NumPy Arrays

Programming for Data Science with Python

1. Introduction

NumPy is the fundamental package for **scientific computing** in Python.

It is a **Python library** that provides:

- a multidimensional array object
- various derived objects (such as masked arrays and matrices)
- an assortment of routines for fast operations on arrays (including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.)

At the **core** of the **NumPy** package, is the ndarray object.

- This encapsulates **n-dimensional arrays** of homogeneous data types - with many operations being performed in compiled code for performance.

There are several **important differences** between **NumPy arrays** and the **standard Python sequences:**

- **NumPy arrays** have a **fixed size** at creation, unlike Python lists (which can grow dynamically).
- Changing the size of a ndarray will create a new array and delete the original.

The **elements** in a **NumPy array** are all required to be of the same data type, thus will be the same size in memory.

The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.

NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.

A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays.

In other words, in order to efficiently use much (perhaps even most) of today's scientific/mathematical Python-based software, just knowing how to use Python's built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

2. NumPy: Class ndarray

2.1 Overview

In NumPy, an array object represents a multidimensional, homogeneous array offixed-size items.

An associated data-type object describes the format of each element in the array:

- its byte-order
- how many bytes it occupies in memory,
- whether it is an integer, a floating point number, or something else.

2.2 Fundamental Concepts

2.2.1 NumPy Arrays: Dimension & Axis

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**. The number of axes is rank.

EXAMPLE:

The coordinates of a point in 3D space [1, 2, 1] is an array of rank 1, because it has one axis.

- This axis has a length of 3.

This array, [[1., 0., 0.], [0., 1., 2.)) has rank 2, i.e., 2 dimensions (it is 2-dimensional).

- Let's see this array as a table of **2 rows and 3 columns**.
    - The first dimension (axis 0) has a length of 2 (2 rows), the second dimension (axis 1) has a length of 3 (3 columns).

A 2-dimensional array has two corresponding axes:

- the first running vertically downwards across rows (axis 0)
- the second running horizontally across columns (axis 1)

Many operation can take place along one of these axes. For example, we can sum each row of an array, in which case we operate along columns, or axis 1.

Run the following 4 code blocks:

In [1]:
```
import numpy as np
x = np.arange(12).reshape((3,4))
x
```

Out[1]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [3]:
```
import numpy as np
y = np.arange(18).reshape((3, 6))
print(y)
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]]
```

In [4]:
```
# Sum up all the elements
x.sum()
```

Out[4]:
```
66
```

In [5]:
```
y.sum()
```

Out[5]:
```
153
```

In [6]:
```
# Sum up elements along the horizontal axis
x.sum(axis = 1)
```

Out[6]:
```
array([ 6, 22, 38])
```

In [8]:
```
y.sum(axis = 1)
```

Out[8]:
```
array([15, 51, 87])
```

In [9]:
```
# Sum up elements along the vertical axis
x.sum(axis = 0)
```

Out[9]:
```
array([12, 15, 18, 21])
```

In [10]:
```
y.sum(axis = 0)
```

Out[10]:
```
array([18, 21, 24, 27, 30, 33])
```

IMPORTANT NOTES: **axis = 0 axis= 0**: Actions occur vertically, i.e., moving along vertically actions on the rows, e.g., drop rows, add rows

IMPORTANT NOTES: **axis = 1 axis = 1**: Actions occur horizontally, i.e., moving along horizontally actions on the columns, e.g., add columns, drop columns

3. Numpy Arrays: Creation

3.1 Overview

NumPy arrays can be created in different ways, with default initial values or manually specified ones.

3.2 NumPy Arrays: Creation: Empty, Ones, Zeros, Full Arrays

empty(shape[, dtype, order]) Returns a new array of given shape and type, without initializing entries.

empty_like(a[, dtype, order, subok]) Returns a new array with the same shape and type as a given array.

eye(N[, M, k, dtype]) Returns a 2-D array with ones on the diagonal and zeros elsewhere.

identity(n[, dtype]) Returns the identity array.

ones(shape[, dtype, order]) Returns a new array of given shape and type, filled with ones.

ones_like(a[, dtype, order, subok]) Returns an array of ones with the same shape and type as a given array.

zeros(shape[, dtype, order]) Returns a new array of given shape and type, filled with zeros.

zeros_like(a[, dtype, order, subok]) Returns an array of zeros with the same shape and type as a given array.

full(shape, fill_value[, dtype, order]) Returns a new array of given shape and type, filled with fill_value.

full_like(a, fill_value[, dtype, order, subok]) Returns a full array with the same shape and type as a given array.

3.2.1 empty(shape[, dtype, order])

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

Parameters:

shape: int or tuple of int → Shape of the empty array

dtype: data-type, optional → Desired output data-type.

order: {'C', 'F'}, optional

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

Returns:

out: ndarray

Array of uninitialized (arbitrary) data of the given shape, dtype, and order.

Object arrays will be initialized to None.

Run the following 5 code blocks:

In [11]:
```
import numpy as np

# An empty 1D array: shape = 8 (1D, 8 elements)
np.empty(8)
```

Out[11]:
```
array([0.000e+000, 0.000e+000, 0.000e+000, 0.000e+000, 0.000e+000,
       8.577e-321, 0.000e+000, 0.000e+000])
```

In [12]:
```
import numpy as np
np.empty(16)
```

Out[12]:
```
array([6.23042070e-307, 4.67296746e-307, 1.69121096e-306, 7.56593696e-307,
       8.34441742e-308, 1.78022342e-306, 6.23058028e-307, 9.79107872e-307,
       6.89807188e-307, 7.56594375e-307, 6.23060065e-307, 1.78021527e-306,
       8.34454050e-308, 1.11261027e-306, 1.15706896e-306, 1.33512173e-306])
```

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

# An empty 1D array of integers: shape = 8
np.empty(8, dtype=int)
```

Out[13]:
```
array([   0,    1,    0,  654, 1240,    0,  768,  654])
```

In [15]:
```
import numpy as np
np.empty(5, dtype = int)
```

Out[15]:
```
array([-786324352,        654,          0,          0,          1])
```

In [16]:
```
import numpy as np

# Store values in rows as "C" style
np.empty(8, dtype=int, order= 'C')
```

Out[16]:
```
array([-785502816,        654, -785502896,        654, -785500656,
              654,    4390944,    7471215])
```

In [17]:
```
import numpy as np
np.empty(5, dtype = float, order= 'F')
```

Out[17]:
```
array([0.00000000e+000, 1.38951915e-311, 1.38951915e-311, 1.38951915e-311,
       1.60220393e-306])
```

In [18]:
```
import numpy as np

# An empty 1D array of integers: shape (a tuple)
np.empty([2, 3], dtype = int)
```

Out[18]:
```
array([[-824141936,        654,          0],
       [         0,          1,          0]])
```

In [19]:
```
import numpy as np
np.empty([5,5], dtype = int)
```

Out[19]:
```
array([[        50,         48,         50,         52,         45],
       [        48,         52,         45,         48,         50],
       [        84,         50,         50,         58,         52],
       [        51,         58,         50,         49,         46],
       [        54,         56,         55,         90, 862335286]])
```

In [20]:
```
import numpy as np

# an 1D array of strings
np.empty(8, dtype=str)
```

Out[20]:
```
array(['', '', '', '', '', '', '', ''], dtype='<U1')
```

3.2.2 empty_like(a[, dtype, order, subok])

Returns a new array with the same shape and type as a given array.

Parameters:

a: array like → The shape and data-type of a define these same attributes of the returned array.

dtype: data-type, optional; → Overrides the data type of the result.

...subok: bool, optional

- If True, then the newly created array will use the sub-class type of 'a'
  - Otherwise it will be a base-class array. Defaults to True.

Returns:

out: ndarray

Array of uninitialized (arbitrary) data with the same shape and type as 'a'.

Run the following 2 code blocks:

In [21]:
```
import numpy as np

a= ([1,2,3], [4,5,6]) # a is array-like

np.empty_like(a)
```

Out[21]:
```
array([[-824141936,        654,          0],
       [         0,     131074,          0]])
```

In [23]:
```
import numpy as np
chicken = (["why", 'did','the', 'chicken'], ['cross', 'the', 'road', '?'])
np.empty_like(chicken)
```

Out[23]:
```
array([['', '', '', ''],
       ['', '', '', '']], dtype='<U7')
```

In [24]:
```
import numpy as np

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

```
np.empty_like(a)
```

Out[24]:
```
array([[7.74860419e-304, 7.74860419e-304, 7.74860419e-304],
       [7.74860419e-304, 7.74860419e-304, 7.74860419e-304]])
```

In [25]:
```
import numpy as np

b = np.array([[9,8,7,6],[4,3,2,1]])
np.empty_like(b)
```

Out[25]:
```
array([[      0,       1,       0, 5570652],
       [   1532,       0,     768,     654]])
```

3.2.3 identity(n[, dtype])

Returns the identity array.

Run the following code block:

In [26]:
```
import numpy as np

np.identity(8, dtype=int)
```

Out[26]:
```
array([[1, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 1]])
```

In [27]:
```
import numpy as np
np.identity(4)
```

Out[27]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

3.2.4 eye(N[, M, k, dtype])

Returns a 2-D array with ones on the diagonal and zeros elsewhere.

Parameters:

N: int → Number of rows in the output.

M: int, optional → Number of columns in the output. If None, defaults to N.

k: int, optional → Index of the diagonal

0 (the default): → to the main diagonal; a positive value refers to an upper diagonal; and a negative value to a lower diagonal.

dtype: data-type, optional → Data-type of the returned array.

Returns:

I: ndarray of shape (N,M)

An array where all elements are equal to zero, except for the k-th diagonal, whose values are equal to one.

Run the following 2 code blocks:

In [28]:
```
import numpy as np
np.eye (8, dtype = int)
```

Out[28]:
```
array([[1, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 1]])
```

In [35]:
```
import numpy as np
np.eye(4, dtype=float)
```

Out[35]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

In [36]:
```
import numpy as np
np.eye (8, k=2)
```

Out[36]:
```
array([[0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.]])
```

In [43]:
```
import numpy as np
np.eye (18, k=2)
```

Out[43]:
```
array([[0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        1., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 1.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.]])
```

3.2.5 ones(shape[, dtype, order])

Returns a new array of given shape and type, filled with ones.

Parameters:

shape: int or sequence of ints → Shape of the new array, e.g., (2, 3) or 2.

dtype: data-type, optiona → The desired data-type for the array, e.g., numpy.int8. Default is numpy.float64.

order: {'C', 'F'}, optional → ...

Returns:

out: ndarray

Array of ones with the given shape, dtype, and order.

Run the following 2 code blocks:

```
In [44]: import numpy as np
         np.ones(8)

Out[44]: array([1., 1., 1., 1., 1., 1., 1., 1.])
```

```
In [46]: import numpy as np
         np.ones(3)

Out[46]: array([1., 1., 1.])
```

```
In [47]: import numpy as np
         np.ones(8, dtype=int)

Out[47]: array([1, 1, 1, 1, 1, 1, 1, 1])
```

```
In [48]: import numpy as np
         np.ones(5, dtype=int)

Out[48]: array([1, 1, 1, 1, 1])
```

3.2.6 zeros(shape[, dtype, order])

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

Run the following code block:

```
In [49]: import numpy as np
         np.zeros(8)

Out[49]: array([0., 0., 0., 0., 0., 0., 0., 0.])
```

```
In [50]: import numpy as np
         np.zeros(5)

Out[50]: array([0., 0., 0., 0., 0.])
```

3.2.7 full(shape, fill_value[, dtype, order])

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

Run the following 2 code blocks:

```
In [51]: import numpy as np
         np.full(8, 2)

Out[51]: array([2, 2, 2, 2, 2, 2, 2, 2])
```

```
In [52]: import numpy as np
         np.full(5,3)
```

Out[52]:
```
array([3, 3, 3, 3, 3])
```

In [53]:
```python
import numpy as np
np.full(8, "Hello")
```

Out[53]:
```
array(['Hello', 'Hello', 'Hello', 'Hello', 'Hello', 'Hello', 'Hello',
       'Hello'], dtype='<U5')
```

In [54]:
```python
import numpy as np
np.full(3, "Who?")
```

Out[54]:
```
array(['Who?', 'Who?', 'Who?'], dtype='<U4')
```

3.2.8 arange([start,]stop, [step,]dtype=None)

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 linspace for these cases.

Parameters:

start: number, optional → Start of interval. The interval includes this value. The default start value is 0.

stop: number → 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).

step: number, 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, start must also be given.

dtype: dtype → The type of the output array.

- If dtype is not given, infer the data type from the other input arguments.

Returns:

arange: ndarray → Array of evenly spaced values.

- For floating point arguments, the length of the result is ceil((stop - start)/step).
- Because of floating point overflow, this rule may result in the last element of out being greater than stop.

Run the following 3 code blocks:

```
In [55]: import numpy as np
         np.arange(8)

Out[55]: array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
In [57]: import numpy as np
         np.arange(15)

Out[57]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
In [58]: import numpy as np
         np.arange(3, 8)

Out[58]: array([3, 4, 5, 6, 7])
```

```
In [60]: import numpy as np
         np.arange(5, 13)

Out[60]: array([ 5,  6,  7,  8,  9, 10, 11, 12])
```

```
In [61]: import numpy as np
         np.arange(3, 19, 2)

Out[61]: array([ 3,  5,  7,  9, 11, 13, 15, 17])
```

```
In [62]: import numpy as np
         np.arange(5, 50, 5)

Out[62]: array([ 5, 10, 15, 20, 25, 30, 35, 40, 45])
```

3.2.9 linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)

Returns **evenly spaced** numbers over a specified interval.

Returns **num evenly spaced samples**, calculated over the interval [start, stop].

The endpoint of the interval can optionally be excluded.

Parameters:

start: scalar → The starting value of the sequence.

stop: scalar → The end value of the sequence, unless the 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:** the step size changes when endpoint is False.

num: int, optional → Number of samples to generate. Default is 50. Must be non-negative.

endpoint: bool, optional

- If True, stop is the last sample.

- Otherwise, it is not included. Default is True.

Retstep: bool, optional

- If True, return (samples, step), where step is the spacing between samples.

dtype: dtype, optional → The type of the output array.

- If dtype is not given, infers the data type from the other input arguments.

Returns: samples : ndarray

- There are num equally spaced samples in the closed interval [start, stop] or the half-open interval [start, stop) (depending on whether endpoint is True or False)

step: float, optional

- Only returned if retstep is True
- Size of spacing between samples.

  ### **Run the following 4 code blocks:**

In [63]:
```
import numpy as np
np.linspace(2.0, 3.0, num=5)
```
Out[63]:
```
array([2.  , 2.25, 2.5 , 2.75, 3.  ])
```

In [71]:
```
import numpy as np
np.linspace(5.0, 50, num=12)
```
Out[71]:
```
array([ 5.        ,  9.09090909, 13.18181818, 17.27272727, 21.36363636,
       25.45454545, 29.54545455, 33.63636364, 37.72727273, 41.81818182,
       45.90909091, 50.        ])
```

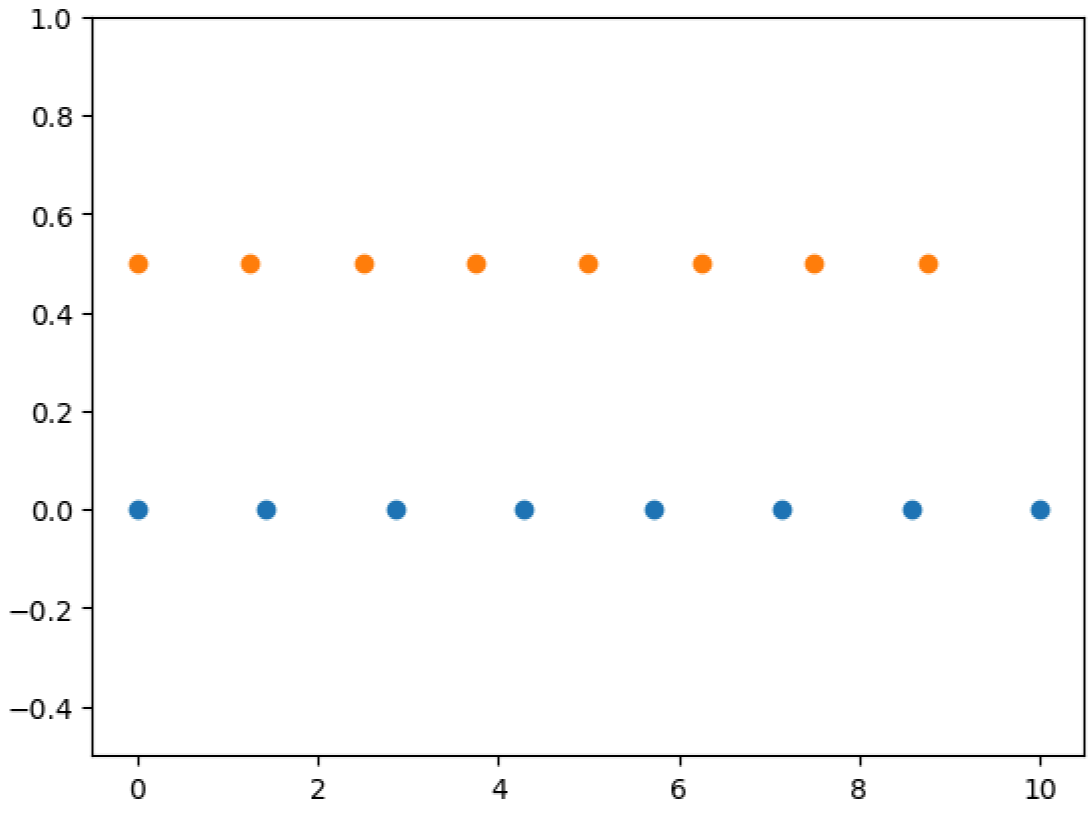
In [66]:
```
import numpy as np
np.linspace(2.0, 3.0, num=5 , endpoint=False)
```
Out[66]:
```
array([2. , 2.2, 2.4, 2.6, 2.8])
```

In [69]:
```
import numpy as np
np.linspace(1.0, 10.0, num=4, endpoint=False)
```
Out[69]:
```
array([1.  , 3.25, 5.5 , 7.75])
```

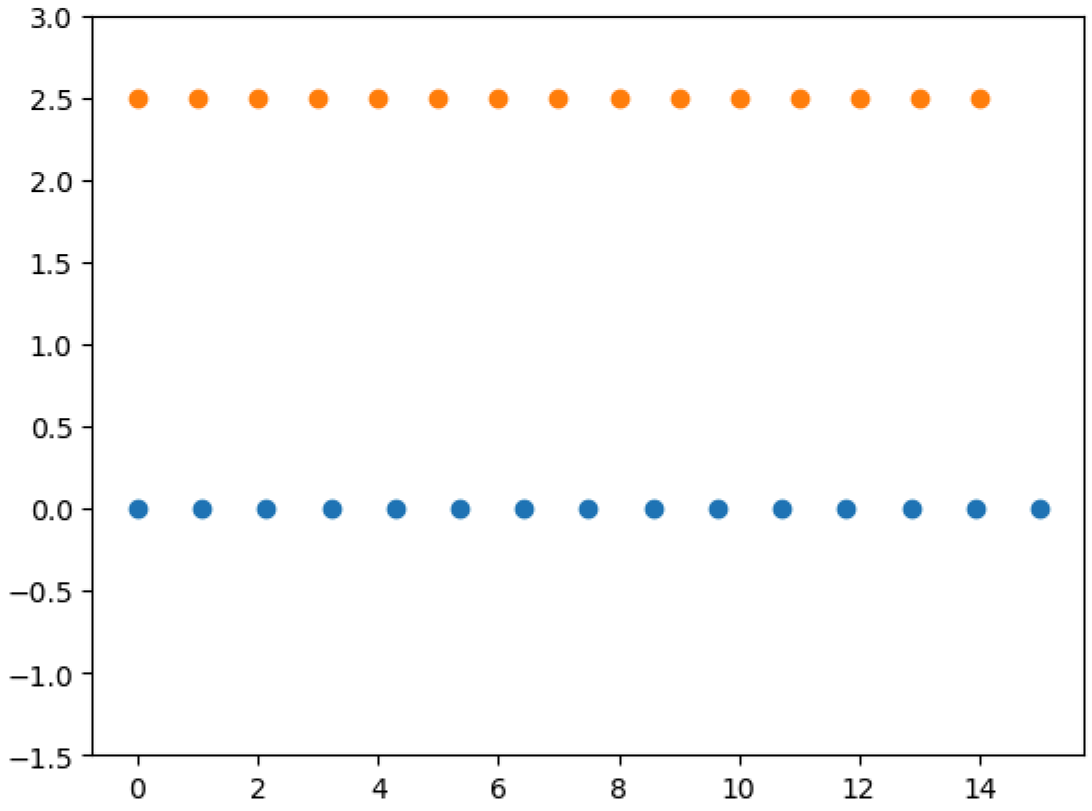
In [72]:
```
import numpy as np
np.linspace(2.0, 3.0, num=5, retstep=True)
```
Out[72]:
```
(array([2.  , 2.25, 2.5 , 2.75, 3.  ]), 0.25)
```

In [73]:
```
import numpy as np
np.linspace(3.0, 15.0, num=6, retstep = True)
```
Out[73]:
```
(array([ 3. ,  5.4,  7.8, 10.2, 12.6, 15. ]), 2.4)
```

In [74]:
```
import numpy as np
import matplotlib.pyplot as plt
N = 8
y = np.zeros(N)
xl = np.linspace(0, 10, N, endpoint=True)
x2 = np.linspace(0, 10, N, endpoint=False)
plt.plot(xl, y, 'o') # [<matplotlib.lines.Line2D object at 0x… >]
plt.plot(x2, y + 0.5, 'o') #[<matplotlib.lines.Line2D object at 0x… >]
plt.ylim([-0.5, 1])
(-0.5, 1)
plt.show()
```



In [76]:
```
import numpy as np
import matplotlib.pyplot as plt
N = 15
y = np.zeros(N)
xl = np.linspace(0, 15, N, endpoint=True)
x2 = np.linspace(0, 15, N, endpoint=False)
plt.plot(xl, y, 'o')
plt.plot(x2, y + 2.5, 'o')
plt.ylim([-1.5, 3])
(3.5, 6)
plt.show()
```



3.3 NumPy Arrays: Creation: from Existing Data

3.3.1 array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)

Create an array from existing data

Parameters:

object: array_like → An array, any object exposing the array interface, an object whose array method returns an array, or any (nested) sequence.

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.
- This argument can only be used to 'upcast' the array. For downcasting, use the .astype(t) method.

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.).

order: {'K', 'A', 'C', 'F'}, optional

- Specify the memory layout of the array. If object is not an array, the newly created array will be in C order (row major) unless 'F' is specified, in which case it will be in Fortran order (column major).
- If object is an array the following holds.

Order: no copy copy = True

K' unchanged F & C order preserved, otherwise most similar order

A' unchanged F order if input is F and not C, otherwise Corder

C' C order C order

F' F order F order

When copy=False and a copy is made for other reasons, the result is the same as if copy=True, with some exceptions for A, see the Notes section.

The default order is 'K'.

subok: bool, optional

- If True, then sub-classes will be passed-through,
  - otherwise the returned array will be forced to be a base-class array (default).

ndmin: int, optional

- Specifies the minimum number of dimensions that the resulting array should have.
- Ones will be pre-pended to the shape as needed to meet this requirement.

Returns:

out: ndarray

An array object satisfying the specified requirements.

Run the following 3 code blocks:

In [77]:
```
import numpy as np
np.array([1, 2, 3])
```

Out[77]:
```
array([1, 2, 3])
```

In [78]:
```
import numpy as np
np.array([12,23,34,45])
```

Out[78]:
```
array([12, 23, 34, 45])
```

In [79]:
```
import numpy as np
np.array([[1, 2], [3, 4]])
```

Out[79]:
```
array([[1, 2],
       [3, 4]])
```

In [80]:
```
import numpy as np
np.array([[56,67,],[78,89]])
```

Out[80]:
```
array([[56, 67],
       [78, 89]])
```

In [81]:
```
np.array([1, 2, 3, 4, 5], ndmin=2)
```

Out[81]:
```
array([[1, 2, 3, 4, 5]])
```

In [82]:
```
np.array([5,4,3,2,1], ndmin=1)
```

Out[82]:
```
array([5, 4, 3, 2, 1])
```

3.3.2 asarray(a, dtype=None, order=None)

Convert the input a into an array

Parameters:

a: array_like → Input data, in any form that can be converted to an array.

- (This includes lists, lists of tuples, tuples, tuples of tuples, tuples of lists and ndarrays.)

dtype: data-type, optional→ By default, the data-type is inferred from the input data.

order: {'C', 'F'}, optional

- Whether to use row-major (C-style) or column-major (Fortran-style) memory representation. Defaults to 'C'.

Returns:

out: ndarray

Array interpretation of a. No copy is performed if the input is already an ndarray with matching dtype and order. If a is a subclass of ndarray, a base class ndarray is returned.

Run the following code block:

In [83]:
```
import numpy as np
a = [1, 2, 3, 4, 5]
b = np.asarray([1, 2, 3, 4, 5])
print (b)
c = np.asarray(a)
print (c)
```

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

In [84]:
```
import numpy as np
x = [10, 9, 8, 7, 6]
y = np.asarray([10, 9, 8, 7, 6])
print(y)
z = np.asarray(x)
print(z)
```

```
[10  9  8  7  6]
[10  9  8  7  6]
```

3.3.3 fromstring(string, dtype=float, count=-1, sep='')

A new 1-D array initialized from raw binary or text data in a string.

Parameters:

string: sir → A string containing the data.

dtype: data-type, optional → The data type of the array; default: float.

- For binary input data, the data must be in exactly this format.

count: int, optional → Read this number of dtype elements from the data.

- If this is negative (the default), the count will be determined from the length of the data.)

sep: str, optional

- If not provided or, equivalently, the empty string, the data will be interpreted as binary data.
- Otherwise, as ASCII text with decimal numbers.
- Also in this latter case, this argument is interpreted as the string separating numbers in the data; extra whitespace between elements is also ignored.

Returns: arr: ndarray

Run the following 2 code blocks:

In [102…
```
import numpy as np
aStr = "This is a sentence."
np.fromstring(aStr, dtype=np.uint8)
```

```
C:\Users\deaun\AppData\Local\Temp\ipykernel_26168\830202303.py:3: DeprecationWarning:
The binary mode of fromstring is deprecated, as it behaves surprisingly on unicode in
puts. Use frombuffer instead
  np.fromstring(aStr, dtype=np.uint8)
```

Out[102]:
```
array([ 84, 104, 105, 115,  32, 105, 115,  32,  97,  32, 115, 101, 110,
       116, 101, 110,  99, 101,  46], dtype=uint8)
```

In [94]:
```
import numpy as np
stringster = 'Is this really a sentence?'
np.fromstring(stringster, dtype=np.uint8)
```

```
C:\Users\deaun\AppData\Local\Temp\ipykernel_26168\1057859692.py:3: DeprecationWarnin
g: The binary mode of fromstring is deprecated, as it behaves surprisingly on unicode
inputs. Use frombuffer instead
  np.fromstring(stringster, dtype=np.uint8)
```

Out[94]:
```
array([ 73, 115,  32, 116, 104, 105, 115,  32, 114, 101,  97, 108, 108,
       121,  32,  97,  32, 115, 101, 110, 116, 101, 110,  99, 101,  63],
      dtype=uint8)
```

In [95]:
```
import numpy as np
aStr = "This is a sentence."
anArray = np.fromstring(aStr, dtype=np.uint8)
print(anArray)
```

```
[ 84 104 105 115  32 105 115  32  97  32 115 101 110 116 101 110  99 101
  46]
```

```
C:\Users\deaun\AppData\Local\Temp\ipykernel_26168\2038893667.py:3: DeprecationWarnin
g: The binary mode of fromstring is deprecated, as it behaves surprisingly on unicode
inputs. Use frombuffer instead
  anArray = np.fromstring(aStr, dtype=np.uint8)
```

In [97]:
```
import numpy as np
reString = 'I guess it is a string.'
np.fromstring(reString, dtype=np.uint8)
```

```
C:\Users\deaun\AppData\Local\Temp\ipykernel_26168\400706221.py:3: DeprecationWarning:
The binary mode of fromstring is deprecated, as it behaves surprisingly on unicode in
puts. Use frombuffer instead
  np.fromstring(reString, dtype=np.uint8)
```

Out[97]:
```
array([ 73,  32, 103, 117, 101, 115, 115,  32, 105, 116,  32, 105, 115,
        32,  97,  32, 115, 116, 114, 105, 110, 103,  46], dtype=uint8)
```

3.3.4 diag(v, k=0)

Extract a diagonal or construct a diagonal array.

See the more detailed documentation for numpy.diagonal if you use this function to extract a diagonal and wish to write to the resulting array; whether it returns a copy or a view depends on what version of numpy you are using.

Parameters:

v: array_like

- If v is a 2-D array, returns a copy of its k-th diagonal.
- If v is a 1-D array, return a 2-D array with v on the k-th diagonal.

k: int, optional

- Diagonal in question. The default is 0.
- Use k>0 for diagonals above the main diagonal.
- k<0 for diagonals below the main diagonal.

Returns:

out: ndarray

The extracted diagonal or constructed diagonal array.

Run the following 4 code blocks:

In [103…

```
import numpy as np
x = np.arange(9).reshape((3, 3))
print(x)
```

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

In [107…

```
import numpy as np
a = np.arange(16).reshape((4, 4))
print(a)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

In [105…

```
import numpy as np
x = np.arange(9).reshape((3,3))
x
```

Out[105]:

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

In [108…

```
import numpy as np
b = np.arange(25).reshape((5,5))
b
```

Out[108]:

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

In [111…

```
import numpy as np
x = np.arange(9).reshape((3,3))
np.diag(x)
```

Out[111]:

```
array([0, 4, 8])
```

In [112…

```
import numpy as np
z = np.arange(16).reshape((4,4))
np.diag(z)
```

Out[112]:

```
array([ 0,  5, 10, 15])
```

In [113…

```
import numpy as np
x = np.arange(9).reshape((3,3))
np.diag(x, k = 1)
```

Out[113]:

```
array([1, 5])
```

In [117…
```
import numpy as np
last = np.arange(25).reshape((5,5))
np.diag(last, k = 2)
```

Out[117]:
```
array([ 2,  8, 14])
```

4. Creation of NumPy Arrays: Simple Methods

4.1. 1-D Arrays

4.1.1 Using ndarray.arange()

Run the following code block:

In [118…
```
import numpy as np
np.arange(8)
```

Out[118]:
```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

In [119…
```
import numpy as np
np.arange(20)
```

Out[119]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

4.2 2-D Arrays

4.2.1 Using ndarray.arange().reshape(a, b)

Run the following 4 code blocks:

In [120…
```
x = np.arange(12).reshape((3,4))
x
```

Out[120]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [121…
```
y = np.arange(20).reshape((4,5))
print(y)
```

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

5. Numpy: Built-In Functions: Array Manipulation

5.1 Overview

reshape(a, newshape, order='C') Gives a new shape to an array without changing its data.

flat() A 1-D iterator over the array.

flatten(order='C') Return a copy of the array collapsed into one dimension.

transpose(a, axes=None) Permute the dimensions of an array.

concatenate((a1, a2, ...), axis=0) Join a sequence of arrays along an existing axis.

split(ary, indices_or_sections, axis=0) Split an array into multiple sub-arrays.

delete(arr, obj, axis=None) Returns a new array with sub-arrays along an axis deleted. For a one dimensional array, this returns those entries not returned by arr[obj].

insert(arr, obj, values, axis=None) Insert values along the given axis before the given indices.

append(arr, values, axis=None) Append values to the end of an array.

resize (a, new_shape) Returns a new array with the specified shape. If the new array is larger than the original array,

- then the new array is filled with repeated copies of a. **NOTE:** this behavior is different from a.resize(new_shape),
- which fills with zeros instead of repeated copies of a.

trim_zeros(filt, trim='fb') Trims the leading and/or trailing zeros from a 1-D array or sequence.

unique(ar, return_index=False, return_inverse=False, return_counts=False, axis=None) Finds the unique elements of an array. Returns the sorted unique elements of an array. There are three optional outputs in addition to the unique elements:

- the indices of the input array that give the unique values,
- the indices of the unique array that reconstruct the input array,
- the number of times each unique value comes up in the input array.

flip(m, axis) Reverse the order of elements in an array along the given axis. The shape of the array is preserved, but the elements are reordered .

fliplr(m) Flips array in the left/right direction. Flips the entries in each row in the left/right direction. Columns are preserved but appear in a different order than before.

flipud (m) Flips array in the up/down direction. Flips the entries in each column in the up/down direction. Rows are preserved, but appear in a different order than before.

tile(A, reps) Construct an array by repeating A the number of times given by reps. If reps has length d, the result will have dimension of max(d, A.ndim).

repeat(a, repeats, axis=None)

Repeats elements of an array