
print(B)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

- The dots (...) represent as many colons as needed to produce a complete indexing tuple.

```
print(B)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

```
B[:,:,1]
```

```
array([[ 1,  5,  9],
       [13, 17, 21]])
```

```
B[...,1]
```

```
array([[ 1,  5,  9],
       [13, 17, 21]])
```

```
B[1,...]
```

```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

```
B[1]
```

```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

```
B[1,:,:]
```

```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

```
#iterating
for i in B:
    print(i)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

```
for i in B.flat:
    print(i)
```

```
0
1
2
3
4
5
6
7
8
```

Slicing

```
print(A)
```

```
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]]
```

```
#slicing
print(A[1:3,3:6] )
```

```
[[11 12 13]
 [19 20 21]]
```

```
#Slicing using advanced index for column
print(A[1:3,[3,4,5]] )
```

```
[[11 12 13]
 [19 20 21]]
```

```
print(A[A>10])
```

```
[11 12 13 14 15 16 17 18 19 20 21 22 23]
```

```
B.ravel()  # returns the array, flattened
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13
       17, 18, 19, 20, 21, 22, 23])
```

```
B.T  # returns the array, transposed
```

```
array([[[ 0, 12],
        [ 4, 16],
        [ 8, 20]],

       [[ 1, 13],
        [ 5, 17],
        [ 9, 21]],

       [[ 2, 14],
        [ 6, 18],
        [10, 22]],

       [[ 3, 15],
        [ 7, 19],
        [11, 23]]])
```

Slice is a View

- Changing value in slice of array will change the original array values

```
print (arr)
arr[0:2,:]
```

```
[[ 1  2  3  4]
 [ 9  8  7  6]
 [19 18 17 16]]
array([[1, 2, 3, 4],
       [9, 8, 7, 6]])
```

```
arr1 = arr[1:2,1:]
```

```
print (arr1)
arr1[0][1] = 55
print (arr1)
print(arr)
```

```
[[8 7 6]]
[[ 8 55  6]]
[[ 1  2  3  4]
 [ 9  8 55  6]
 [19 18 17 16]]
```

Stacking -splitting arrays

```
a = np.floor(10*np.random.random((2,2)))
print(a)
b = np.floor(8*np.random.random((2,2)))
print(b)
```

```
[[ 1.  4.]
 [ 5.  0.]]
[[ 3.  1.]
 [ 0.  1.]]
```

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

```
array([[ 1.,  4.,  3.,  1.],
       [ 5.,  0.,  0.,  1.]])
```

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

```
array([[ 1.,  4.],
       [ 5.,  0.],
       [ 3.,  1.],
       [ 0.,  1.]])
```

```
c=np.hstack((a,b))
np.hsplit(c,2)
```

```
[array([[ 1.,  4.],
        [ 5.,  0.]]), array([[ 3.,  1.],
        [ 0.,  1.]])]
```

Array arithmetic

```
x = np.arange(4)
print("x =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
print("x / 2 =", x / 2)
print("x // 2 =", x // 2) # floor division
```

```
x = [0 1 2 3]
x + 5 = [5 6 7 8]
x - 5 = [-5 -4 -3 -2]
x * 2 = [0 2 4 6]
x / 2 = [ 0.   0.5  1.   1.5]
x // 2 = [0 0 1 1]
```

```
print("-x = ", -x)            # unary ufunc for negation
print("x ** 2 = ", x ** 2)  # ** operator for exponentiation
print("x % 2 = ", x % 2)     #  % operator for modulus
```

```
-x =  [ 0 -1 -2 -3]
x ** 2 =  [0 1 4 9]
x % 2 =  [0 1 0 1]
```

Array Striding

numpy.ndarray.strides

- strides : how many bytes to skip in memory to move to the next position along a certain axis.
- a.flags.contiguous returns true if the memory is contiguously allocated
  - In case of python it is true
  - Not in case of Fortran

```
[29] print (arr)
     print("Type       : " ,type(arr))
     print("Item Size : " ,arr.itemsize)
     print("dtype      : " ,arr.dtype)
     print("arr.strides: " ,arr.strides)
```

```
[[ 1  2  3  4]
 [ 9  8  7  6]
 [19 18 17 16]]
Type       :  <class 'numpy.ndarray'>
Item Size :  8
dtype      :  int64
arr.strides:  (32, 8)
```

Array Striding

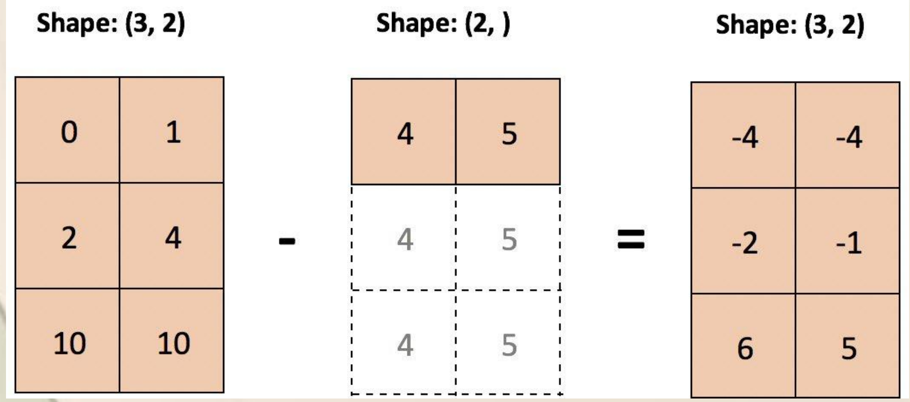
```
[24] print (arr)

     print(arr[:,::2])
     print(arr[:,::1])
     print(arr[:,::4])
```

```
[[ 1  2  3  4]
 [ 9  8  7  6]
 [19 18 17 16]]
[[ 1  3]
 [ 9  7]
 [19 17]]
[[ 1  2  3  4]
 [ 9  8  7  6]
 [19 18 17 16]]
[[ 1]
 [ 9]
 [19]]
```

Broadcasting

- **Broadcasting** refers to the ability of NumPy to treat arrays of different shapes during arithmetic operations. Arithmetic operations on arrays are usually done on corresponding elements.




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

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

```
B= np.arange(5,8)
print(B)
```

```
[5 6 7]
```

```
A+B
```

```
array([[ 5,  7,  9],
       [ 8, 10, 12],
       [11, 13, 15],
       [14, 16, 18]])
```

Computation using numpy

```
import numpy as np
np.random.seed(0)
```

```
def compute_reciprocals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1.0/values[i]
    return output
```

```
values = np.random.randint(1,10, size=5)
print(values)
```

```
[6 1 4 4 8]
```

```
print(compute_reciprocals(values))
```

```
[ 0.16666667  1.          0.25        0.25        0.125     ]
```

```
%timeit compute_reciprocals(values)
```

```
9.19 µs ± 146 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
big_array = np.random.randint(1,100,size=1000000)
%timeit compute_reciprocals(big_array)
```

```
1.55 s ± 17.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit (1.0/big_array)
```

```
3.38 ms ± 26.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Computation using numpy

```
L = np.random.random(100)
```

```
sum(L)
```

```
50.461758453195614
```

```
np.sum(L)
```

```
50.461758453195642
```

```
big_array = np.random.rand(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)
```

```
129 ms ± 1.19 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
1.07 ms ± 22.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
min(big_array), max(big_array)
```

```
(7.0712031718933588e-07, 0.99999972076563337)
```

```
np.min(big_array), np.max(big_array)
```

```
(7.0712031718933588e-07, 0.99999972076563337)
```

```
%timeit min(big_array)
%timeit np.min(big_array)
```

```
51.9 ms ± 430 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
456 µs ± 3.29 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```