NumPy Arrays I

```
import numpy as np
```

An array is also a data-structure which contains multiple values. Working with NumPy arrays is much easier than working with Python lists.

The term NumPy stands for Numeric (or Numerical) Python. It is one of the most popular modules amongst Statisticians and Data Scientists across the globe to perform heavy mathematical computations. NumPy arrays are also faster than Python lists in terms of code execution time.

For all practical purposes, **an array is the same as a list**. Fundamentally, they have been programmed differently by the inventors of Python language.

Activity: NumPy Arrays - The `ones()` & `zeros()` Functions

To get started with NumPy arrays, let's create an array having each item as `1` with dimensions `5` blocks, `3` rows and `3` columns.

To create such an array, you first need to `import` the `numpy` module at the beginning of the code. Then use the `ones()` function from the `numpy` module.

The `ones()` function requires two inputs. The first input should be the required dimension of the array and the second input should define the data-type of the items in the required array.

```
a=np.ones(1)
print(a)
```

```
[1.]
```

```
type(a)
```

```
numpy.ndarray
```

```
#Create an array containing '1' as its items using the 'ones()' function from the 'num
a1=np.ones((1,7))
a1
```

```
array([[1., 1., 1., 1., 1., 1., 1.]])
```

```
a= np.ones((5), dtype=int)
a
```

```
array([1, 1, 1, 1, 1])
```

```
a1=np.ones((2,7),dtype=int)
a1
```

```
array([[1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1]])
```

```
a1=np.zeros((3),dtype=int)
a1
```

```
array([0, 0, 0])
```

```
a1=np.zeros((3,5),dtype=int)
a1
```

```
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
```

```
a2=np.ones((4,2,2,4))
a2
```

```
array([[[[1., 1., 1., 1.],
         [1., 1., 1., 1.]],
```

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


       [[[1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.]]],


       [[[1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.]]],


       [[[1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.]]]])
```

```
type(a1)
```

```
numpy.ndarray
```

```
a=[11,12,13, 14, 15]
```

```
type(a1)
```

```
numpy.ndarray
```

```
a1=np.array(a)
print(a)
```

```
[11, 12, 13, 14, 15]
```

```
l1=[5.0,3.0,6.4,9.4, 5.5]

a4=np.array(l1)
a4
```

```
array([5. , 3. , 6.4, 9.4, 5.5])
```

```
Start coding or generate with AI.
```

```
type(a4)
```

```
numpy.ndarray
```

```
type(l1)
```

```
import numpy as np
```

Note:

1. Always import the `numpy` module at the beginning of the code with `np` as its alias (or nickname). Throughout the course, we will always refer to the `numpy` module as `np` unless otherwise specified.
2. The de facto data-type of an item in a NumPy array is `float`.

Using the `type()` function, you can verify whether an array is a NumPy array or not.

```
# Using the 'type()' function, verify whether the 'ones' array is a NumPy array or not
type(a1)
```

As you can see, the `type()` function has returned `numpy.ndarray` as an output which confirms that the `ones` array is indeed a NumPy array.

Similarly, you try to create a NumPy array having each item as `0` with dimensions `4` blocks, `2` rows and `3` columns using the `zeros()` function. Make sure that each item is an integer.

```
# Using 'zeros()' function, create a NumPy array having 4 blocks, 2 rows and 3 columns
a3=np.zeros((4,2,3))
a3
```

```
a3=np.zeros((4,2,3))
a3
```

Recreate the above array without providing the data-type as input. The output should be a three-dimensional array with each item as `0.` which is a floating-point number.

Here's the desired output:

```
array([[[0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.]]])
```

```
# Using 'zeros()' function, create a numpy array having 4 blocks, 2 rows and 3 columns
```

As you can see, all the items of the `zero_float` array are of `float` data-type.

Activity: The `dtype` Keyword

To check the data-type of the values stored in a NumPy array, you can use the `dtype` keyword.

```
# Using the 'dtype' keyword, verify whether the 'zeros_float' array items are floating
a1.dtype
```

```
dtype('int64')
```

For the time being ignore the number `64`. As you can see, the `dtype` keyword has returned `dtype('float64')` as an output confirming that all the items of the `zeros_float` array are floating-point values.

```
# Using the 'dtype' keyword, verify whether the 'ones' array items are int values or n
```

For the time being ignore the number `64`. As you can see, the `dtype` keyword has returned `dtype('int64')` as an output confirming that all the items of the `ones` array are integer values.

The `type()` function is a function of Python. It returns the type of Python object. The data-types or more accurately the **primitive data-types** `int`, `float`, `str` and `bool` are actually Python objects. Similarly, Python lists, NumPy module, `random` modules are also Python objects. The `type()` function will tell you what type of Python objects they are.

```
# Run the code shown below to see that the 'type()' function returns the type of Pytho
```

```
# 1. The 'int' Python object.
print(type(list))
# 2. The 'float' Python object.

# 3. The 'str' Python object.

# 4. The 'list' Python object

# 5. 'numpy' is a 'module' Python object.

# 6. 'random' is a 'module' Python object.
```

```
<class 'type'>
```

Start coding or generate with AI.

Start coding or generate with AI.

In the output, ignore the term `class`. As you can see, the `type()` function indeed returns the type of Python object for an entity in Python.

Activity: The `ndim` And `shape` Keywords

The `ndim` Keyword: Using the `ndim` keyword, you can find out the dimension of a NumPy array. The `ones`, `zeros` and `zeros_float` arrays are three-dimensional arrays. So, the `ndim` keyword should return `3` for each of these arrays.

```
# Using the 'ndim' array, find out the dimension of 'ones', 'zeros' and 'zeros_float'
a1.ndim
```

```
1
```

```
a2.ndim
```

```
4
```

```
a4.ndim
```

```
a4
```

The `shape` Keyword: Using the `shape` keyword, you can find out the number of blocks, rows and columns present in NumPy arrays.

```
# Using the 'shape' keyword find out the number of blocks, rows and columns in the 'on
a4.shape
```

```
(5,)
```

```
a1
```

```
a2.shape
```

```
(4, 2, 2, 4)
```

```
a2.size
```

```
64
```

```
len(a2)
```

```
4
```

```
Start coding or generate with AI.
```

As you can see:

- The `ones` array has 5 blocks, 3 rows and 3 columns.
- The `zeros` array has 4 blocks, 2 rows and 3 columns.
- The `zeros_ones` array also has 4 blocks, 2 rows and 3 columns.

Activity: The `arange()` And `linspace()` Functions^^

The are `arange()` and `linspace()` functions in NumPy module create an array of numbers such that the difference between a pair of any two consecutive numbers in the array is constant. However, there is a slight difference in their outputs.

The `arange()` Function: It returns a one-dimensional array containing numbers between two specific numbers. It is similar to the `range()` function in the `for` loop.

```
# Create a one-dimensional array containing numbers between 5 and 20 using the 'arange

a6=np.arange(5,21,3)
a6
```

```
array([ 5,  8, 11, 14, 17, 20])
```

Here, the `arange()` function has created an array of numbers from `5` to `20`. The difference between a pair of any two consecutive numbers is `1`. The `arange()` function will not include the last number specified in the array created.

If you want the difference between two consecutive numbers to be greater than `1` (say `4`) in an array, then you can specify the difference as an additional input to the `arange()` function.

```
# Create an array of numbers from 1 to 24 such that the difference between two consecu
a=np.arange(1,24,4)
a
```

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

As you can see, the `arange()` function has created an array of numbers from `1` to `24` such that the difference between two consecutive numbers is `4`.

Note: If the difference between two consecutive numbers is not specified, then the `arange()` function will assume that the difference is `1`.

You can also create an array containing the floating-point numbers using the `arange()` function by providing `dtype=float` as an additional input to the function.

```
# Create an array of numbers from 1 to 24 such that the difference between two consecu
a7=np.linspace(1,24,4)
```

```
a7=np.linspace(1,24,12)
a7
```

```
array([ 1.        ,  3.09090909,  5.18181818,  7.27272727,  9.36363636,
       11.45454545, 13.54545455, 15.63636364, 17.72727273, 19.81818182,
       21.90909091, 24.        ])
```

```
a7
```

```
array([ 1.        ,  8.66666667, 16.33333333, 24.        ])
```

```
# Create an array containing 4 numbers between 1 to 25.
```

As you can see, the `linspace()` function has created an array containing `4` numbers between `1` and `25` (including `25`) such that the difference between two consecutive numbers is constant. Take a pair of any two consecutive numbers in `array5`, the difference

between them will be `8.`

Notice the difference between the outputs of the `arange()` and `linspace()` functions for exactly the same inputs. The `arange()` function returns `6` numbers such that the difference between them is exactly `4` whereas the `linspace()` function returns exactly `4` numbers such that the difference between two consecutive numbers is constant.

In other words:

- In the `arange()` function, you specify the value of the difference between two consecutive numbers as the third input.
- Whereas, in the `linspace()` function, you specify the length of the array to be created as the third input.

Let's create another array using the `linspace()` function to get a better understanding. Let's create an array containing `10` numbers between `1` and `25` such that the difference between two consecutive numbers is constant.

```
# Create an array containing 10 numbers between 1 and 25 such that the difference betw
```

Take a pair of any two consecutive numbers in the `array5`, the difference between them will be `2.66666666`

Activity: The `reshape()` Function

The `reshape()` function reshapes an array from one configuration to another without changing the items.

The `zeros` array has 4 blocks, 2 rows and 3 columns. Suppose you want to reshape it into another array having `6` rows and `4` columns without changing the items, then you can use the `reshape()` function.

```
# Reshape the 'zeros' array so that it has 6 rows and 4 columns.
# Before reshaping
a8=a3.reshape(4,6)
a8
# After reshaping
```

```
a2
```

```
array([[[[1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.]]],


       [[[1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.]]],


       [[[1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.]]],


       [[[1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.]]]])
```

```
ar=a2.reshape( 4,4,4 )
ar
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-2938899299.py in <cell line: 0>()
```

```
a9=a3.reshape(24)
```

```
a9
```

```
a1=np.ones((27))
a1
```

The `reshape()` function has reshaped the `zeros` array to a two-dimensional array having 6 rows and 4 columns. Inside the `reshape()` function, you have to provide the new configuration.

This operation is only valid when the number of items in the new configuration is exactly the same as the number of items in the original configuration.

In this case, there are 24 items (4 blocks $\times$ 2 rows $\times$ 3 columns = 24 items) in the original configuration. So the new configuration must also have 24 items.

You can also convert a multi-dimensional array into a one-dimensional array using the `reshape()` function. Let's convert `ones` into a one-dimensional array using the `reshape()` function. It has 45 items. The reshaped array must also contain 45 items.

```
# Reshape the 'ones' array so that it has just 1 row.
```

As you can see, the three-dimensional array is converted into a one-dimensional array using the `reshape()` function.

Activity: The `random.randint()` Function

The NumPy module also has its `random.randint()` function. It takes two inputs. The first input is the range of two integers between which a random number is to be created. The second input is the `size` parameter which defines the number of random integers to be created.

Syntax: `np.random.randint(num1, num2, size=N)`

where `num1` and `num2` are starting numbers and `N` is the number of items to be contained in the array.

```
import random
```

```
a=random.randint(1,10)
b=random.randint(100,1000)
print(a)
print(b)
```

```
8
208
```

```
#Create a one-dimensional NumPy array containing 15 random integers between 50 and 100.
a9=np.random.randint(50,100,(2,6,5))
a9
```

```
array([[[80, 63, 62, 65, 71],
        [54, 51, 50, 60, 81],
        [96, 57, 64, 73, 61],
        [68, 67, 75, 56, 92],
        [80, 91, 53, 89, 58],
        [58, 77, 79, 92, 99]],

       [[97, 65, 78, 69, 99],
        [57, 62, 83, 67, 67],
        [95, 76, 87, 78, 77],
        [69, 94, 63, 71, 60],
        [53, 66, 91, 65, 80],
        [81, 67, 89, 64, 81]]])
```

Similarly, you can create a multi-dimensional NumPy array containing random integers by specifying the dimensions of the array to be created in the `size` parameter.

```
# Create a two-dimensional NumPy array containing 15 random integers between 50 and 10
```

```
# Create a three-dimensional NumPy array containing 40 random integers between 50 and
```

Activity: The `array()` Function^

We can convert a Python list into a NumPy array. To do this, use the `array()` function which takes a Python list as an input.

Let's first create a list of natural numbers from `1` to `15`. Then convert the list into a NumPy array using the `array()` function.

```
#  Convert a Python list into a NumPy array using the 'array()' function.
```

To check whether a Python list is converted into a NumPy array or not, you can use the `type()` function.

```
#  Verify whether the 'arr' array is a NumPy array or a Python list.
```

```
a10=np.array([1,4,2,7,9,8,6,6,0,4,6,7])
```

```
a10
```

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

```
type(a10)
```

```
numpy.ndarray
```

```
s=np.sum(a10)
print(s)
```

```
60
```

```
l1=[1,4,2,7,9,8,6,6,0,4,6,7]
```

```
a10*a10
```

```
array([ 1, 16,  4, 49, 81, 64, 36, 36,  0, 16, 36, 49])
```

```
l1*l1
```

```
np.median(a10)
```

```
np.float64(6.0)
```

```
np.log(a10)
```

```
/tmp/ipython-input-1912220576.py:1: RuntimeWarning: divide by zero encountered in log
  np.log(a10)
array([0.        , 1.38629436, 0.69314718, 1.94591015, 2.19722458,
       2.07944154, 1.79175947, 1.79175947,       -inf, 1.38629436,
       1.79175947, 1.94591015])
```

```
from scipy import stats

stats.mode(a10)
```

```
ModeResult(mode=np.int64(6), count=np.int64(3))
```

```
np.mean(a10)
```

Activity: The `list()` Function

We can convert a NumPy array into a Python list using the `list()` function.

Let's convert the `arr` NumPy array into a Python list using the `list()` function.

```
# Convert a NumPy array into a Python list using the 'list()' function.
l2=list(a10)
print(l2)
```

```
[np.int64(1), np.int64(4), np.int64(2), np.int64(7), np.int64(9), np.int64(8), np.int64(6), np.int64(6), np.int64(0), np.int
```

Let's verify whether the conversion happened or not using the `type()` function.

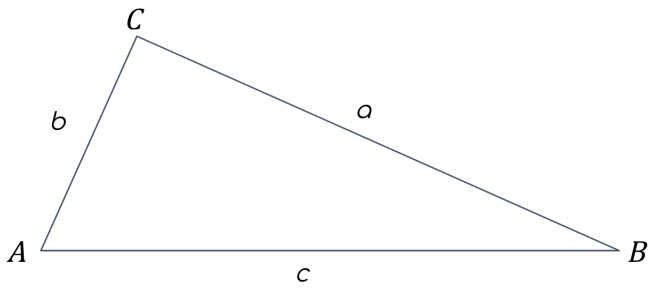
```
# Verify whether the 'arr_converted' list is Python list or not.
type(l2)
```

```
list
```

As you can see, the output is `list`. Hence, we successfully converted the NumPy array to a Python list using the `list()` function.

Activity: The `sqrt()` Function^^^

One of the most notorious concepts in maths is Heron's formula. It calculates the area of a triangle having sides a, b and c.



According to the Heron's formula, the area of a triangle is given by

$$\text{Area} = \sqrt{s(s-a)(s-b)(s-c)}$$

where

$$s = \frac{a+b+c}{2}$$

i.e., the semi-perimeter of a triangle.

Let's create a function which takes the lengths of the sides of a triangle and checks whether the lengths indeed represent the lengths of the sides of a triangle.

The sum of lengths of any two sides of a triangle is always greater than the length of the third side.

If the lengths meet the criteria for being the lengths of the sides of a triangle, then the function should return `True`. Else it should return `False`.

```
# Create a function to check whether the input lenghts represent the lengths of the tr
def area(a,b,c):
  if((a+b>c and b+c>a and a+c>b)) == True:
    ar=np.sqrt(s(s-a))
  return
```

```
```

```
#define the length of sides of a triangle
```

Now, let's create a function to calculate the area of a triangle using the Heron's formula. The function must include the `is_triangle()` function and `np.sqrt()` functions. If the invalid side lengths are passed as input to the function, then it should return `Invalid triangle.` message else it should return the area of a triangle.

```
#  Create a function to calculate the area of a triangle using the Heron's formula.
```

Alternatively, you can raise a number to the power `0.5` to calculate the square root of a number, but it will return a complex number as an output for a negative number. On the other hand, the `np.sqrt()` function returns `nan` for negative numbers. It stands for **not a number** and denotes a NumPy null value.
