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

Chapter#4 NumPy Basics: Arrays and Vectorized Computation

Creating ndarrays Pg#88 (106)

In [3]:
```
# Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
data2
```

Out[3]:
```
[[1, 2, 3, 4], [5, 6, 7, 8]]
```

Table 4-1. Array creation functions

Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype; copies the input data by default
`asarray`	Convert input to ndarray, but do not copy if the input is already an ndarray
`arange`	Like the built-in `range` but returns an ndarray instead of a list
`ones`, `ones_like`	Produce an array of all 1s with the given shape and dtype; `ones_like` takes another array and produces a ones array of the same shape and dtype
`zeros`, `zeros_like`	Like `ones` and `ones_like` but producing arrays of 0s instead
`empty`, `empty_like`	Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros`
`full`, `full_like`	Produce an array of the given shape and dtype with all values set to the indicated "fill value" `full_like` takes another array and produces a filled array of the same shape and dtype
`eye`, `identity`	Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)

In [2]:
```
# asarray()
# input is not an array
list1 = [12,13,14]
asArray = np.asarray(list1)
asArray
```

Out[2]:
```
array([12, 13, 14])
```

In [4]:
```
# asarray()
# input is already an array
arr1 = [12,13,14]
asArray1 = np.asarray(arr1)
asArray
```

Out[4]:
```
array([12, 13, 14])
```

In [9]:
```
# np.ones() >>> as an arg u can specify shape
onesArr = np.ones((2,3))
onesArr
```

Out[9]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

In [10]:
```
# np.ones_like() >>> as an arg u specify an array, it will infer the shape from that array and create ones ar
ray

testArr = np.array([12,14,16,18,20])

onesLikeArr = np.ones_like(testArr)
onesLikeArr
```

Out[10]:
```
array([1, 1, 1, 1, 1])
```

In [15]:
```
testArr1 = np.array([[1,2,3], [4,5,6]])

emptyLikeArr = np.empty_like(testArr1)
print(emptyLikeArr)
```

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

In [22]:
```
# np.full()  >>> 1st arg="shape", 2nd arg="fill_value"
# fill_value is what we want all elements to be...
# Here we want that all our elements should be 10

fullArr = np.full((3,3),10)
fullArr
```

Out[22]:
```
array([[10, 10, 10],
       [10, 10, 10],
       [10, 10, 10]])
```

In [23]:
```
# np.full_like()  >>> 1st 1rg ="arrName",2nd arg="fill_value"
# 1st arg will infer or preserve shape of array and create an array accordance with that shape
# and 2nd arg will fill whatever value we would want

testArr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

fullLikeArr = np.full_like(testArr, 9)
fullLikeArr
```

Out[23]:
```
array([[9, 9, 9, 9, 9],
       [9, 9, 9, 9, 9]])
```

In [25]:
```
# np.eye() >>> arg="N" bcoz it creates N x N identity matrix, ones on the diagonal and 0 elsewhere

eyeArr = np.eye(4)
eyeArr
```

Out[25]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

In [27]:
```
# np.identity() arg="N"

identityArr = np.identity(5)
identityArr
```

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

Data Types for ndarrays Pg#90 (108)

- It’s important to be cautious when using the numpy.string_ type, as string data in NumPy is fixed size and may truncate input without warning. pandas has more intuitive out-of-the-box behavior on non-numeric data.
- **Calling `astype` always creates a `new array (a copy of the data)`, even if the new dtype is the same as the old dtype.**

`Basic Indexing and Slicing` Pg#94 (112)

- array slices are views on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array.

`Boolean Indexing` Pg#99 (117)

In [5]:
```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
data = np.array([[ 0.0929, 0.2817, 0.769 , 1.2464],
                 [ 1.0072, -1.2962, 0.275 , 0.2289],
                 [ 1.3529, 0.8864, -2.0016, -0.3718],
                 [ 1.669 , -0.4386, -0.5397, 0.477 ],
                 [ 3.2489, -1.0212, -0.5771, 0.1241],
                 [ 0.3026, 0.5238, 0.0009, 1.3438],
                 [-0.7135, -0.8312, -2.3702, -1.8608]])
```

In [6]:
```
data[names == "Bob"]
```

Out[6]:
```
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])
```

To select everything but 'Bob', you can either use `!=` or negate the condition using `~`:

In [9]:
```
a = data[~(names == 'Bob')]

b = data[names != "Bob"]

print(a)
print("=====================================================")
print(b)
```

```
[[ 1.0072e+00 -1.2962e+00  2.7500e-01  2.2890e-01]
 [ 1.3529e+00  8.8640e-01 -2.0016e+00 -3.7180e-01]
 [ 3.2489e+00 -1.0212e+00 -5.7710e-01  1.2410e-01]
 [ 3.0260e-01  5.2380e-01  9.0000e-04  1.3438e+00]
 [-7.1350e-01 -8.3120e-01 -2.3702e+00 -1.8608e+00]]
=====================================================
[[ 1.0072e+00 -1.2962e+00  2.7500e-01  2.2890e-01]
 [ 1.3529e+00  8.8640e-01 -2.0016e+00 -3.7180e-01]
 [ 3.2489e+00 -1.0212e+00 -5.7710e-01  1.2410e-01]
 [ 3.0260e-01  5.2380e-01  9.0000e-04  1.3438e+00]
 [-7.1350e-01 -8.3120e-01 -2.3702e+00 -1.8608e+00]]
```

The ~ operator can be useful when you want to `invert a general condition` :

In [10]:
```
cond = names == 'Bob'
```

In [11]:
```
data[cond]
```

Out[11]:
```
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])
```

In [12]:
```
data[~cond]
```

Out[12]:
```
array([[ 1.0072e+00, -1.2962e+00,  2.7500e-01,  2.2890e-01],
       [ 1.3529e+00,  8.8640e-01, -2.0016e+00, -3.7180e-01],
       [ 3.2489e+00, -1.0212e+00, -5.7710e-01,  1.2410e-01],
       [ 3.0260e-01,  5.2380e-01,  9.0000e-04,  1.3438e+00],
       [-7.1350e-01, -8.3120e-01, -2.3702e+00, -1.8608e+00]])
```

Selecting data from an array by boolean indexing always creates a copy of the data, even if the returned array is unchanged.

Fancy Indexing

Keep in mind that fancy indexing, unlike slicing, always copies the data into a new array.

In [3]:
```
arr = np.arange(32).reshape((8, 4))
arr
```

Out[3]:
```
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, 25, 26, 27],
       [28, 29, 30, 31]])
```

In [4]:
```
# the result of fancy indexing is always one-dimensional.
arr[[1, 5, 7, 2], [0, 3, 1, 2]]
```

Out[4]:
```
array([ 4, 23, 29, 10])
```

In [5]:
```
arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
```

Out[5]:
```
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Transposing Arrays and Swapping Axes

- Transposing is a special form of reshaping that similarly returns a `view` on the underlying data `without copying` anything. Arrays have the `transpose` method and also the special `T` attribute
- For higher dimensional arrays, transpose will accept a tuple of axis numbers to permute the axes
- ndarray has the method `swapaxes`, which takes a `pair of axis numbers and switches the indicated axes` to rearrange the data

In [6]:
```
arr = np.arange(16).reshape((2, 2, 4))
arr
```

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

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
```

In [9]:
```
# swapaxes similarly returns a view on the data without making a copy.
arr.swapaxes(1, 2)
```

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

       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

Universal Functions: Fast Element-Wise Array Functions

- a ufunc can return multiple arrays, `modf` is one example, a vectorized version of the built-in Python divmod; it returns the fractional and integral parts of a floating-point array

In [10]:
```
arr = np.arange(7)
arr
```

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

In [11]:
```
np.sqrt(arr)
```

Out[11]:
```
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974])
```

In [13]:
```
# original array is unchanged
arr
```

Out[13]:
```
array([0, 1, 2, 3, 4, 5, 6])
```

In [20]:
```
# Ufuncs accept an optional out argument that allows them to operate in-place on arrays:
np.sqrt(arr,arr)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-615c19e48a1a> in <module>
      1 # Ufuncs accept an optional out argument that allows them to operate in-place on arrays:
----> 2 np.sqrt(arr,arr)

TypeError: ufunc 'sqrt' output (typecode 'd') could not be coerced to provided output parameter (typecode
 'l') according to the casting rule ''same_kind''
```

Array Oriented Programming with Arrays

- suppose we wished to evaluate the function `sqrt(x^2 + y^2)` across a regular grid of values.
- The `np.meshgrid` function `takes two 1D arrays` and `produces two 2D matrices` corresponding to all pairs of (x, y) in the two arrays:

In [10]:
```
points = np.arange(-5, 5, 0.01)  # 1000 equally spaced points
```

In [11]:
```
xs, ys = np.meshgrid(points, points)
```

In [12]:
```
xs
```

Out[12]:
```
array([[-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       ...,
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99]])
```

In [13]:
```
ys
```

Out[13]:
```
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

In [14]:
```
# Now, evaluating the function sqrt(x^2 + y^2):

z = np.sqrt(xs ** 2, ys ** 2)
z
```

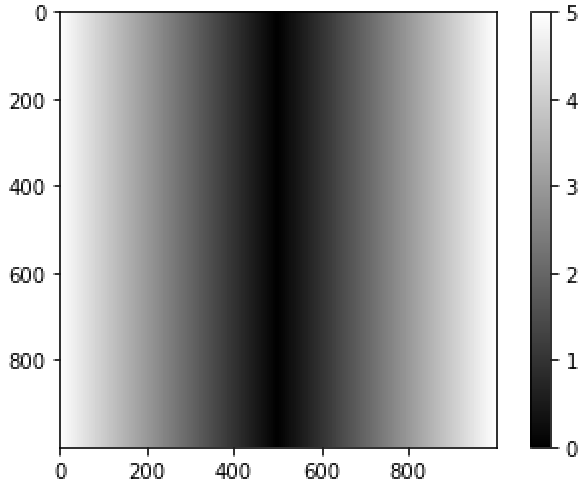
Out[14]:
```
array([[5.  , 4.99, 4.98, ..., 4.97, 4.98, 4.99],
       [5.  , 4.99, 4.98, ..., 4.97, 4.98, 4.99],
       [5.  , 4.99, 4.98, ..., 4.97, 4.98, 4.99],
       ...,
       [5.  , 4.99, 4.98, ..., 4.97, 4.98, 4.99],
       [5.  , 4.99, 4.98, ..., 4.97, 4.98, 4.99],
       [5.  , 4.99, 4.98, ..., 4.97, 4.98, 4.99]])
```

In [20]:
```
import matplotlib.pyplot as plt
```

In [19]:
```
# from matplotlib import pyplot as plt
```

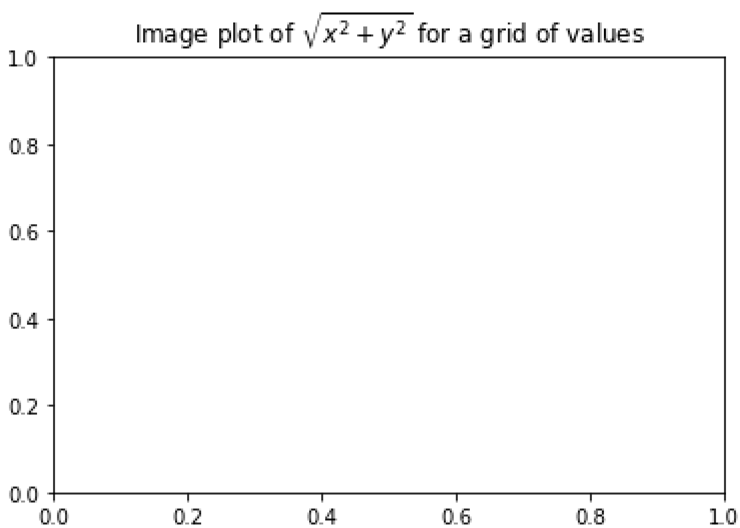
In [21]:
```
plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
```

Out[21]:
```
<matplotlib.colorbar.Colorbar at 0x7e17ad8>
```



In [22]:
```
plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")
```

Out[22]:
```
Text(0.5, 1.0, 'Image plot of $\\sqrt{x^2 + y^2}$ for a grid of values')
```



Methods for boolean arrays:

- any(), all()
- These methods also work with non-boolean arrays, where non-zero elements evaluate to True.

Unique and other Set Logic:

- Contrast `np.unique` with the pure Python alternative: `sorted(set(arrName))`
- `np.in1d`, tests membership of the values in one array in another, returning a `boolean array`:

In [3]:
```
values = np.array([6, 0, 0, 3, 2, 5, 6])
np.in1d(values, [2, 3, 6])
```

Out[3]:
```
array([ True, False, False,  True,  True, False,  True])
```

File Input and Output with Arrays

- NumPy is able to save and load data to and from disk either in text or binary format. We only discuss NumPy's built-in binary format.
- Arrays are saved `by default` in an `uncompressed raw binary format` with file extension `.npy`.
- You save multiple arrays in an `uncompressed archive` using `np.savez` and passing the `arrays as keyword arguments`
- When loading an `.npz file`, you get back a `dict-like object` that loads the individual arrays lazily

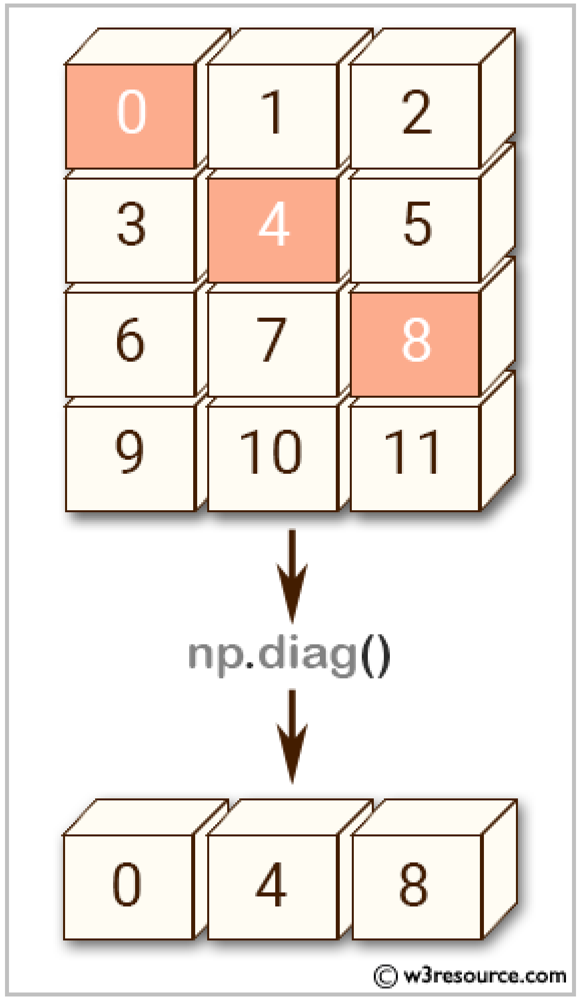
Linear Algebra

1) `np.diag(arr, k)`

- **The diag() function is used to extract a diagonal or construct a diagonal array.**
- Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array,
- or convert a 1D array into a square matrix with zeros on the off-diagonal
- `K` is diagonal and is `optional parameter`. `The default is 0`. Use `k>0 for diagonals above the main diagonal`, and `k<0 for diagonals below the main diagonal`

Example 1:

Pictorial Representation



In [3]:
```
a = np.arange(12).reshape((4,3))
a
```

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

In [4]:
```
np.diag(a)
```

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

Example 2:

In [5]:
```
a = np.arange(12).reshape((4,3))
np.diag(a, k=1)
```

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

Example 3

In [6]:
```
a = np.arange(12).reshape((4,3))
np.diag(a, k=-1)
```

Out[6]:
```
array([ 3,  7, 11])
```

Example 4

convert a 1D array into a square matrix with zeros on the off-diagonal

In [8]:
```
a = np.arange(12).reshape((4,3))
a
```

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

In [12]:
```
b = np.diag(a)
b
```

Out[12]:
```
array([0, 4, 8])
```

In [13]:
```
np.diag(b)
```

Out[13]:
```
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 8]])
```

In [9]:
```
np.diag(np.diag(a))
```

Out[9]:
```
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 8]])
```

np.trace()

- Compute the sum of the diagonal elements

In [15]:
```
arr = np.eye(4, dtype="int32")
arr
```

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

In [18]:
```
# arr.trace()

np.trace(arr)
```

Out[18]:
```
4
```

Pseudorandom Number Generation

- you can get a 4 × 4 array of samples from the standard normal distribution using normal:

```
samples = np.random.normal(size=(4, 4))
```

In [2]:
```
samples = np.random.normal(size=(4, 4))
samples
```

Out[2]:
```
array([[-0.198571  ,  0.54952326, -1.60395172,  0.50184499],
       [-0.8050926 , -0.70441228, -0.55273046, -0.48594149],
       [-1.54220781, -0.91798573,  0.4067151 , -0.82948599],
       [ 1.76680114, -0.48113257, -0.34890996,  1.15075424]])
```

In [3]:
```
from random import normalvariate
```

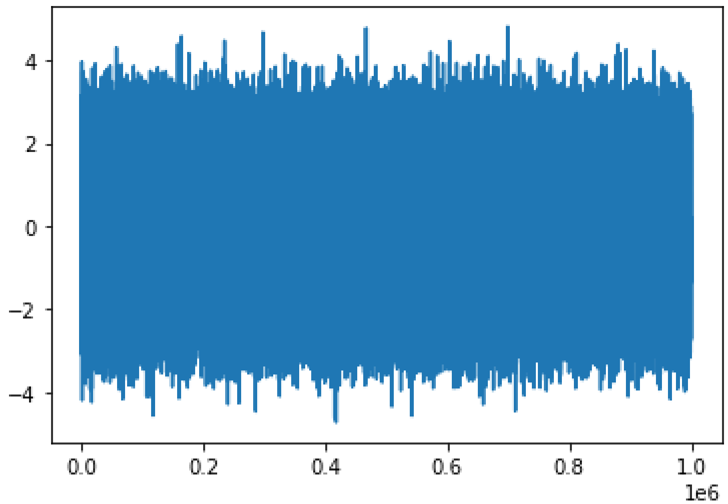
In [4]:
```
N = 1000000
```

In [11]:
```
# normalvariate(mean, StandarDeviation)

import matplotlib.pyplot as plt

nums = []
for i in range(N):
    samples = normalvariate(0, 1)
    nums.append(samples)

# Plotting a graph
plt.plot(nums)
plt.show()
```



We say that `these are pseudorandom numbers` because they are `generated by an algorithm with deterministic behavior based on the seed of the random number generator`. You can `change NumPy's random number generation seed using np.random.seed:`

In [12]:
```
np.random.seed(1234)
```

The data generation functions in numpy.random `use a global random seed`. `To avoid global state`, you can `use numpy.random.RandomState to create a random number generator isolated from others:`

In [13]:
```
randomNumberGenerator = np.random.RandomState(1234)
randomNumberGenerator
```

Out[13]:
```
RandomState(MT19937) at 0x77F2778
```

In [14]:
```
randomNumberGenerator.randn(10)
```

Out[14]:
```
array([ 0.47143516, -1.19097569,  1.43270697, -0.3126519 , -0.72058873,
        0.88716294,  0.85958841, -0.6365235 ,  0.01569637, -2.24268495])
```

In [ ]: