lab1-lab-2-Numpy

February 23, 2023

[41]:
```
def fun():
    L = []
    for n in range(10000000):
        L.append(n**2)
    return L
```

[42]:
```
%timeit fun()
```

```
2.41 s ± 12.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[39]:
```
# can we optimise the above code?
def fun2():
    return [n**2 for n in range(10000000)]
```

[40]:
```
%timeit fun2()
```

```
2.18 s ± 16.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[14]:
```
# how to find out documentation of any inbuilt function
# np.append?
```

[43]:
```
# find the error in the below code.
def func1(a, b):
    return a / b

def func2(x):
    a = x
    b = x - 1
    return func1(a, b)
```

[45]:
```
# func2(1)
```

0.1 Numpy

[46]:
```
import numpy as np
```

[47]:
```
# import matplotlib.pyplot as plt
```

[48]:
```
np.__version__
```

[48]:
```
'1.21.6'
```

[51]:
```
x = [1,2,4]
print(type(x))
x = np.array(x)
print(type(x))
```

```
<class 'list'>
<class 'numpy.ndarray'>
```

[52]:
```
# integer array:
np.array([1, 4, 2, 5, 3])
```

[52]:
```
array([1, 4, 2, 5, 3])
```

[61]:
```
x = np.random.rand(3,4)
print(type(x))
x = np.array(x)
print("\n\n")
print(type(x))
```

```
<class 'numpy.ndarray'>



<class 'numpy.ndarray'>
```

[68]:
```
# nested lists result in multi-dimensional arrays
np.array([range(i, i + 3) for i in [2, 4, 6]])
```

[68]:
```
array([[2, 3, 4],
       [4, 5, 6],
       [6, 7, 8]])
```

[77]:
```
# Create a length-10 integer array filled with zeros
np.zeros((3,5), dtype=int)
```

[77]:
```
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
```

[76]:
```
# Create a 3x5 floating-point array filled with ones
np.ones((3, 5), dtype=float)
```

[76]:
```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

[80]:
```python
# Create a 3x5 array filled with 3.14
np.full((3, 5), 3.14)
```

[80]:
```
array([[3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14]])
```

[88]:
```python
# Create an array filled with a linear sequence
# Starting at 0, ending at 20, stepping by 2
# (this is similar to the built-in range() function)
np.arange(0, 20, 2)
```

[88]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

[92]:
```python
# Create an array of five values evenly spaced between 0 and 1
np.linspace(0, 1, 5)
```

[92]:
```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

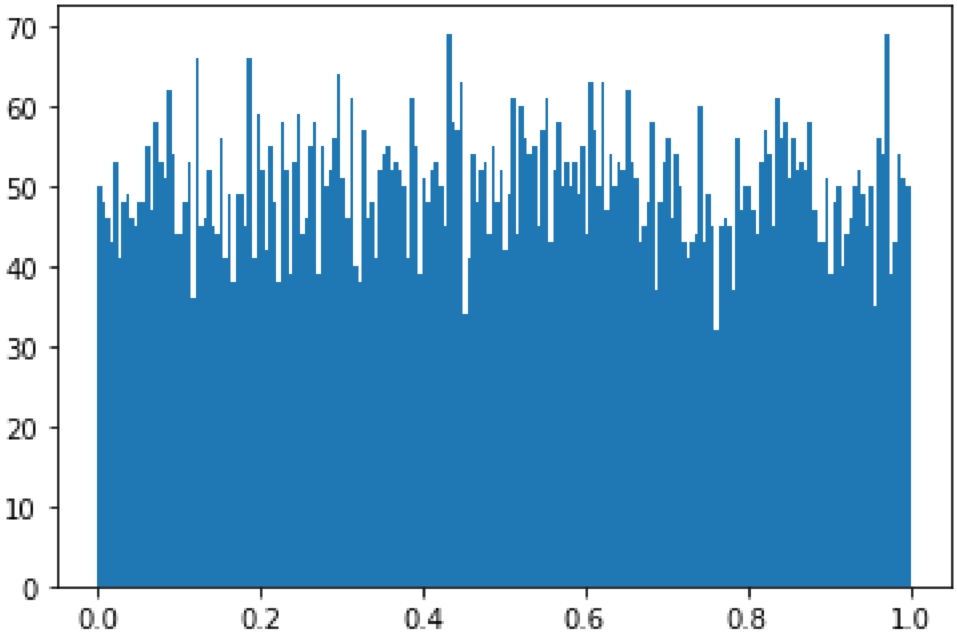
[31]:
```python
# Create a 3x3 array of uniformly distributed
# random values between 0 and 1
x = np.random.random((10000))
```

[99]:
```python
import matplotlib.pyplot as plt
```

[100]:
```python
x
```

[100]:
```
array([4.87137687e-01, 4.57356103e-01, 3.83035179e-01, 8.13774995e-01,
       1.79689897e-01, 4.45249867e-01, 4.28580573e-01, 3.66515480e-01,
       6.94895119e-01, 3.64226033e-04])
```

[121]:
```python
plt.hist(x, bins=200);
```

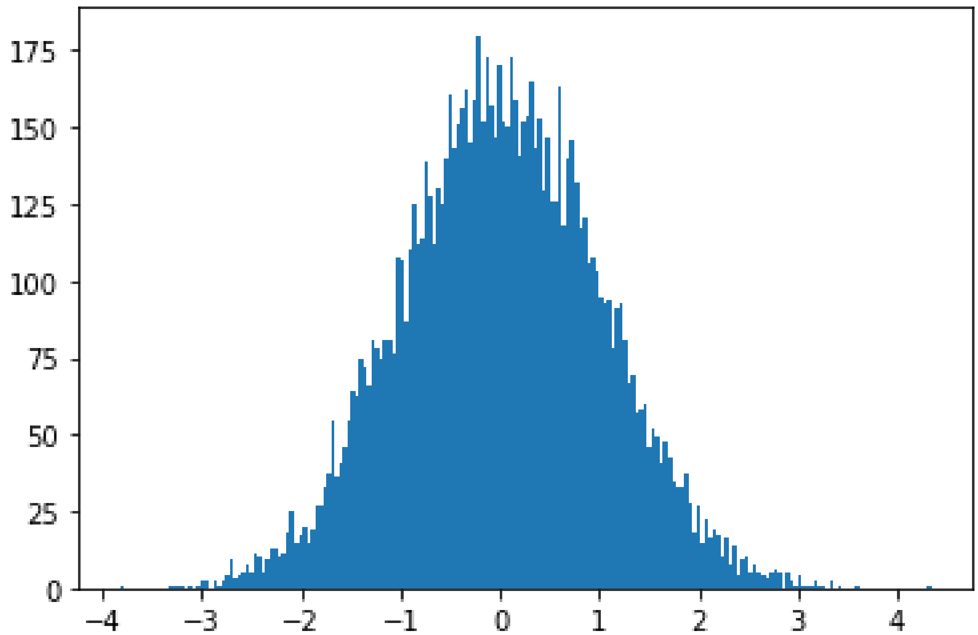


[30]:
```
# Create a 3x3 array of normally distributed random values
# with mean 0 and standard deviation 1
np.random.normal(0, 1, (3, 3))
```

[30]:
```
array([[ 0.26010113,  0.40018921,  0.35291973],
       [-0.39158533, -0.34958723,  0.06219367],
       [-1.56564613,  1.23545203,  0.50353038]])
```

[123]:
```
x = np.random.normal(0, 1, (10000))
```

[124]:
```
plt.hist(x, bins=200);
```



[132]:
```
# Create a 3x3 array of random integers in the interval [0, 10)
np.random.randint(0, 100, 10)
```

[132]:
```
array([39, 51, 11, 44, 66, 95, 90, 16, 77, 89])
```

[134]:
```
# Create a 3x3 identity matrix
np.eye(3)
```

[134]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

[133]:
```
# Data Types
np.zeros(10, dtype='int16')
```

[133]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int16)
```

[ ]:
```
# Data Types
np.zeros(10, dtype=np.int16)
```

0.1.1 Numpy Arrays

[44]:
```
np.random.seed(0)  # seed for reproducibility
x1 = np.random.randint(10, size=6)  # One-dimensional array
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
```

```
x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
```

[47]:
```
x2
```

[47]:
```
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

[48]:
```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

[139]:
```
# convert list to numpy array
x = [[1,2,3], [4,5,6], [7,8,9], [1,1,1]] # 4 x 3
y = [[1,2,3,1], [4,5,6,1], [7,8,9,4]]  # 3 x 4
type(x)
# convert to numpy array
```

[139]:
```
list
```

0.1.2 Numpy Array Indexing and Slicing

[49]:
```
x2
```

[49]:
```
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

[153]:
```
# with 2D array
x2[0:2,1]
```

[153]:
```
array([5, 6])
```

[ ]:
```
# with 2D array
```

[100]:
```
x2
```

[100]:
```
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

[101]:
```
x2[:2, :3]
```

[101]: array([[3, 5, 2],
        [7, 6, 8]])

[50]:
```
x2[::-1, ::-1]
```

[50]: array([[7, 7, 6, 1],
        [8, 8, 6, 7],
        [4, 2, 5, 3]])

0.1.3 Be careful while creating copies

[58]:
```
x2_sub_copy = x2[:2, :2]
print(x2_sub_copy)
```

```
[[3 5]
 [7 6]]
```

[59]:
```
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
```

```
[[42  5]
 [ 7  6]]
```

[60]:
```
print(x2)
```

```
[[42  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

0.1.4 Reshaping

[72]:
```
grid = np.arange(1, 10).reshape((3,3))
print(grid)
```

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

[70]:
```
x = np.array([1, 2, 3])
print(x.shape)
# row vector via reshape
x = x.reshape((1, 3))
print(x.shape)
```

```
(3,)
(1, 3)
```

0.1.5 Concatenation

[71]:
```python
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

[71]:
```
array([1, 2, 3, 3, 2, 1])
```

[74]:
```python
grid, grid
```

[74]:
```
(array([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]]),
 array([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]]))
```

[80]:
```python
# concatenate along the first axis
np.concatenate([grid, grid], axis=0) # rows will change,columns are expanded
```

[80]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9],
       [1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

[83]:
```python
# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
```

[83]:
```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6],
       [7, 8, 9, 7, 8, 9]])
```

[85]:
```python
x = np.array([1, 2, 3]) # 1,3



grid = np.array([[9, 8, 7],
                 [6, 5, 4]]) # (2,3)

# vertically stack the arrays
np.vstack([grid, x])
```

[85]:
```
array([[9, 8, 7],
       [6, 5, 4],
       [1, 2, 3]])
```

[87]:
```python
grid.shape
```

[87]: (2, 3)

[88]:
```
y = np.array([[99],
              [99]])
```

[89]:
```
y.shape
```

[89]: (2, 1)

[86]:
```
# horizontally stack the arrays
#y = np.array([[99],
#               [99]])


np.hstack([grid, y])
```

[86]:
```
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
```

0.1.6 Example of matrix multiplication

[91]:
```
# vanilla python code to multiply two matrix

def matmul_python(x, y):
    result = [[0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0]] # 4 x 4
    colY = len(y[0])
    rowX = len(x)
    for row in range(rowX):
        for col in range(colY):
            for i,itemX in enumerate(x[row]):
                result[row][col]+=itemX*y[i][col]
    return result
```

[94]:
```
x = [[1,2,3], [4,5,6], [7,8,9], [1,1,1]] # 4 x 3
y = [[1,2,3,1], [4,5,6,1], [7,8,9,4]]  # 3 x 4
%timeit matmul_python(x,y)
```

```
9.8 µs ± 29.1 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

[95]:
```
x = np.array(x)
y = np.array(y)
%timeit np.matmul(x,y)
```

```
941 ns ± 1.8 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

[ ]:
```
# Hint: 1 us = 1000ns
```

0.1.7 Summary Stats of Data

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

[91]:
```
# np.random.seed(0) # <- what happens with this line?
L = np.random.random(100)
sum(L)
```

[91]: 52.13902574466732

[92]:
```
np.sum(L)
```

[92]: 52.139025744667336

[93]:
```
big_array = np.random.rand(1000000)
%timeit sum(big_array)
```

```
71.1 ms ± 187 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[94]:
```
%timeit np.sum(big_array)
```

```
201 µs ± 2.36 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[95]:
```
# maximum and minimum
min(big_array), max(big_array)
```

[95]: (1.4057692298008462e-06, 0.9999994392723005)

[96]:
```
np.min(big_array), np.max(big_array)
```

[96]: (1.4057692298008462e-06, 0.9999994392723005)

[97]:
```
# sum of multidimensional data
M = np.random.random((3, 4))
print(M)
```

```
[[0.35747479 0.16525562 0.22801649 0.01116173]
 [0.96484772 0.93042983 0.28571123 0.18633432]
 [0.84657553 0.11751108 0.43646768 0.48335078]]
```

[99]:
```
M.shape, #(3,1)
```

[99]: (3, 4)

[98]:
```
M.sum()
```

[98]: 5.013136800754203

[109]:
```
M.sum(axis=1)
```

[109]:
```
array([0.76190863, 2.3673231 , 1.88390506])
```

[106]:
```
M.min(axis=0) # find sum of each column
```

[106]:
```
array([0.35747479, 0.11751108, 0.22801649, 0.01116173])
```

[107]:
```
M.min(axis=1)
```

[107]:
```
array([0.01116173, 0.18633432, 0.11751108])
```

- `axis=0` is column
- `axis=1` is row
- The axis keyword specifies the dimension of the array that will be collapsed, rather than the dimension that will be returned

0.1.8 Broadcasting

[110]:
```
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

[110]:
```
array([5, 6, 7])
```

[113]:
```
a + 5
#[0,1,2] + [5,5,5]
```

[113]:
```
array([5, 6, 7])
```

[115]:
```
M, a
```

[115]:
```
(array([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]]),
 array([0, 1, 2]))
```

[116]:
```
M = np.ones((3, 3))
M+a # (3,3) (1,3)
```

[116]:
```
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
```

Here the one-dimensional array a is stretched, or broadcast across the second dimension in order to match the shape of M `numpy.newaxis` is used to increase the dimension of the existing array by one more dimension, when used once.

[117]:
```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis] # same as saying, "I want every element spread␣
↪across row but make it a 2d array"
```

[124]:
```
a.reshape(3,1)
```

[124]:
```
array([[0],
       [1],
       [2]])
```

[123]:
```
b
```

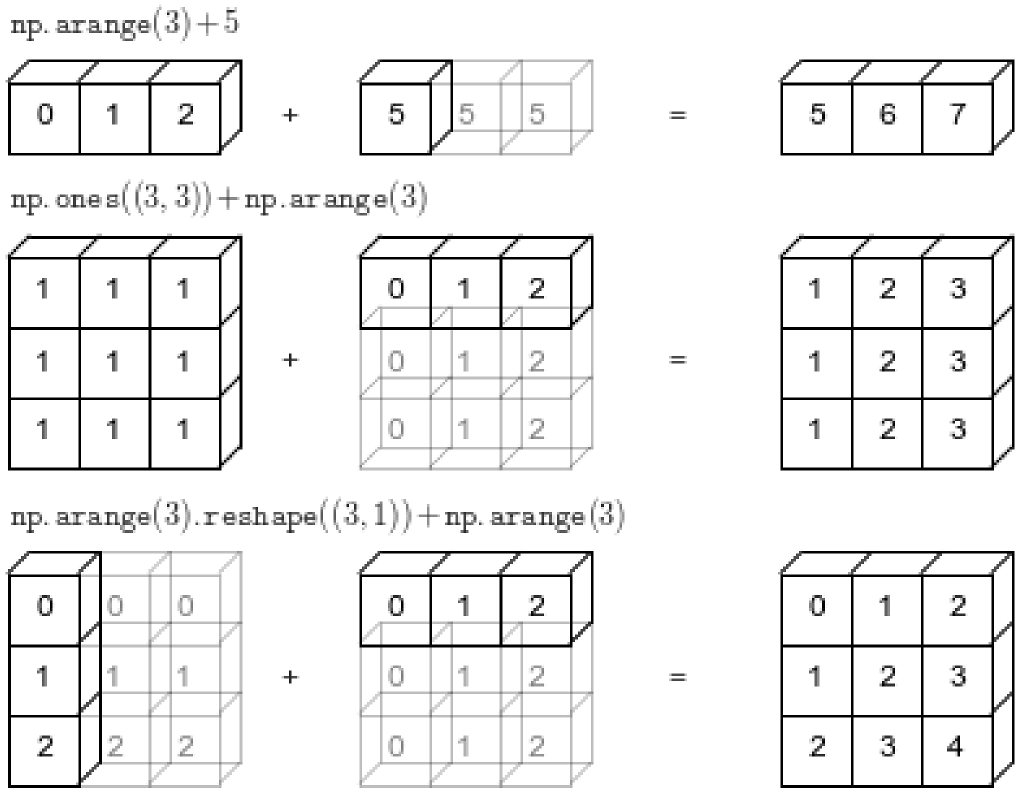
[123]:
```
array([[0],
       [1],
       [2]])
```

[120]:
```
print(a.shape)
print(b.shape)
```

```
(3,)
(3, 1)
```

[121]:
```
a + b
```

[121]:
```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```



0.1.9 Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

- Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.
- Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
- Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

[135]:
```
# example of Rule 1
M = np.ones((2, 3)) # (2,3)
a = np.arange(3) # (1,3) -> (2,3)
```

[129]:
```
M, a
```

[129]:
```
(array([[1., 1., 1.],
        [1., 1., 1.]]),
 array([0, 1, 2]))
```

[136]:
```
M+a
```

[136]:
```
array([[1., 2., 3.],
       [1., 2., 3.]])
```

[140]:
```
# example of Rule 2
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

[141]:
```
a+b
```

[141]:
```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

[ ]:
```
# example of Rule 3
```

[142]:
```
M = np.ones((3, 2)) # (3,2)
a = np.arange(3)    #(3,3)
```

[143]:
```
M+a
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [143], in <cell line: 1>()
----> 1 M+a

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

0.1.10 Usefulness of Broadcasting

[144]:
```
# mean centering
X = np.random.random((10, 3))
```

[147]:
```
X
```

[147]:
```
array([[0.34284584, 0.00846426, 0.85343236],
       [0.38841378, 0.3388289 , 0.93265682],
       [0.46910315, 0.74570254, 0.9263706 ],
       [0.17372444, 0.02774391, 0.05559845],
       [0.67021314, 0.43505254, 0.90258482],
       [0.16114474, 0.56536336, 0.56557665],
       [0.28697744, 0.68061159, 0.54282346],
       [0.87071569, 0.97289904, 0.49027435],
       [0.80780529, 0.20026995, 0.33832142],
       [0.61004664, 0.87544312, 0.45195328]])
```

[146]:
```
Xmean = X.mean(0)
Xmean
```

[146]:
```
array([0.47809902, 0.48503792, 0.60595922])
```

[148]:
```
X_centered = X - Xmean # (10,3) (10,3) ->
```

[149]:
```
X_centered
```

[149]:
```
array([[-0.13525318, -0.47657366,  0.24747314],
       [-0.08968523, -0.14620902,  0.32669759],
       [-0.00899587,  0.26066462,  0.32041138],
       [-0.30437457, -0.45729401, -0.55036077],
       [ 0.19211412, -0.04998538,  0.2966256 ],
       [-0.31695427,  0.08032544, -0.04038257],
       [-0.19112157,  0.19557367, -0.06313576],
       [ 0.39261667,  0.48786112, -0.11568487],
       [ 0.32970628, -0.28476797, -0.2676378 ],
       [ 0.13194762,  0.3904052 , -0.15400594]])
```

0.1.11 Easy Sorting

[150]:
```
x = np.array([2, 1, 4, 3, 5])
np.sort(x)
```

[150]:
```
array([1, 2, 3, 4, 5])
```

[151]:
```
x = np.array([2, 1, 4, 3, 5])
i = np.argsort(x)
print(i)
```

```
[1 0 3 2 4]
```

[155]:
```
np.argmin(x)
```

[155]:
```
1
```

[156]:
```
# with 2d array
rand = np.random.RandomState(42)
X = rand.randint(0, 10, (4, 6))
print(X)
```

```
[[6 3 7 4 6 9]
 [2 6 7 4 3 7]
 [7 2 5 4 1 7]
 [5 1 4 0 9 5]]
```

[ ]:
```python
# sort each column of X
np.sort(X, axis=0)
```

[ ]:
```python
# sort each row of X
np.sort(X, axis=1)
```

0.1.12 Partial Sorts: Partitioning

[158]:
```python
x = np.array([7, 2, 3, 1, 6, 5, 4])
np.partition(x, 2)
```

[158]:
```
array([1, 2, 3, 7, 6, 5, 4])
```

0.1.13 2D plots

[165]:
```python
%matplotlib inline
import matplotlib.pyplot as plt
```

[159]:
```python
# x and y have 50 steps from 0 to 5
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 50)[:, np.newaxis]
```

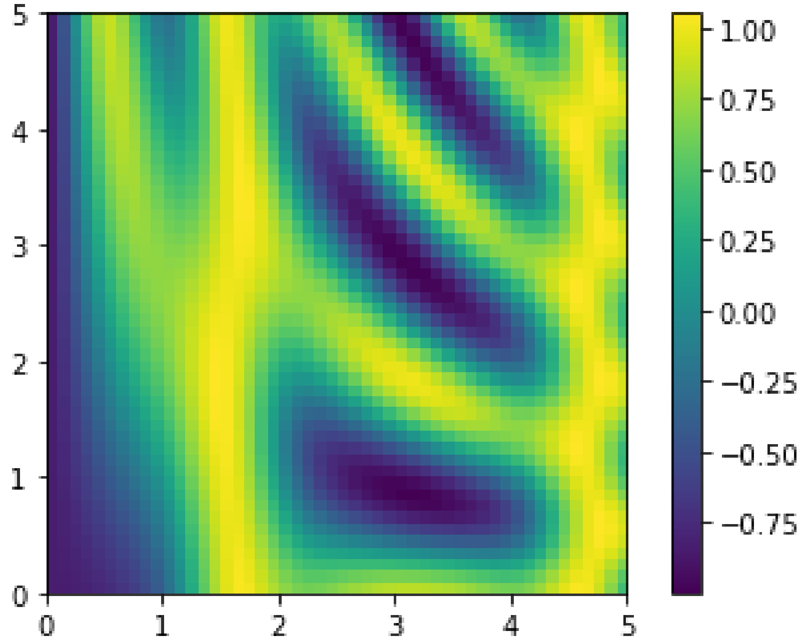
[168]:
```python
# can you explain weather the input to trignometric functions are in degrees or␣
↪radians?

# z = f(x,y)
z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

[169]:
```python
z.shape
```

[169]:
```
(50, 50)
```

[170]:
```python
plt.imshow(z, origin='lower', extent=[0, 5, 0, 5],
           cmap='viridis')
plt.colorbar();
```



0
1
2
3
4
5
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75


```
[ ]:
```