NumPy

What will you learn?

1. Introduction to NumPy
2. Advantages
3. **Functionalities of NumPy** : Creation, Manipulation
4. Boolean Indexing
5. NumPy Broadcasting

Introduction

NumPy stands for Numerical Python. It has following features :

POWERFUL N-DIMENSIONAL ARRAYS : Fast and versatile, the NumPy vectorization, indexing, and broadcasting concepts are the de-facto standards of array computing today.

NUMERICAL COMPUTING TOOLS : NumPy offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more.

INTEROPERABLE : NumPy supports a wide range of hardware and computing platforms, and plays well with distributed, GPU, and sparse array libraries.

PERFORMANT : The core of NumPy is well-optimized C code. Enjoy the flexibility of Python with the speed of compiled code.

EASY TO USE : NumPy's high level syntax makes it accessible and productive for programmers from any background or experience level.

OPEN SOURCE : Distributed under a liberal BSD license, NumPy is developed and maintained publicly on GitHub by a vibrant, responsive, and diverse community.

Advantages over normal lists

You may think, we already have python lists with us to use. So why should we ever opt for NumPy? Let us see some advantages of NumPy over normal python lists.

Memory Consumption

NumPy arrays use much less memory as compared to normal lists. We can easily verify this.

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

li_arr = [i for i in range(100)]     # Create list of 100 elements
np_arr = np.arange(100)              # Create numpy array of 100 elements
```

In [2]:
```
## Size of 100 elements in numpy array
print(np_arr.itemsize * np_arr.size)
```

```
400
```

In [3]:
```
## Size of 100 elements in python list
print(sys.getsizeof(1) * len(li_arr))
```

```
2800
```

Time Execution

NumPy arrays are much faster as compared to python list. We can easily verify this.

In [4]:
```
import time
import numpy as np
```

In [5]:
```
size = 10000
```

In [6]:
```
def addition_using_list():
  t1 = time.time()
  a = range(size)
  b = range(size)
  c = [a[i] + b[i] for i in range(size)]
  t2 = time.time()
  return t2 - t1
```

In [7]:
```
def addition_using_numpy():
  t1 = time.time()
  a = np.arange(size)
  b = np.arange(size)
  c = a + b
  t2 = time.time()#mili second
  return t2 - t1
```

In [8]:
```
t_list = addition_using_list()
t_numpy = addition_using_numpy()
print("List = ", t_list * 1000)
print("NumPy = ", t_numpy * 1000)
```

```
List =  2.9916763305664062
NumPy =  0.9624958038330078
```

Convinient to Use

It is much easier to perform basic operations in NumPy arrays

Why is NumPy Faster Than Lists?

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

This behavior is called locality of reference in computer science.

This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Creating NumPy Arrays

In [9]:
```
import numpy as np    ## Use np as an alias for numpy
```

np.array()

NumPy is used to work with arrays. The array object in NumPy is called ndarray.

We can create a NumPy ndarray object by using the array() function.

Syntax

```
numpy.array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0, like
=None)
```

Parameters

1. **object :** array_like

   An array, any object exposing the array interface, an object whose **array** method returns an array, or any (nested) sequence.
2. **dtype :** data-type, optional

   The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence.
3. **copy :** bool, optional

   If true (default), then the object is copied. Otherwise, a copy will only be made if **array** returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (dtype, order, etc.).

In [10]:
```
a = [1, 2, 3]
b = np.array(a)
print(b)
print(type(b))
```

```
[1 2 3]
<class 'numpy.ndarray'>
```

In [11]:
```
a = [1, 2, 3, '5', 4.5]
b = np.array(a, dtype = str)
print(b)
```

```
['1' '2' '3' '5' '4.5']
```

In [12]:
```
a = [1, 2, 3, '5', 4.5]
b = np.array(a * 3)
print(b)
```

```
['1' '2' '3' '5' '4.5' '1' '2' '3' '5' '4.5' '1' '2' '3' '5' '4.5']
```

np.ones()

Syntax

```
numpy.ones(shape, dtype=None, order='C', *, like=None)
```

This function returns a new array of given shape and type, filled with ones.

Parameters

1. **shape :** int or sequence of ints

   Shape of the new array, e.g., (2, 3) or 2.
2. **dtype :** data-type, optional

   The desired data-type for the array, e.g., numpy.int8. Default is numpy.float64.
3. **order :** {'C', 'F'}, optional, default: C

   Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.
4. **like :** array_like

   Reference object to allow the creation of arrays which are not NumPy arrays. If an array-like passed in as like supports the **array_function** protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument.

In [13]:
```
b = np.ones(3, dtype = int)
b
```

Out[13]:
```
array([1, 1, 1])
```

In [14]:
```
b = np.ones((2, 3), dtype = int)
b
```

Out[14]:
```
array([[1, 1, 1],
       [1, 1, 1]])
```

np.zeros()

Syntax

```
numpy.zeros(shape, dtype=float, order='C', *, like=None)
```

This returns a new array of given shape and type, filled with zeros.

Parameters

1. **shape :** int or tuple of ints

   Shape of the new array, e.g., (2, 3) or 2.
2. **dtype :** data-type, optional

   The desired data-type for the array, e.g., numpy.int8. Default is numpy.float64.
3. **order :** {'C', 'F'}, optional, default: 'C'

   Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

In [ ]:
```
b = np.zeros(3, dtype = int)
b
```

In [ ]:
```
b = np.zeros((3, 4))
b
```

np.full()

Syntax

```
numpy.full(shape, fill_value, dtype=None, order='C', *, like=None)
```

Return a new array of given shape and type, filled with fill_value.

Parameters

1. **shape :** int or sequence of ints

   Shape of the new array, e.g., (2, 3) or 2.
2. **fill_value :** scalar or array_like

   Fill value.
3. **dtype :** data-type, optional

   The desired data-type for the array The default, None, means np.array(fill_value).dtype.
4. **order :** {'C', 'F'}, optional

   Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

In [ ]:
```
b = np.full((3, 3), 5, dtype = float)
b
```

np.empty()

Syntax

```
numpy.empty(shape, dtype=float, order='C', *, like=None)
```

This returns a new array of given shape and type, without initializing entries.

Parameters

1. **shape :** int or tuple of int

   Shape of the empty array, e.g., (2, 3) or 2.
2. **dtype :** data-type, optional

   Desired output data-type for the array, e.g, numpy.int8. Default is numpy.float64.
3. **order :** {'C', 'F'}, optional, default: 'C'

   Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

In [ ]:
```
np.empty([2, 2])            # uninitialized
```

In [ ]:
```
np.empty([2, 2], dtype=int)             # uninitialized
```

Note: You will get a different result each time you execute using empty, as the array formed will always be uninitialised.

np.arange()

Syntax

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

This returns evenly spaced values within a given interval.

Values are generated within the half-open interval [start, stop) (in other words, the interval including start but excluding stop). For integer arguments the function is equivalent to the Python built-in range function, but returns an ndarray rather than a list.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use numpy.linspace for these cases.

Parameters

1. **start :** integer or real, optional

   Start of interval. The interval includes this value. The default start value is 0.
2. **stop :** integer or real

   End of interval. The interval does not include this value, except in some cases where step is not an integer and floating point round-off affects the length of out.
3. **step :** integer or real, optional

   Spacing between values. For any output out, this is the distance between two adjacent values, out[i+1] - out[i]. The default step size is 1. If step is specified as a position argument, start must also be given.
4. **dtype :** dtype

   The type of the output array. If dtype is not given, infer the data type from the other input arguments.

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

In [ ]:
```
b = np.arange(2, 10)
b
```

In [ ]:
```
b = np.arange(2, 20, 2)
b
```

np.linspace()

Syntax

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=
0)[source]
```

This returns evenly spaced numbers over a specified interval.

The endpoint of the interval can optionally be excluded.

Parameters

1. **start :** array_like The starting value of the sequence.
2. **stop :** array_like

   The end value of the sequence, unless endpoint is set to False. In that case, the sequence consists of all but the last of num + 1 evenly spaced samples, so that stop is excluded. Note that the step size changes when endpoint is False.
3. **num :** int, optional

   Number of samples to generate. Default is 50. Must be non-negative.
4. **endpoint :** bool, optional

   If True, stop is the last sample. Otherwise, it is not included. Default is True.
5. **retstep :** bool, optional

   If True, return (samples, step), where step is the spacing between samples.
6. **dtype :** dtype, optional

   The type of the output array. If dtype is not given, the data type is inferred from start and stop. The inferred dtype will never be an integer; float is chosen even if the arguments would produce an array of integers.

1. **axis :** int, optional

   The axis in the result to store the samples. Relevant only if start or stop are array-like. By default (0), the samples will be along a new axis inserted at the beginning. Use -1 to get an axis at the end.

In [ ]:
```
b = np.linspace(2, 10)
b
print(b[1] - b[0])
print(b[3] - b[2])
```

In [ ]:
```
b = np.linspace(2, 10, 5, dtype = int, endpoint = False)
b
```

The difference between np.arange() and np.linspace() is that, using arange() we actually have control over step value, and using linspace() we have control over the number of values we want to generate.

np.identity()

Syntax

```
numpy.identity(n, dtype=None, *, like=None)
```

This returns the identity array.

The identity array is a **square array** with ones on the main diagonal.

Parameters

1. **n :** int Number of rows (and columns) in n x n output.
2. **dtype :** data-type, optional Data-type of the output. Defaults to float.

In [ ]:
```
b = np.identity(3)
b
```

np.eye()

Syntax

```
numpy.eye(N, M=None, k=0, dtype=<class 'float'>, order='C', *, like=None)
```

This returns a **2-D array** with ones on the diagonal and zeros elsewhere.

Parameters

1. **N :** int

   Number of rows in the output.
2. **M :** int, optional

   Number of columns in the output. If None, defaults to N.
3. **k :** int, optional

   Index of the diagonal: 0 (the default) refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.
4. **dtype :** data-type, optional

   Data-type of the returned array.
5. **order :** {‘C’, ‘F’}, optional

   Whether the output should be stored in row-major (C-style) or column-major (Fortran-style) order in memory.

In [ ]:
```
b = np.eye(3, 4)
b
```

np.random.rand()

Syntax

```
np.random.rand(d0, d1, ..., dn)
```

This returns random values in a given shape.

Parameters

1. **d0, d1, ..., dn :** int, optional

   The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

In [ ]:
```
## To generate values in range 0 - 10, we simply multiply by 10
b = np.random.rand(10) * 10
b
```

In [ ]:
```
b = np.random.rand(2, 3)
b
```

np.random.randint()

Syntax

```
random.randint(low, high=None, size=None, dtype=int)
```

This returns random integers from low (inclusive) to high (exclusive).

Parameters

1. **low :** int or array-like of ints

   Lowest (signed) integers to be drawn from the distribution (unless high=None, in which case this parameter is one above the highest such integer).
2. **high :** int or array-like of ints, optional

   If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None). If array-like, must contain integer values
3. **size :** int or tuple of ints, optional

   Output shape. If the given shape is, e.g., (m, n, k), then m *n* k samples are drawn. Default is None, in which case a single value is returned.
4. **dtype :** dtype, optional

   Desired dtype of the result. Byteorder must be native. The default value is int.

In [ ]:
```
np.random.randint(2, size=10)
```

In [ ]:
```
np.random.randint(1, size=10)
```

In [ ]:
```
np.random.randint(5, size=(2, 4))
```

In [ ]:
```
np.random.randint(1, [3, 5, 10])
```

Indexing and Slicing

Since we already know how to use indices and perform slicing in normal python lists, we shall see in comparison, how a numpy array does similar things.

1-D Arrays

In [ ]:
```
import numpy as np

li = [1, 2, 3, 4, 5]
arr = np.array(li)

print(li)
print(arr)
```

In [ ]:
```
print(arr.data)
print(arr.shape)
print(arr.dtype)
## Refers to the memory gap between two elements of numpy array
print(arr.strides)
```

In [ ]:
```
## Accessing elements
print(li[3])
print(arr[3])
```

In [ ]:
```
## Accessing more than one elements
print(li[1:4])
print(arr[1:4])
```

2-D Arrays

In [ ]:
```
li_2d = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]]
arr_2d = np.array(li_2d)

print(li_2d)
print(arr_2d)
```

In [ ]:
```
print(arr_2d.data)
print(arr_2d.shape)
print(arr_2d.dtype)
print(arr_2d.strides)
```

In [ ]:
```
## Accessing the elements
print(li_2d[2][1])
print(arr_2d[2][1])
print(arr_2d[2, 1])   # We may use comma also
```

In [ ]:
```
## Slicing
print(li_2d[1][:3])
print(arr_2d[1, :3])
```

For normal lists, we cannot get elements in column axis. Suppose we want to get elements of 1st, 2nd and 3rd row belonging only to the 2nd column, we may try this:

In [ ]:
```
print(li_2d[0:3][2])
```

But this does not work. Lets break it down into steps

In [ ]:
```
x = li_2d[0:3]
print(x)
y = x[2]
print(y)
```

This is easily possible in numpy arrays :

In [ ]:
```
print(arr_2d[0 : 3, 2])
```

We may also get multiple sliced rows and columns. Lets see how :

In [ ]:
```
print(arr_2d[2:4, 1:3])
print(li_2d[2:4][1:3])
```

Mathematical Operations

Lets see how numpy arrays simplify mathematical operations for us.

Arithmetic Operations

In [ ]:
```
import numpy as np

li = [1, 2, 3, 4, 5]
a = np.random.randint(1, 20, 5)
b = np.random.randint(1, 20, 5)

print(li)
print(a)
print(b)
```

In [ ]:
```
## Adding 1 to each element in list
li = [i+1 for i in li]
li
```

In [ ]:
```
## Adding 1 to each element in numpy array
a = a + 1
a
```

In [ ]:
```
## Adding two numpy arrays
c = a + b
c
```

In [ ]:
```
## Subtracting two numpy arrays
d = a - b
d
```

In [ ]:
```
## Multiplying two numpy arrays
e = a * b
e
```

In [ ]:
```
## Dividing two numpy arrays
f = a / b
f
```

In [ ]:
```
h = a ** b
h
```

In [ ]:
```
## Some misc operations
print(a)
print(a.sum())    ## Sum of all elements
print(a.mean())   ## Mean of all elements
print(a.min())    ## Minimum of all elements
print(a.argmin()) ## Index of minimum of all elements
print(a.max())    ## Maximum of all elements
print(a.argmax()) ## Index of maximum of all elements
```

Logical Operations

In [ ]:
```
print(a)
print(b)
```

In [ ]:
```
a > b
```

In [ ]:
```
a < b
```

In [ ]:
```
a == b
```

In [ ]:
```
print(np.logical_or(a, b))
print(np.logical_and(a, b))
```

In [ ]:
```
a[2] = 0
print(a)
print(np.logical_not(a))
```

Try all these operations for 2-D arrays also.

Boolean Indexing

NumPy also permits the use of a boolean-valued array as an index, to perform advanced indexing on an array. In its simplest form, this is an extremely intuitive and elegant method for selecting contents from an array based on logical conditions.

1D Arrays

Let us see an example

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

In [ ]:
```
b = np.random.randint(1, 20, 8)
print(b)
```

In [ ]:
```
print(b > 10)
```

In [ ]:
```
bool_arr = b > 10
print(bool_arr)
new_arr = b[bool_arr]
print(new_arr)
```

As you see, we are able to extract those elements for which the condition b > 10 was True.

In shorthand, we may do the following :

In [ ]:
```
new_arr = b[b > 10]
new_arr
```

Some more examples :

In [ ]:
```
new_arr = b[(b > 10) & (b < 18)]
new_arr
```

In [ ]:
```
print(b)
c = b
c
```

In the next example, we will update some elements on basis of a condition.

In [ ]:
```
c[:3] = 19
print(c)
c[c > 15] = 100
c
```

In [ ]:
```
print(b)
print(b[b == 100])
```

In [ ]:
```
## To get those indiced where element is 100
ind = np.where(b == 100)
ind
```

2D Arrays

2D arrays work the same as 1D arrays. Lets look at few examples

In [ ]:
```
import numpy as np
a = np.random.randint(1, 30, (5, 6))
print(a)
```

In [ ]:
```
bool_arr = a > 20
print(bool_arr)
```

In [ ]:
```
ans = a[bool_arr]
print(ans)
```

In [ ]:
```
b = a
print(a)
```

In [ ]:
```
b[bool_arr] = 100
print(b)
```

We just updated all the values greater than 20 to 100.

In [ ]:
```
c = np.random.randint(1, 10, (2, 2))
print(c)
```

In [ ]:
```
c_bool = np.array([[True, False], [False, True], [True, True]])
print(c_bool)
```

In [ ]:
```
print(c[c_bool])   ## Error generated because dimensions of boolean array and
c are not same.
```

Lets see how we can work over a specific column.

In [ ]:
```
print(b)
```

In [ ]:
```
bool_arr = b[:, 3] == 100
print(bool_arr)
```

Lets update the array.

In [ ]:
```
b[bool_arr, 3] = 99
print(b)
```

NumPy Broadcasting

The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes. Broadcasting provides a means of vectorizing array operations so that looping occurs in C instead of Python. It does this without making needless copies of data and usually leads to efficient algorithm implementations. There are, however, cases where broadcasting is a bad idea because it leads to inefficient use of memory that slows computation.

NumPy operations are usually done on pairs of arrays on an element-by-element basis. In the simplest case, the two arrays must have exactly the same shape, as in the following example:

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

In [ ]:
```
a = np.array([1.0, 2.0, 3.0])
b = np.array([2.0, 2.0, 2.0])
a * b
```

NumPy’s broadcasting rule relaxes this constraint when the arrays’ shapes meet certain constraints. The simplest broadcasting example occurs when an array and a scalar value are combined in an operation:

In [ ]:
```
a = np.array([1.0, 2.0, 3.0])
b = 2.0
a * b
```

The result is equivalent to the previous example where b was an array. We can think of the scalar b being stretched during the arithmetic operation into an array with the same shape as a. The new elements in b are simply copies of the original scalar. The stretching analogy is only conceptual. NumPy is smart enough to use the original scalar value without actually making copies so that broadcasting operations are as memory and computationally efficient as possible.

The code in the second example is more efficient than that in the first because broadcasting moves less memory around during the multiplication (b is a scalar rather than an array).

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimensions and works its way left. Two dimensions are compatible when

1. they are equal, or
2. one of them is 1

If these conditions are not met, a ValueError: operands could not be broadcast together exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the size that is not 1 along each axis of the inputs.

In [ ]:
```
x = np.random.randint(1, 10, (3, 3))
y = np.random.randint(1, 10, (3, 3))
print(x)
print(y)
```

In [ ]:
```
ans = x - y
print(ans)
```

In [ ]:
```
x = np.random.randint(1, 10, (3, 3))
y = np.random.randint(1, 10, (3))
print(x)
print(y)
```

In [ ]:
```
ans = x - y
print(ans)
```

In [ ]:
```
x = np.random.randint(1, 10, (3, 2))
y = np.random.randint(1, 10, (2, 3))
print(x)
print(y)
```

Transpose

To make these two arrays compatible, we may **transpose** one array.

In [ ]:
```
y = np.transpose(y)
print(y)
ans = x - y
print(ans)
```

Reshape

We may also **reshape** our array. Lets see an example :

In [ ]:
```
x = np.arange(16)
x
```

In [ ]:
```
y = np.random.randint(1, 10, (4, 4))
y
```

In [ ]:
```
x * y    ## Error generated
```

Lets reshape 'x'

In [ ]:
```
x = np.reshape(x, (4, 4))
x
```

Now we may multiply, subtract, add or divide 'x' and 'y'.

In [ ]:
```
x * y
```

In [ ]:
```
x - y
```

In [ ]:
```
x + y
```

In [ ]:
```
x / y
```

In [ ]:
```
x // y
```

In [ ]:
```
x % y
```