```
import torch
print(torch.__version__)
```

```
2.5.1+cu121
```

```
if torch.cuda.is_available():
    print("GPU is available!")
    print(f"Using GPU: {torch.cuda.get_device_name(0)}")
else:
    print("GPU not available. Using CPU.")
```

```
GPU is available!
Using GPU: Tesla T4
```

Creating a Tensor

```
# using empty
a = torch.empty(2,3)
```

```
# check type
type(a)
```

```
torch.Tensor
```

```
# using zeros
torch.zeros(2,3)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

```
# using ones
torch.ones(2,3)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

```
# using rand
torch.rand(2,3)
```

```
tensor([[0.1192, 0.4828, 0.9105],
        [0.4266, 0.4203, 0.2719]])
```

```
# use of seed
torch.rand(2,3)
```

```
tensor([[0.0064, 0.4451, 0.1580],
        [0.8119, 0.4629, 0.3227]])
```

```
# manual_seed
torch.manual_seed(100)
torch.rand(2,3)
```

```
tensor([[0.1117, 0.8158, 0.2626],
        [0.4839, 0.6765, 0.7539]])
```

```
torch.manual_seed(100)
torch.rand(2,3)
```

```
tensor([[0.1117, 0.8158, 0.2626],
        [0.4839, 0.6765, 0.7539]])
```

```
# using tensor
torch.tensor([[1,2,3],[4,5,6]])
```

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

```
# other ways

# arange
print("using arange ->", torch.arange(0,10,2))

# using linspace
print("using linspace ->", torch.linspace(0,10,10))

# using eye
print("using eye ->", torch.eye(5))

# using full
print("using full ->", torch.full((3, 3), 5))
```

```
using arange -> tensor([0, 2, 4, 6, 8])
using linspace -> tensor([ 0.0000,  1.1111,  2.2222,  3.3333,  4.4444,  5.5556,  6.6667,  7.7778,
         8.8889, 10.0000])
using eye -> tensor([[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 1.]])
using full -> tensor([[5, 5, 5],
        [5, 5, 5],
        [5, 5, 5]])
```

Tensor Shapes

```
x = torch.tensor([[1,2,3],[4,5,6]])
x
```

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

```
x.shape
```

```
torch.Size([2, 3])
```

```
torch.empty_like(x)
```

```
tensor([[     134842408414608,       96429401449888,     3503044046096754],
        [-2738188572259385344,  7020378893157883663,  8389754676633367105]])
```

```
torch.zeros_like(x)
```

```
tensor([[0, 0, 0],
        [0, 0, 0]])
```

```
torch.ones_like(x)
```

```
tensor([[1, 1, 1],
        [1, 1, 1]])
```

```
torch.rand_like(x, dtype=torch.float32)
```

```
tensor([[0.4076, 0.6037, 0.9820],
        [0.0117, 0.3891, 0.5056]])
```

Tensor Data Types

```
# find data type
x.dtype
```

```
torch.int64
```

```
# assign data type
torch.tensor([1.0,2.0,3.0], dtype=torch.int32)
```

```
tensor([1, 2, 3], dtype=torch.int32)
```

```
torch.tensor([1,2,3], dtype=torch.float64)
```

```
tensor([1., 2., 3.], dtype=torch.float64)
```

```
# using to()
x.to(torch.float32)
```

```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

Data Type	Dtype	Description
32-bit Floating Point	`torch.float32`	Standard floating-point type used for most deep learning tasks. Provides a balance between precision and memory usage.
64-bit Floating Point	`torch.float64`	Double-precision floating point. Useful for high-precision numerical tasks but uses more memory.
16-bit Floating Point	`torch.float16`	Half-precision floating point. Commonly used in mixed-precision training to reduce memory and computational overhead on modern GPUs.

Data Type	Dtype	Description
BFloat16	`torch.bfloat16`	Brain floating-point format with reduced precision compared to `float16`. Used in mixed-precision training, especially on TPUs.
8-bit Floating Point	`torch.float8`	Ultra-low-precision floating point. Used for experimental applications and extreme memory-constrained environments (less common).
8-bit Integer	`torch.int8`	8-bit signed integer. Used for quantized models to save memory and computation in inference.
16-bit Integer	`torch.int16`	16-bit signed integer. Useful for special numerical tasks requiring intermediate precision.
32-bit Integer	`torch.int32`	Standard signed integer type. Commonly used for indexing and general-purpose numerical tasks.
64-bit Integer	`torch.int64`	Long integer type. Often used for large indexing arrays or for tasks involving large numbers.
8-bit Unsigned Integer	`torch.uint8`	8-bit unsigned integer. Commonly used for image data (e.g., pixel values between 0 and 255).
Boolean	`torch.bool`	Boolean type, stores `True` or `False` values. Often used for masks in logical operations.
Complex 64	`torch.complex64`	Complex number type with 32-bit real and 32-bit imaginary parts. Used for scientific and signal processing tasks.
Complex 128	`torch.complex128`	Complex number type with 64-bit real and 64-bit imaginary parts. Offers higher precision but uses more memory.
Quantized Integer	`torch.qint8`	Quantized signed 8-bit integer. Used in quantized models for efficient inference.
Quantized Unsigned Integer	`torch.quint8`	Quantized unsigned 8-bit integer. Often used for quantized tensors in image-related tasks.

Mathematical operations

1. Scalar operation

```
x = torch.rand(2,2)
x
```

```
tensor([[0.9594, 0.1334],
        [0.8591, 0.6639]])
```

```
# addition
x + 2
# substraction
x - 2
# multiplication
x * 3
# division
x / 3
# int division
(x * 100)//3
# mod
((x * 100)//3)%2
# power
x**2
```

```
tensor([[0.9205, 0.0178],
        [0.7380, 0.4408]])
```

2. Element wise operation

```
a = torch.rand(2,3)
b = torch.rand(2,3)
```

```
print(a)
print(b)
```

```
tensor([[0.0554, 0.7498, 0.3761],
        [0.6721, 0.5572, 0.2686]])
tensor([[0.3466, 0.1875, 0.7433],
        [0.6713, 0.9844, 0.0096]])
```

```
# add
a + b
# sub
a - b
# multiply
a * b
# division
a / b
# power
a ** b
# mod
a % b
```

```
tensor([[0.0554, 0.1872, 0.3761],
        [0.0008, 0.5572, 0.0091]])
```

```
c = torch.tensor([1, -2, 3, -4])
```

```
# abs
torch.abs(c)
```

```
tensor([1, 2, 3, 4])
```

```
# negative
torch.neg(c)
```

```
tensor([-1,  2, -3,  4])
```

```
d = torch.tensor([1.9, 2.3, 3.7, 4.4])
```

```
# round
torch.round(d)
```

```
tensor([2., 2., 4., 4.])
```

```
# ceil
torch.ceil(d)
```

```
tensor([2., 3., 4., 5.])
```

```
# floor
torch.floor(d)
```

```
tensor([1., 2., 3., 4.])
```

```
# clamp
torch.clamp(d, min=2, max=3)
```

```
tensor([2.0000, 2.3000, 3.0000, 3.0000])
```

3. Reduction operation

```
e = torch.randint(size=(2,3), low=0, high=10, dtype=torch.float32)
e
```

```
tensor([[1., 2., 9.],
        [7., 6., 5.]])
```

```
# sum
torch.sum(e)
# sum along columns
torch.sum(e, dim=0)
# sum along rows
torch.sum(e, dim=1)
```

```
tensor([18, 13])
```

```
# mean
torch.mean(e)
# mean along col
torch.mean(e, dim=0)
```

```
tensor([4., 4., 7.])
```

```
# median
torch.median(e)
```

```
tensor(5.)
```

```
# max and min
torch.max(e)
torch.min(e)
```

```
tensor(1.)
```

```
# product
torch.prod(e)
```

```
tensor(3780.)
```

```
# standard deviation
torch.std(e)
```

```
tensor(3.0332)
```

```
# variance
torch.var(e)
```

```
tensor(9.2000)
```

```
# argmax
torch.argmax(e)
```

```
tensor(2)
```

```
# argmin
torch.argmin(e)
```

```
tensor(0)
```

4. Matrix operations

```
f = torch.randint(size=(2,3), low=0, high=10)
g = torch.randint(size=(3,2), low=0, high=10)

print(f)
print(g)
```

```
tensor([[3, 7, 8],
        [7, 9, 9]])
tensor([[1, 1],
        [4, 2],
        [4, 7]])
```

```
# matrix multiplcation
torch.matmul(f, g)
```

```
tensor([[63, 73],
        [79, 88]])
```

```
vector1 = torch.tensor([1, 2])
vector2 = torch.tensor([3, 4])

# dot product
torch.dot(vector1, vector2)
```

```
tensor(11)
```

```
# transpose
torch.transpose(f, 0, 1)
```

```
tensor([[3, 7],
        [7, 9],
        [8, 9]])
```

```
h = torch.randint(size=(3,3), low=0, high=10, dtype=torch.float32)
h
```

```
tensor([[8., 2., 2.],
        [4., 8., 8.],
        [5., 0., 4.]])
```

```
# determinant
torch.det(h)
```

```
tensor(224.)
```

```
# inverse
torch.inverse(h)
```

```
tensor([[ 0.1429, -0.0357,  0.0000],
        [ 0.1071,  0.0982, -0.2500],
        [-0.1786,  0.0446,  0.2500]])
```

5. Comparison operations

```
i = torch.randint(size=(2,3), low=0, high=10)
j = torch.randint(size=(2,3), low=0, high=10)

print(i)
print(j)
```

```
tensor([[5, 0, 1],
        [1, 1, 3]])
tensor([[2, 6, 3],
        [5, 2, 5]])
```

```
# greater than
i > j
# less than
i < j
# equal to
i == j
# not equal to
i != j
# greater than equal to

# less than equal to
```

```
tensor([[True, True, True],
        [True, True, True]])
```

6. Special functions

```
k = torch.randint(size=(2,3), low=0, high=10, dtype=torch.float32)
k
```

```
tensor([[0., 6., 4.],
        [4., 8., 6.]])
```

```
# log
torch.log(k)
```

```
tensor([[1.6094, 1.3863, 0.0000],
        [1.9459, 0.0000, 2.0794]])
```

```
# exp
torch.exp(k)
```

```
tensor([[1.4841e+02, 5.4598e+01, 2.7183e+00],
        [1.0966e+03, 2.7183e+00, 2.9810e+03]])
```

```
# sqrt
torch.sqrt(k)
```

```
tensor([[2.2361, 2.0000, 1.0000],
        [2.6458, 1.0000, 2.8284]])
```

```
# sigmoid
torch.sigmoid(k)
```

```
tensor([[0.9933, 0.9820, 0.7311],
        [0.9991, 0.7311, 0.9997]])
```

```
# softmax
torch.softmax(k, dim=0)
```

```
tensor([[0.0180, 0.1192, 0.1192],
        [0.9820, 0.8808, 0.8808]])
```

```
# relu
torch.relu(k)
```

```
tensor([[0., 6., 4.],
        [4., 8., 6.]])
```

Inplace Operations

```
m = torch.rand(2,3)
n = torch.rand(2,3)

print(m)
print(n)
```

```
tensor([[0.2018, 0.1089, 0.7653],
        [0.6473, 0.3401, 0.6406]])
tensor([[0.4749, 0.1632, 0.6222],
        [0.5379, 0.2758, 0.7180]])
```

```
m.add_(n)
```

```
tensor([[0.6767, 0.2721, 1.3875],
        [1.1851, 0.6159, 1.3585]])
```

```
m
```

```
tensor([[0.6767, 0.2721, 1.3875],
        [1.1851, 0.6159, 1.3585]])
```

```
n
```

```
tensor([[0.4749, 0.1632, 0.6222],
        [0.5379, 0.2758, 0.7180]])
```

```
torch.relu(m)
```

```
tensor([[0.6767, 0.2721, 1.3875],
        [1.1851, 0.6159, 1.3585]])
```

```
m.relu_()
```

```
tensor([[0.6767, 0.2721, 1.3875],
        [1.1851, 0.6159, 1.3585]])
```

```
m
```

```
tensor([[0.6767, 0.2721, 1.3875],
        [1.1851, 0.6159, 1.3585]])
```

Copying a Tensor

```
a = torch.rand(2,3)
a
```

```
tensor([[0.5153, 0.9985, 0.6783],
        [0.2776, 0.6227, 0.2982]])
```

```
b = a
```

```
b
```

```
tensor([[0.5153, 0.9985, 0.6783],
        [0.2776, 0.6227, 0.2982]])
```

```
a[0][0] = 0
```

```
a
```

```
tensor([[0.0000, 0.9985, 0.6783],
        [0.2776, 0.6227, 0.2982]])
```

```
b
```

```
tensor([[0.0000, 0.9985, 0.6783],
        [0.2776, 0.6227, 0.2982]])
```

```
id(a)
```

```
134838567488544
```

```
id(b)
```

```
134838567488544
```

```
b = a.clone()
```

```
a
```

```
tensor([[0.0000, 0.9985, 0.6783],
        [0.2776, 0.6227, 0.2982]])
```

```
b
```

```
tensor([[0.0000, 0.9985, 0.6783],
        [0.2776, 0.6227, 0.2982]])
```

```
a[0][0] = 10
```

```
a
```

```
tensor([[10.0000,  0.9985,  0.6783],
        [ 0.2776,  0.6227,  0.2982]])
```

```
b
```

```
tensor([[0.0000, 0.9985, 0.6783],
        [0.2776, 0.6227, 0.2982]])
```

```
id(a)
```

```
134838567488544
```

```
id(b)
```

```
134838569581552
```

Start coding or generate with AI.