**软件开发应用基础实践**

**【实验目的】**

1. **使用Jupyter Notebook编程**
2. **掌握python基本代码编程**

**【实验内容】**

**1.学习****前向反向传播**

**2.学会类的创建和调用**

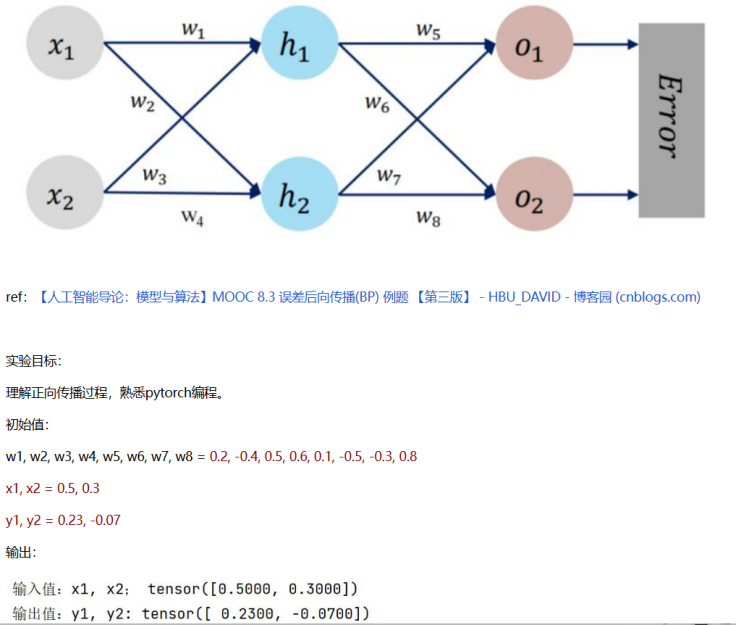
**3.导入使用numpy库**

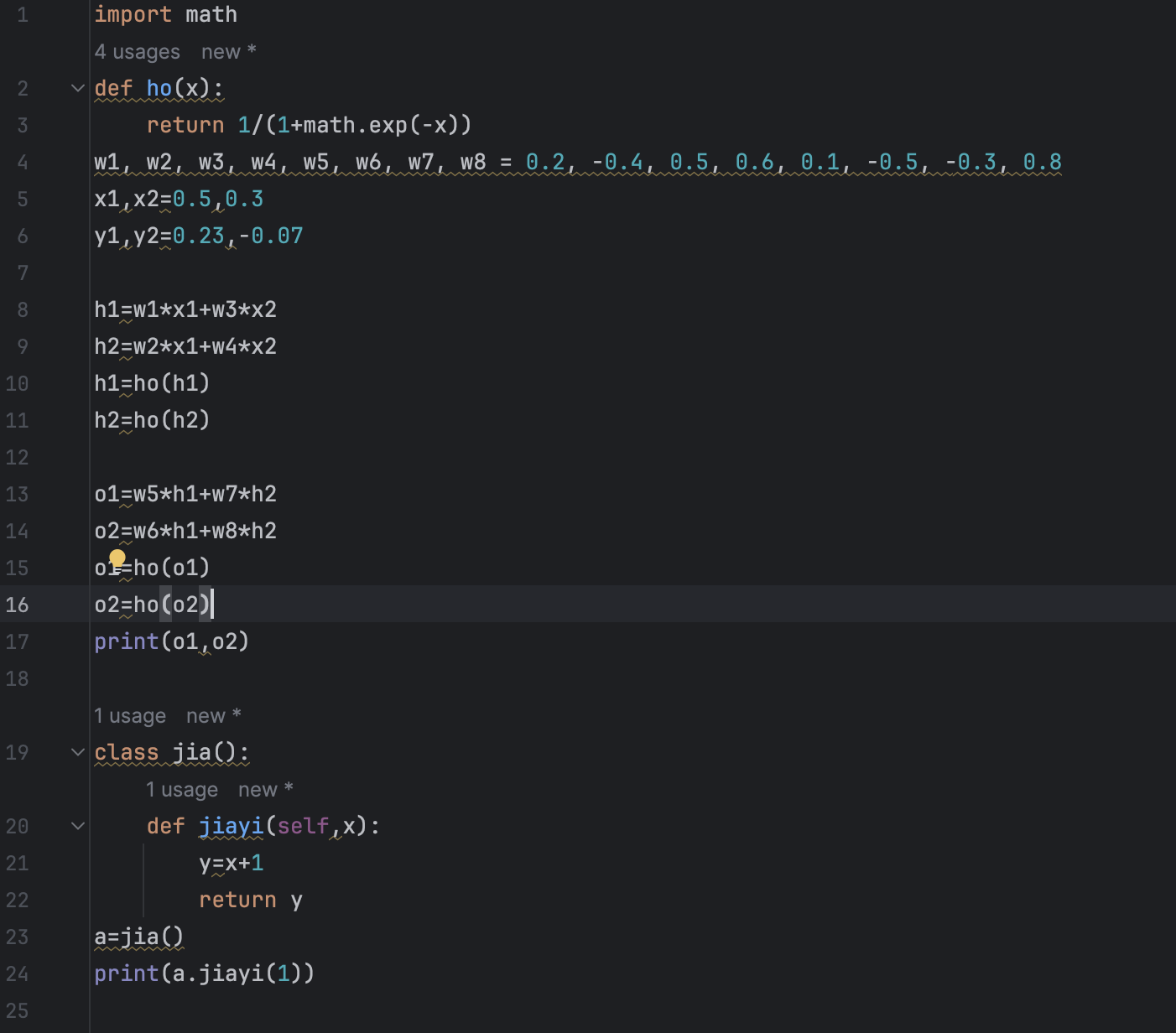
**4.学会****使用numpy库建立矩阵，使用库中的函数方法，**

**5.使用matplotlib.pyplot库绘图**

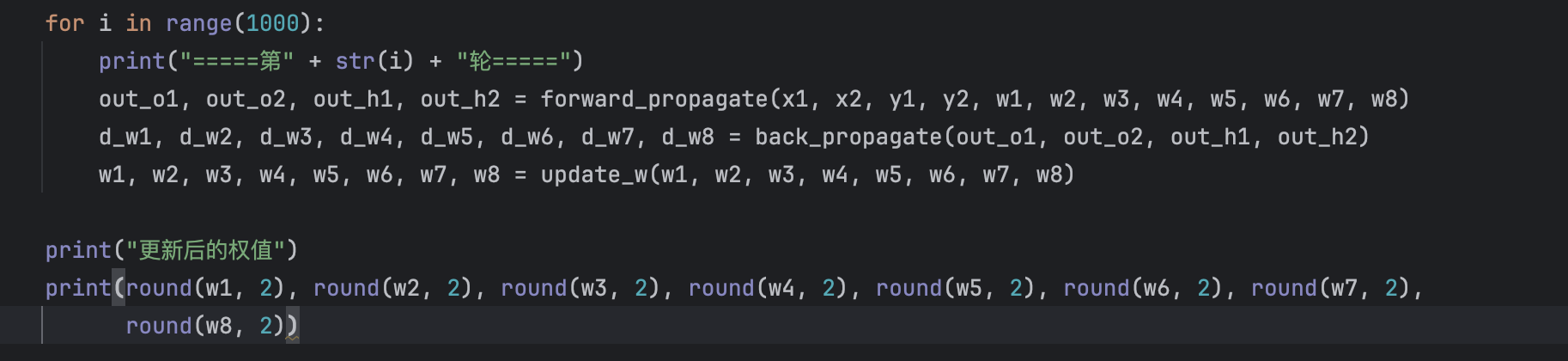
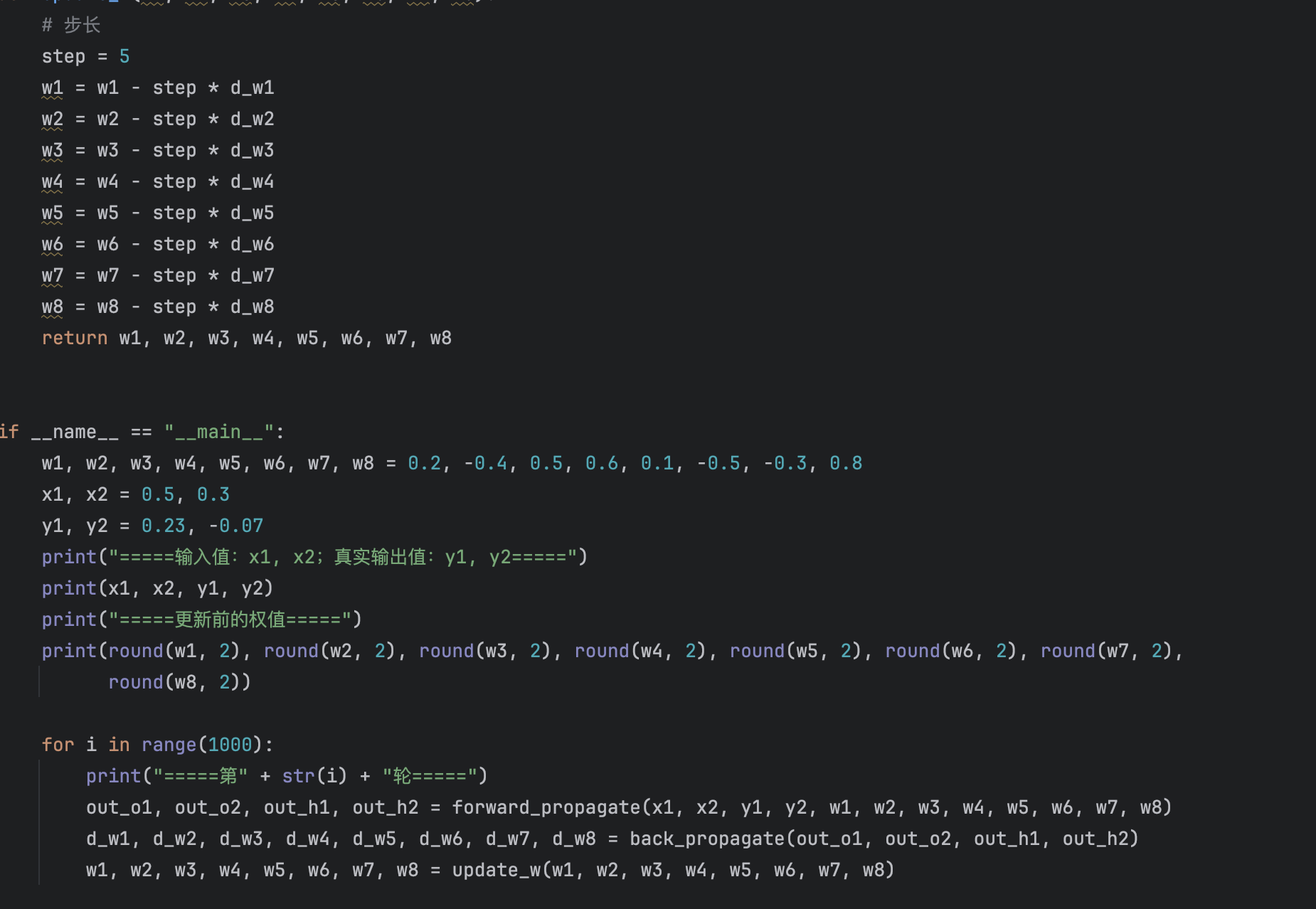
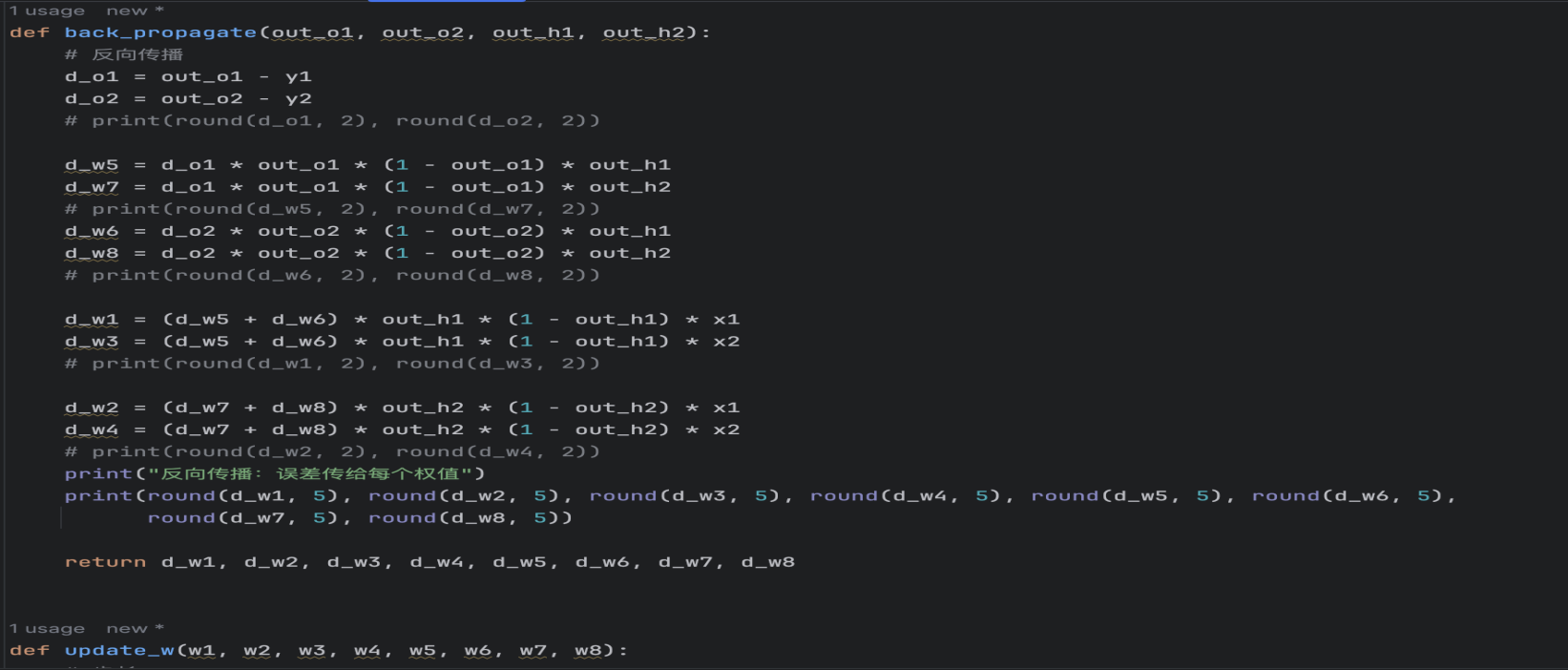
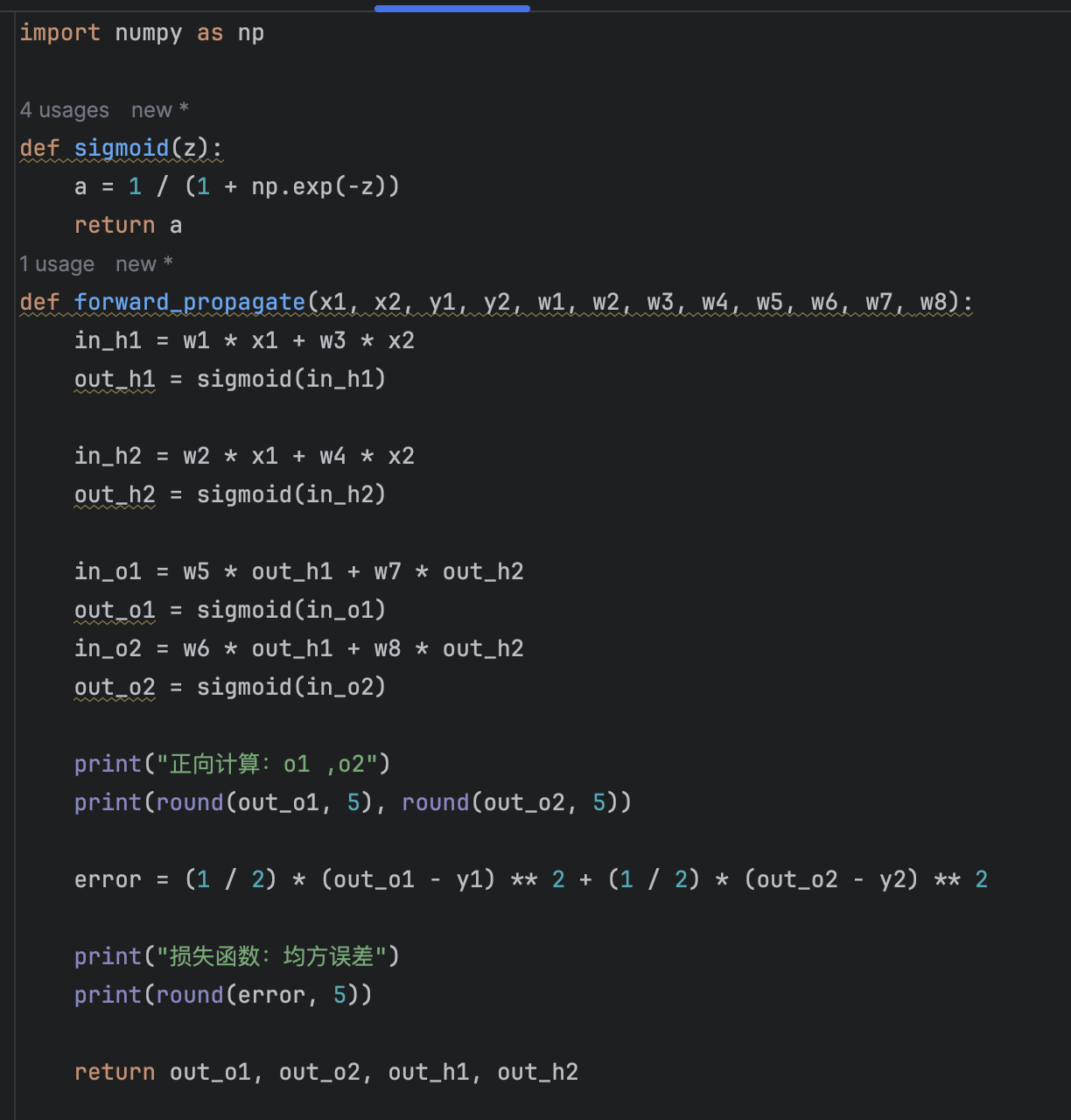
**【实验结果/结论】**

**前向传播**





**前向反向传播**



**类的创建和调用**

class jia():

def jiayi(self,x):

y=x+1

return y

a=jia()

print(a.jiayi(1))

**导入numpy库**

import numpy as np

**使用numpy库**

建立一个一维数组 a 初始化为[4,5,6],

输出a 的类型（type）

输出a的各维度的大小（shape）

输出 a的第一个元素（值为4）

import numpy as np

a=np.array([4,5,6])

print('数组a:',a)

print('a的类型：',type(a))

print('a的各维度的大小：',a.shape)

print('a的第一个元素',a[0])

建立一个二维数组 b,初始化为 [ [4, 5, 6],[1, 2, 3]]

1. 输出各维度的大小（shape）
2. 输出 b(0,0)，b(0,1),b(1,1)

这三个元素（对应值分别为4,5,2）

import numpy as np

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

print ('二维数组b:', b)

print('b的各维度的大小：',b. shape)

print ('b的三个元素：',b[0, 0], b[0, 1], b[1, 1])

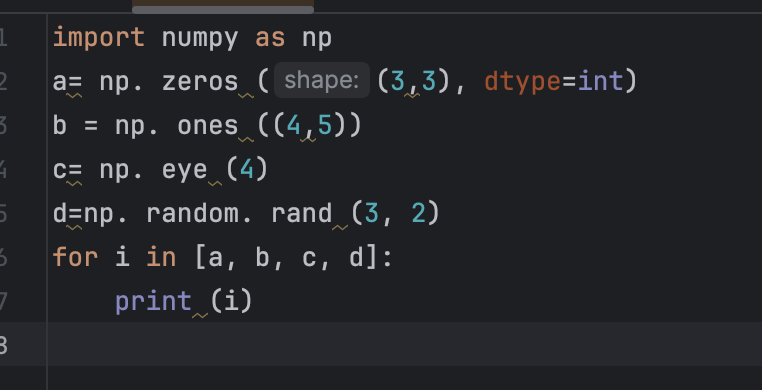
**使用numpy库建立矩阵，使用库中的函数方法**

1. 建立一个全0矩阵 a, 大小为 3x3; 类型为整型

（提示: dtype = int）

1. 建立一个全1矩阵b,大小为4x5;
2. 建立一个单位矩阵c ,大小为4x4;

(4)生成一个随机数矩阵d,大小为 3x2.



建立一个数组 a,(值为[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]] ) ,

1. 打印a;
2. 输出 下标为(2,3),(0,0) 这两个数组元素的值

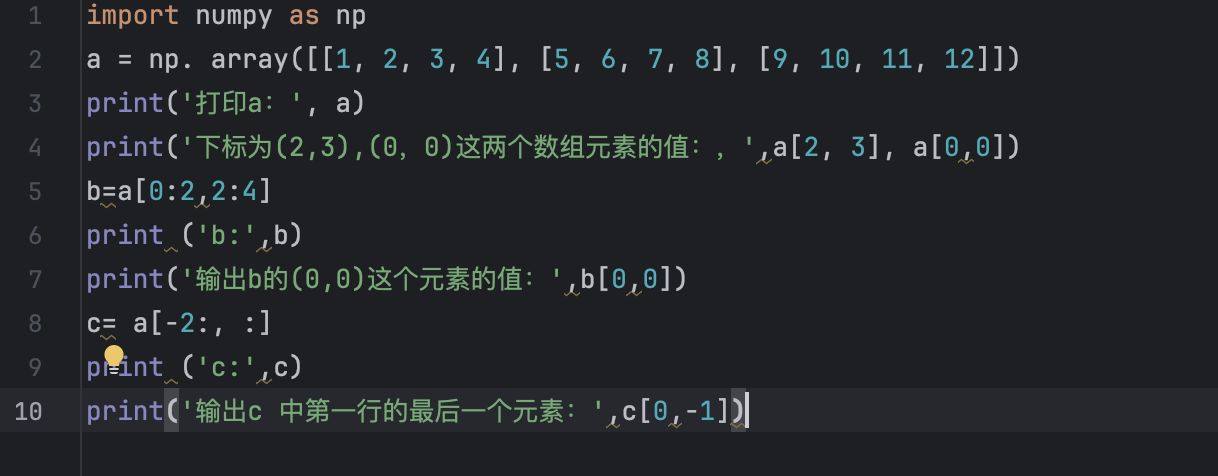
把上一题的 a数组的 0到1行 2到3列，放到b里面去，（此处不需要从新建立a,直接调用即可）

(1),输出b;

(2) 输出b 的（0,0）这个元素的值

把第5题中数组a的最后两行所有元素放到 c中，（提示： a[1:2, :]）

1. 输出 c ;
2. 输出 c 中第一行的最后一个元素（提示，使用 -1 表示最后一个元素）



#### 8.建立数组a,初始化a为[[1, 2], [3, 4], [5, 6]]，输出 （0,0）（1,1）（2,0）这三个元素（提示： 使用 print(a[[0, 1, 2], [0, 1, 0]]) ）

#### import numpy as np

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

#### print(a[[0,1,2],[0,1,0]])

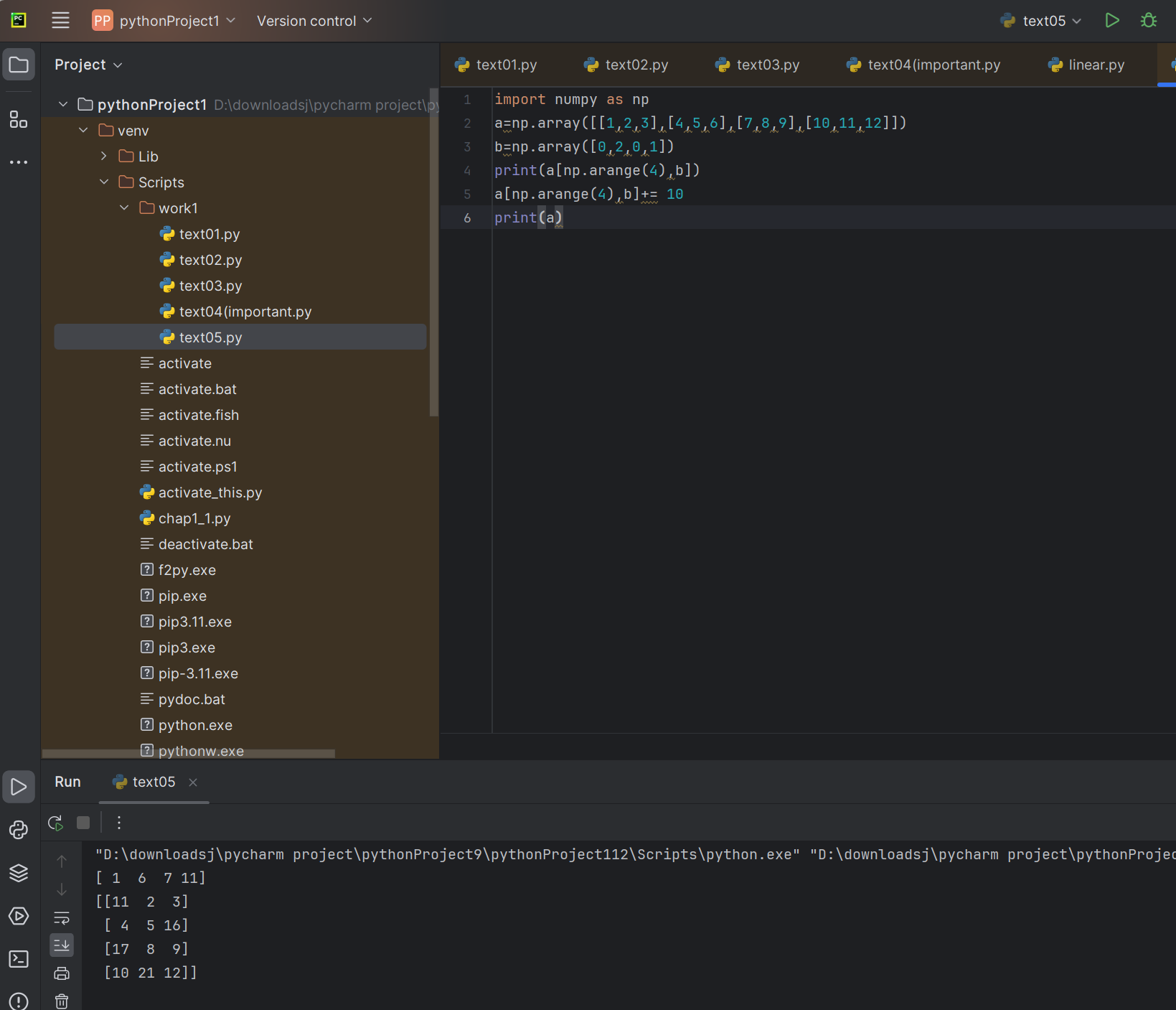
9. 建立矩阵

a ,初始化为[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]，

输出(0,0),(1,2),(2,0),(3,1)

(提示使用 b = np.array([0, 2, 0, 1]) print(a[np.arange(4), b]))

10. 对9 中输出的那四个元素，每个都加上10，然后重新输出矩阵



11. 执行 x = np.array([1, 2])，然后输出 x 的数据类型

import numpy as np

x=np.array([1,2])

print(x.dtype)

12.执行 x = np.array([1.0, 2.0]) ，然后输出 x 的数据类类型

import numpy as np

x=np.array([1.0,2.0])

print (x.dtype)

13.执行 x = np.array([[1, 2], [3, 4]], dtype=np.float64) ，y = np.array([[5, 6], [7, 8]], dtype=np.float64)，然后输出 x+y ,和 np.add(x,y)

import numpy as np

x=np.array ([[1,2],[3,4]],dtype=np.float64)

y =np.array([[5,6],[7,8]],dtype=np.float64)

print(x+y)

print(np.add(x,y))

14. 利用 13题目中的x,y 输出 x-y 和 np.subtract(x,y)

15. 利用13题目中的x，y 输出 x\*y ,和 np.multiply(x, y) 还有 np.dot(x,y),比较差异。然后自己换一个不是方阵的试试

16. 利用13题目中的x,y,输出 x / y .(提示 ： 使用函数 np.divide())

17. 利用13题目中的x,输出 x的 开方。(提示： 使用函数 np.sqrt() )

18.利用13题目中的x,y ,执行 print(x.dot(y)) 和 print(np.dot(x,y))

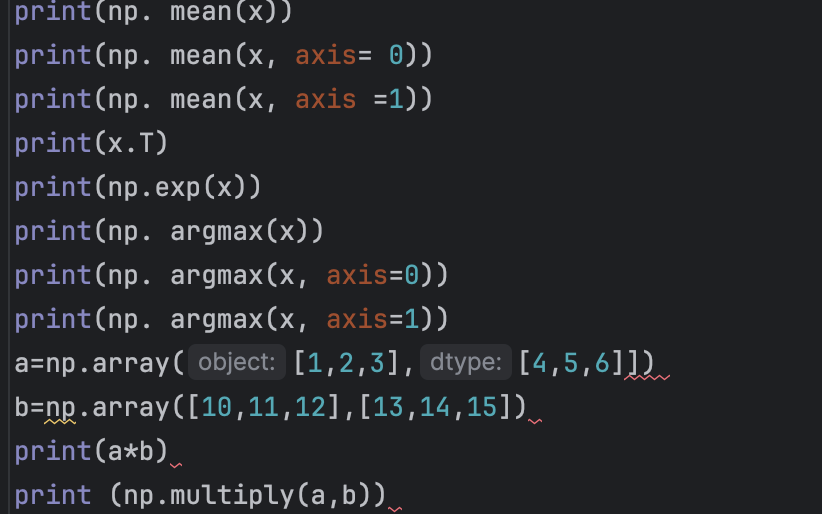
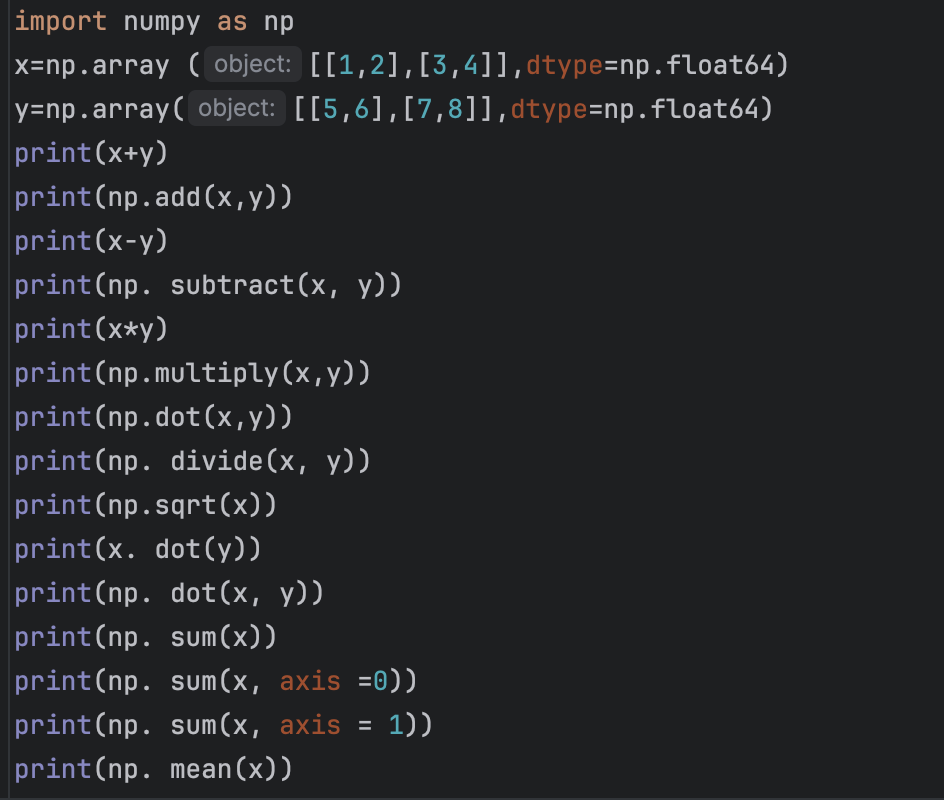
19.利用13题目中的 x,进行求和。提示：输出三种求和 (1)print(np.sum(x)): (2)print(np.sum(x，axis =0 )); (3)print(np.sum(x,axis = 1))

20.利用13题目中的 x,进行求平均数（提示：输出三种平均数(1)print(np.mean(x)) (2)print(np.mean(x,axis = 0))(3) print(np.mean(x,axis =1))）

21.利用13题目中的x，对x 进行矩阵转置，然后输出转置后的结果，（提示： x.T 表示对 x 的转置）

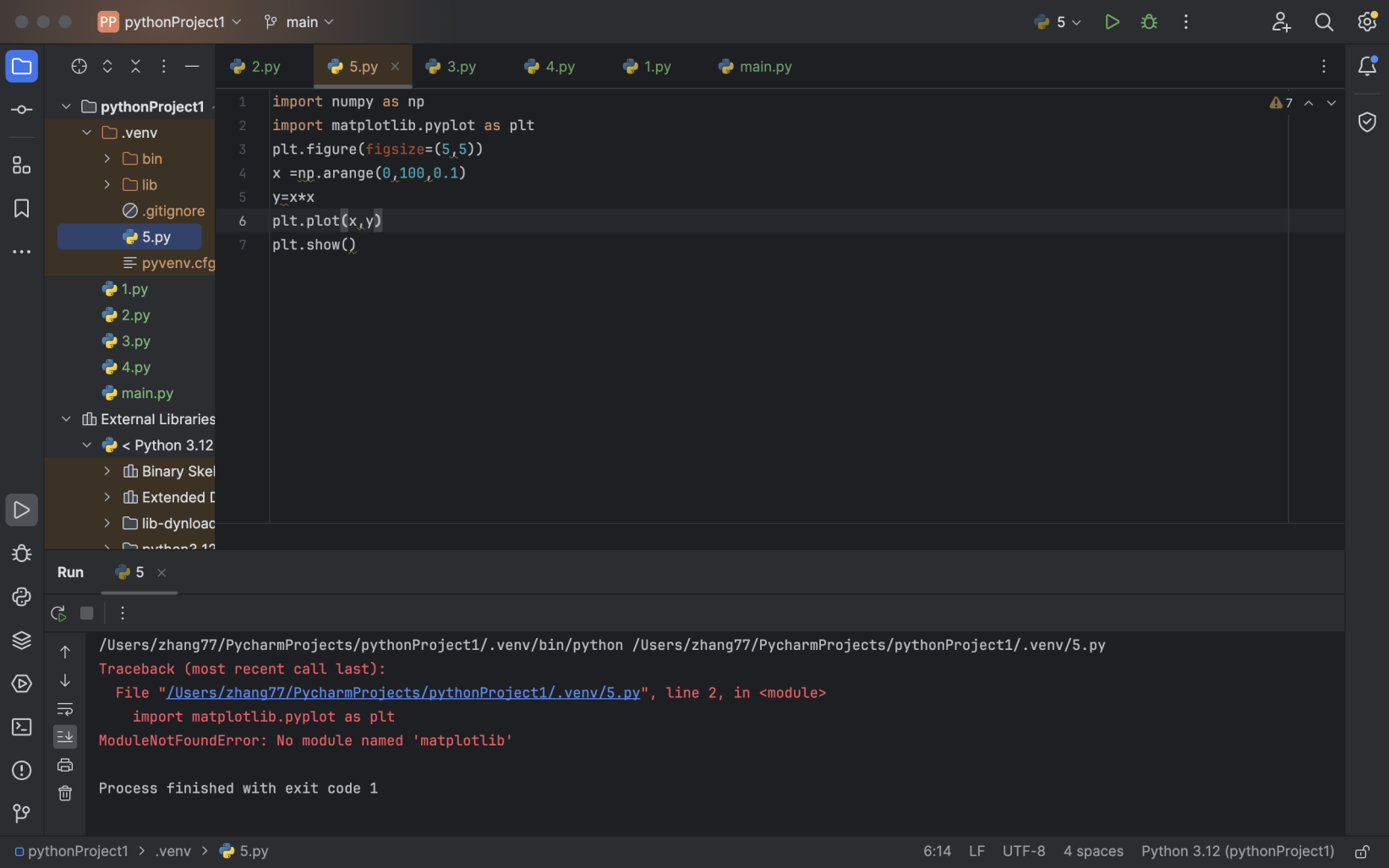
22.利用13题目中的x,求e的指数（提示： 函数 np.exp()）

23.利用13题目中的 x,求值最大的下标（提示(1)print(np.argmax(x)) ,(2) print(np.argmax(x, axis =0))(3)print(np.argmax(x),axis =1))



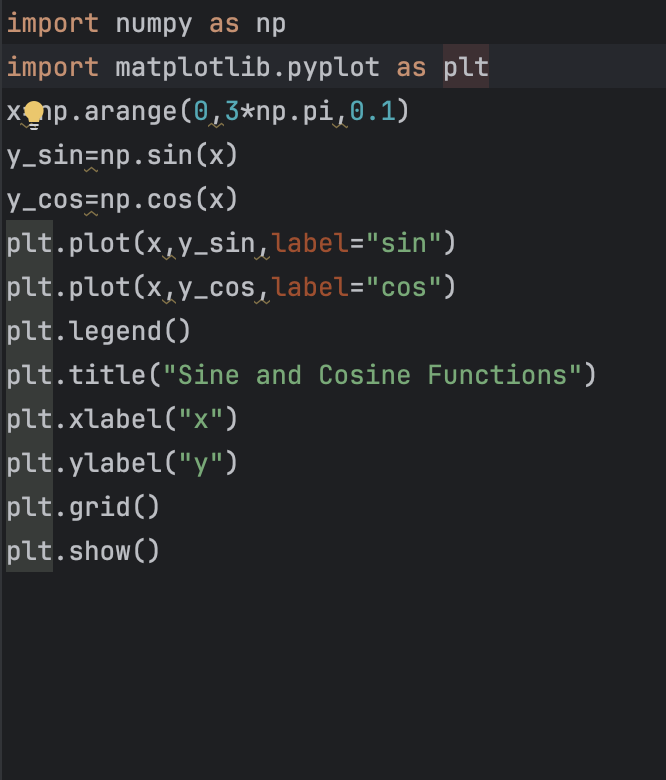
**使用matplotlib.pyplot库绘图**

24,画图，y=x\*x 其中 x = np.arange(0, 100, 0.1) （提示这里用到 matplotlib.pyplot 库）



报错了

#### 画图。画正弦函数和余弦函数， x = np.arange(0, 3 \* np.pi, 0.1)(提示：这里用到 np.sin() np.cos() 函数和 matplotlib.pyplot 库)



**其他练习**

a = torch.empty((3,3))

print(a)

b= torch.empty((1,5))

print(b)

import torch

import numpy as np

my\_tensor = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.float32,requires\_grad=True)

print (my\_tensor)

print(my\_tensor.dtype)

print(my\_tensor.device)

print(my\_tensor.shape)

print(my\_tensor.requires\_grad) # Other common initialization methods

x = torch.empty(size = (3, 3))

x = torch.zeros((3, 3))

x = torch.rand((3, 3))

x = torch.ones((3, 3))

x = torch.eye(5, 5) # I, eye

x = torch.arange(start=0, end=5, step=1)

x = torch.linspace(start=0.1, end=1, steps=10)

x = torch.empty(size=(1, 5)).normal\_(mean=0, std=1)

x = torch.empty(size=(1,5)).uniform\_(0, 1)

x = torch.diag(torch.ones(3))# How to initialize and convert tensors to other types (int, float, double)

tensor = torch.arange(4)

print(tensor.bool()) # boolean True/False

print(tensor.short()) # int16

print(tensor.long()) #int64 (Important)

print(tensor.half()) # float16 36

print(tensor.float()) #float32 (Important)

print(tensor.double()) #float64 # Array to Tensor conversion and vice-versa

np\_array = np.zeros((5, 5))

tensor = torch.from\_numpy(np\_array)

np\_array\_back = tensor.numpy

import torch

import numpy as np

x=torch.tensor([1,2,3])

y=torch.tensor([9,8,7])

z1=torch.empty(3)

torch.add(x,y,out=z1)

z2 =torch.add(x,y)

z=x+y

z=x-y

z=torch.true\_divide(x,y)

t=torch.zeros(3)

t.add\_(x)

t+=x

z=x.pow(2)

z=x\*\*2

z=x>0

z=x<0

x1=torch.rand((2,5))

x2=torch.rand((5,3))

x3=torch.mm(x1,x2)

matrix\_exp = torch.rand(5, 5)

print(matrix\_exp.matrix\_power(3))

z = x \* y

print(z)

z = torch.dot(x, y)

print (z)

# Batch Matrix Multiplication

batch = 32

n=10

m = 20

p= 30

tensor1 = torch.rand((batch, n, m))

tensor2 = torch.rand((batch, m, p))

out\_bmm = torch.bmm(tensor1, tensor2)

# (batch, n, p)

# Example of Broadcasting

x1 = torch.rand((5,5))

x2 = torch.rand((1,5))

z = x1 - x2

z = x1 \*\* x2

# Other useful tensor operations

sum\_x = torch.sum(x, dim=0)

values, indices = torch.max(x, dim=0)

values, indices = torch.min(x, dim=0)

mean\_x=torch.mean(x.float(),dim=0)

z=torch.eq(x,y)

sorted\_y,indices=torch.sort(y,dim=0,descending=False)

z=torch.clamp(x,min=0)

x=torch.tensor([1,0,1,1,1],dtype=torch.bool)

z=torch.any(x)

z=torch.all(x)

import torch

batch\_size=10

features=25

x=torch.rand((batch\_size,features))

print(x[0].shape)

print(x[:,0].shape)

print(x[2,0:10])#0:10→[0,1,2，…，9]

x[0,0]=100

#Fancy indexing

x=torch.arange(10)

indices=[2,5,8]

print(x[indices])

x=torch.rand((3,5))

rows=torch.tensor([1,0])

cols=torch.tensor([4,0])

print(x[rows,cols].shape) #More advanced indexing

x=torch.arange(10)

print(x[(x<2)&(x>8)])

print(x[x.remainder(2)==0])

#Useful operations

print(torch.where(x>5,x,x\*2))

print(torch.tensor([0,0,1,2,2,3,4]).unique())

print(x.ndimension()) # 5x5×5

print(x.numel())

import torch

x = torch.arange(9)

x\_3x3 = x.view(3, 3)

print(x\_3x3)

x\_3x3 = x.reshape(3, 3)

y = x\_3x3.t() # [0, 1, 4, 7, 2, 5, 8]

print(y.contiguous().view(9))

x1=torch.rand((2,5))

x2=torch.rand((2,5))

print(torch.cat((x1,x2),dim=0).shape)

print(torch.cat((x1,x2),dim=1).shape)

z=x1.view(-1)

print(z.shape)

batch =64

x=torch.rand((batch,2,5))

z=x.view(batch,-1)

print(z.shape)

z=x.permute(0,2,1)

print(z.shape)

x=torch.arange(10)#[10]

print(x.unsqueeze(0).shape)

print(x.unsqueeze(1).shape)

x=torch.arange(10).unsqueeze(0).unsqueeze(1)#1x1x10

z=x.squeeze(1)

print(z.shape)

import torch

import torch.nn.functional as F # Parameterless functions, like (some) activation functions

import torchvision.datasets as datasets # Standard datasets

import torchvision.transforms as transforms # Transformations we can perform on our dataset for augmentation

from torch import optim # For optimizers like SGD, Adam, etc.

from torch import nn # All neural network modules

from torch.utils.data import (

DataLoader,

) # Gives easier dataset managment by creating mini batches etc.

from tqdm import tqdm # For nice progress bar!

# Here we create our simple neural network. For more details here we are subclassing and

# inheriting from nn.Module, this is the most general way to create your networks and

# allows for more flexibility. I encourage you to also check out nn.Sequential which

# would be easier to use in this scenario but I wanted to show you something that

# "always" works and is a general approach.

class NN(nn.Module):

def \_\_init\_\_(self, input\_size, num\_classes):

"""

Here we define the layers of the network. We create two fully connected layers

Parameters:

input\_size: the size of the input, in this case 784 (28x28)

num\_classes: the number of classes we want to predict, in this case 10 (0-9)

"""

super(NN, self).\_\_init\_\_()

# Our first linear layer take input\_size, in this case 784 nodes to 50

# and our second linear layer takes 50 to the num\_classes we have, in

# this case 10.

self.fc1 = nn.Linear(input\_size, 50)

self.fc2 = nn.Linear(50, num\_classes)

def forward(self, x):

"""

x here is the mnist images and we run it through fc1, fc2 that we created above.

we also add a ReLU activation function in between and for that (since it has no parameters)

I recommend using nn.functional (F)

Parameters:

x: mnist images

Returns:

out: the output of the network

"""

x = F.relu(self.fc1(x))

x = self.fc2(x)

return x

# Set device cuda for GPU if it's available otherwise run on the CPU

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# Hyperparameters

input\_size = 784

num\_classes = 10

learning\_rate = 0.001

batch\_size = 64

num\_epochs = 3

# Load Data

train\_dataset = datasets.MNIST(

root="dataset/", train=True, transform=transforms.ToTensor(), download=True

)

test\_dataset = datasets.MNIST(

root="dataset/", train=False, transform=transforms.ToTensor(), download=True

)

train\_loader = DataLoader(dataset=train\_dataset, batch\_size=batch\_size, shuffle=True)

test\_loader = DataLoader(dataset=test\_dataset, batch\_size=batch\_size, shuffle=True)

# Initialize network

model = NN(input\_size=input\_size, num\_classes=num\_classes).to(device)

# Loss and optimizer

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=learning\_rate)

# Train Network

for epoch in range(num\_epochs):

for batch\_idx, (data, targets) in enumerate(tqdm(train\_loader)):

# Get data1 to cuda if possible

data = data.to(device=device)

targets = targets.to(device=device)

# Get to correct shape

data = data.reshape(data.shape[0], -1)

# Forward

scores = model(data)

loss = criterion(scores, targets)

# Backward

optimizer.zero\_grad()

loss.backward()

# Gradient descent or adam step

optimizer.step()

# Check accuracy on training & test to see how good our model

def check\_accuracy(loader, model):

"""

Check accuracy of our trained model given a loader and a model

Parameters:

loader: torch.utils.data1.DataLoader

A loader for the dataset you want to check accuracy on

model: nn.Module

The model you want to check accuracy on

Returns:

acc: float

The accuracy of the model on the dataset given by the loader

"""

num\_correct = 0

num\_samples = 0

model.eval()

# We don't need to keep track of gradients here so we wrap it in torch.no\_grad()

with torch.no\_grad():

# Loop through the data1

for x, y in loader:

# Move data1 to device

x = x.to(device=device)

y = y.to(device=device)

# Get to correct shape

x = x.reshape(x.shape[0], -1)

# Forward pass

scores = model(x) #[64,10]

\_, predictions = scores.max(1)

# Check how many we got correct

num\_correct += (predictions == y).sum()

# Keep track of number of samples

num\_samples += predictions.size(0)

model.train()

return num\_correct / num\_samples

# Check accuracy on training & test to see how good our model

print(f"Accuracy on training set: {check\_accuracy(train\_loader, model)\*100:.2f}")

print(f"Accuracy on test set: {check\_accuracy(test\_loader, model)\*100:.2f}")

import torch.nn as nn

import torch

import torch.nn.functional as F # Parameterless functions, like (some) activation functions

import torchvision.datasets as datasets # Standard datasets

import torchvision.transforms as transforms # Transformations we can perform on our dataset for augmentation

from torch import optim # For optimizers like SGD, Adam, etc.

from torch import nn # All neural network modules

from torch.utils.data import (

DataLoader,

) # Gives easier dataset managment by creating mini batches etc.

class Tudui(nn.Module):

def \_\_init\_\_(self, \*args, \*\*kwargs) -> None:

super().\_\_init\_\_(\*args, \*\*kwargs)

self.fc1 = nn.Linear(2,2)

self.fc2 = nn.Linear(2,2)

def forward(self,x):

out = self.fc1(x)

out = torch.relu(out)

out =self.fc2(out)

out = torch.relu(out)

return out

x = torch.tensor([0.5,0.3])

tudui = Tudui()

out1 = tudui(x)

print(out1)

import torch.nn

import torchvision

import torch.nn as nn

from torch.utils.data import DataLoader

from tqdm import tqdm

train\_data = torchvision.datasets.MNIST(root='data',train=True,transform=torchvision.transforms.ToTensor(),download=True)

test\_data = torchvision.datasets.MNIST(root='data',train=False,transform=torchvision.transforms.ToTensor(),download=True)

train\_loader = DataLoader(dataset=train\_data,batch\_size=64,shuffle=True)

test\_loader = DataLoader(dataset=test\_data,batch\_size=64,shuffle=True)

class Tudui(nn.Module):

    def \_\_init\_\_(self, \*args, \*\*kwargs) -> None:

        super().\_\_init\_\_(\*args, \*\*kwargs)

        self.fc1 = nn.Linear(28\*28,50)

        self.fc2 = nn.Linear(50,10)

    def forward(self,x):

        out = torch.relu(self.fc1(x))

        out = torch.relu(self.fc2(out))

        return out

tudui = Tudui()

loss\_fn = torch.nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(params=tudui.parameters(),lr=0.01)

for epoch in range(5):

    total\_loss = 0

    for idx,(data, label) in enumerate(tqdm(train\_loader)):

        data = data.reshape(data.shape[0],-1)

        out = tudui(data)  #[64,1,28,28] out是前向传播的结果

        loss = loss\_fn(out,label)

        total\_loss += loss

        if idx % 100 == 0:

            print('总损失值为：',total\_loss)

        optimizer.zero\_grad()

        loss.backward()

        optimizer.step()