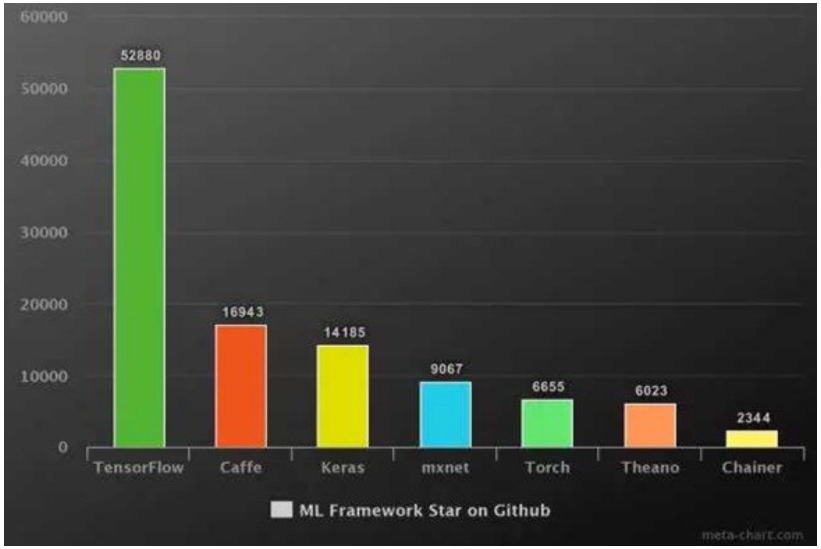
Lab 7 - TensorFlow

www.huawei.com

Content

What Is TensorFlow

- TensorFlow is Google's second-generation software library for dataflow programming.
- The TensorFlow framework supports various deep learning algorithms, as well as many computing platforms other than those for deep learning. The system stability is high.
- TensorFlow is open-source, which facilitates maintenance and update and improves the development efficiency.



Copyright Huawei Technologies Co., Ltd

What Is TensorFlow

- TensorFlow can train and run the deep neural networks for
  - image recognition
  - handwritten digit classification
  - recurrent neural network
  - word embedding
  - natural language processing
  - video detection, and many more.
- TensorFlow is run on multiple CPUs or GPUs and also mobile operating systems.

TensorFlow Characteristics

Flexible and scalable

Multi-language

GPU

Cross-platform

Powerful computation

Distributed

Copyright Huawei Technologies Co., Ltd

What Can We Do Using TensorFlow (1)

- Self-driving cars
- Music creation
- Image recognition
- Speech recognition
- Language models
- Human activity recognition
- Automated theorem proving
- Gaming (for example, play MarioKart racing games), etc.

Copyright Huawei Technologies Co., Ltd

What Can We Do Using TensorFlow (2)

Artistic style transfer

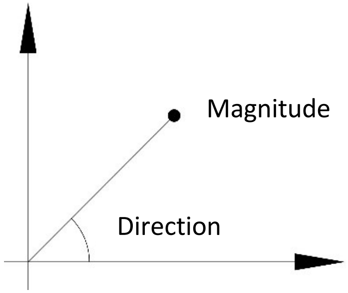
Facial recognition

Copyright Huawei Technologies Co., Ltd

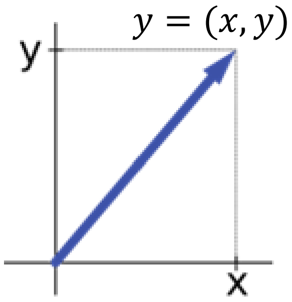
Basics of TensorFlow

- The word TensorFlow is made by two words:
  - Tensor: a multi-dimensional array
    - Tensor is the primary data structure in TensorFlow programs. Tensors are N-dimensional (where N could be 1, 2, 3, 4, or very large) data structures.
  - Flow: the flow of data in operation.
- In a running graph, tensors are the data that flows between nodes.

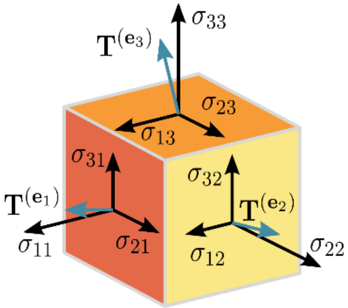
Basics of TensorFlow



Scalar



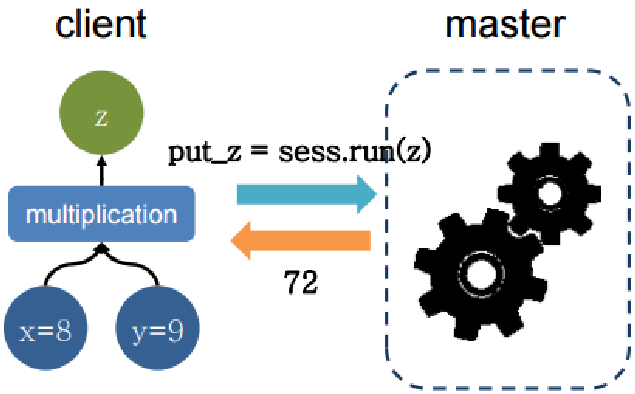
Vector



Tensor

Tensor

+



Data flow graph

Flow

Copyright Huawei Technologies Co., Ltd

Tensor

- In TensorFlow, tensors are classified into constant tensors and variable tensors.
  - **A defined constant tensor** has an unchangeable value and dimension, and a defined variable tensor has a changeable value and an unchangeable dimension.
  - In neural networks, **variable tensors** are generally used as matrices for storing weights and other information, and are a type of trainable data.
    - Constant tensors can be used for storing hyperparameters or other structured data.

Constant Tensor Creation

- Constant Tensor: Common methods for creating a constant tensor include.
  - **tf.constant():** creates a constant tensor.
  - **tf.zeros(), tf.zeros_like(), tf.ones(), and tf.ones_like():** create an all-zero or all-one constant tensor.
  - **tf.fill():** creates a tensor with a user-defined value.
  - **tf.random:** creates a tensor with a known distribution.
  - Creating a list object by using NumPy, and then converting the list object into a tensor by using **tf.convert_to_tensor.**

Constant Tensor Creation

```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[1., 2.],
      [3., 4.]], dtype=float32)>
```

- tf.constant(value, dtype=None, shape=None, name='Const'):
  - value: A constant value (or list) of output type dtype.
  - dtype: The type of the elements of the resulting tensor.
  - shape: Optional dimensions of resulting tensor.
  - name: Optional name for the tensor.

```
import tensorflow as tf

const_a = tf.constant([[1, 2, 3, 4]],shape=[2,2], dtype=tf.float32)

# Create a 2x2 matrix with values 1, 2, 3, and 4.

Print(const_a)
```

Constant Tensor Creation

- View common attributes

```
print("value of the constant const_a:", const_a.numpy())

print("data type of the constant const_a:", const_a.dtype)

print("shape of the constant const_a:", const_a.shape)

print("name of the device that is to generate the constant const_a:", const_a.device)
```

```
Value of the constant const_a: [[1. 2.]
 [3. 4.]]
Data type of the constant const_a: <dtype: 'float32'>
Shape of the constant const_a: (2, 2)
Name of the device that is to generate the constant const_a:
/job:localhost/replica:0/task:0/device:CPU:0
```

Constant Tensor Creation

- tf.zeros(), tf.zeros_like(), tf.ones(), and tf.ones_like():
  - tf.zeros(): Create a constant with the value 0.
  - tf.zeros(shape, dtype=tf.float32, name=None):
    - shape: A list of integers, a tuple of integers, or a 1-D Tensor of type int32
    - dtype: The DType of an element in the resulting Tensor.
    - name: Optional string. A name for the operation.

```
zeros_b = tf.zeros(shape=[2, 3], dtype=tf.int32)
# Create a 2x3 matrix with all values being 0.
```

Constant Tensor Creation

- Create a tensor whose value is 0 based on the input tensor, with its shape being the same as that of the input tensor.
  - tf.zeros_like(input, dtype=None, name=None):
  - input_tensor: A Tensor or array-like object.
  - dtype: A type for the returned Tensor. Must be float16, float32, float64, int8, uint8, int16, uint16, int32, int64, complex64, complex128, bool or string (optional).
  - name: A name for the operation (optional).

```
zeros_like_c = tf.zeros_like(const_a)
#View generated data.
print( zeros_like_c.numpy() )
```

```
Output:
-----------
array([[0., 0.],
       [0., 0.]], dtype=float32)
```

Constant Tensor Creation

- tf.fill(): Create a tensor and fill it with a scalar value.
  - tf.fill(dims, value, name=None):
  - dims: A 1-D sequence of non-negative numbers. Represents the shape of the output tf.Tensor. Entries should be of type: int32, int64.
  - value: A value to fill the returned tf.Tensor.
  - name: Optional string. The name of the output tf.Tensor.

```
fill_d = tf.fill([3,3], 8)
# Create a 3x3 matrix with all values being 8.
#View data.
print( fill_d.numpy() )
```

```
Output:
------------
array([[8, 8, 8],
[8, 8, 8],
      [8, 8, 8]])
```

Constant Tensor Creation

- **tf.random():** This module is used to generate a tensor with a specific distribution.
  - Common methods in this module include:
    - tf.random.uniform(), tf.random.normal(), and tf.random.shuffle(). The following describes how to use tf.random.normal().
  - Create a tensor that conforms to a normal distribution.

```
random_e = tf.random.normal([5,5],mean=0,stddev=1.0, seed = 1)
#View the created data.
random_e.numpy()
```

```
Output:
-------------
array([[-0.8521641, 2.0672443, -0.94127315, 1.7840577, 2.9919195 ],
       [-0.8644102, 0.41812655, -0.85865736, 1.0617154, 1.0575105],
       [ 0.22457163, -0.02204755, 0.5084496, -0.09113179, -1.3036906 ],
       [-1.1108295, -0.24195422, 2.8516252, -0.7503834, 0.1267275 ],
       [0.9460202, 0.12648873, -2.6540542, 0.0853276, 0.01731399]],
      dtype=float32)
```

Constant Tensor Creation

- tf.random()

tf.random.normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)

Parameter	Description
shape	A 1-D integer Tensor or Python array. The shape of the output tensor.
mean	A Tensor or Python value of type dtype, broadcastable with stddev. The mean of the normal distribution.
stddev	A Tensor or Python value of type dtype, broadcastable with mean. The standard deviation of the normal distribution.
dtype	The type of the output.
Seed	A Python integer. Used to create a random seed for the distribution. See tf.random.set_seed for behavior
Name	A name for the operation (optional).

Constant Tensor Creation

- Create a list object by using NumPy, and then convert the list object into a tensor by using tf.convert_to_tensor.
- **tf.convert_to_tensor** can be used to convert a Python data type into a tensor data type available to TensorFlow
- tf.convert_to_tensor(value, dtype=None, dtype_hint=None, name=None)

Parameter	Description
value	An object whose type has a registered Tensor conversion function.
dtype	Optional element type for the returned tensor. If missing, the type is inferred from the type of value.
dtype_hint	Optional element type for the returned tensor, used when dtype is None. In some cases, a caller may not have a dtype in mind when converting to a tensor, so dtype_hint can be used as a soft preference. If the conversion to dtype_hint is not possible, this argument has no effect.
name	Optional name to use if a new Tensor is created.

Constant Tensor Creation

- tf.convert_to_tensor()

```
#Create a list.
list_f = [1,2,3,4,5,6]
#View the data type.
print( type(list_f) )

tensor_f = tf.convert_to_tensor(list_f, dtype=tf.float32)
print( tensor_f )
```

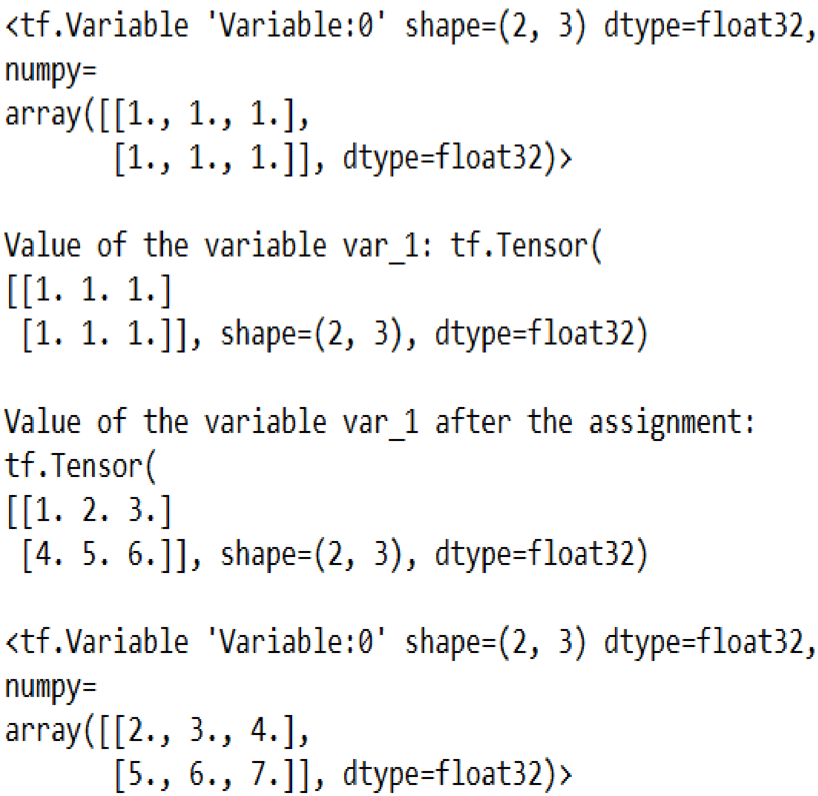
```
Output:
-----------
list

<tf.Tensor: shape=(6,), dtype=float32,
numpy=array([1., 2., 3., 4., 5., 6.],
dtype=float32)>
```

Variable Tensor Creation

- In TensorFlow, variables are operated using the **tf.Variable** class.
- tf.Variable indicates a tensor.
- The value of **tf.Variable can be changed by running an arithmetic operation** on tf.Variable.
- Variable values can be read and changed.

Variable Tensor Creation

```python
import tensorflow as tf
#Create a variable.
#Only the initial value needs to be provided
var_1 = tf.Variable(tf.ones([2,3]))
print( var_1 ,'\n')

#Read the variable value.
print("Value of the variable var_1:",
      var_1.read_value(),'\n')

#Assign a variable value.
var_value_1=[[1,2,3],[4,5,6]]

var_1.assign(var_value_1)
print("Value of the variable var_1 after the assignment:",
      var_1.read_value())

#Variable addition
var_1.assign_add(tf.ones([2,3]))
print(var_1)
```

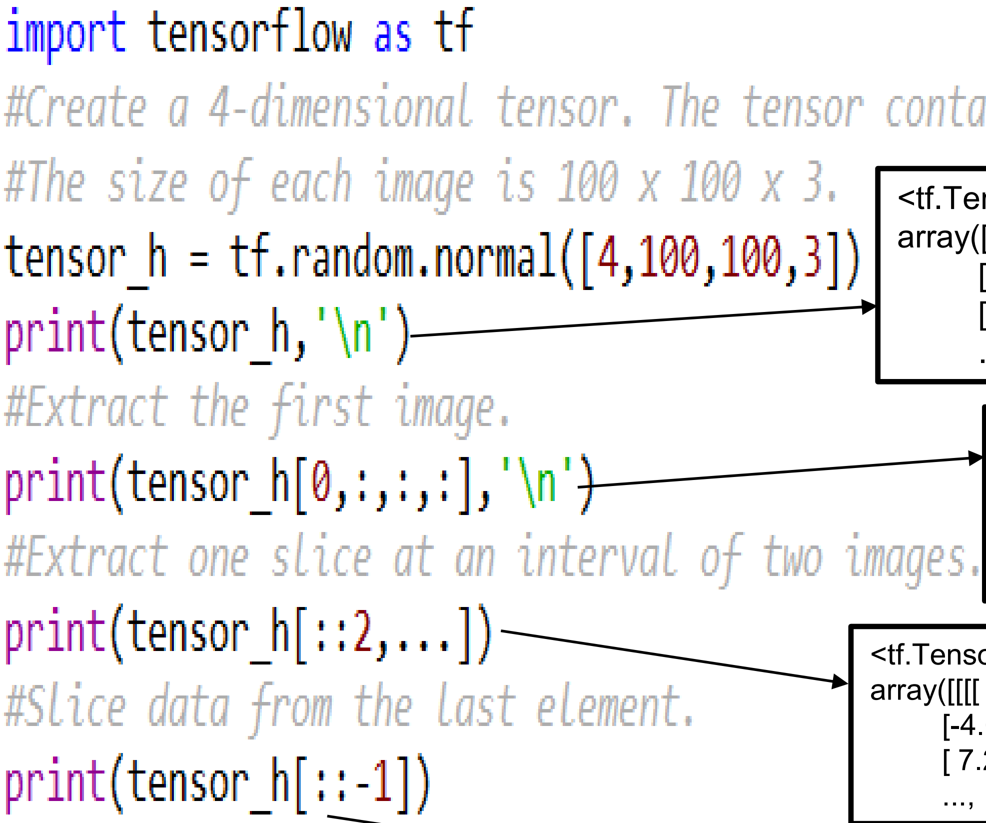


```
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32,
numpy=
array([[1., 1., 1.],
       [1., 1., 1.]], dtype=float32)>

Value of the variable var_1: tf.Tensor(
[[1. 1. 1.]
 [1. 1. 1.]], shape=(2, 3), dtype=float32)

Value of the variable var_1 after the assignment:
tf.Tensor(
[[1. 2. 3.]
 [4. 5. 6.]], shape=(2, 3), dtype=float32)

<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32,
numpy=
array([[2., 3., 4.],
       [5., 6., 7.]], dtype=float32)>
```

Tensor Slicing and Indexing

- **Slicing:** Tensor slicing methods include:
  - **[start: end]:** extracts a data slice from the start position to the end position of the tensor.
  - **[start:end:step] or [::step]:** extracts a data slice at an interval of step from the start position to the end position of the tensor.
  - **[::-1]:** slices data from the last element.
  - **'...':** indicates a data slice of any length.

Tensor Slicing and Indexing



```python
import tensorflow as tf
#Create a 4-dimensional tensor. The tensor contains four images.
#The size of each image is 100 x 100 x 3.
tensor_h = tf.random.normal([4,100,100,3])
print(tensor_h,'\n')
#Extract the first image.
print(tensor_h[0,:,:,:],'\n')
#Extract one slice at an interval of two images.
print(tensor_h[::2,...])
#Slice data from the last element.
print(tensor_h[::-1])
```

```
<tf.Tensor: shape=(4, 100, 100, 3), dtype=float32, numpy=
array([[[[ 1.68444023e-01, -7.46562362e-01, -4.34964240e-01],
         [-4.69263226e-01, 6.26460612e-01, 1.21065331e+00],
         [ 7.21675277e-01, 4.61057723e-01, -9.20868576e-01],
         ...,
```

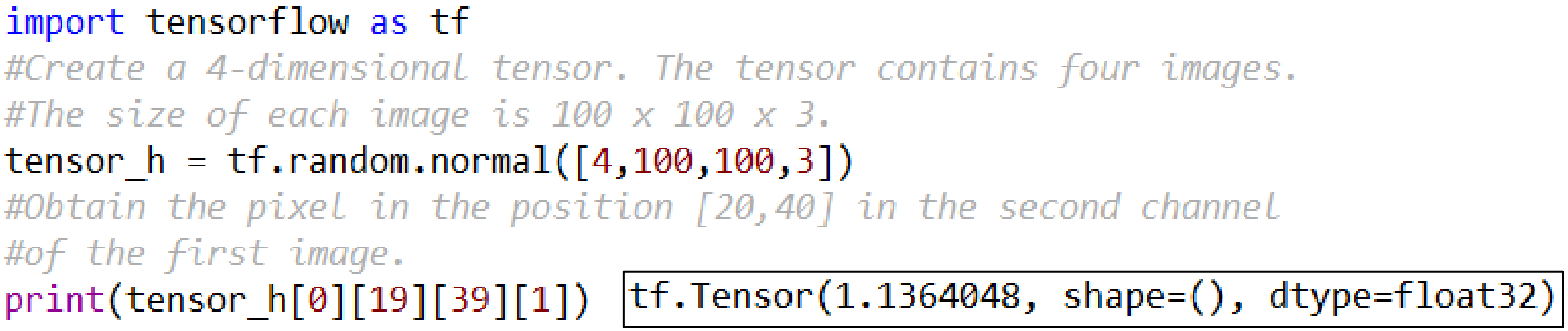
```
<tf.Tensor: shape=(100, 100, 3), dtype=float32, numpy=
array([[[ 1.68444023e-01, -7.46562362e-01, -4.34964240e-01],
        [-4.69263226e-01, 6.26460612e-01, 1.21065331e+00],
        [ 7.21675277e-01, 4.61057723e-01, -9.20868576e-01],
        ...,
```

```
<tf.Tensor: shape=(2, 100, 100, 3), dtype=float32, numpy=
array([[[[ 1.68444023e-01, -7.46562362e-01, -4.34964240e-01],
         [-4.69263226e-01, 6.26460612e-01, 1.21065331e+00],
         [ 7.21675277e-01, 4.61057723e-01, -9.20868576e-01],
         ...,
```

```
<tf.Tensor: shape=(4, 100, 100, 3), dtype=float32, numpy=
array([[[[-1.70684665e-01, 1.52386248e+00, -1.91677585e-01],
         [-1.78917408e+00, -7.48436213e-01, 6.10363662e-01],
         [ 7.64770031e-01, 6.06725179e-02, 1.32704067e+00],
         ...,
```

Tensor Slicing and Indexing

- **Indexing**: The basic format of an index is a[d1][d2][d3].



```
import tensorflow as tf
#Create a 4-dimensional tensor. The tensor contains four images.
#The size of each image is 100 x 100 x 3.
tensor_h = tf.random.normal([4,100,100,3])
#Obtain the pixel in the position [20,40] in the second channel
#of the first image.
print(tensor_h[0][19][39][1])
```

```
tf.Tensor(1.1364048, shape=(), dtype=float32)
```

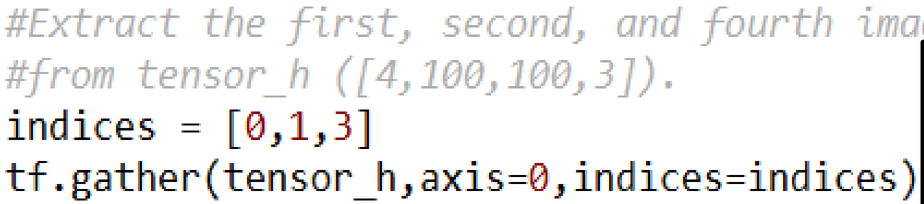
- If the indexes of data to be extracted are nonconsecutive, **tf.gather** and **tf.gather_nd** are commonly used for data extraction in TensorFlow.
- To extract data from a particular dimension:
- tf.gather(params, indices, axis=None)

params: input tensor

indices: index of the data to be extracted

axis: dimension of the data to be extracted

Tensor Slicing and Indexing

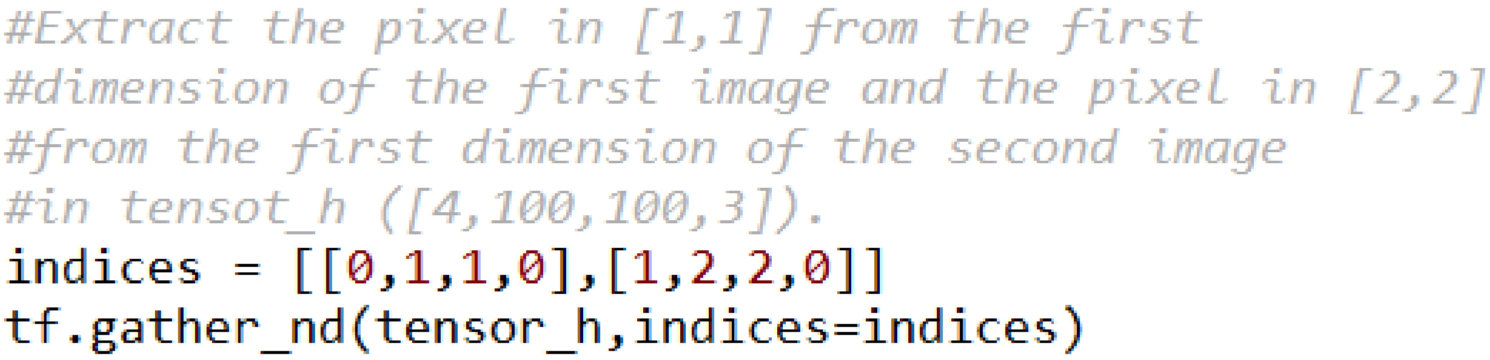


```
#Extract the first, second, and fourth images
#from tensor_h ([4,100,100,3]).
indices = [0,1,3]
tf.gather(tensor_h,axis=0,indices=indices)
```

```
Output:
-----------
<tf.Tensor: shape=(3, 100, 100, 3), dtype=float32, numpy=
array([[[[ 1.68444023e-01, -7.46562362e-01, -4.34964240e-01],
         [-4.69263226e-01, 6.26460612e-01, 1.21065331e+00],
         [ 7.21675277e-01, 4.61057723e-01, -9.20868576e-01],
         ...,
```

- **tf.gather_nd** allows data extraction from multiple dimensions:
- tf.gather_nd(params, indices, batch_dims=0, name=None):
    - params: The tensor from which to gather values.
    - indices: Must be one of the following types: int32, int64. Index tensor.
    - Name: A name for the operation (optional).
    - batch_dims: An integer or a scalar 'Tensor'. The number of batch dimensions.

Tensor Slicing and Indexing



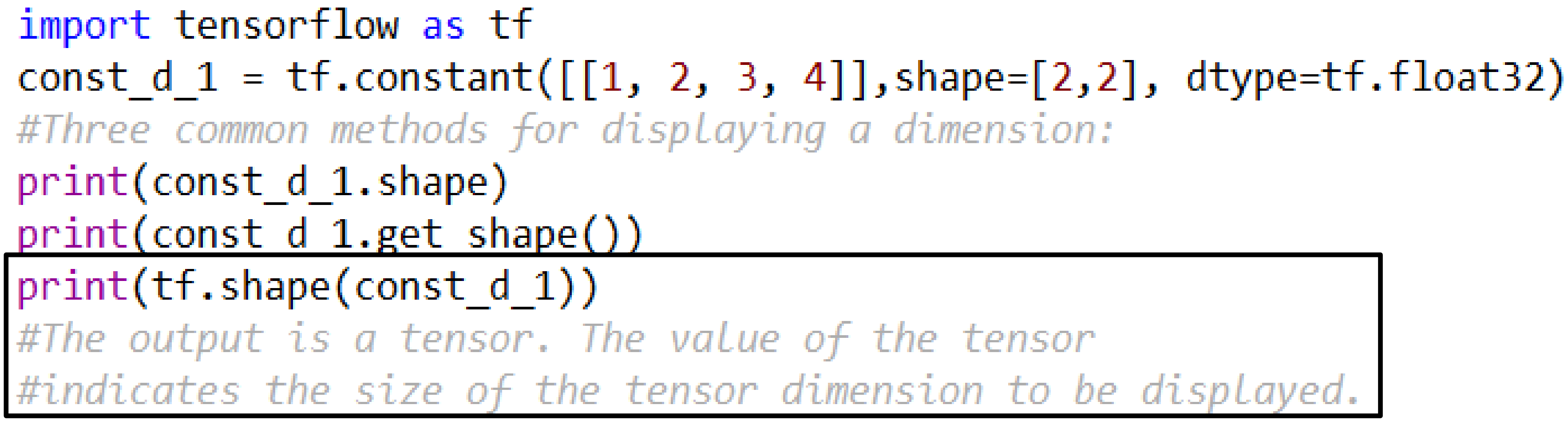
```
#Extract the pixel in [1,1] from the first
#dimension of the first image and the pixel in [2,2]
#from the first dimension of the second image
#in tensot_h ([4,100,100,3]).
indices = [[0,1,1,0],[1,2,2,0]]
tf.gather_nd(tensor_h,indices=indices)
```

```
Output:
----------
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([0.5705869, 0.9735735], dtype=float32)>
```

Tensor Dimension Modification

```
Output:
-----------
(2, 2)
(2, 2)
tf.Tensor([2 2], shape=(2,), dtype=int32)
```

- Dimension display:

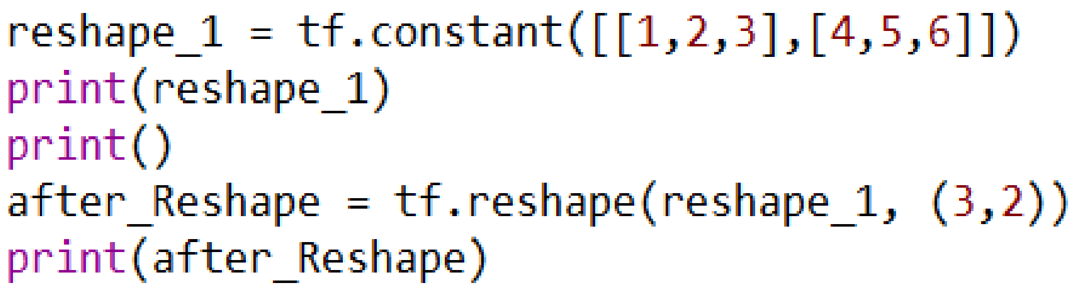


```python
import tensorflow as tf
const_d_1 = tf.constant([[1, 2, 3, 4]],shape=[2,2], dtype=tf.float32)
#Three common methods for displaying a dimension:
print(const_d_1.shape)
print(const_d_1.get_shape())
print(tf.shape(const_d_1))
#The output is a tensor. The value of the tensor
#indicates the size of the tensor dimension to be displayed.
```

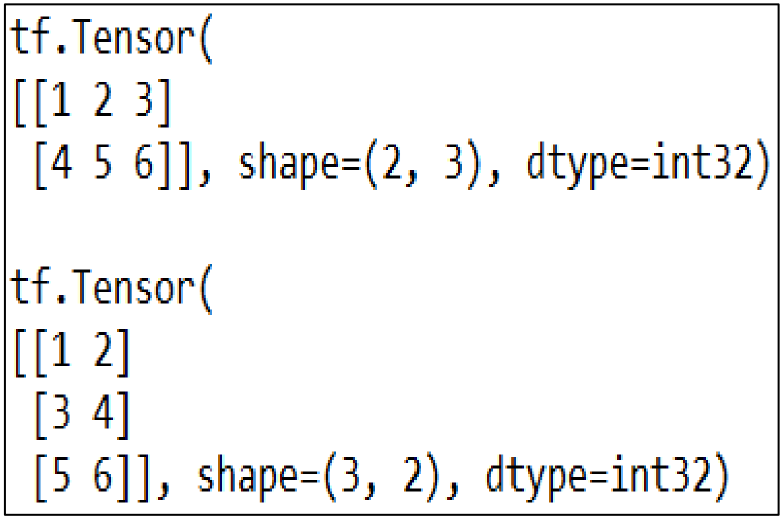
- **.shape** and **.get_shape()** return TensorShape objects,
- **tf.shape(x)** returns Tensor objects

Tensor Dimension Modification

- Dimension Reshaping:
- tf.reshape(tensor,shape,name=None):
  - tensor: input tensor
  - shape: dimension of the reshaped tensor



```
reshape_1 = tf.constant([[1,2,3],[4,5,6]])
print(reshape_1)
print()
after_Reshape = tf.reshape(reshape_1, (3,2))
print(after_Reshape)
```

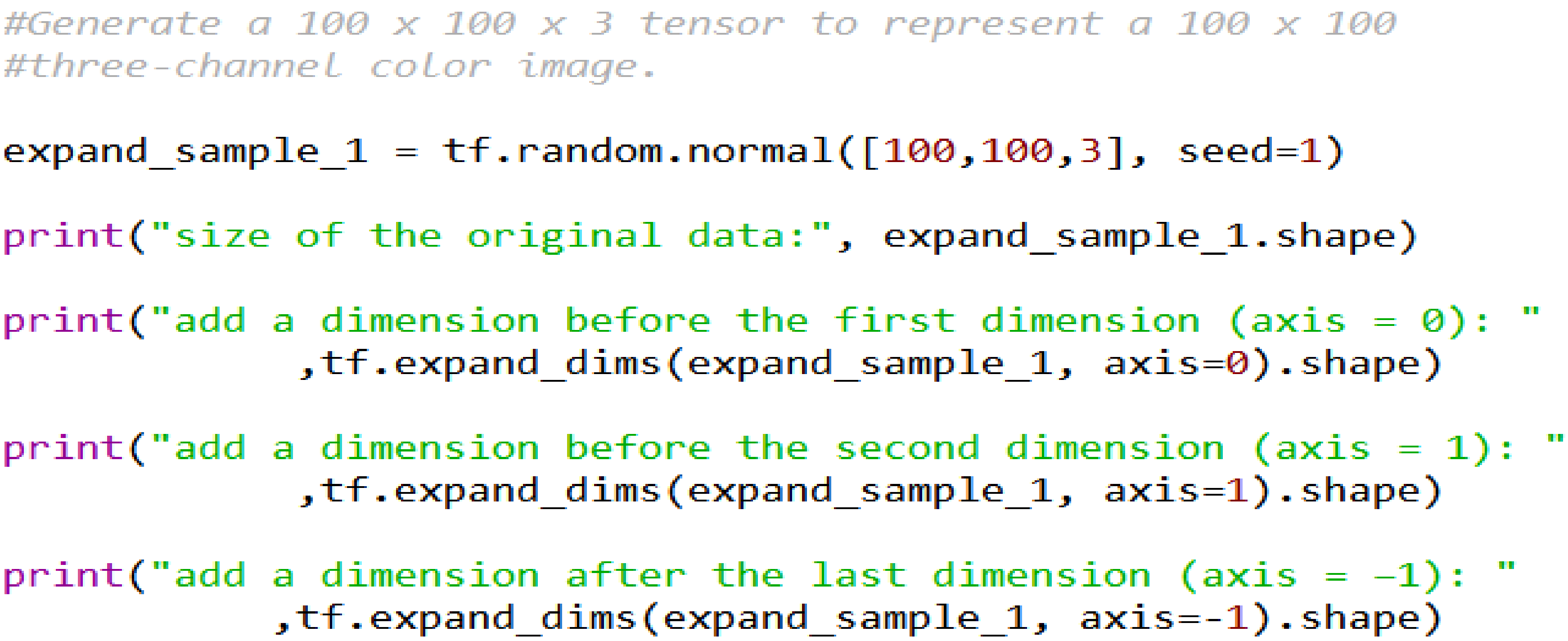


```
tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)

tf.Tensor(
[[1 2]
 [3 4]
 [5 6]], shape=(3, 2), dtype=int32)
```

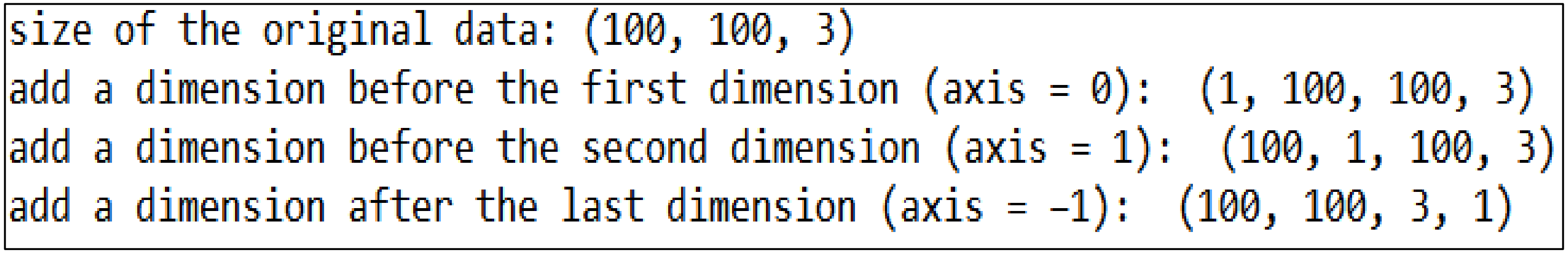
Tensor Dimension Modification

- **Dimension Expansion**
    - tf.expand_dims(input, axis, name=None):
        - input: input tensor
        - axis: adds a dimension after the axis dimension.
- When the number of dimensions of the input data is D, the axis must fall in the range of [–(D + 1), D] (included). A negative value indicates adding a dimension in reverse order.

Tensor Dimension Modification



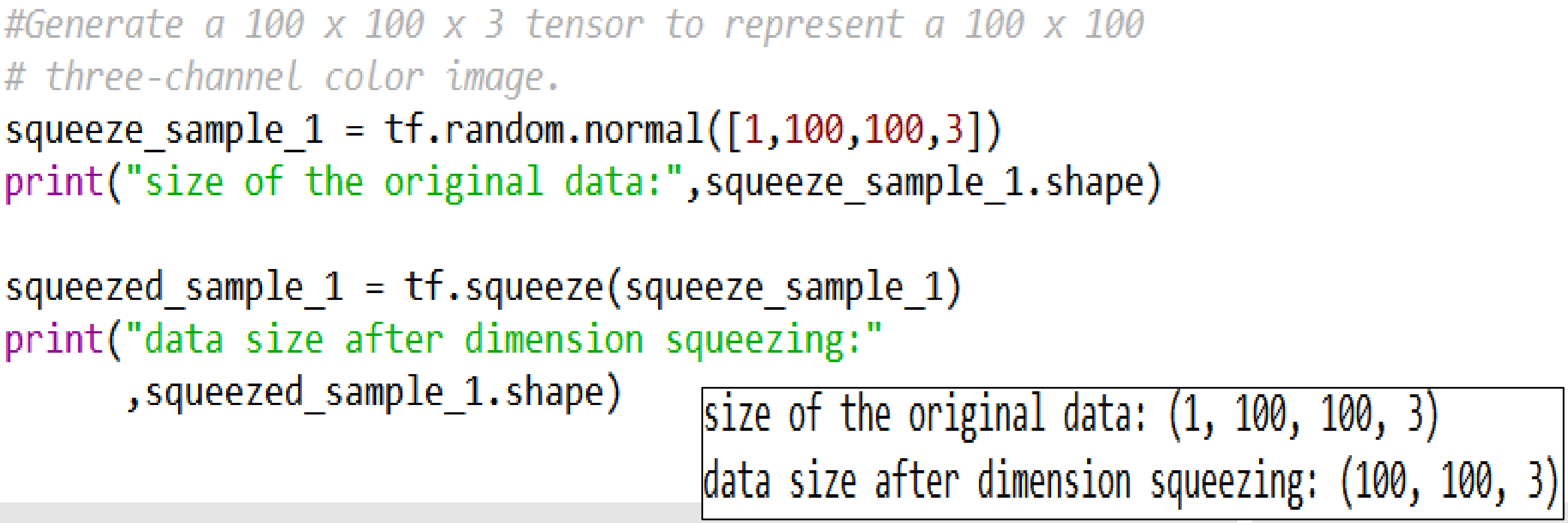
```
#Generate a 100 x 100 x 3 tensor to represent a 100 x 100
#three-channel color image.

expand_sample_1 = tf.random.normal([100,100,3], seed=1)

print("size of the original data:", expand_sample_1.shape)

print("add a dimension before the first dimension (axis = 0): "
            ,tf.expand_dims(expand_sample_1, axis=0).shape)

print("add a dimension before the second dimension (axis = 1): "
            ,tf.expand_dims(expand_sample_1, axis=1).shape)

print("add a dimension after the last dimension (axis = -1): "
           ,tf.expand_dims(expand_sample_1, axis=-1).shape)
```



```
size of the original data: (100, 100, 3)
add a dimension before the first dimension (axis = 0):  (1, 100, 100, 3)
add a dimension before the second dimension (axis = 1):  (100, 1, 100, 3)
add a dimension after the last dimension (axis = -1):  (100, 100, 3, 1)
```

Tensor Dimension Modification

- Dimension Squeeze: Remove dimension of size 1 from shape of tensor.
  - tf.squeeze(input, axis=None, name=None):
    - input: input tensor
    - axis: If axis is set to 1, dimension 1 needs to be deleted.

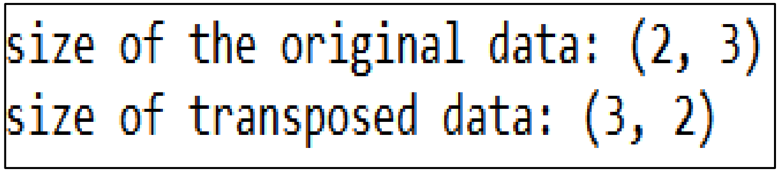


```
#Generate a 100 x 100 x 3 tensor to represent a 100 x 100
# three-channel color image.
squeeze_sample_1 = tf.random.normal([1,100,100,3])
print("size of the original data:",squeeze_sample_1.shape)

squeezed_sample_1 = tf.squeeze(squeeze_sample_1)
print("data size after dimension squeezing:"
      ,squeezed_sample_1.shape)
```

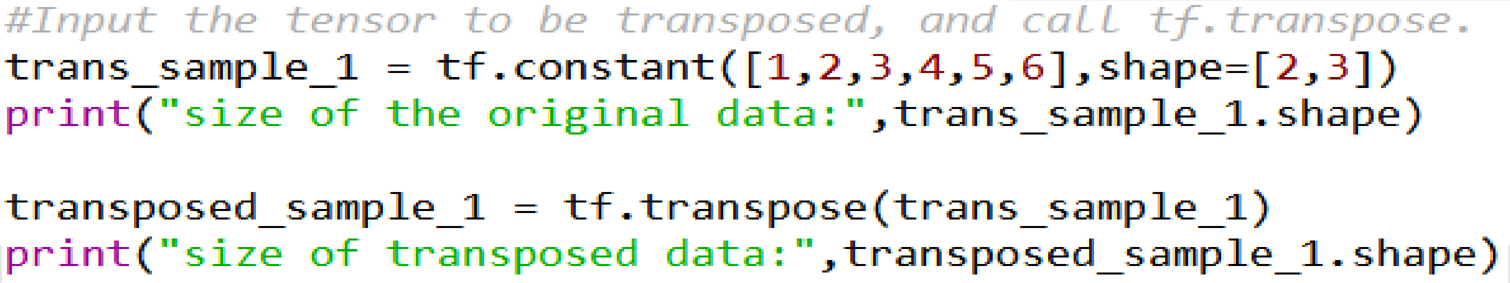
```
size of the original data: (1, 100, 100, 3)
data size after dimension squeezing: (100, 100, 3)
```

Tensor Dimension Modification

- Transpose:
  - tf.transpose(a, perm=None, conjugate=False, name='transpose'):
    - a: input tensor
    - perm: tensor size sequence, generally used to transpose high-dimensional arrays
    - conjugate: indicates complex number transpose.
    - name: tensor name



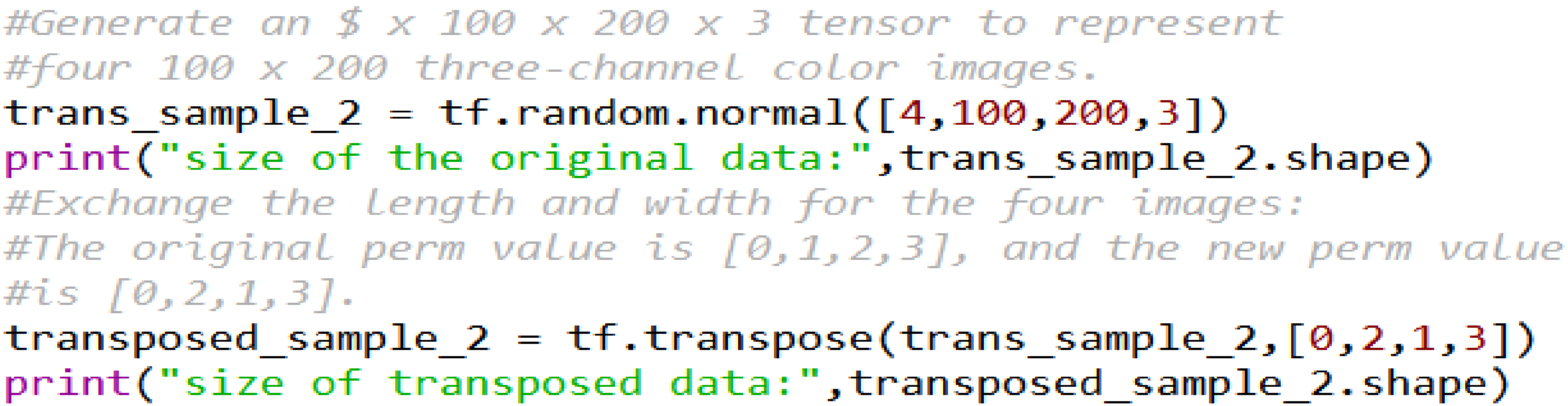
```
size of the original data: (2, 3)
size of transposed data: (3, 2)
```



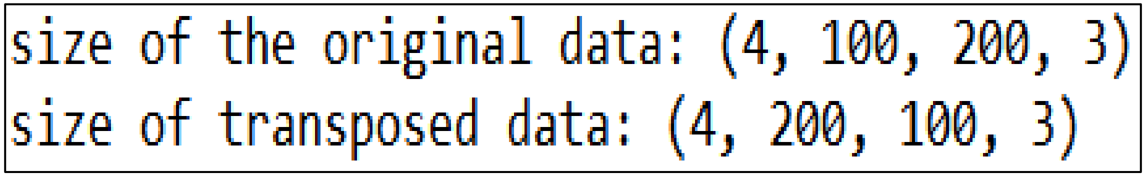
```
#Input the tensor to be transposed, and call tf.transpose.
trans_sample_1 = tf.constant([1,2,3,4,5,6],shape=[2,3])
print("size of the original data:",trans_sample_1.shape)

transposed_sample_1 = tf.transpose(trans_sample_1)
print("size of transposed data:",transposed_sample_1.shape)
```

Tensor Dimension Modification

- Data dimensions can be transposed by changing the sequence of values in perm.



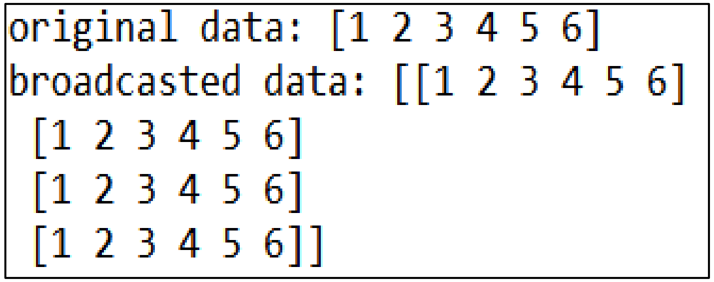
```
#Generate an $ x 100 x 200 x 3 tensor to represent
#four 100 x 200 three-channel color images.
trans_sample_2 = tf.random.normal([4,100,200,3])
print("size of the original data:",trans_sample_2.shape)
#Exchange the length and width for the four images:
#The original perm value is [0,1,2,3], and the new perm value
#is [0,2,1,3].
transposed_sample_2 = tf.transpose(trans_sample_2,[0,2,1,3])
print("size of transposed data:",transposed_sample_2.shape)
```



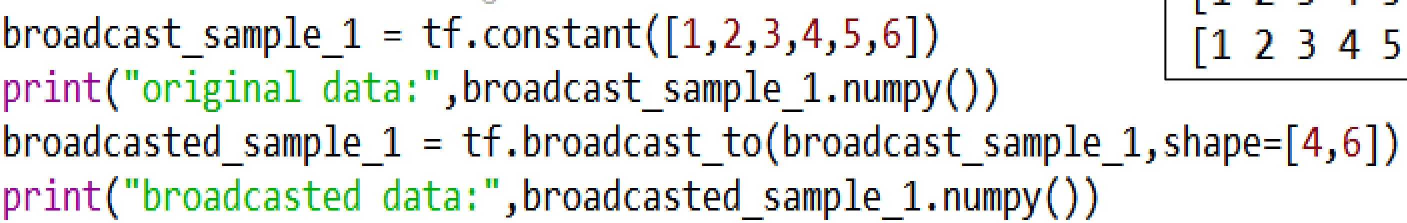
```
size of the original data: (4, 100, 200, 3)
size of transposed data: (4, 200, 100, 3)
```

Tensor Dimension Modification

- Broadcast (broadcast_to)
  - broadcast_to is used to broadcast data from a low dimension to a high dimension.
  - tf.broadcast_to(input, shape, name=None):
    - input: input tensor
    - shape: size of the output tensor

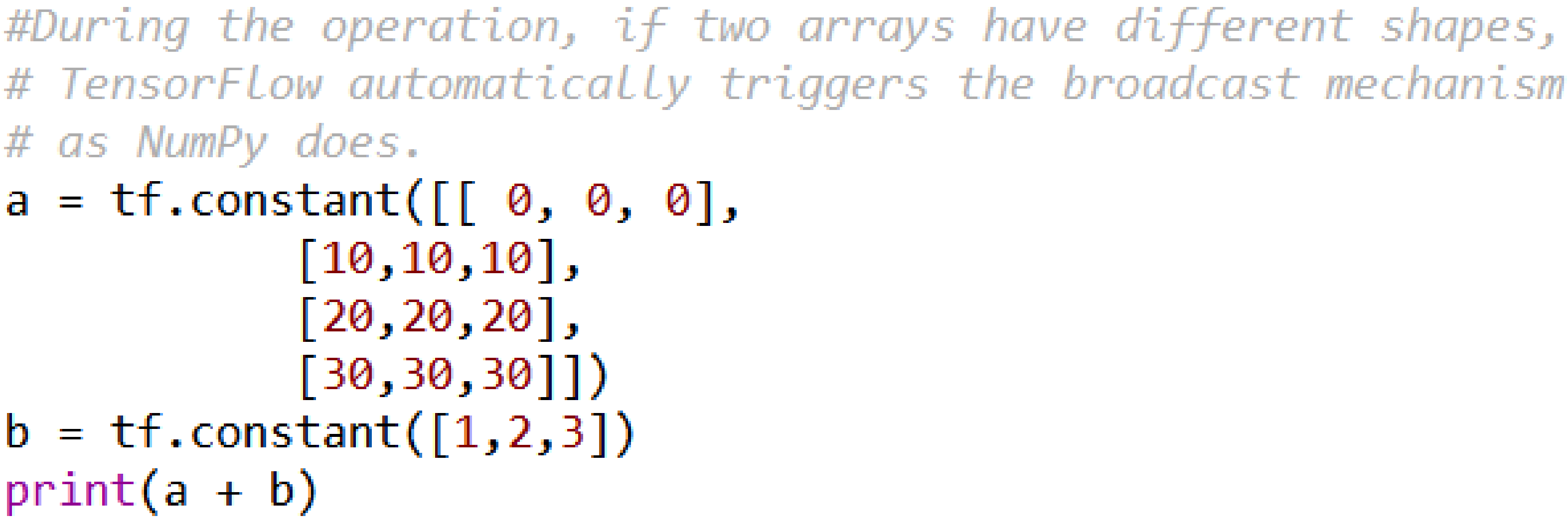


```
original data: [1 2 3 4 5 6]
broadcasted data: [[1 2 3 4 5 6]
 [1 2 3 4 5 6]
 [1 2 3 4 5 6]
 [1 2 3 4 5 6]]
```

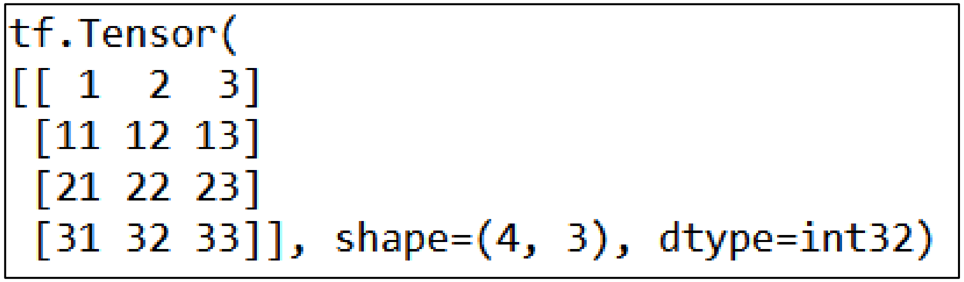


```
broadcast_sample_1 = tf.constant([1,2,3,4,5,6])
print("original data:",broadcast_sample_1.numpy())
broadcasted_sample_1 = tf.broadcast_to(broadcast_sample_1,shape=[4,6])
print("broadcasted data:",broadcasted_sample_1.numpy())
```

Tensor Dimension Modification



```
#During the operation, if two arrays have different shapes,
# TensorFlow automatically triggers the broadcast mechanism
# as NumPy does.
a = tf.constant([[ 0, 0, 0],
                 [10,10,10],
                 [20,20,20],
                 [30,30,30]])
b = tf.constant([1,2,3])
print(a + b)
```



```
tf.Tensor(
[[ 1  2  3]
 [11 12 13]
 [21 22 23]
 [31 32 33]], shape=(4, 3), dtype=int32)
```

Arithmetic Operations on Tensors

- Main arithmetic operations include:
  - addition (tf.add)
  - subtraction (tf.subtract)
  - multiplication (tf.multiply)
  - division (tf.divide)
  - logarithm (tf.math.log)
  - powers (tf.pow)

Arithmetic Operations on Tensors

- Add operation

```
a = tf.constant([[3, 5], [4, 8]])
b = tf.constant([[1, 6], [2, 9]])
print(tf.add(a, b))
```

```
Output:
-----------
tf.Tensor(
[[ 4 11]
 [ 6 17]], shape=(2, 2), dtype=int32)
```

- Matrix Multiplication

```
tf.matmul(a,b)
```

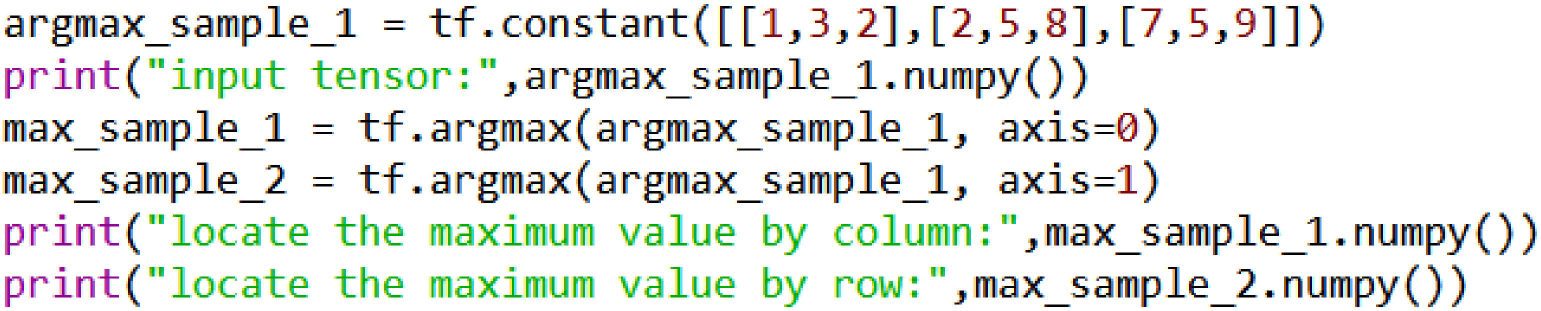
```
Output:
------------
tf.Tensor(
[[13 63]
 [20 96]], shape=(2, 2), dtype=int32)
```

Tensor Statistics Collection

- Methods for collecting tensor statistics include:
  - tf.reduce_min/max/mean(): calculates the minimum, maximum, and mean values.
  - tf.argmax()/tf.argmin(): calculates the positions of the maximum and minimum values.
  - tf.equal(): checks whether two tensors are equal by element.
  - tf.unique(): removes duplicate elements from tensors.
  - tf.nn.in_top_k(prediction, target, K): calculates whether the predicted value is equal to the actual value, and returns a Boolean tensor.

Tensor Statistics Collection

- tf.argmax(input,axis):
  - input: input tensor
  - axis: maximum output value in the axis dimension



```python
argmax_sample_1 = tf.constant([[1,3,2],[2,5,8],[7,5,9]])
print("input tensor:",argmax_sample_1.numpy())
max_sample_1 = tf.argmax(argmax_sample_1, axis=0)
max_sample_2 = tf.argmax(argmax_sample_1, axis=1)
print("locate the maximum value by column:",max_sample_1.numpy())
print("locate the maximum value by row:",max_sample_2.numpy())
```

```
Output:
--------------
input tensor: [[1 3 2]
 [2 5 8]
 [7 5 9]]
locate the maximum value by column: [2 1 2]
locate the maximum value by row: [1 2 2]
```

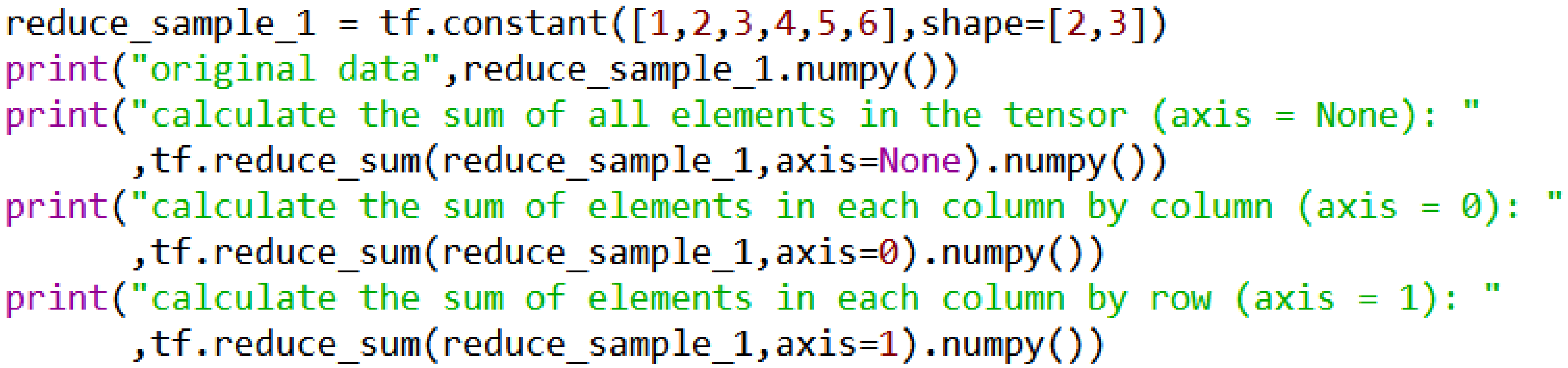
Dimension-based Arithmetic Operations

- In TensorFlow, a series of operations of tf.reduce_* reduce tensor dimensions.
- The series of operations can be performed on dimensional elements of a tensor, for example, calculating the mean value by row and calculating a product of all elements in the tensor.
- Common operations include:
  - tf.reduce_sum (addition),
    tf.reduce_min (minimum),
    tf.reduce_mean (mean value),
    tf.reduce_any (logical OR),
    tf.reduce_prod (multiplication),
    tf.reduce_max (maximum),
    tf.reduce_all (logical AND),
    tf.reduce_logsumexp (log(sum(exp)))

Tensor Statistics Collection

- Calculate the sum of elements in all dimensions of a tensor.
- tf.reduce_sum(input_tensor, axis=None, keepdims= False, name=None):
  - input_tensor: The tensor to reduce. Should have numeric type.
  - axis: The dimensions to reduce. If None (the default), reduces all dimensions. Must be in the range [-rank(input_tensor),rank(input_tensor)].
  - keepdims: If true, retains reduced dimensions with length 1.
  - name: A name for the operation (optional).

Tensor Statistics Collection



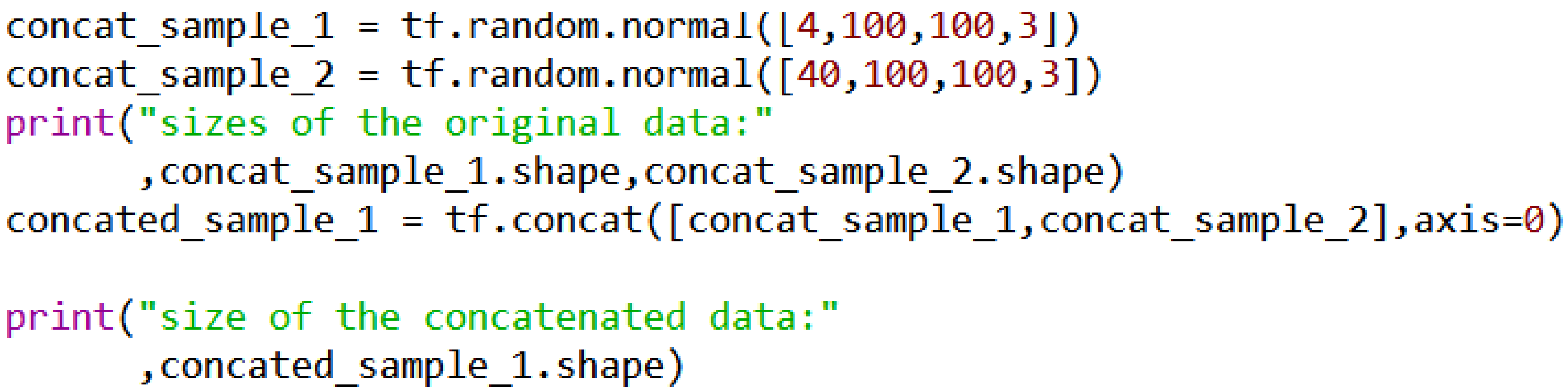
```
reduce_sample_1 = tf.constant([1,2,3,4,5,6],shape=[2,3])
print("original data",reduce_sample_1.numpy())
print("calculate the sum of all elements in the tensor (axis = None): "
      ,tf.reduce_sum(reduce_sample_1,axis=None).numpy())
print("calculate the sum of elements in each column by column (axis = 0): "
      ,tf.reduce_sum(reduce_sample_1,axis=0).numpy())
print("calculate the sum of elements in each column by row (axis = 1): "
      ,tf.reduce_sum(reduce_sample_1,axis=1).numpy())
```

```
Output:
----------
original data [[1 2 3]
 [4 5 6]]
calculate the sum of all elements in the tensor (axis = None):  21
calculate the sum of elements in each column by column (axis = 0):  [5 7 9]
calculate the sum of elements in each column by row (axis = 1):  [ 6 15]
```

Tensor Concatenation and Splitting

- Tensor Concatenation operations include:
  - tf.contact(): concatenates vectors based on the specified dimension, while keeping other dimensions unchanged.
  - tf.stack(): changes a group of R dimensional tensors to R+1 dimensional tensors, with the dimensions changed after the concatenation.
- tf.concat(values, axis, name='concat'):
  - values: input tensor
  - axis: dimension to concatenate
  - name: operation name

Tensor Concatenation

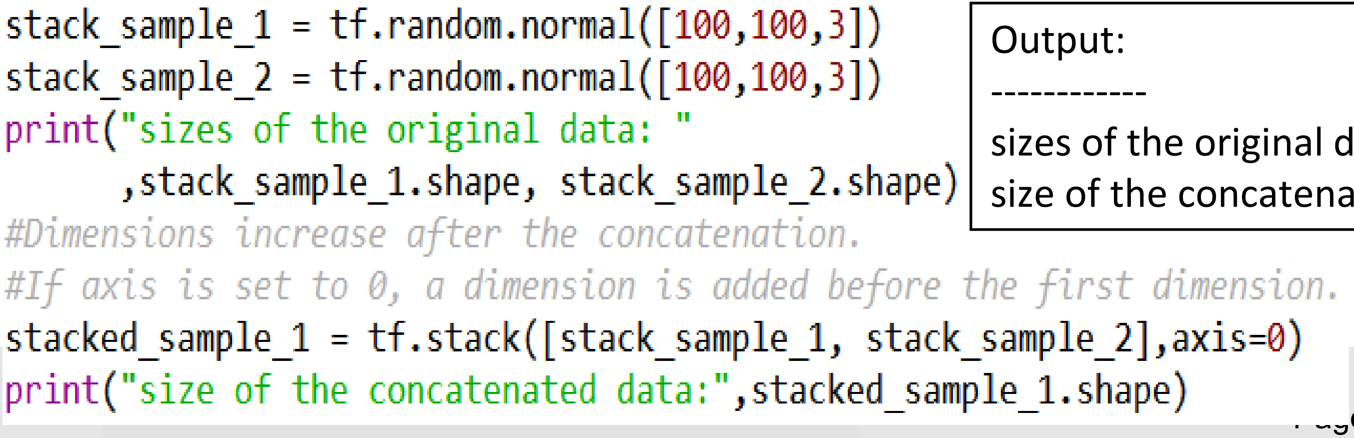


```
concat_sample_1 = tf.random.normal([4,100,100,3])
concat_sample_2 = tf.random.normal([40,100,100,3])
print("sizes of the original data:"
      ,concat_sample_1.shape,concat_sample_2.shape)
concated_sample_1 = tf.concat([concat_sample_1,concat_sample_2],axis=0)

print("size of the concatenated data:"
      ,concated_sample_1.shape)
```

```
Output:
------------
sizes of the original data: (4, 100, 100, 3) (40, 100, 100, 3)
size of the concatenated data: (44, 100, 100, 3)
```

Tensor Concatenation

- A dimension can be added to an original matrix in the same way. axis determines the position of the dimension.
- tf.stack(values, axis=0, name='stack'):
  - values: A list of Tensor objects with the same shape and type.
  - axis: The axis to stack along. Defaults to the first dimension. Negative values wrap around, so the valid range is [-(R+1), R+1).
  - name: A name for this operation (optional).



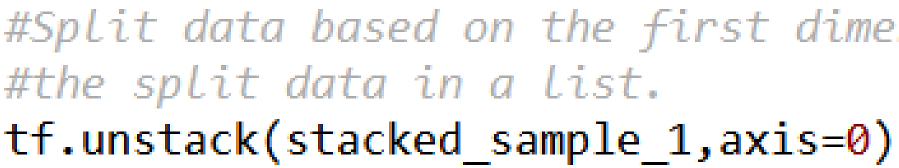
```
stack_sample_1 = tf.random.normal([100,100,3])
stack_sample_2 = tf.random.normal([100,100,3])
print("sizes of the original data: "
      ,stack_sample_1.shape, stack_sample_2.shape)
#Dimensions increase after the concatenation.
#If axis is set to 0, a dimension is added before the first dimension.
stacked_sample_1 = tf.stack([stack_sample_1, stack_sample_2],axis=0)
print("size of the concatenated data:",stacked_sample_1.shape)
```

```
Output:
------------
sizes of the original data:  (100, 100, 3) (100, 100, 3)
size of the concatenated data: (2, 100, 100, 3)
```

Tensor Splitting

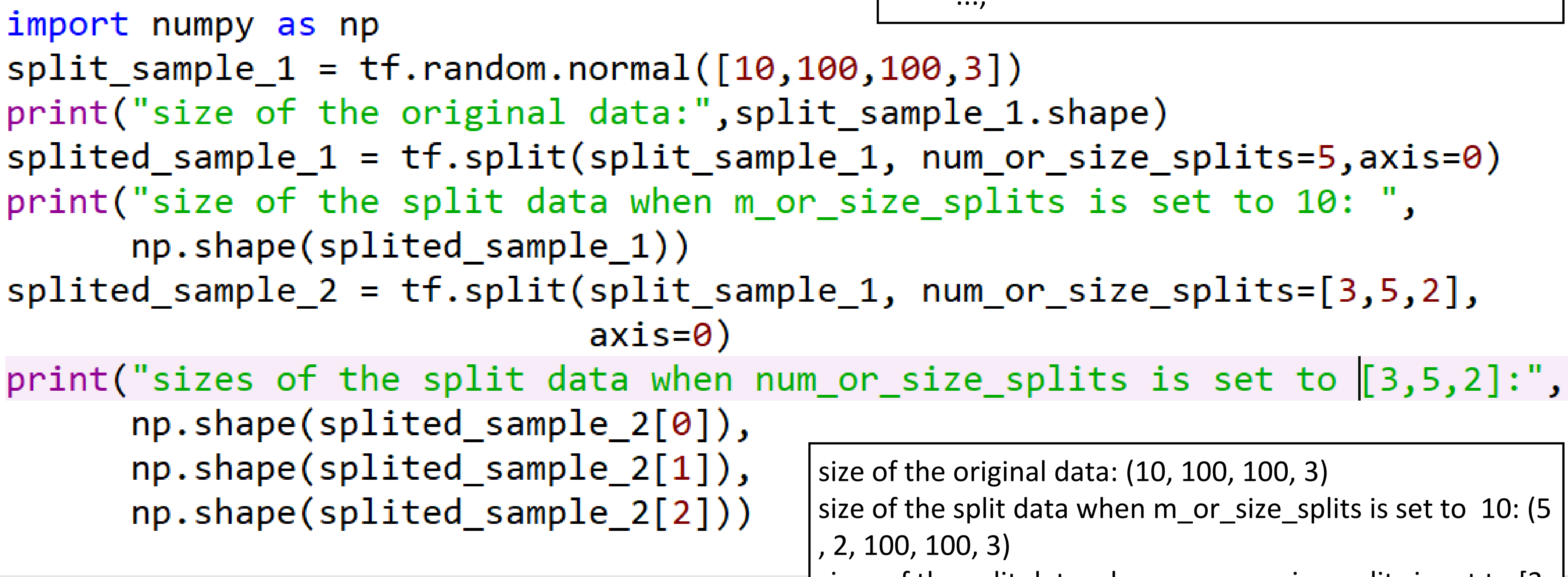
- tensor splitting operations include:
  - tf.unstack(): splits a tensor by a specific dimension.
  - tf.split(): splits a tensor into a specified number of sub tensors based on a specific dimension.
  - tf.split() is more flexible than tf.unstack().
- tf.unstack(value, num=None, axis=0,name='unstack'):
  - value: input tensor
  - num: indicates that a list containing num elements is output. The value of num must be the same as the number of elements in the specified dimension. This parameter can generally be ignored.
  - axis: specifies the dimension based on which the tensor is split.
  - name: operation name

Tensor Splitting



```
#Split data based on the first dimension and output
#the split data in a list.
tf.unstack(stacked_sample_1,axis=0)
```

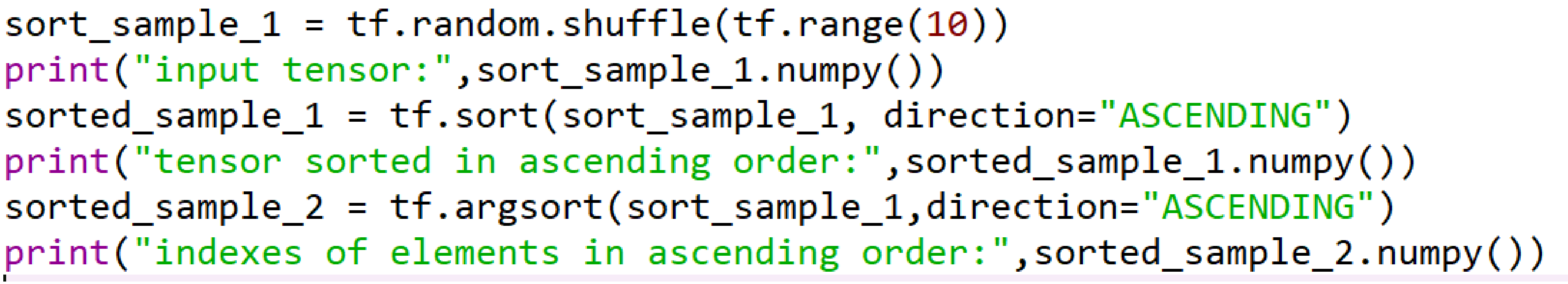
```
[<tf.Tensor: shape=(100, 100, 3), dtype=float32,
numpy=
 array([[[ 0.0665694 ,  0.7110351 ,  1.907618  ],
        [ 0.84416866,  1.5470593 , -0.5084871 ],
        [-1.9480026 , -0.9899087 , -0.09975405],
        ...,
```



```
import numpy as np
split_sample_1 = tf.random.normal([10,100,100,3])
print("size of the original data:",split_sample_1.shape)
splited_sample_1 = tf.split(split_sample_1, num_or_size_splits=5,axis=0)
print("size of the split data when m_or_size_splits is set to 10: ",
      np.shape(splited_sample_1))
splited_sample_2 = tf.split(split_sample_1, num_or_size_splits=[3,5,2],
                            axis=0)
print("sizes of the split data when num_or_size_splits is set to [3,5,2]:",
      np.shape(splited_sample_2[0]),
      np.shape(splited_sample_2[1]),
      np.shape(splited_sample_2[2]))
```

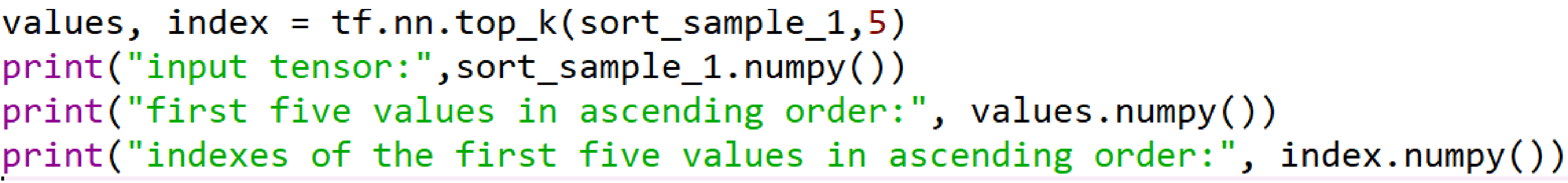
```
size of the original data: (10, 100, 100, 3)
size of the split data when m_or_size_splits is set to  10: (5
, 2, 100, 100, 3)
sizes of the split data when num_or_size_splits is set to [3,
5,2]: (3, 100, 100, 3) (5, 100, 100, 3) (2, 100, 100, 3)
```

Tensor Sorting



```python
sort_sample_1 = tf.random.shuffle(tf.range(10))
print("input tensor:",sort_sample_1.numpy())
sorted_sample_1 = tf.sort(sort_sample_1, direction="ASCENDING")
print("tensor sorted in ascending order:",sorted_sample_1.numpy())
sorted_sample_2 = tf.argsort(sort_sample_1,direction="ASCENDING")
print("indexes of elements in ascending order:",sorted_sample_2.numpy())
```

```
input tensor: [7 2 3 9 6 5 1 8 0 4]
tensor sorted in ascending order: [0 1 2 3 4 5 6 7 8 9]
indexes of elements in ascending order: [8 6 1 2 9 5 4 0 7 3]
```



```python
values, index = tf.nn.top_k(sort_sample_1,5)
print("input tensor:",sort_sample_1.numpy())
print("first five values in ascending order:", values.numpy())
print("indexes of the first five values in ascending order:", index.numpy())
```

```
input tensor: [7 2 3 9 6 5 1 8 0 4] first five values in
ascending order: [9 8 7 6 5] indexes of the first five
values in ascending order: [3 7 0 4 5]
```

Thanks

www.huawei.com