Tutorial 10: Python - NumPy & Matplotlib

CS 104/108

Spring, 2024-25

TA: Mohana Evuri
Credits: Guramrit Singh

Topics

- NumPy
- Arithmetic in NumPy
- Broadcasting in NumPy
- Matplotlib

NumPy

NumPy

- NumPy is a Python library used for working with arrays/matrices.
- Used in variety of numerical computations involving matrix multiplications, fourier transforms etc.
- NumPy vectorization involves performing mathematical operations on entire arrays, eliminating the need to loop through individual elements.
- NumPy arrays are called ndarrays (short for n-dimensional arrays), these arrays come with a lot of functions support.
- NumPy is much faster than ordinary python lists because they are stored contiguously in memory (locality).
- NumPy documentation can be found at https://numpy.org/doc/stable/
- Reference for this tutorial: https://www.w3schools.com/python/numpy/

Creating a numpy array

basic.py > ...

```python
import numpy as np

a = np.array([1, 2, 3, 4, 5])
b = np.array((6, 9, 10))

print("a's type:", type(a))

print("array a:", a)
print("array b:", b)

print("a's shape:", a.shape)
print("a's number of dimensions:", a.ndim)
```

```
~/De/c/t/tutorial_10 ····· 07:33:49 PM
❯ python3 basic.py
a's type: <class 'numpy.ndarray'>
array a: [1 2 3 4 5]
array b: [ 6  9 10]
a's shape: (5,)
a's number of dimensions: 1
~/De/c/t/tutorial_10 ····· 07:33:51 PM
❯
```

- To create an ndarray, we can pass a list, tuple or any array-like object into the array method, and it will be converted into an ndarray.
- The demo shows creation of ndarray using list and tuple.
- We use to shape attribute to get the dimensions of the ndarray.
- We use to ndim attribute to get the number of dimensions of the ndarray.

Data types

dt.py > ...

```python
import numpy as np

a = np.array([1, 2, 3, 4])
b = np.array(["apple", "banana", "cherry"])

print("array a:", a)
print("a dtype:", a.dtype)
print("array b:", b)
print("b dtype:", b.dtype)

c = np.array([12, 21, 32, 40], dtype='S')
print("array c:", c)
print("c dtype:", c.dtype)

d = np.array([1, 2, 3, 4], dtype='f')
print("array d:", d)
print("d dtype:", d.dtype)

e = np.array([1.2, 2.3, 3.4], dtype='float32')
print("e dtype:", e.dtype)
f = e.astype('i')
print("array f:", f)
print("f dtype:", f.dtype)

g = np.array([1, 0, 3, 0, 5])
print("g dtype:", g.dtype)
h = g.astype(bool)
print("array h:", h)
print("h dtype:", h.dtype)
```

```
~/De/c/t/tutorial_10 ···· 12:39:18 AM
❯ python3 dt.py
array a: [1 2 3 4]
a dtype: int64
array b: ['apple' 'banana' 'cherry']
b dtype: <U6
array c: [b'12' b'21' b'32' b'40']
c dtype: |S2
array d: [1. 2. 3. 4.]
d dtype: float32
e dtype: float32
array f: [1 2 3]
f dtype: int32
g dtype: int64
array h: [ True False  True False  True]
h dtype: bool
~/De/c/t/tutorial_10 ···· 12:39:25 AM
❯
```

- We use the dtype attribute to get the datatype of the ndarray.
- We use the dtype parameter of array function to create an array of expected data type.
- The astype function creates a copy of the array, and allows you to specify the data type as a parameter.

Different dimensional arrays

dim.py > ...

```python
import numpy as np

a = np.array(1)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

print("Dim of a:", a.ndim)
print("Dim of b:", b.ndim)
print("Dim of c:", c.ndim)
print("Dim of d:", d.ndim)

e = np.array([[1, 2], [3, 4]], ndmin=5)
f = np.array([[[1], [2]], [[3], [4]], [[5], [6]]], ndmin=2)

print("Array e:", e)
print("Dim of e:", e.ndim)

print("Array f:", f)
print("Dim of f:", f.ndim)
```

```
~/De/c/tutorials/tutorial_10 · 08:24:53 PM
❯ python3 dim.py
Dim of a: 0
Dim of b: 1
Dim of c: 2
Dim of d: 3
Array e: [[[[[1 2]
    [3 4]]]]]
Dim of e: 5
Array f: [[[1]
  [2]]

 [[3]
  [4]]

 [[5]
  [6]]]
Dim of f: 3
~/De/c/tutorials/tutorial_10 · 08:24:54 PM
❯
```

- You can create any dimensional arrays using array function of np.
- You can also specify the minimum dimension of ndarray required by ndmin parameter of the array function.
- See the above example, in case of array ‘e’, the dimension became 5, whereas for ‘f’ it is 3.

Array indexing and slicing

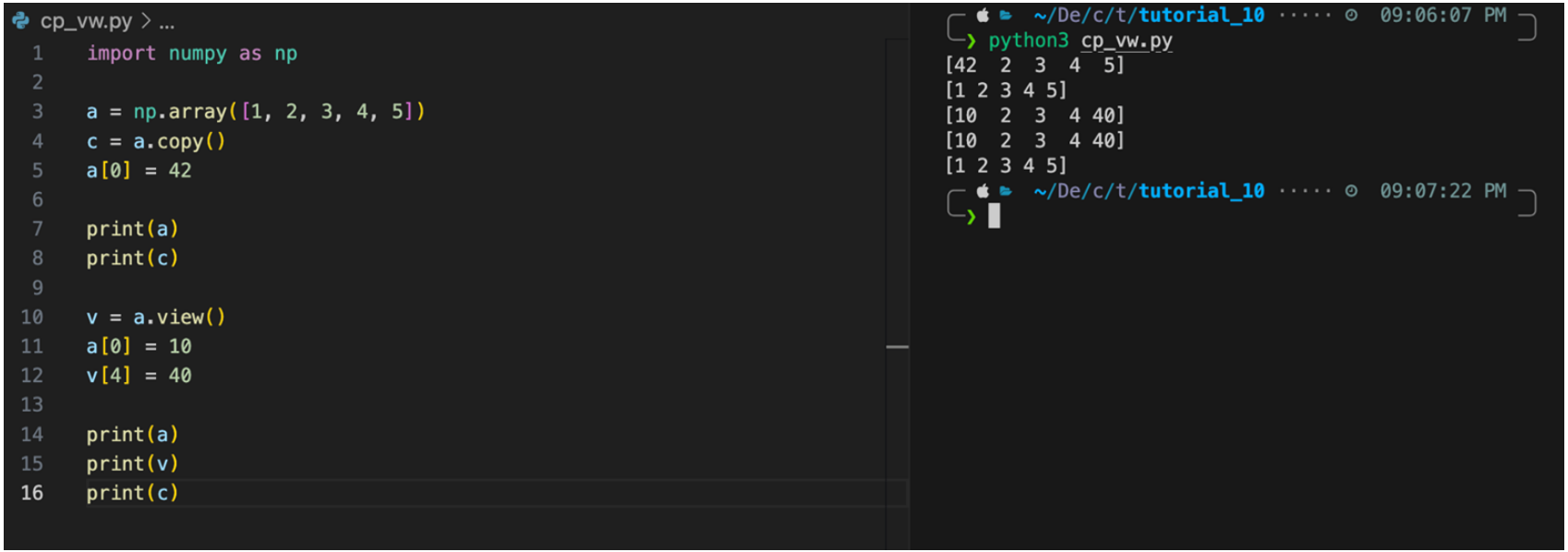
ind_sli.py > ...

```python
import numpy as np

a = np.array(1)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

print("Only element of a:", a)
print("2nd element of b:", b[1])
print("3rd element of 2nd row of c:", c[1, 2])
print("2nd element of 1st row of 2nd array of d:", d[1, 0, 1])

print("First 4 elements of b:", b[:4])
print("Last 2 elements of b:", b[-2:])
print("Middle 3 elements of b:", b[1:4])
print("Every even index element of b:", b[::2])
print("Every element of odd row and even column of c:", c[1::2, ::2])
```

```
~/De/c/t/tutorial_10 ····· 08:46:43 PM
❯ python3 ind_sli.py
Only element of a: 1
2nd element of b: 2
3rd element of 2nd row of c: 6
2nd element of 1st row of 2nd array of d: 8
First 4 elements of b: [1 2 3 4]
Last 2 elements of b: [4 5]
Middle 3 elements of b: [2 3 4]
Every even index element of b: [1 3 5]
Every element of odd row and even column of c: [[ 4
6]
 [10 12]]
~/De/c/t/tutorial_10 ····· 08:46:46 PM
❯
```

- You can access any element in an array by referring to it's index number. (0 - indexing)
- For slicing, we pass slice instead of index like this- [start:end].
- You can also define the step, like this- [start:end:step].

Copy vs view



- The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.
- The copy *owns* the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.
- The view *does not own* the data and any changes made to the view will affect the original array, and any

Reshaping

reshape.py > ...

```python
import numpy as np

a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
print("array a:", a)

b = a.reshape(4, 3)
print("array b:", b)

c = a.reshape(2, 3, 2)
print("array c:", c)

d = a.reshape(2, -1, 3)
print("array d:", d)
```

```
~/De/c/t/tutorial_10 ····· 09:46:34 AM
❯ python3 reshape.py
array a: [ 1  2  3  4  5  6  7  8  9 10 11 12]
array b: [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
array c: [[[ 1  2]
  [ 3  4]
  [ 5  6]]

 [[ 7  8]
  [ 9 10]
  [11 12]]]
array d: [[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]
~/De/c/t/tutorial_10 ····· 09:46:35 AM
❯
```

- The shape of an array is the number of elements in each dimension. We used the shape attribute to get the shape of an ndarray. We can also change the shape of an ndarray by using the reshape function.
- You can reshape into any shape as long as the number of elements are equal in both the shapes.
- All dimensions need to be non-negative integers. You are allowed to have one “unknown” dimension while reshaping, it is represented by -1, and numpy will compute it.

Joining arrays

join.py > ...

```python
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print("array a:", a)
print("array b:", b)

c = np.concatenate((a, b))
print("array c:", c)

d = np.array([[1, 2], [3, 4]])
e = np.array([[5, 6], [7, 8]])
print("array d:", d)
print("array e:", e)

f = np.concatenate((d, e), axis=0)
print("array f:", f)
g = np.concatenate((d, e), axis=1)
print("array g:", g)

h = np.vstack((d, e))
print("array h:", h)

i = np.hstack((d, e))
print("array i:", i)
```

```
~/De/c/t/tutorial_10 ····· 10:15:15 AM
❯ python3 join.py
array a: [1 2 3]
array b: [4 5 6]
array c: [1 2 3 4 5 6]
array d: [[1 2]
 [3 4]]
array e: [[5 6]
 [7 8]]
array f: [[1 2]
 [3 4]
 [5 6]
 [7 8]]
array g: [[1 2 5 6]
 [3 4 7 8]]
array h: [[1 2]
 [3 4]
 [5 6]
 [7 8]]
array i: [[1 2 5 6]
 [3 4 7 8]]
~/De/c/t/tutorial_10 ····· 10:15:16 AM
❯
```

- Axis in an ndarrays are numbered starting with 0. 0 - row, 1 - column and so on …
- We pass a sequence of arrays that we want to join to the concatenate function, along with the axis. By default the axis is taken to be 0.
- hstack function is used to stack along the rows whereas vstack function is used to stack along the columns.

Splitting array

split.py > ...

```python
import numpy as np

a = np.array([1, 2, 3, 4, 5, 6])
b = np.array_split(a, 3)
print("array b:", b)
c  = np.array_split(a, 4)
print("array c:", c)

d = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])
e = np.array_split(d, 2)
print("array e:", e)
f = np.array_split(d, 2, axis=1)
print("array f:", f)

g = np.hsplit(d, 2)
print("array g:", g)
h = np.vsplit(d, 3)
print("array h:", h)
```

```
~/De/c/t/tutorial_10 ····· 11:06:46 AM
❯ python3 split.py
array b: [array([1, 2]), array([3, 4]), array([5, 6])]
array c: [array([1, 2]), array([3, 4]), array([5]), arr
ay([6])]
array e: [array([[1, 2],
       [3, 4],
       [5, 6]]), array([[ 7,  8],
       [ 9, 10],
       [11, 12]])]
array f: [array([[ 1],
       [ 3],
       [ 5],
       [ 7],
       [ 9],
       [11]]), array([[ 2],
       [ 4],
       [ 6],
       [ 8],
       [10],
       [12]])]
array g: [array([[ 1],
       [ 3],
       [ 5],
       [ 7],
       [ 9],
       [11]]), array([[ 2],
       [ 4],
       [ 6],
       [ 8],
       [10],
       [12]])]
array h: [array([[1, 2],
       [3, 4]]), array([[5, 6],
       [7, 8]]), array([[ 9, 10],
       [11, 12]])]
```

- Joining merges multiple arrays into one and splitting breaks one array into multiple.
- We use array_split for splitting arrays, we pass it the array we want to split and the number of splits. The function returns a list of ndarrays. You can specify the axis to split along. (Default axis is 0)
- If the array has less elements than required, it will adjust from the end accordingly.

Search

where.py > ...

```
1  import numpy as np
2
3  a = np.array([1, 2, 3, 4, 5, 4, 4])
4  b = np.where(a == 4)
5  print("array b:", b)
6  c = np.where(a % 2 == 0)
7  print("array c:", c)
```

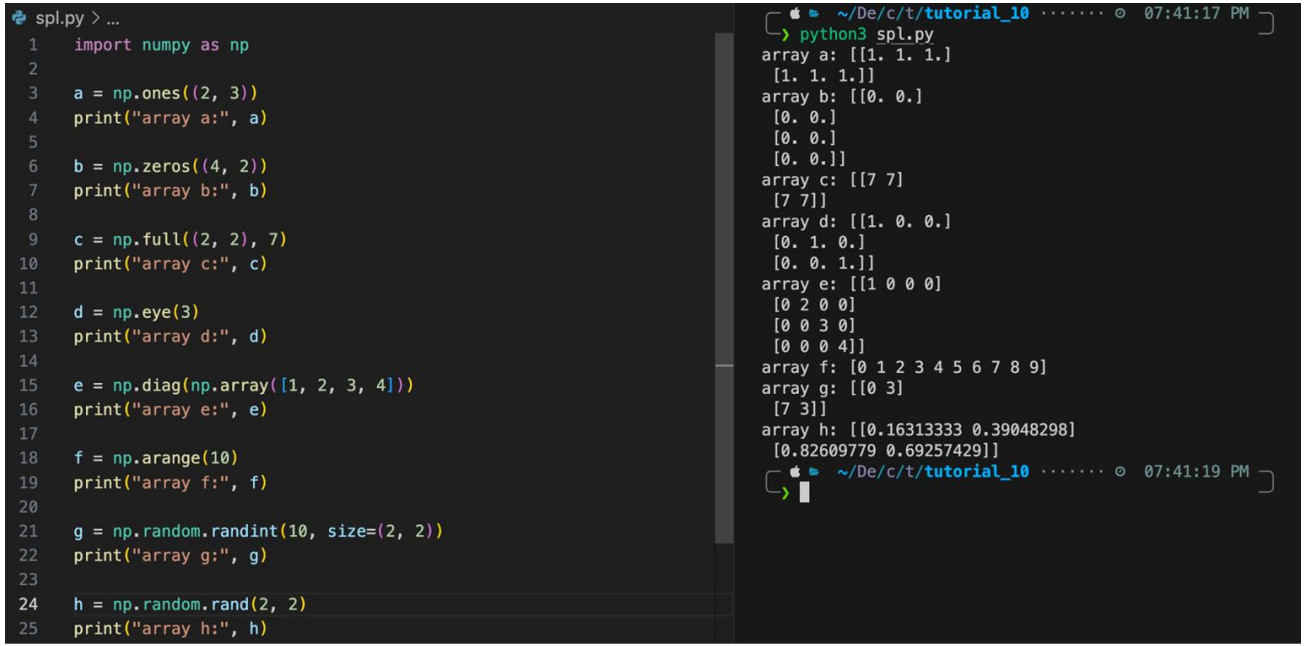
```
~/De/c/t/tutorial_10 ····· 11:20:37 AM
❯ python3 where.py
array b: (array([3, 5, 6]),)
array c: (array([1, 3, 5, 6]),)
~/De/c/t/tutorial_10 ····· 11:20:38 AM
❯
```

- You can search an array for a certain value, and return the indexes that get a match.
- To search an array, use the where method.

Filtering

- Getting some elements out of an existing array and creating a new array out of them is called filtering.
- In NumPy, you filter an array using a boolean index list.
- If the value at an index is True that element is contained in the filtered array, if the value at that index is False that element is excluded from the filtered array.

Special arrays



- There are a lot of special arrays available in NumPy. A few of them are as follows
  - ones : an array of 1s
  - zeros : an array of 0s
  - full : an array having all elements equal to fill_value which is the second parameter of full function
  - eye : an identity matrix of dimension specified
  - diag : constructs a diagonal array out of the array given
  - arange : returns evenly spaced values within a given interval
  - random : returns random numbers of required size

Arithmetic in NumPy

Vectorization

vect.py > ...

```python
import numpy as np

x = [1, 2, 3, 4]
y = [4, 5, 6, 7]
z = []

for i in range(len(x)):
    z.append(x[i] + y[i])

w = np.add(x, y)

print("array x:", x)
print("array y:", y)
print("array z:", z)
print("array w:", w)
print("z type:", type(z))
print("w type:", type(w))

a = np.array([1, 2, 3, 4])
b = np.array([3, 0, 1, 2])

c = a + b
print("array c:", c)
```

```
~/De/c/t/tutorial_10 ····· 08:05:18 PM
❯ python3 vect.py
array x: [1, 2, 3, 4]
array y: [4, 5, 6, 7]
array z: [5, 7, 9, 11]
array w: [ 5  7  9 11]
z type: <class 'list'>
w type: <class 'numpy.ndarray'>
array c: [4 2 4 6]
~/De/c/t/tutorial_10 ····· 08:05:19 PM
❯
```

- Converting iterative statements into a vector based operation is called vectorization.
- It is faster as modern CPUs are optimized for such operations.
- The example shows the iterative method as well as the NumPy's add function.
- The + operator can be used in place of add function for ndarrays.

Simple arithmetic

op.py > ...

```python
import numpy as np

a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [1, 2]])
print("array a:", a)
print("array b:", b)
c = a + b
print("array c:", c)
d = a - b
print("array d:", d)
e = a * b
print("array e:", e)
f = a / b
print("array f:", f)
g = a ** b
print("array g:", g)
h = a % b
print("array h:", h)
i = a // b
print("array i:", i)
```

```
~/De/c/t/tutorial_10 ····· 08:14:25 PM
❯ python3 op.py
array a: [[1 2]
 [3 4]]
array b: [[5 6]
 [1 2]]
array c: [[6 8]
 [4 6]]
array d: [[-4 -4]
 [ 2  2]]
array e: [[ 5 12]
 [ 3  8]]
array f: [[0.2        0.33333333]
 [3.         2.        ]]
array g: [[ 1 64]
 [ 3 16]]
array h: [[1 2]
 [0 0]]
array i: [[0 0]
 [3 2]]
~/De/c/t/tutorial_10 ····· 08:14:26 PM
❯
```

- You could use arithmetic operators + - * / ** % // directly between NumPy arrays. All of these are element-wise operations between ndarrays.

Sum and product

sum_prod.py > ...

```python
import numpy as np

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

b = np.sum(a)
print("array b:", b)
c = np.sum(a, axis=0)
print("array c:", c)
d = np.sum(a, axis=1)
print("array d:", d)
e = np.sum(a, axis=2)
print("array e:", e)

f = np.prod(a)
print("array f:", f)
g = np.prod(a, axis=0)
print("array g:", g)
h = np.prod(a, axis=1)
print("array h:", h)
i = np.prod(a, axis=2)
print("array i:", i)
```

```
~/De/c/t/tutorial_10 ····· 08:31:51 PM
❯ python3 sum_prod.py
array a: [[[1 2]
  [3 4]]

 [[5 6]
  [1 2]]]
array b: 24
array c: [[6 8]
 [4 6]]
array d: [[4 6]
 [6 8]]
array e: [[ 3  7]
 [11  3]]
array f: 1440
array g: [[ 5 12]
 [ 3  8]]
array h: [[ 3  8]
 [ 5 12]]
array i: [[ 2 12]
 [30  2]]
~/De/c/t/tutorial_10 ····· 08:31:53 PM
❯
```

- sum can be used to get sum of all elements in the array or sum along some axis as specified.
- prod can be used to get product of all elements in the array or product along some axis as specified.

Trigonometric operations

trig.py > ...

```
1   import numpy as np
2
3   a = np.array([[0, np.pi/6], [np.pi/4, np.pi/3]])
4   print("array a:", a)
5   b = np.sin(a)
6   print("array b:", b)
7   c = np.cos(a)
8   print("array c:", c)
9   d = np.tan(a)
10  print("array d:", d)
11
12  e = np.array([[0, 30], [45, 60]])
13  print("array e:", e)
14  f = np.radians(e)
15  print("array f:", f)
16  g = np.degrees(f)
17  print("array g:", g)
```

```
~/De/c/t/tutorial_10 ····· 08:40:59 PM
❯ python3 trig.py
array a: [[0.         0.52359878]
 [0.78539816 1.04719755]]
array b: [[0.         0.5       ]
 [0.70710678 0.8660254 ]]
array c: [[1.         0.8660254 ]
 [0.70710678 0.5       ]]
array d: [[0.         0.57735027]
 [1.         1.73205081]]
array e: [[ 0 30]
 [45 60]]
array f: [[0.         0.52359878]
 [0.78539816 1.04719755]]
array g: [[ 0. 30.]
 [45. 60.]]
~/De/c/t/tutorial_10 ····· 08:41:40 PM
❯
```

- NumPy provides sin, cos and tan that take values in radians and produce the corresponding sin, cos and tan values.
- You can use radians function to convert degrees to radians and degrees function to convert

Matrix operations

mat.py > ...

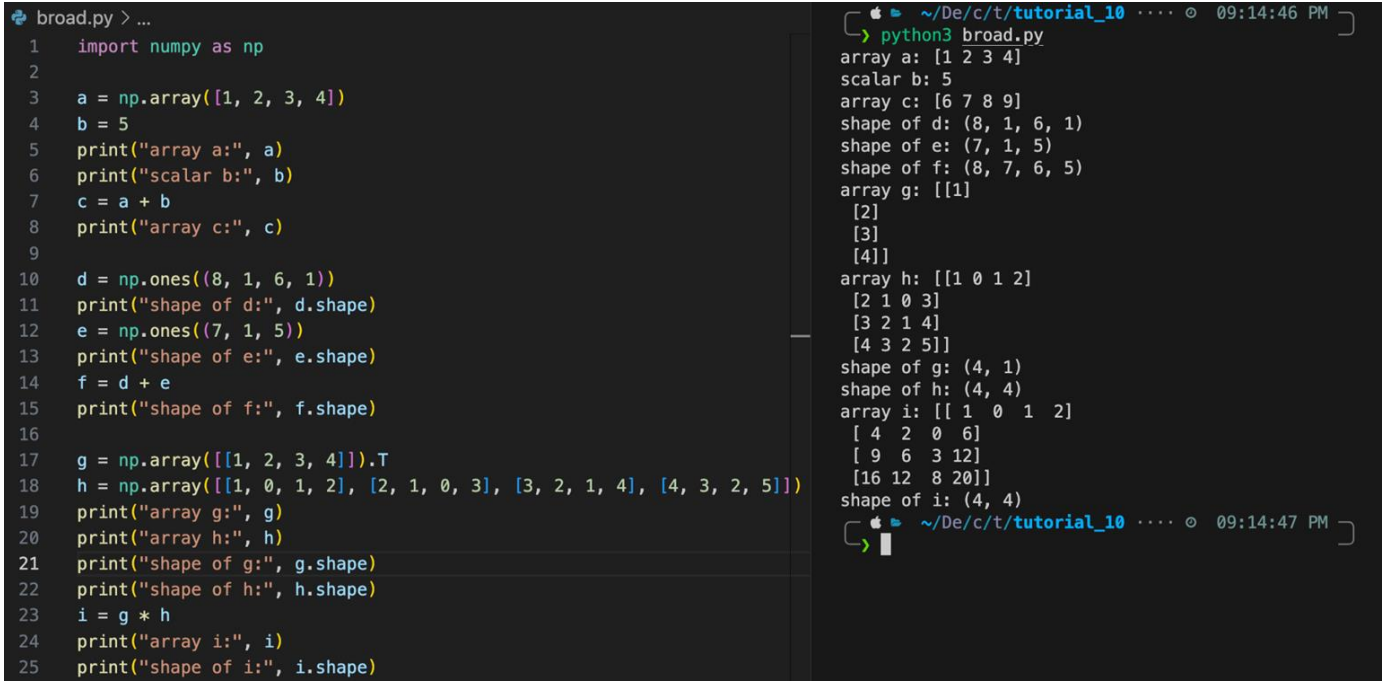
```python
import numpy as np

a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [1, 2]])
print("array a:", a)
print("array b:", b)

c = np.multiply(a, b)
print("array c:", c)
e = np.matmul(a, b)
print("array e:", e)
f = a @ b
print("array f:", f)

g = np.transpose(a)
print("array g:", g)
h = a.T
print("array h:", h)

i = np.linalg.inv(a)
print("array i:", i)

j = np.linalg.det(a)
print("array j:", j)
```

```
~/De/c/t/tutorial_10 ····· 08:55:31 PM
❯ python3 mat.py
array a: [[1 2]
 [3 4]]
array b: [[5 6]
 [1 2]]
array c: [[ 5 12]
 [ 3  8]]
array e: [[ 7 10]
 [19 26]]
array f: [[ 7 10]
 [19 26]]
array g: [[1 3]
 [2 4]]
array h: [[1 3]
 [2 4]]
array i: [[-2.   1. ]
 [ 1.5 -0.5]]
array j: -2.0000000000000004
~/De/c/t/tutorial_10 ····· 08:55:33 PM
❯
```

- We see the following matrix operations in the above example (There are a lot more to explore :)) :
  - matmul : matrix multiplication (alternate is to use @ operator)
  - transpose : transpose of the matrix (alternative is to use T attribute)
  - linalg.inv : inverse of the matrix
  - linalg.det : determinant of the matrix
- Note that multiply function is different from dot function.

Broadcasting in NumPy

Broadcasting



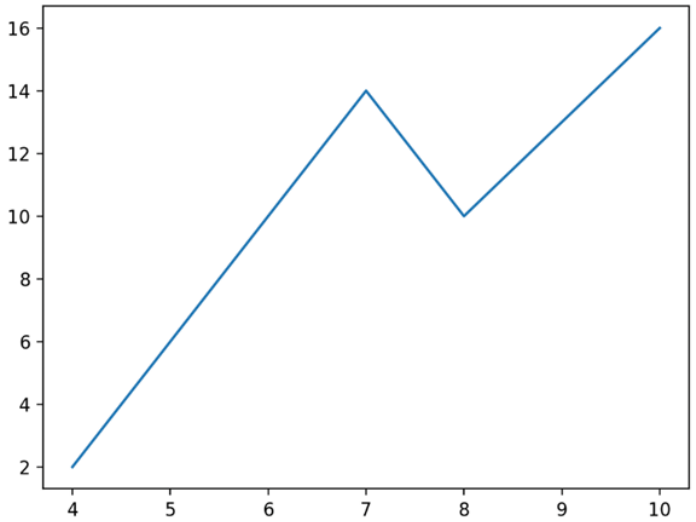
```
broad.py > ...
1   import numpy as np
2
3   a = np.array([1, 2, 3, 4])
4   b = 5
5   print("array a:", a)
6   print("scalar b:", b)
7   c = a + b
8   print("array c:", c)
9
10  d = np.ones((8, 1, 6, 1))
11  print("shape of d:", d.shape)
12  e = np.ones((7, 1, 5))
13  print("shape of e:", e.shape)
14  f = d + e
15  print("shape of f:", f.shape)
16
17  g = np.array([[1, 2, 3, 4]]).T
18  h = np.array([[1, 0, 1, 2], [2, 1, 0, 3], [3, 2, 1, 4], [4, 3, 2, 5]])
19  print("array g:", g)
20  print("array h:", h)
21  print("shape of g:", g.shape)
22  print("shape of h:", h.shape)
23  i = g * h
24  print("array i:", i)
25  print("shape of i:", i.shape)
```

```
~/De/c/t/tutorial_10 ···· 09:14:46 PM
❯ python3 broad.py
array a: [1 2 3 4]
scalar b: 5
array c: [6 7 8 9]
shape of d: (8, 1, 6, 1)
shape of e: (7, 1, 5)
shape of f: (8, 7, 6, 5)
array g: [[1]
 [2]
 [3]
 [4]]
array h: [[1 0 1 2]
 [2 1 0 3]
 [3 2 1 4]
 [4 3 2 5]]
shape of g: (4, 1)
shape of h: (4, 4)
array i: [[ 1  0  1  2]
 [ 4  2  0  6]
 [ 9  6  3 12]
 [16 12  8 20]]
shape of i: (4, 4)
~/De/c/t/tutorial_10 ···· 09:14:47 PM
❯
```

- The term broadcasting refers to how numpy treats arrays with different dimension during arithmetic operations which lead to certain constraints, the smaller array is broadcast across the larger array so that they have compatible shapes.
- When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimension and works its way left. Two dimensions are compatible when
    - they are equal, or
    - one of them is 1

Matplotlib

Basic plot

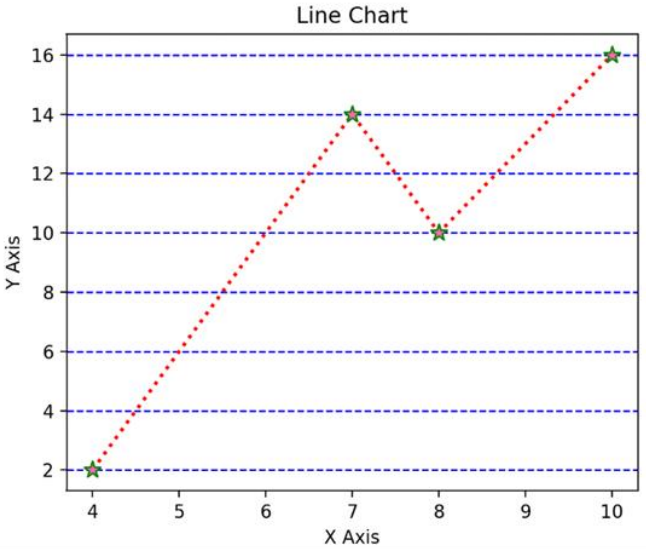


plot.py > ...

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.array([4, 7, 8, 10])
y = np.array([2, 14, 10, 16])

plt.plot(x, y)
plt.show()
```

- Matplotlib is a graph plotting library in python that serves as a visualization utility.
- Most of the Matplotlib utilities lies under the pyplot submodule, and are usually imported under the plt alias
- By default, the plot function draws a line from point to point.
- Parameter 1 is an array containing the points on the x-axis.
- Parameter 2 is an array containing the points on the y-axis.

Styled plot



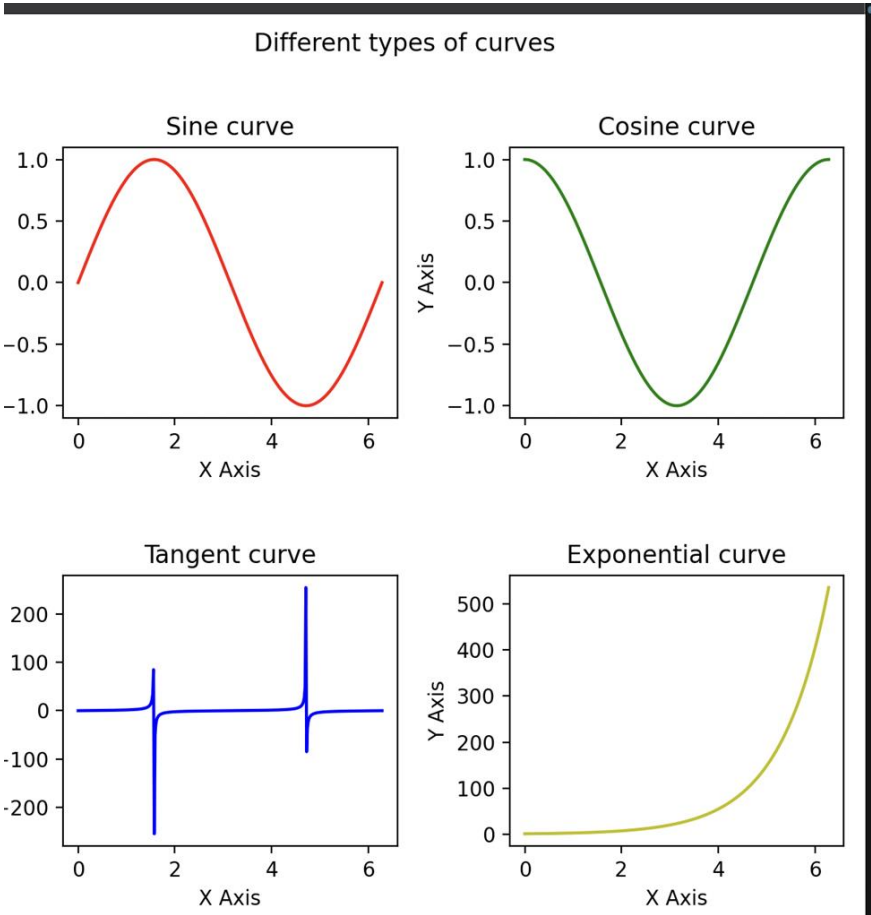
style.py > ...

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.array([4, 7, 8, 10])
y = np.array([2, 14, 10, 16])

plt.plot(x, y, marker = '*', ls = ':', lw = 2, c = 'r', ms = 10, mec = 'g', mfc = 'hotpink')
plt.title('Line Chart')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.grid(axis='y', color='b', ls='--', lw=1)
plt.show()
```

- There are a lot of styling features available in pyplot.
- In the above example, we used the following parameters:

  marker (m), linestyle (ls), linewidth (lw), color (c), markersize (ms), markeredgecolor (mec), markerfacecolor (mfc).

Subplots

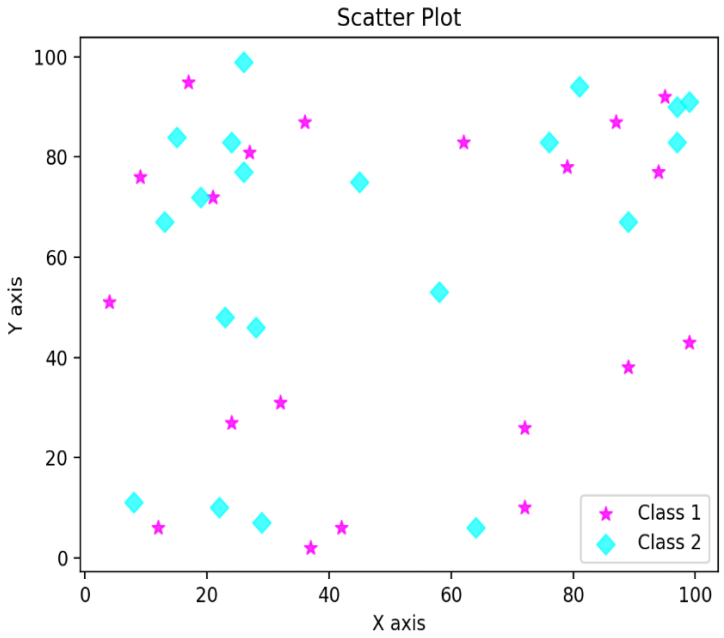


subplot.py > ...

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 400)
y0 = np.sin(x)
plt.subplot(2, 2, 1)
plt.plot(x, y0, c = 'r')
plt.title('Sine curve')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')

y1 = np.cos(x)
plt.subplot(2, 2, 2)
plt.plot(x, y1, c = 'g')
plt.title('Cosine curve')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')

y2 = np.tan(x)
plt.subplot(2, 2, 3)
plt.plot(x, y2, c = 'b')
plt.title('Tangent curve')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')

y3 = np.exp(x)
plt.subplot(2, 2, 4)
plt.plot(x, y3, c = 'y')
plt.title('Exponential curve')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')

plt.suptitle('Different types of curves')
plt.tight_layout()
plt.show()
```

- With the subplot function you can draw multiple plots in one figure
- The subplot function takes three arguments that describes the layout of the figure.
- The layout is organized in rows and columns, which are represented by the *first* and *second* argument.
- The third argument represents the index of the current plot.

Scatter plot

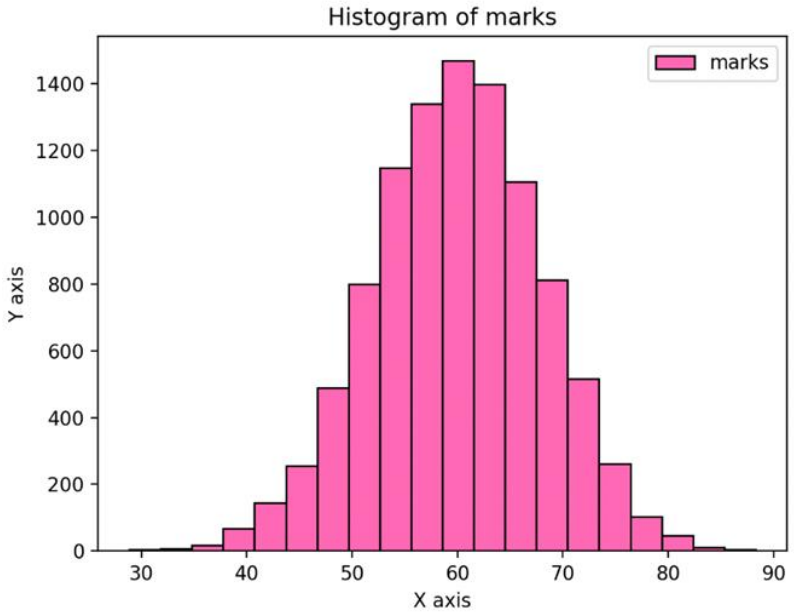


scatter.py > ...

```python
import numpy as np
import matplotlib.pyplot as plt

x1 = np.random.randint(100, size=20)
y1 = np.random.randint(100, size=20)

x2 = np.random.randint(100, size=20)
y2 = np.random.randint(100, size=20)

plt.scatter(x1, y1, c='magenta', alpha=0.8, marker='*', s=50, label='Class 1')
plt.scatter(x2, y2, c='cyan', alpha=0.7, marker='D', s=50, label='Class 2')

plt.title('Scatter Plot')
plt.xlabel('X axis')
plt.ylabel('Y axis')
plt.legend()

plt.savefig('scatter.png')
```

- With Pyplot, you can use the scatter function to draw a scatter plot.
- The scatter function plots one dot for each observation. It needs two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis.
- You can save any plot using the savefig function that takes the filename to which the image will be saved as an argument.

Histogram



hist.py > ...

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.random.normal(60, 8, 10000)

plt.hist(x, bins=20, color='hotpink', edgecolor='black', label='marks')
plt.title('Histogram of marks')
plt.xlabel('X axis')
plt.ylabel('Y axis')
plt.legend()
plt.show()
```

- A histogram is a graph showing frequency distributions.
- It is a graph showing the number of observations within each given interval.
- In Matplotlib, we use the hist function to create histograms.

Exercises

Exercise 1

We talked about NumPy being faster in arithmetic operations than ordinary python lists due to vectorization. Let's test it now.

Create a 1000*1000 matrix A and compute its square such that the matrix is given by: A[i, j] = 1000*i + j

Do it using ordinary python lists as well as numpy operations and compare the time taken.

Solution 1

exer_1.py > np_array

```python
import numpy as np
import time

def py_list(dim):
    matrix = [[dim*i+j for j in range(dim)] for i in range(dim)]
    square = [[0 for j in range(dim)] for i in range(dim)]
    start = time.time()
    for i in range(dim):
        for j in range(dim):
            for k in range(dim):
                square[i][j] += matrix[i][k] * matrix[k][j]
    end = time.time()
    print(f'Python list elapsed time: {end-start} seconds')

def np_array(dim):
    matrix = np.arange(dim*dim).reshape(dim, -1)
    start = time.time()
    square = matrix @ matrix
    end = time.time()
    print(f'NumPy array elapsed time: {end-start} seconds')

if __name__ == '__main__':
    dim = 1000
    py_list(dim)
    np_array(dim)
```

```
~/De/c/t/tutorial_10 ····· 01:06:12 AM
❯ python3 exer_1.py
Python list elapsed time: 79.5338089466095 seconds
NumPy array elapsed time: 0.4554412364959717 seconds
~/De/c/t/tutorial_10
❯
```

Notice how fast NumPy is as compared to Python lists.

Exercise 2

We will see use of broadcasting in this exercise.

You are given a set of let's say five 2d points as an ndarray and you want to compute the euclidean distance between each pair of points and store it into a 5*5 ndarray.

Naive approach will be to use loops to do this task, but your *aim* is to do it *without use of any loops*.

Solution 2

exer_2.py > ...

```python
import numpy as np

def distance(arr):
    diff = arr.reshape(arr.shape[0], 1, arr.shape[1]) - arr
    dist = np.sqrt(np.sum(diff**2, axis=2))
    return dist

if __name__ == '__main__':
    arr = np.random.randint(0, 5, size=(5, 2))
    print(arr)
    print(distance(arr))
```

```
~/De/c/tutorials/tutorial_10 ··· 01:24:54 AM
❯ python3 exer_2.py
[[4 0]
 [1 4]
 [0 0]
 [2 2]
 [4 4]]
[[0.         5.         4.         2.82842712 4.        ]
 [5.         0.         4.12310563 2.23606798 3.        ]
 [4.         4.12310563 0.         2.82842712 5.65685425]
 [2.82842712 2.23606798 2.82842712 0.         2.82842712]
 [4.         3.         5.65685425 2.82842712 0.        ]]
~/De/c/tutorials/tutorial_10 ··· 01:24:56 AM
❯
```

1. Notice how we first changed the shape of arr before computing the difference between co-ordinates of the points.
2. After reshaping, the dimensions mismatch and hence numpy use broadcasting to compute the difference.
3. Finally we find the distance using sum, power and sqrt from the difference computed.

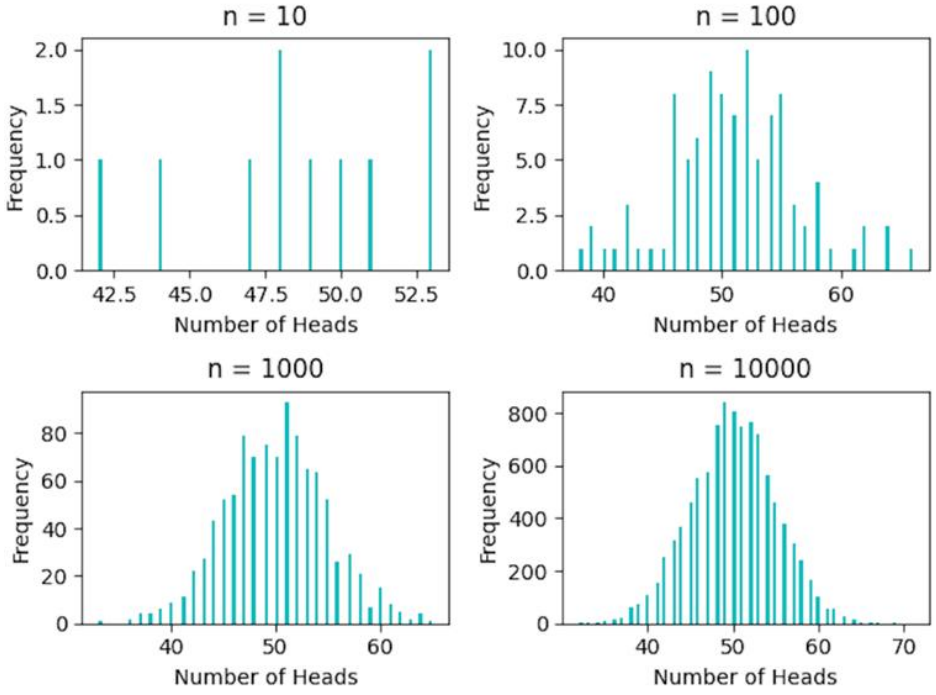
Exercise 3

Say you toss 100 fair coins simultaneously and want to programmatically record the total number of heads. One simultaneous toss of the 100 coins counts as a single trial. You are given a list of number of trials ([10,100,1000,10000]). For every value in this list, we want to count the number of heads and plot a histogram. The histogram should take the shape of a binomial distribution and get more accurate with larger numbers of trials.

You need to define two functions:

1. toss(num_trials): Takes the number of times the experiment is to be performed as an argument and returns a numpy array of the same size with the number of heads obtained in each trial.
2. plot_hist(outcomes, index):- Takes an array of the number of heads obtained for *k* trials from toss(*k*) and plots a histogram based on these counts. Generate subplot at index giving as the argument for each value in trials_list and save this plot as exer_3.png

Solution 3



exer_3.py > ...

```python
import numpy as np
import matplotlib.pyplot as plt

def toss(number_trials):
    return np.sum(np.random.randint(0, 2, (number_trials, 100)), axis=1)

def plot_hist(outcomes, index):
    plt.subplot(2, 2, index)
    plt.hist(outcomes, bins=100, color='c')
    plt.xlabel('Number of Heads')
    plt.ylabel('Frequency')
    plt.title(f'n = {outcomes.shape[0]}')
    plt.tight_layout()
    plt.savefig('exer_3.png')

if __name__ == '__main__':
    list_trials = np.array([10, 100, 1000, 10000])
    for i in range(list_trials.shape[0]):
        plot_hist(toss(list_trials[i]), i+1)
```

Thank You !!!