Basic Python

ML4HEP 2026 Pre-school Lectures

Subir Sarkar, SINP

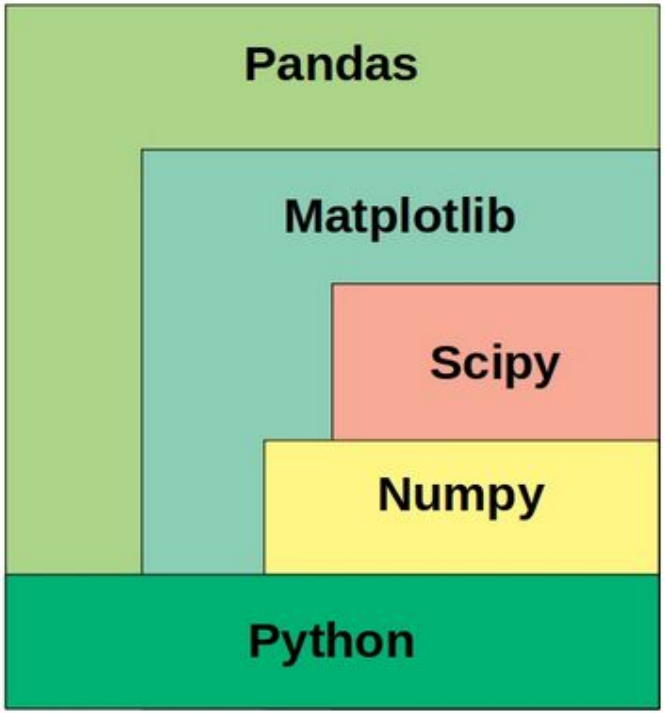
Lecture 7, 14/05/2026

What is Numpy

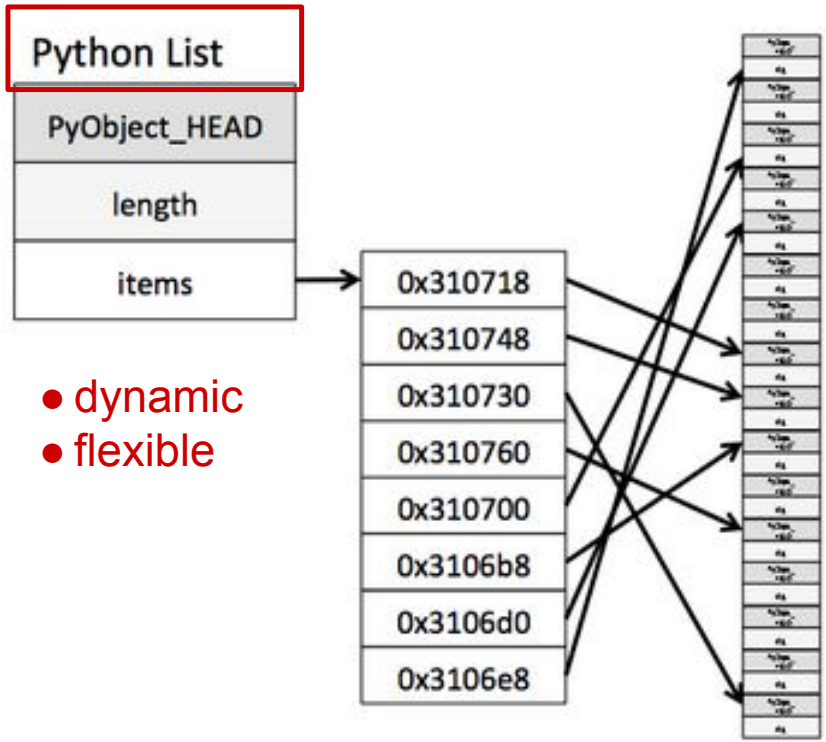
- Numpy is a C extension module for array oriented computing
  - adds powerful data structures like multi-dimensional arrays
    - fixed size arrays holding data of the same underlying type for memory and speed optimization
  - precompiled mathematical and numerical functions for performance
  - performance similar to C with flexibility of python

What is Numpy

- Numpy is a C extension module for array oriented computing
  - adds powerful data structures like multi-dimensional arrays
    - fixed size arrays holding data of the same underlying type for memory and speed optimization
  - precompiled mathematical and numerical functions for performance
  - performance similar to C with flexibility of python
- Numpy forms the basis of scientific computing in Python
  - Linear Algebra
  - Image and signal processing
  - Machine learning



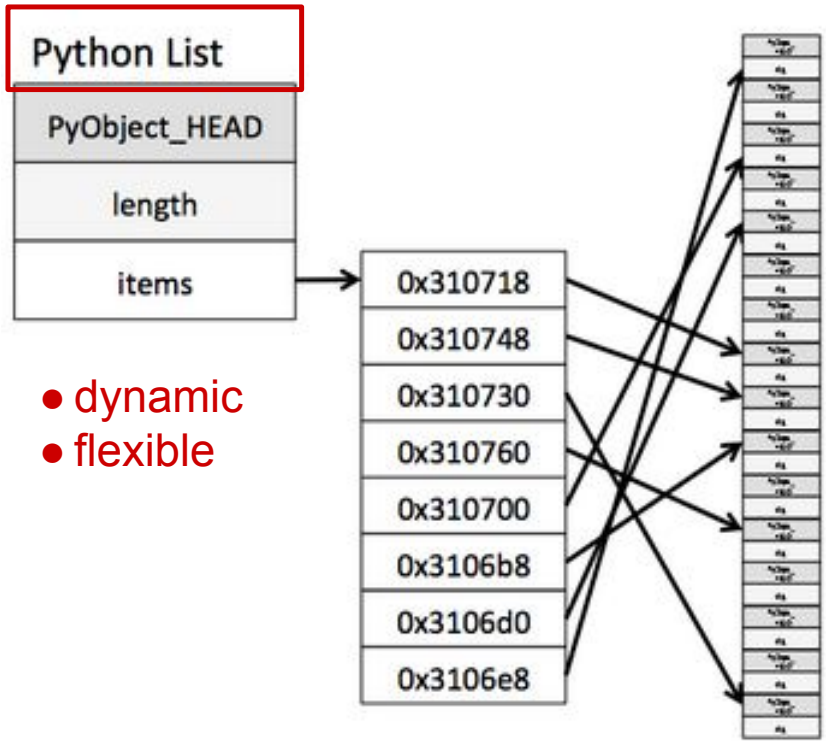
List vs Numpy Array



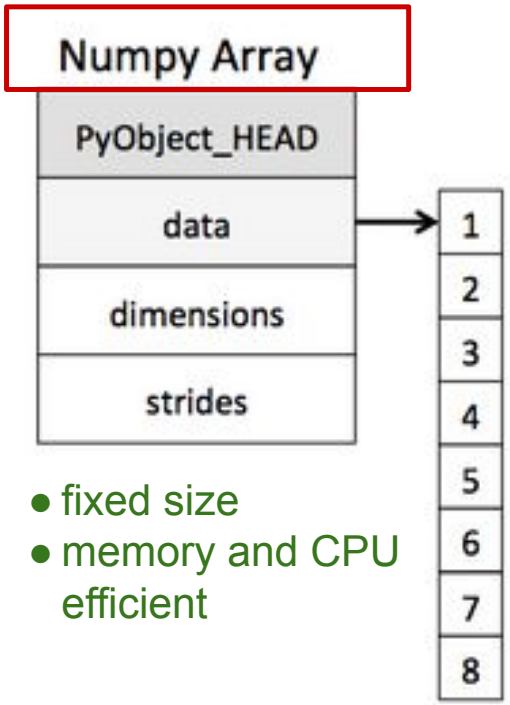
- dynamic
- flexible

https://jakevdp.github.io/PythonDataScienceHandbook/02.01-understanding-data-types.html

List vs Numpy Array



- dynamic
- flexible



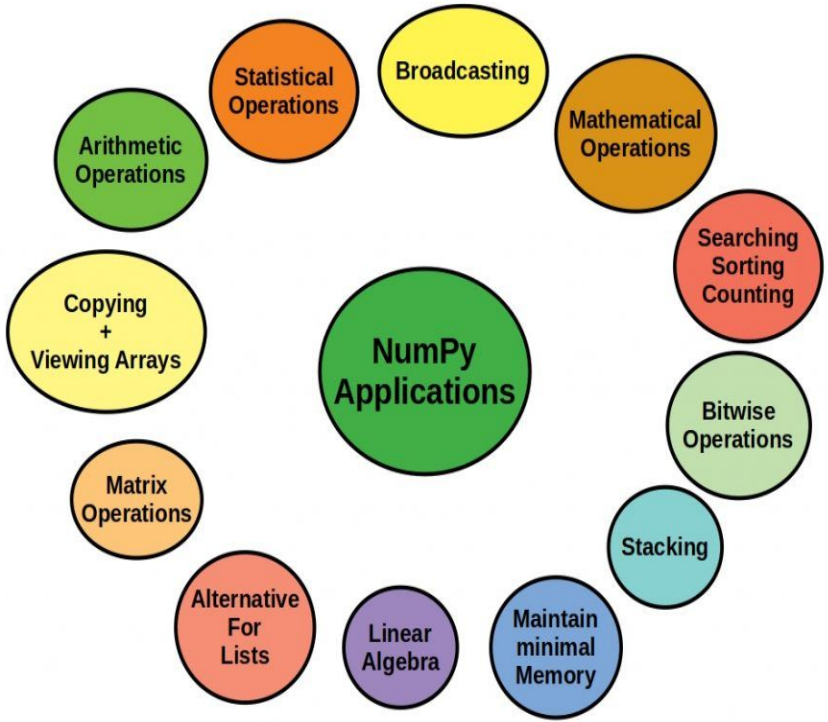
- fixed size
- memory and CPU efficient

https://jakevdp.github.io/PythonDataScienceHandbook/02.01-understanding-data-types.html

- The *strides* of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis.

Numpy Features

- Smaller memory footprint
- CPU efficiency
- Natural look and feel (e.g algebraic operation)
  - vectorization
  - broadcasting
  - operator overloading



https://starship-knowledge.com/awesome-python-data-science-libraries

Vectorization

- Modern CPUs have "vector" or Single Instruction Multiple Data (SIMD) capabilities which apply the same operation simultaneously to two, four, or more pieces of data
- An over-simplified view of Vectorization
  - process of rewriting a loop to process 4 elements of the array of size N simultaneously N/4 times instead of doing one element at a time

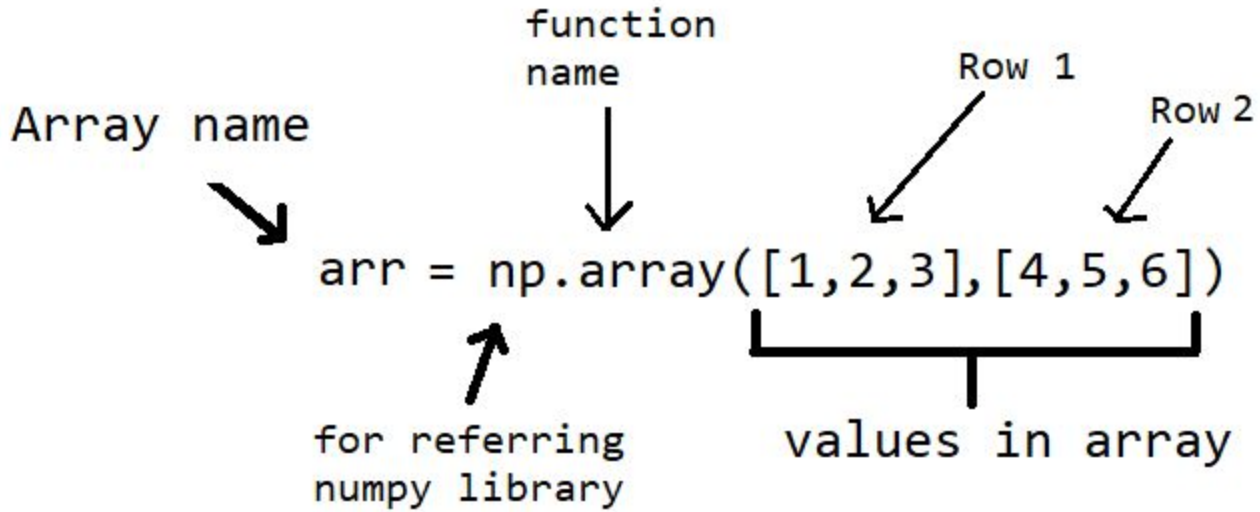
client code:

```
const int N = 16;
int A[N], B[N], C[N];
for (int i = 0; i < N; ++i)
  C[i] = A[i] + B[i];
```

will be unrolled as, assuming 4 independent SIMD instructions

```
for (int i = 0; i < N; i += 4) {
  C[i]   = A[i]   + B[i];
  C[i+1] = A[i+1] + B[i+1];
  C[i+2] = A[i+2] + B[i+2];
  C[i+3] = A[i+3] + B[i+3];
}
```

Arrays from Python lists



numpy.array

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

In [6]: np.array([1, 2, 3]) # one can also use a tuple
Out[6]: array([1, 2, 3])

In [3]: np.array([[1, 2], [3, 4]])
Out[3]: array([[1, 2],
               [3, 4]])

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

ndarray (alias array)

ndarray Attributes	Description
ndarray.ndim	number of axes (dimensions) of the array
ndarray.shape	dimensions of the array
ndarray.size	total number of elements of the array
ndarray.dtype	type of the elements in the array
ndarray.itemsize	size in bytes of each element of the array
ndarray.data	buffer containing the actual elements of the array

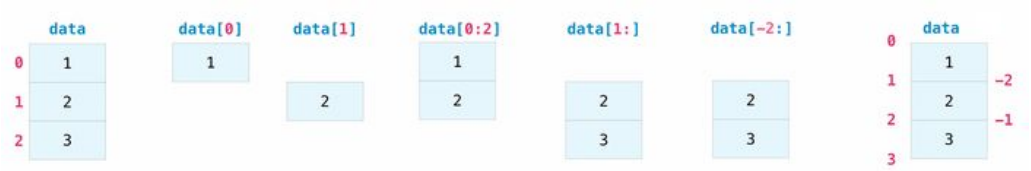
```
>>> import numpy as np
>>> a = np.array([[0,1,2,3,4],[5,6,7,8,9],[10,11,12,13,14]])

>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.ndim
2
>>> a.shape
(3, 5)
```

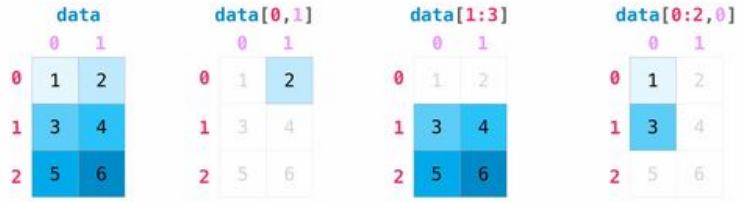
```
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<class 'numpy.ndarray'>
```

Array Access

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

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

In [6]: print(a[1])
2

In [5]: print(a[0:2])
[1 2]

In [8]: print(a[1:])
[2 3]

In [9]: print(a[-2])
2
```

```
In [11]: m = np.array([[1, 2], [3, 4], [5, 6]])

In [12]: print(m)
[[1 2]
 [3 4]
 [5 6]]

In [14]: print(m[0])
[1 2]

In [15]: print(m[0,1])
2

In [16]: print(m[1:3])
[[3 4]
 [5 6]]

In [17]: print(m[0:2,0])
[1 3]
```





https://numpy.org/doc/stable/user/absolute_beginners.html

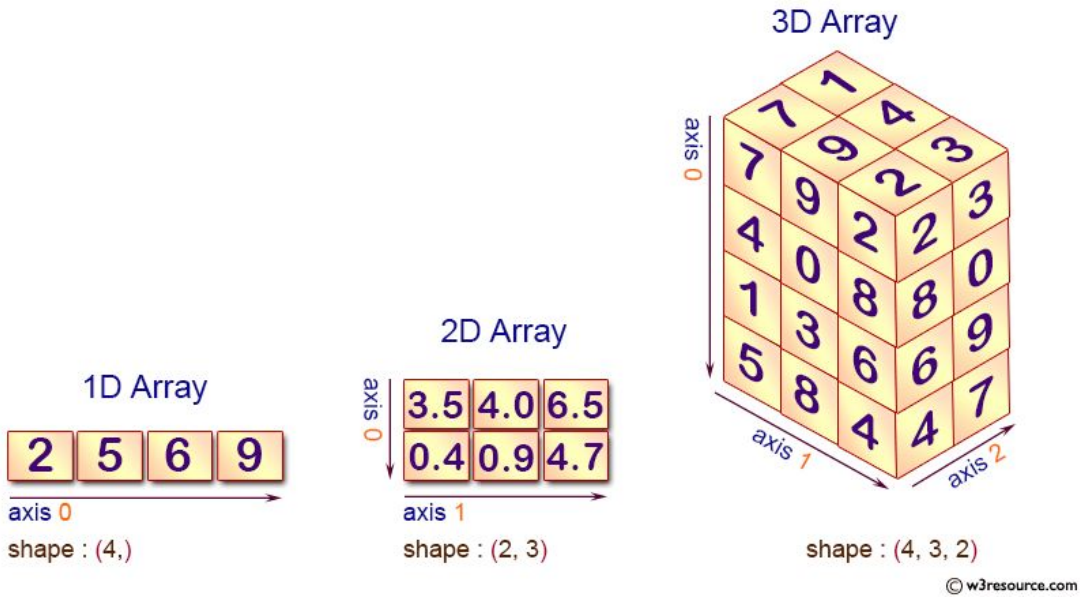
Array Elements w/ conditions

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

In [8]: a = np.arange(1,13, dtype=np.int64).reshape(3,4)

In [14]: print(a[a < 9])
         [1 2 3 4 5 6 7 8]

In [15]: even_only = a%2==0
         print(a[even_only])
         [ 2  4  6  8 10 12]

In [16]: print(a[(a > 3) & (a < 10)]) # note the convention
         [4 5 6 7 8 9]

In [17]: print(np.nonzero(a < 6))
         (array([0, 0, 0, 0, 1]), array([0, 1, 2, 3, 0]))

In [18]: print(np.count_nonzero(a < 6))
         5
```

also see: `numpy.flatnonzero(array)`, numpy.where(condition, …), `numpy.argwhere(array)`

array axes

https://www.w3resource.com/numpy/array-creation/index.php



Arrays from scratch

numpy.arange

```
numpy.arange([start, ]stop, [step, ]dtype=None, *, like=None)
```

```
In [79]: np.arange(15, dtype=np.float64)
Out[79]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.,
                13., 14.])

In [70]: np.arange(0, 20, 2)
Out[70]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

In [76]: np.arange(1, 2, 0.1)
Out[76]: array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])

In [77]: np.arange(10, 1, -1)
Out[77]: array([10,  9,  8,  7,  6,  5,  4,  3,  2])

In [78]: np.arange(10, 0, -3)
Out[78]: array([10,  7,  4,  1])
```

Python List vs Numpy Array Memory Footprint

In [8]:
```
import numpy as np
from sys import getsizeof as sizeof

print(sizeof(999)) # size of an integer
```

```
28
```

In [9]:
```
# Empty List
a = []
print(sizeof(a))
```

```
56
```

In [10]:
```
# list with N elements
b = list(range(1000))
print(sizeof(b)) # empty list + size of the pointers to 1000 integer objects
```

```
8056
```

In [11]:
```
# Numpy array
c = np.arange(1000)
print(c.size * c.itemsize)  +  numpy.array overhead
```

```
8000
```

Python List vs Numpy Array CPU Performance

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

# size of arrays and lists
size = 1000000

# lists
lsta = range(size)
lstb = range(size)

# numpy arrays
arra = np.arange(size)
arrb = np.arange(size)

# multiply the lists and store the result in another list
%timeit lstc = [(a * b) for a, b in zip(lsta, lstb)]
```

```
79.8 ms ± 4.07 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [2]:
```
# multiply the Numpy arrays and store the result in another Numpy array
%timeit arrc = arra * arrb
```

```
1.89 ms ± 470 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

reshape, resize

```
In [135]: a = np.floor(10 * np.random.random((3, 4)))

In [136]: print(a)
[[6. 3. 0. 7.]
 [0. 8. 4. 9.]
 [2. 0. 9. 2.]]

In [137]: a.ravel() # view
Out[137]: array([6., 3., 0., 7., 0., 8., 4., 9., 2., 0., 9., 2.])

In [144]: print(a.reshape(4, 3)) # view
[[6. 3. 0.]
 [7. 0. 8.]
 [4. 9. 2.]
 [0. 9. 2.]]

In [145]: print(a.transpose()) # (a.T) view
[[6. 0. 2.]
 [3. 8. 0.]
 [0. 4. 9.]
 [7. 9. 2.]]

In [148]: a.resize((3, 4)) # changes a

In [149]: print(a)
[[6. 3. 0. 7.]
 [0. 8. 4. 9.]
 [2. 0. 9. 2.]]
```

zeros, ones, full, empty, identity

```
In [10]: import numpy as np
         from math import pi

In [9]: np.zeros(10, dtype=float)
Out[9]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [11]: np.ones((3, 5), dtype=np.float64)
Out[11]: array([[1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.]])

In [7]: np.full((3, 5), pi)
Out[7]: array([[3.14159265, 3.14159265, 3.14159265, 3.14159265, 3.14159265],
               [3.14159265, 3.14159265, 3.14159265, 3.14159265, 3.14159265],
               [3.14159265, 3.14159265, 3.14159265, 3.14159265, 3.14159265]])

In [6]: np.empty(5)
Out[6]: array([1.39758760e-316, 0.00000000e+000, 7.38191927e+271, 2.25563599e-153,
               1.41818073e+195])

In [7]: np.empty((2, 3))
Out[7]: array([[1.39824135e-316, 0.00000000e+000, 0.00000000e+000],
               [0.00000000e+000, 0.00000000e+000, 0.00000000e+000]])

In [11]: np.identity(4)
Out[11]: array([[1., 0., 0., 0.],
                [0., 1., 0., 0.],
                [0., 0., 1., 0.],
                [0., 0., 0., 1.]])
```

Random numbers

```
In [7]: np.random.random((3, 3))

Out[7]: array([[0.20469884, 0.69742749, 0.45218705],
               [0.11663043, 0.29524859, 0.74615482],
               [0.46118038, 0.31752885, 0.88003084]])

In [8]: np.random.normal(0, 1, (3, 3))

Out[8]: array([[-1.07596625, -1.04521417,  0.94400503],
               [-0.11817752, -0.42738321, -0.23172611],
               [-1.15899737,  1.38676666, -1.64432531]])

In [3]: np.random.randint(0, 100, (3, 3))

Out[3]: array([[ 7, 68, 23],
               [ 3, 85, 53],
               [ 0, 11, 75]])

In [14]: np.random.uniform(5,10,size = 4)

Out[14]: array([8.5184692 , 8.72702707, 6.82632451, 5.36786343])

In [15]: np.random.uniform(size = (2,3))

Out[15]: array([[0.83883722, 0.94502378, 0.88101416],
                [0.00827861, 0.58941644, 0.52890729]])
```

linspace, logspace, geomspace

```
In [39]: ## numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)
         np.linspace(0, 1, num=20) # Generate evenly spaced numbers

Out[39]: array([0.     , 0.05263, 0.10526, 0.15789, 0.21053, 0.26316, 0.31579,
                0.36842, 0.42105, 0.47368, 0.52632, 0.57895, 0.63158, 0.68421,
                0.73684, 0.78947, 0.84211, 0.89474, 0.94737, 1.     ])

In [40]: ## numpy.logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None, axis=0)
         np.logspace(2.0, 3.0, num=4)

Out[40]: array([ 100.     ,  215.44347,  464.15888, 1000.     ])

In [42]: y = np.linspace(2.0, 3.0, num=4)
         (10**y).astype(np.float64)

Out[42]: array([ 100.     ,  215.44347,  464.15888, 1000.     ])

In [43]: ## numpy.geomspace(start, stop, num=50, endpoint=True, dtype=None, axis=0)
         np.geomspace(1, 1000, num=4) # Geometric progression

Out[43]: array([   1.,   10.,  100., 1000.])

In [44]: np.geomspace(1, 256, num=9)

Out[44]: array([  1.,   2.,   4.,   8.,  16.,  32.,  64., 128., 256.])

In [45]: np.geomspace(1000, 1, num=4)

Out[45]: array([1000.,  100.,   10.,    1.])
```

numpy.eye

```
In [17]: print(np.eye(4,dtype=np.int8))
```

```
[[1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]]
```

```
In [18]: print(np.eye(4,k=1,dtype=np.int8))
```

```
[[0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]
 [0 0 0 0]]
```

```
In [19]: print(np.eye(4,k=-1,dtype=np.int8))
```

```
[[0 0 0 0]
 [1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]]
```

save/load data

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

In [2]:
```
a = np.arange(15).reshape(3,5)
np.save("matrix_data.npy", a)
```

In [3]:
```
b = np.load('matrix_data.npy')
print(b)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

In [4]:
```
np.savetxt('matrix_data.txt', a)
```

In [5]:
```
c = np.loadtxt('matrix_data.txt')
print(c)
```

```
[[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14.]]
```

Arrays operations

Vectorization and Broadcasting

- Vectorization
  - Array element-by-element computation, i.e explicit loops, indices pushed to “behind the scenes” in optimized, pre-compiled C code
  - Python code
    - clean, concise, easy to follow
    - resembles standard mathematical notation
- Broadcasting
  - describes implicit element-by-element behavior of operations on arrays of different shapes

https://numpy.org/doc/stable/user/absolute_beginners.html

Array Operations

```
In [81]: a = np.random.random((3, 2))
         print(a)
```

```
[[0.07644 0.15268]
 [0.40377 0.33181]
 [0.54763 0.92259]]
```

```
In [65]: print(a.sum(), a.sum(axis=0), a.sum(axis=1))
```

```
2.240318562104763 [1.4351  0.80522] [0.84843 0.87412 0.51778]
```

```
In [82]: print(a.min(), a.max())
```

```
0.07644475727068445 0.9225939469162023
```

```
In [67]: print(a.min(axis=0), a.max(axis=0), a.min(axis=1), a.max(axis=1))
```

```
[0.39084 0.12694] [0.5808  0.38497] [0.38497 0.29331 0.12694] [0.46346 0.5808  0.39084]
```

```
In [68]: print(a.cumsum(axis=0))
```

```
[[0.46346 0.38497]
 [1.04426 0.67829]
 [1.4351  0.80522]]
```

```
In [69]: print(a.cumsum(axis=1))
```

```
[[0.46346 0.84843]
 [0.5808  0.87412]
 [0.39084 0.51778]]
```

sort, concatenate

In [13]:
```
import numpy as np
```

In [14]:
```
a = np.array([2, 1, 5, 3, 7, 4, 6, 8])
```

In [15]:
```
print(np.sort(a))
```

```
[1 2 3 4 5 6 7 8]
```

In [16]:
```
print(np.argsort(a))
```

```
[1 0 3 5 2 6 4 7]
```

In [17]:
```
a.sort()  # in-place sort
print(a)
```

```
[1 2 3 4 5 6 7 8]
```

In [19]:
```
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
print(np.concatenate((a, b)))
```

```
[1 2 3 4 5 6 7 8]
```

In [20]:
```
x = np.array([[1, 2], [3, 4]])
y = np.array([[5, 6]])
np.concatenate((x, y), axis=0)
```

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

reverse

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

In [2]: a = np.arange(15)
        print(a)

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]

In [3]: print(np.flip(a)) # view, returns a copy

[14 13 12 11 10  9  8  7  6  5  4  3  2  1  0]

In [4]: b = np.arange(15).reshape(3,5)

In [5]: print(b)

[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]

In [6]: print(np.flip(b))

[[14 13 12 11 10]
 [ 9  8  7  6  5]
 [ 4  3  2  1  0]]

In [7]: print(np.flip(b, axis=0))

[[10 11 12 13 14]
 [ 5  6  7  8  9]
 [ 0  1  2  3  4]]

In [8]: print(np.flip(b, axis=1))

[[ 4  3  2  1  0]
 [ 9  8  7  6  5]
 [14 13 12 11 10]]
```

```
In [11]: b[1] = np.flip(b[1])

In [12]: print(b)

[[ 0  1  2  3  4]
 [ 9  8  7  6  5]
 [10 11 12 13 14]]

In [13]: b[:,1] = np.flip(b[:,1])

In [14]: print(b)

[[ 0 11  2  3  4]
 [ 9  8  7  6  5]
 [10  1 12 13 14]]
```

Finding Unique Elements

In [1]:
```
import numpy as np
a = np.array([11, 11, 12, 13, 14, 15, 16, 17, 12, 13, 11, 14, 18, 19, 20])
```

In [3]:
```
print(np.unique(a))
```

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

In [5]:
```
values, indices = np.unique(a, return_index=True)
print(indices)
```

```
[ 0  2  3  4  5  6  7 12 13 14]
```

In [8]:
```
values, count = np.unique(a, return_counts=True)
print(values, '\n', count)
```

```
[11 12 13 14 15 16 17 18 19 20]
 [3 2 2 2 1 1 1 1 1 1]
```

In [9]:
```
b = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [1, 2, 3, 4]])
print(np.unique(b, axis=0))
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Comparing Arrays

In [26]:
```
a = np.array([5e5, 1e-7, 4.000004e6])
b = np.array([5.00001e5, 1e-7, 4e6])
print(a,b)
```

```
[5.000000e+05 1.000000e-07 4.000004e+06] [5.00001e+05 1.00000e-07 4.00000e+06]
```

In [27]:
```
rtol = 1e-05
atol = 1e-08
np.abs(a - b) <= (atol + rtol * np.abs(b))
```

Out[27]:
```
array([ True,  True,  True])
```

In [30]:
```
## numpy.isclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)
np.isclose(a,b,rtol=rtol,atol=atol)
```

Out[30]:
```
array([ True,  True,  True])
```

In [31]:
```
## numpy.allclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)
np.allclose(a,b,rtol=rtol,atol=atol)
```

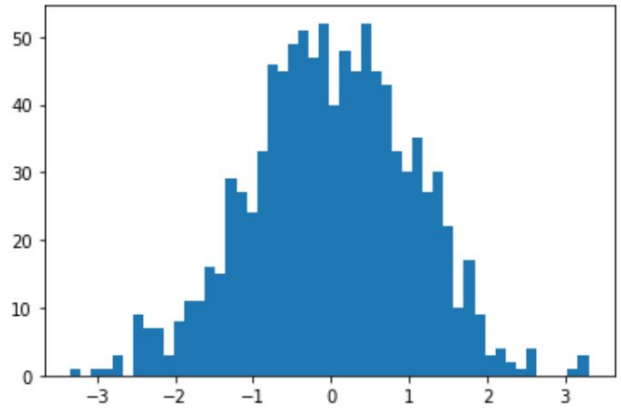
Out[31]:
```
True
```

In [35]:
```
np.equal(np.array([1,2,3,4]),np.array([0,2,3,4]))
```

Out[35]:
```
array([False,  True,  True,  True])
```

Statistics

```
In [1]: a = np.random.normal(0,1,1000)

In [13]: print(np.mean(a))
         -0.034478607630422944

In [14]: print(np.median(a))
         -0.013434852528562493

In [15]: print(np.std(a))
         1.0055207559124688

In [17]: print(np.var(a))
         1.0110719905707826

In [18]: print(np.average(a))
         -0.034478607630422944

In [19]: print(np.ptp(a))
         6.70344163147174
```



Histogramming

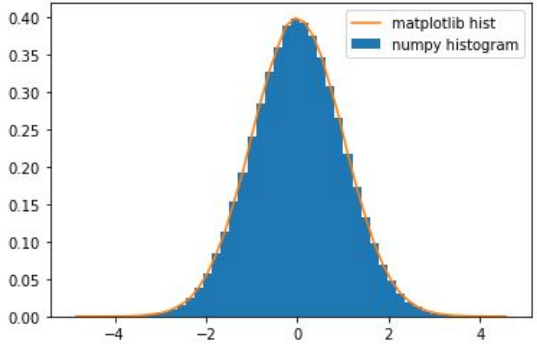
histogram(a[, bins, range, normed, weights, ...])	Compute the histogram of a dataset.
histogram2d(x, y[, bins, range, normed, ...])	Compute the bi-dimensional histogram of two data samples.
histogramdd(sample[, bins, range, normed, ...])	Compute the multidimensional histogram of some data.
bincount(x, /[, weights, minlength])	Count number of occurrences of each value in array of non-negative ints.
histogram_bin_edges(a[, bins, range, weights])	Function to calculate only the edges of the bins used by the histogram function.
digitize(x, bins[, right])	Return the indices of the bins to which each value in input array belongs.

https://numpy.org/doc/stable/reference/routines.statistics.html

numpy.histogram

- `numpy.histogram(array, options)` generates data and returns
  - an array with histogram bin content and
  - an array containing the bin edges

```
In [14]: import numpy as np
         import matplotlib.pyplot as plt
         ndist = np.random.normal(0, 1.0, 1000000)
         plt.hist(ndist, bins=50, density=True)
         # n: bin content, bins: bin edges
         (n, bins) = np.histogram(ndist, bins=50, density=True)
         print(type(n))
         print(type(bins))
         plt.plot(.5 * (bins[1:] + bins[:-1]), n)
         plt.legend(["matplotlib hist", "numpy histogram"], loc ="upper right")

         <class 'numpy.ndarray'>
         <class 'numpy.ndarray'>

Out[14]: <matplotlib.legend.Legend at 0x7ff56c6b0910>
```



Numpy Universal Functions (ufuncs)

- Comparison: `<, <=, ==, !=, >= >`
- Arithmetic: `+, -, *, /, reciprocal, square`
- Math
  - Exponential: `exp, log, log10, power, sqrt`
  - Trigonometric: `sin, cos, tan, arcsin, arccos, arctan`
  - Hyperbolic: `sinh, cosh, tanh, arcsinh, arccosh, arctanh`
- bitwise operations: `&, |, ~, ^, left_shift, right_shift`
- logical operations: `and, not, or, logical_xor`
- predicates: `isfinite, isinf, isnan, signbit`
- others: `abs,ceil, floor, mod, modf, round, sign`

numpy.newaxis

```
In [21]: a = np.array([1, 2, 3, 4, 5, 6])
         a.shape

Out[21]: (6,)

In [30]: print(a[np.newaxis, :])

         [[1 2 3 4 5 6]]

In [31]: print(a[np.newaxis, :].shape)

         (1, 6)

In [28]: print(a[:,np.newaxis])

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

In [29]: print(a[:, np.newaxis].shape)

         (6, 1)
```

numpy.expand_dims

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

In [10]: a = np.arange([0,1,2,3,4,5,6,7,8,9], dtype=np.float64)

In [11]: print(a)
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]

In [12]: print(np.expand_dims(a,axis=0)) # view
[[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]]

In [13]: print(np.expand_dims(a,axis=1)) # view
[[0.]
 [1.]
 [2.]
 [3.]
 [4.]
 [5.]
 [6.]
 [7.]
 [8.]
 [9.]]
```