2d.Images.ndarrays

November 2, 2024

1 Image Representation of ndarrays

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
%config InlineBackend.figure_formats = 'retina'
```

Modify plotting layout for images, i.e. put xtick/xlabels at the top of the image.

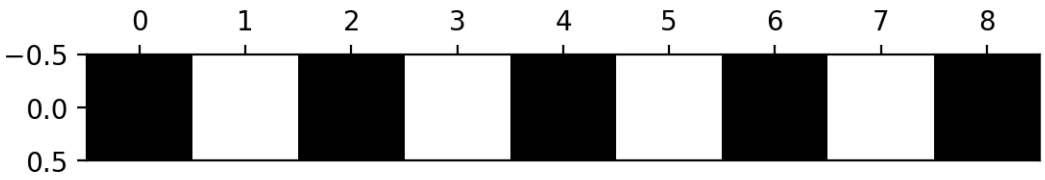
[ ]:
```
matplotlib.rcParams.update( {
    "xtick.top" : True,
    "xtick.labeltop" : True,
    "xtick.bottom": False,
    "xtick.labelbottom": False,
}
)
```

1.1 Display a numpy array of 0's and 1's.

[ ]:
```
shades = np.array( [ 0, 1, 0, 1, 0, 1, 0, 1, 0, ] )

print(list(shades))
print(shades.shape)
print()

# Convert from 1-D array to 2-D matrix
img = np.array( [
    shades,
] )

print(img)
print(img.shape)
print()

plt.imshow(img, cmap='gray') ;
```

```
[0, 1, 0, 1, 0, 1, 0, 1, 0]
(9,)
```

```
[[0 1 0 1 0 1 0 1 0]]
(1, 9)
```

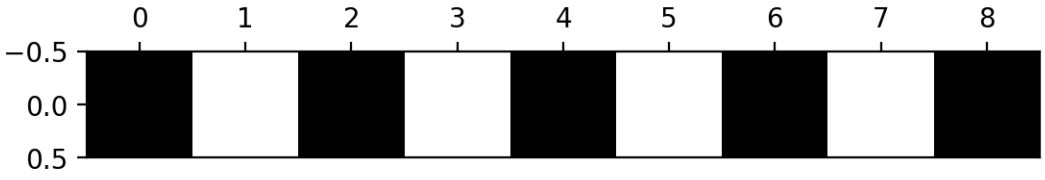


1.2 Using reshape

Go from 1-D to 2-D using the `reshape()` method.

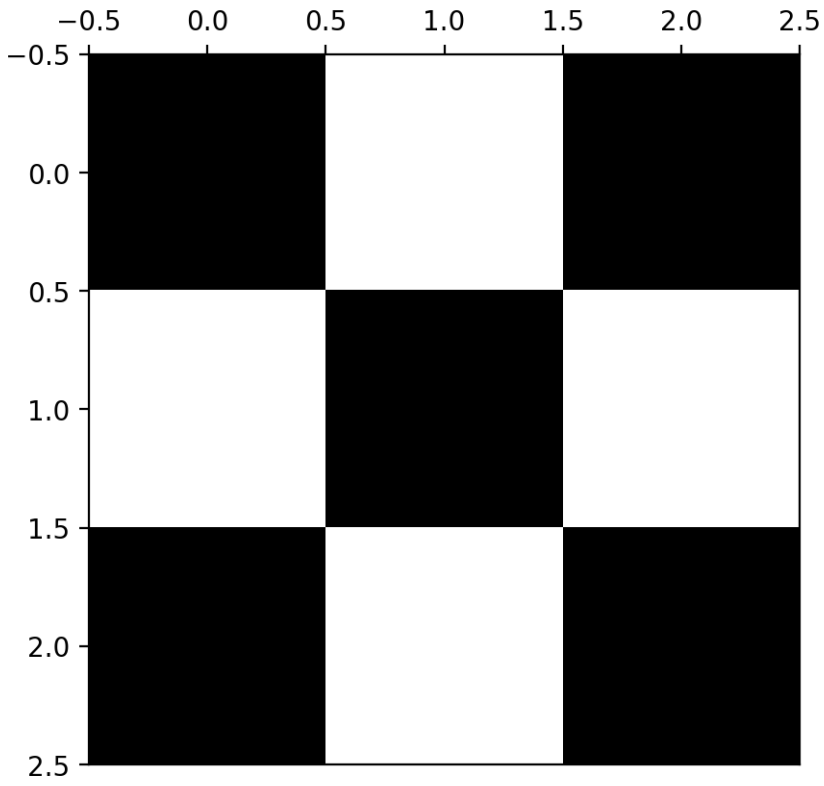
[ ]:
```
shades = np.array( [ 0, 1, 0, 1, 0, 1, 0, 1, 0, ] )

print(list(shades))
print(shades.shape)
print()

# reshape into rows, columns ( -1 == whatever it takes to work )
img = shades.reshape(1,-1)
print(img)
print(img.shape)
print()

plt.imshow(img, cmap='gray') ;
```

```
[0, 1, 0, 1, 0, 1, 0, 1, 0]
(9,)

[[0 1 0 1 0 1 0 1 0]]
(1, 9)
```



[ ]:
```
# reshape into rows, columns ( -1 == whatever it takes to work )
img = shades.reshape(3,-1)
print(img)
print(img.shape)
print()

plt.imshow(img, cmap='gray') ;
```

```
[[0 1 0]
 [1 0 1]
 [0 1 0]]
(3, 3)
```

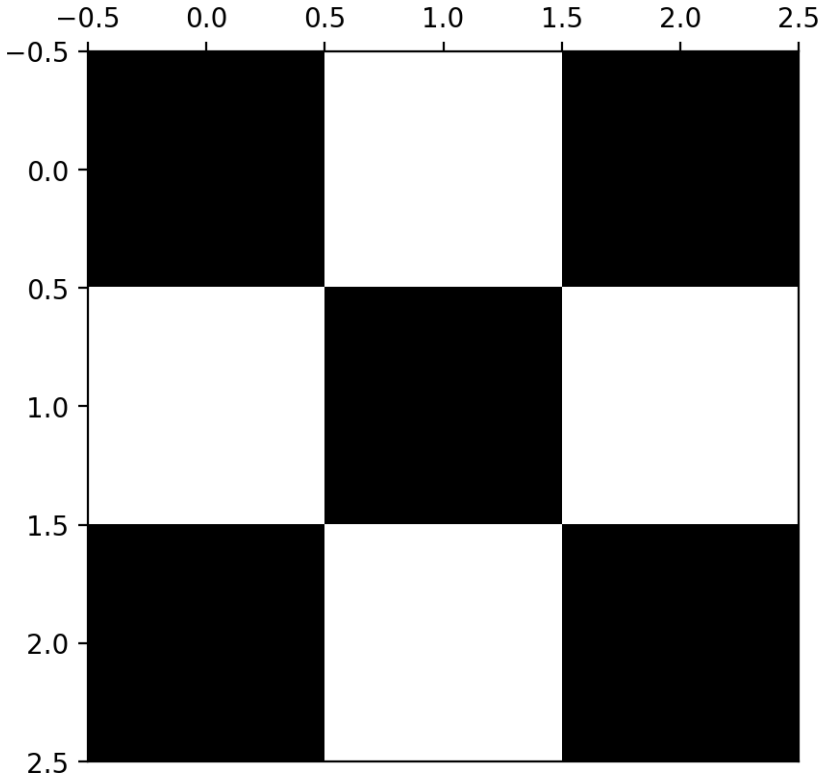


1.3 Using list repetition.

[ ]:
```
# repeating list [0, 1] using '*' list operator
shades = np.array( [ 0, 1, ] * 4 + [0] )

print(list(shades))
print(shades.shape)
print()
```

```
# reshape into rows, columns ( -1 == whatever it takes to work )
img = shades.reshape(3,-1)
print(img)
print(img.shape)
print()

plt.imshow(img, cmap='gray') ;
```

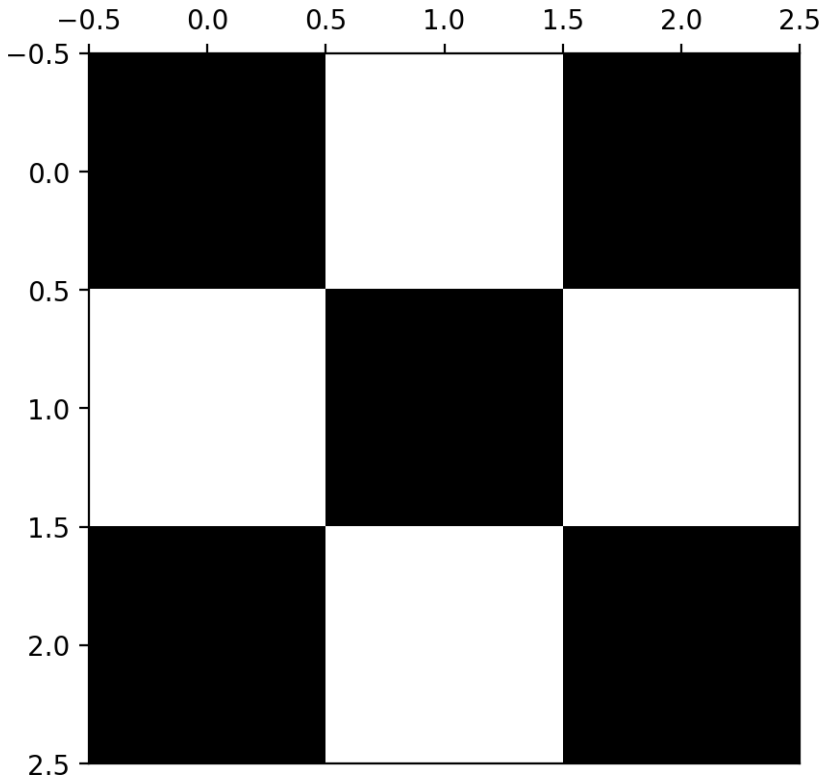
```
[0, 1, 0, 1, 0, 1, 0, 1, 0]
(9,)

[[0 1 0]
 [1 0 1]
 [0 1 0]]
(3, 3)
```



[ ]:
```
# repeating list [0, 1] using np.tile()
shades = np.tile( [ 0, 1, ], 5 ).flatten()[:9]

print(list(shades))
```

```
print(shades.shape)
print()

# reshape into rows, columns ( -1 == whatever it takes to work )
img = shades.reshape( 3, -1 )
print(img)
print(img.shape)
print()

plt.imshow(img, cmap='gray') ;
```

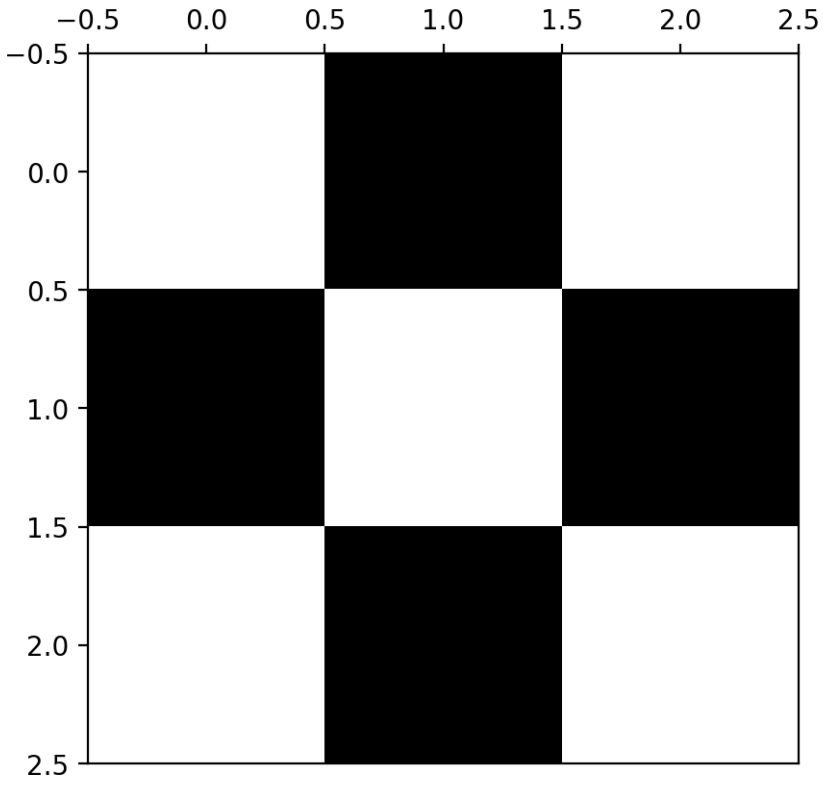
```
[0, 1, 0, 1, 0, 1, 0, 1, 0]
(9,)

[[0 1 0]
 [1 0 1]
 [0 1 0]]
(3, 3)
```



1.4 Using `logical_not()` to invert an array

[ ]:
```
shades = np.array( ( [ 0, 1, ] * 4 + [0] ) )

print(shades)
print(shades.shape)
print()

# reshape into rows, columns ( -1 == whatever it takes to work )
img = np.logical_not( shades ).reshape(3,-1)
print(img + 0 )
print(img.shape)
print()

plt.imshow(img, cmap='gray') ;
```

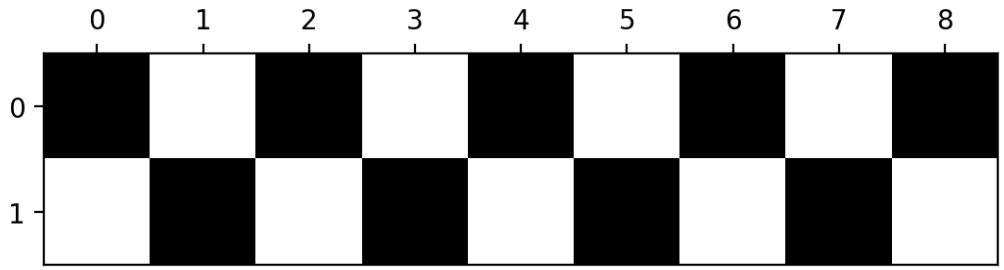
```
[0 1 0 1 0 1 0 1 0]
(9,)

[[1 0 1]
 [0 1 0]
 [1 0 1]]
(3, 3)
```



1.5 Using append() to combine arrays

[ ]:
```
shades = np.array( ( [ 0, 1, ] * 4 + [0] ) )

print(list(shades))
print(list(np.logical_not( shades ) + 0 ))
print(shades.shape)
print()

# reshape into rows, columns ( -1 == whatever it takes to work )
img = np.append( shades, np.logical_not( shades )).reshape(2,-1)
print(img)
print(img.shape)
print()

plt.imshow(img, cmap='gray') ;
```

```
[0, 1, 0, 1, 0, 1, 0, 1, 0]
[1, 0, 1, 0, 1, 0, 1, 0, 1]
(9,)

[[0 1 0 1 0 1 0 1 0]
 [1 0 1 0 1 0 1 0 1]]
(2, 9)
```

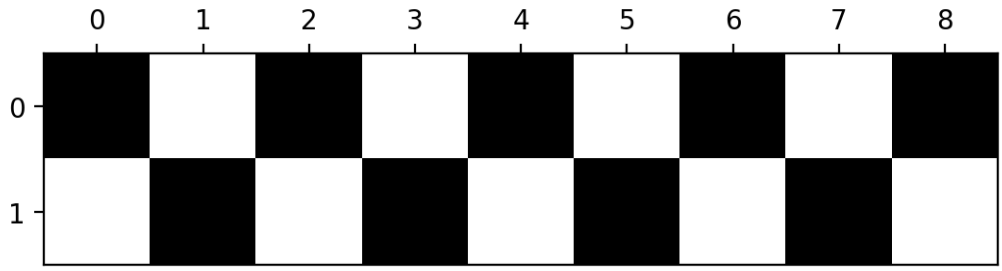


1.6 Using modulo

[ ]:
```
# Since the 0's and 1's alternate, this can be accomplished with a modulo ( i.e.
  ↪ remainder ) operator
shades = np.arange(18) % 2

print(list(shades))
```

```
print(shades.shape)
print()

# reshape into rows, columns ( -1 == whatever it takes to work )
img = shades.reshape(2,-1)
print(img)
print(img.shape)
print()

plt.imshow(img, cmap='gray') ;
```

```
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
(18,)

[[0 1 0 1 0 1 0 1 0]
 [1 0 1 0 1 0 1 0 1]]
(2, 9)
```

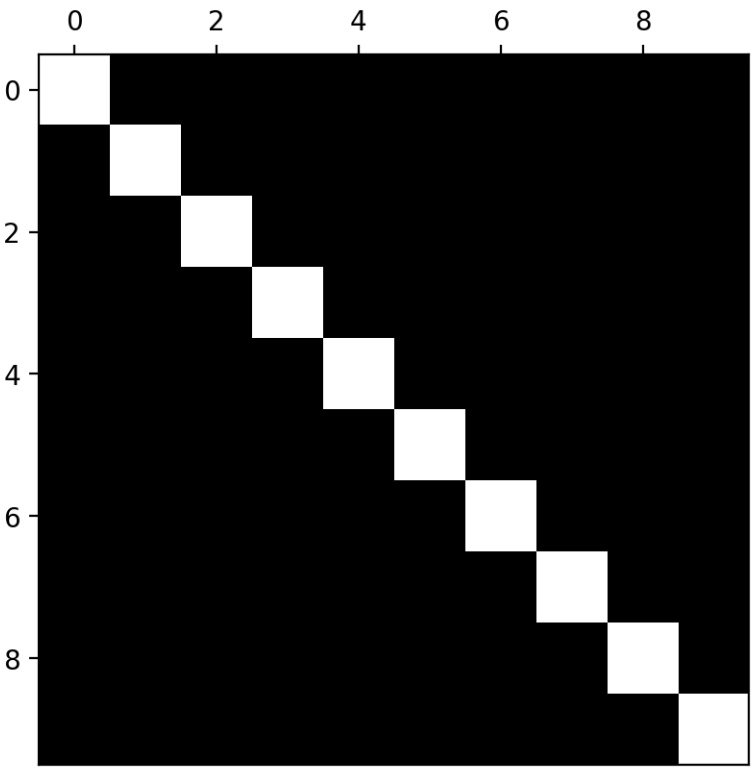


1.7 Create a diagonal line

[ ]:
```
img = np.zeros((10, 10))
np.fill_diagonal(img, 1)

print(img)
print(img.shape)
print()

plt.imshow(img, cmap='gray') ;
```

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

```
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
(10, 10)
```



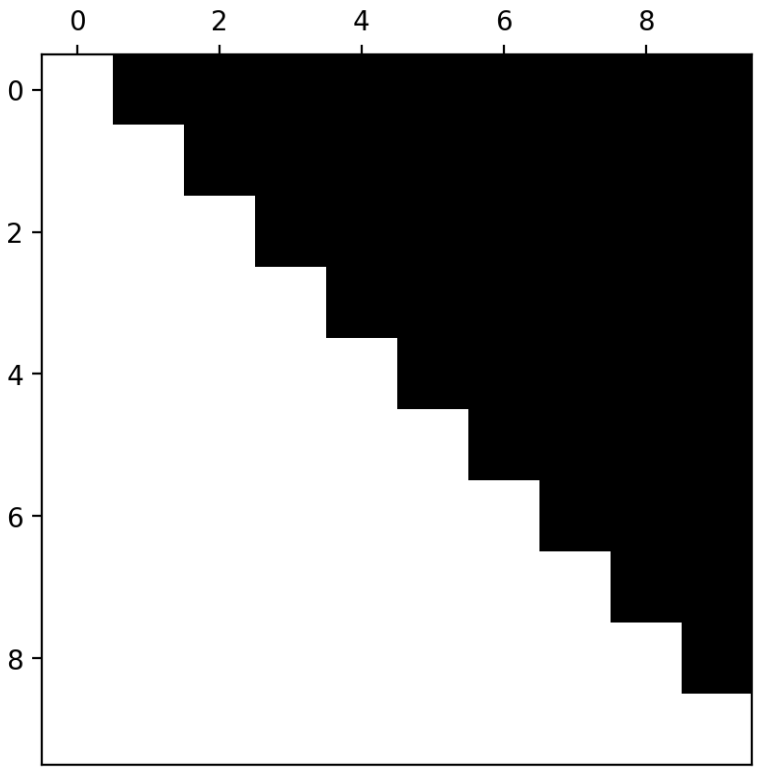
1.8 Keep lower triangle

`tril()` keeps the lower triangle and fills the upper triangle with zeros. By default, the diagonal is not included. To include the diagonal, add the option `-1`.

[ ]:
```
img = np.ones((10, 10))
img = np.tril(img,)

print(img)
print(img.shape)
print()

plt.imshow(img, cmap='gray') ;
```

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

```
 [1. 1. 1. 1. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
(10, 10)
```



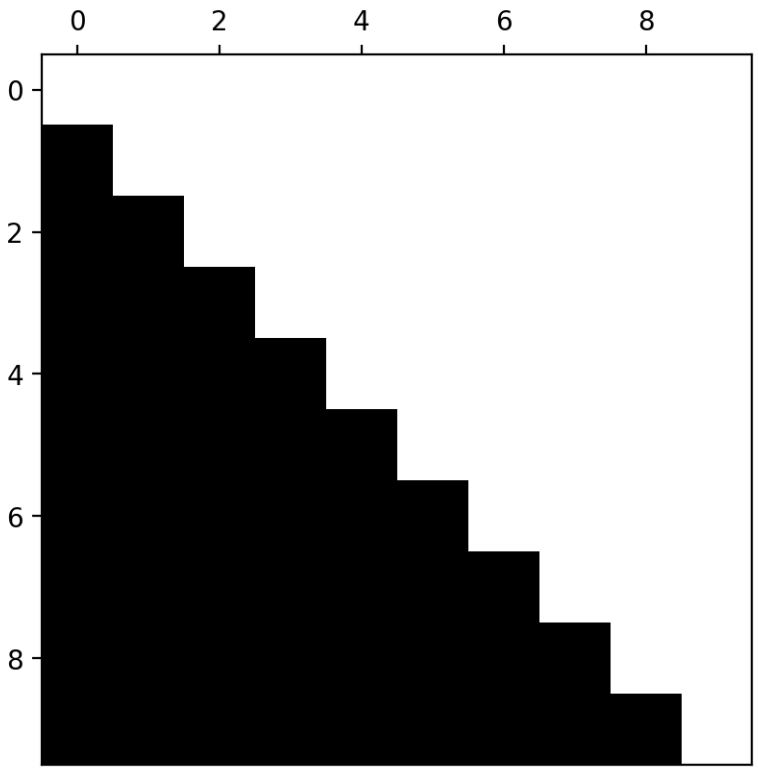
1.9 Keep upper triangle

`triu()` keeps the upper triangle and fills the lower triangle with zeros. By default, the diagonal is not included. To include the diagonal, add the option `-1`.

[ ]:
```
img = np.ones((10, 10))
img = np.triu(img)

print(img)
print(img.shape)
print()
```

```
plt.imshow(img, cmap='gray') ;
```

```
[[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [0. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [0. 0. 1. 1. 1. 1. 1. 1. 1. 1.]
 [0. 0. 0. 1. 1. 1. 1. 1. 1. 1.]
 [0. 0. 0. 0. 1. 1. 1. 1. 1. 1.]
 [0. 0. 0. 0. 0. 1. 1. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
(10, 10)
```



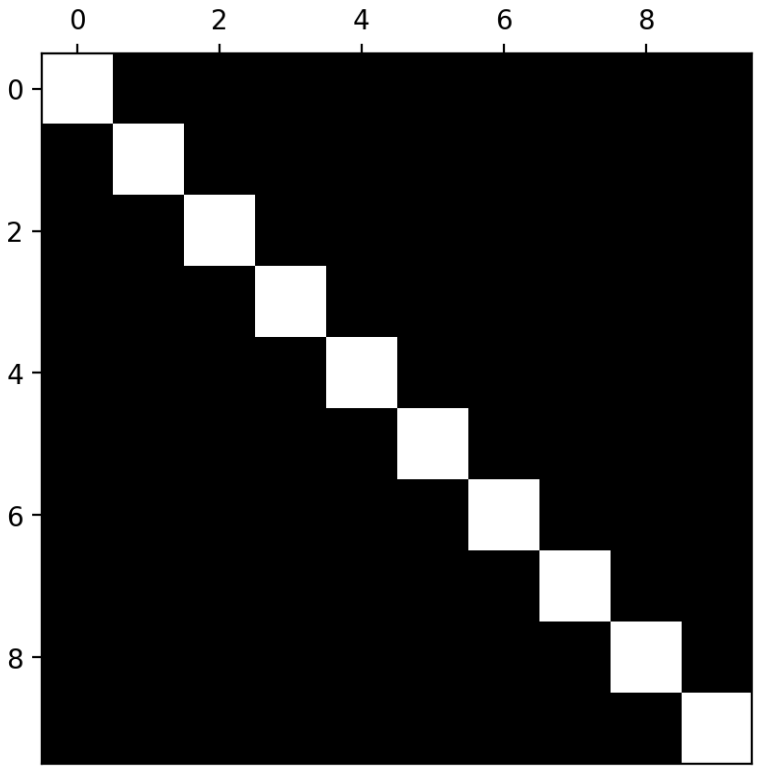
1.10 Keep the diagonal: `triu()` with `tril()`

The result is equivalent to using `fill_diagonal()`

[ ]:
```
img = np.ones((10, 10))
img = np.triu(img)
img = np.tril(img)
```

```
print(img)
print(img.shape)
print()

plt.imshow(img, cmap='gray') ;
```

```
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
(10, 10)
```

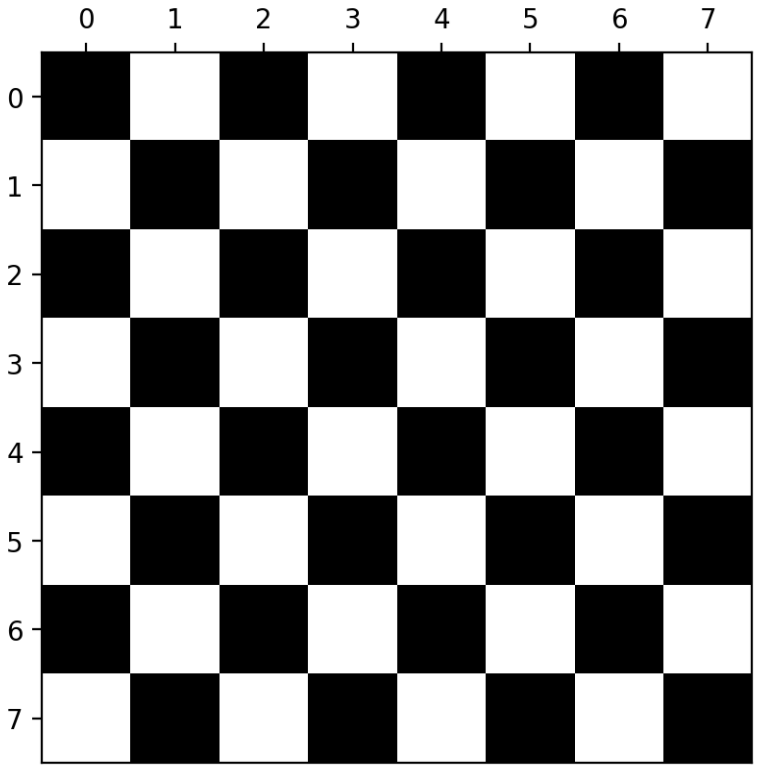


1.11 Checkerboard

[ ]:
```
shape = (8, 8)
indices =  np.indices(shape)
print("== Indices: \n", indices)
print(indices.shape)
print()

sum_of_i = indices.sum( axis = 0 )
print("== Sum of the indices: \n", sum_of_i)
print(sum_of_i.shape)
print()

checkerboard = sum_of_i % 2
img = checkerboard
print("== Checkerboard: \n", img)
print(img.shape)
print()

plt.imshow(img, cmap='gray') ;
```

```
== Indices:
 [[[0 0 0 0 0 0 0 0]
  [1 1 1 1 1 1 1 1]
  [2 2 2 2 2 2 2 2]
  [3 3 3 3 3 3 3 3]
  [4 4 4 4 4 4 4 4]
  [5 5 5 5 5 5 5 5]
  [6 6 6 6 6 6 6 6]
  [7 7 7 7 7 7 7 7]]

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

== Sum of the indices:
 [[ 0  1  2  3  4  5  6  7]
 [ 1  2  3  4  5  6  7  8]
 [ 2  3  4  5  6  7  8  9]
 [ 3  4  5  6  7  8  9 10]
 [ 4  5  6  7  8  9 10 11]
 [ 5  6  7  8  9 10 11 12]
```

```
 [ 6  7  8  9 10 11 12 13]
 [ 7  8  9 10 11 12 13 14]]
(8, 8)

== Checkerboard:
 [[0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]]
(8, 8)
```



[ ]:
```
# The frequencies of sums of rolling two dies.
print(np.asarray(np.unique(sum_of_i[1:7,1:7], return_counts=True)).transpose())
```

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

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

1.12 Black and White numbers

Black is always the minimum in the set of numbers.

White can be any positive number that is the maximum in the set of numbers.

Notice how we use numpy's universal function feature to muliply all values in an array by a single value.

[ ]:
```
shades = np.array( [ 0, 1, 0, 1, 0, 1, 0, 1, 0, ] )

print(list(shades))
print(shades.shape)
print()
```

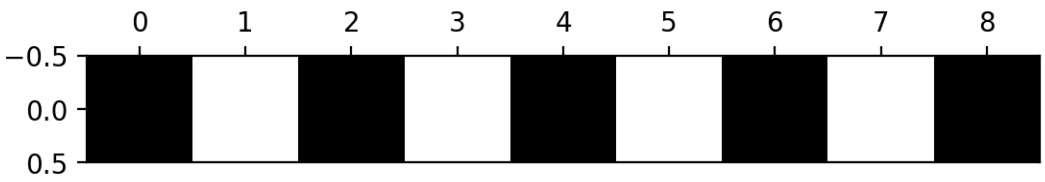
```
[0, 1, 0, 1, 0, 1, 0, 1, 0]
(9,)
```

[ ]:
```
# multiply each value in matrix by a single value ( i.e. a scalar )
shades = (shades * 3.14) + 4

print(list(shades))
print(shades.shape)
print()
```

```
[4.0, 13.8596, 4.0, 13.8596, 4.0, 13.8596, 4.0, 13.8596, 4.0]
(9,)
```

[ ]:
```
# convert from 1-D to 2-D
img = shades.reshape(1, -1)

print(img)
print(img.shape)
print()

plt.imshow(img, cmap='gray') ;
```

```
[[ 4.     13.8596  4.     13.8596  4.     13.8596  4.     13.8596  4.    ]]
(1, 9)
```

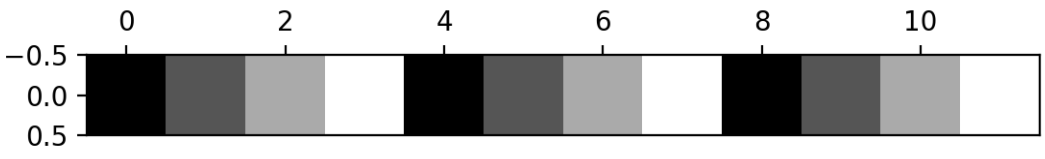


1.13 Greyscale

Not everything has to be black and white.

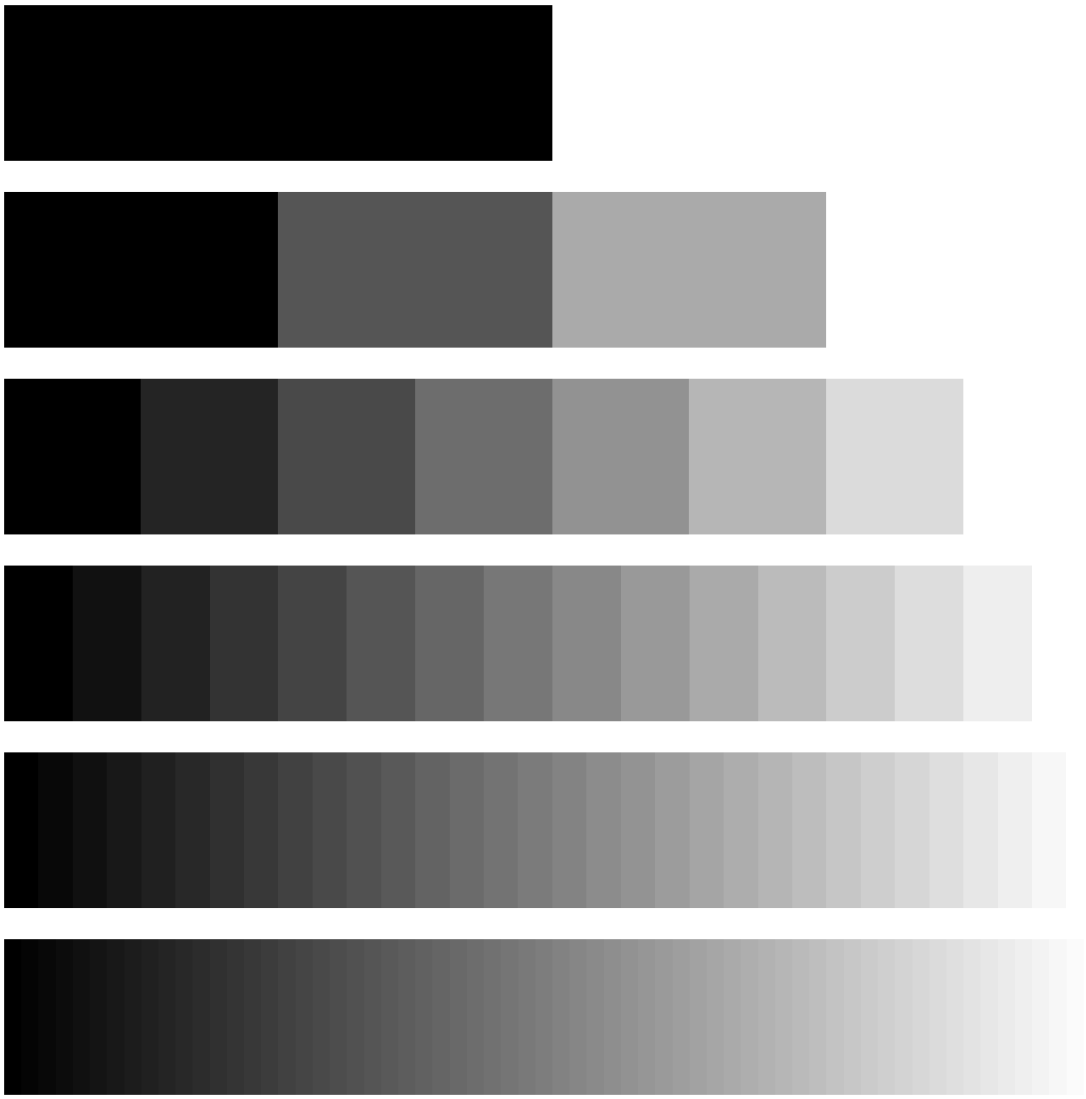
[ ]:
```
# Using modulo to generate numbers in addition to 0, 1.
n = 12
m = 4
shades = np.arange( n ) % m
shades = ( shades / (m-1) ) * 255

print(list(shades))
print(shades.shape)
print()

# reshape into rows, columns ( -1 == whatever it takes to work )
img = shades.reshape(1,-1)
print(list(img))
print(img.shape)
print()

plt.imshow(img, cmap='gray') ;
```

```
[0.0, 85.0, 170.0, 255.0, 0.0, 85.0, 170.0, 255.0, 0.0, 85.0, 170.0, 255.0]
(12,)

[array([  0.,  85., 170., 255.,   0.,  85., 170., 255.,   0.,  85., 170.,
       255.])]
(1, 12)
```



Very fine grey scales.

[ ]:
```
fig, sps = plt.subplots( 6, 1, figsize=(10,10))

for i, sp in enumerate(sps.flatten()):
  img = np.linspace( 0, 255, 2**(i+1) ).reshape(1,-1)
  sp.axis(False)
  sp.imshow(img, cmap='gray', aspect='auto')
```



[ ]:
```

```