DATA 586: Advanced Machine Learning

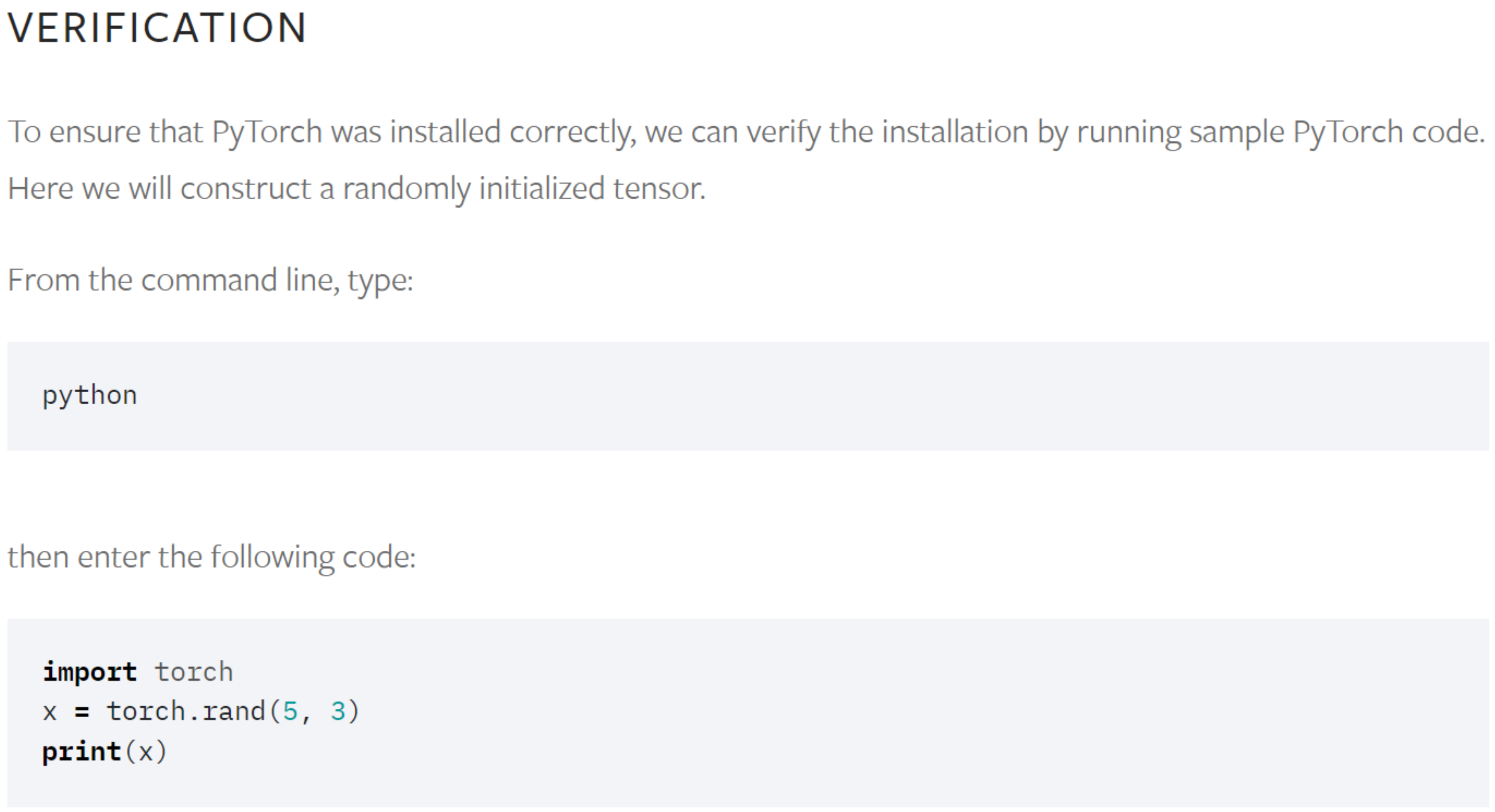
2023W2

Shan Du

Software Preparation

1. Anaconda: Python will be included. You can go to https://www.anaconda.com/download or just use https://pytorch.org/get-started/locally/
2. PyTorth: https://pytorch.org/get-started/locally/

Software Preparation



VERIFICATION

To ensure that PyTorch was installed correctly, we can verify the installation by running sample PyTorch code. Here we will construct a randomly initialized tensor.

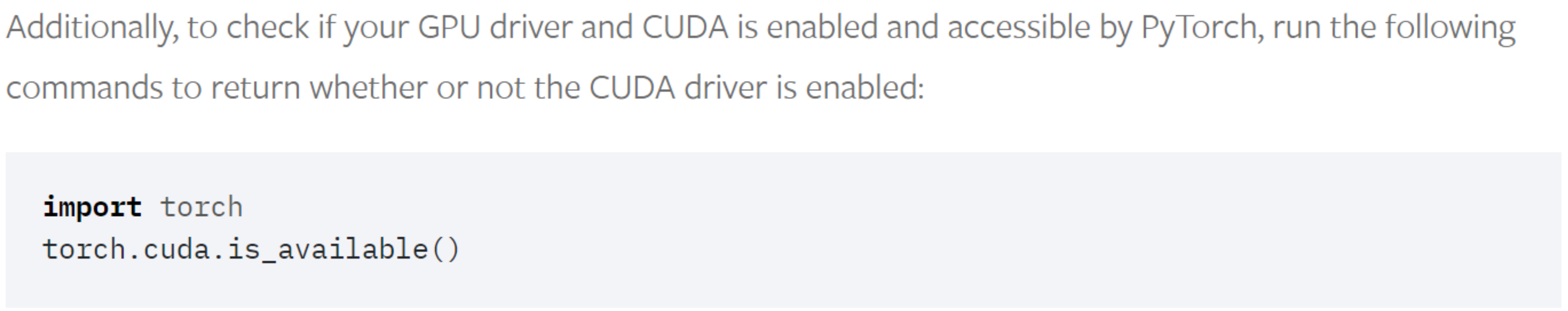
From the command line, type:

```
python
```

then enter the following code:

```
import torch
x = torch.rand(5, 3)
print(x)
```

Software Preparation



Additionally, to check if your GPU driver and CUDA is enabled and accessible by PyTorch, run the following commands to return whether or not the CUDA driver is enabled:

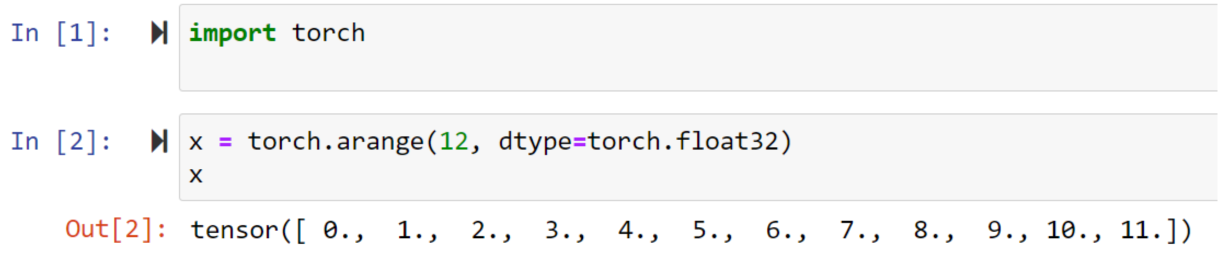
```
import torch
torch.cuda.is_available()
```

Install the *d2l* package that was developed in order to encapsulate frequently used functions and classes found throughout the textbook:

pip install d2l==1.0.0b0

Data Manipulation - Getting Started

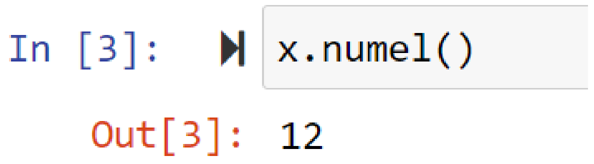
- A tensor represents an array of numerical values. With one axis, a tensor is called a *vector*. With two axes, a tensor is called a *matrix*. With $k > 2$ axes, we just refer to the object as a k^{th}*order tensor*.
- PyTorch provides a variety of functions for creating new tensors prepopulated with values. For example, by invoking *arange(n)*, we can create a vector of evenly spaced values, starting at 0 (included) and ending at n (not included). By default, the interval size is 1. Unless otherwise specified, new tensors are stored in main memory and designated for CPU-based computation.



```
In [1]:  import torch

In [2]:  x = torch.arange(12, dtype=torch.float32)
         x

Out[2]: tensor([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

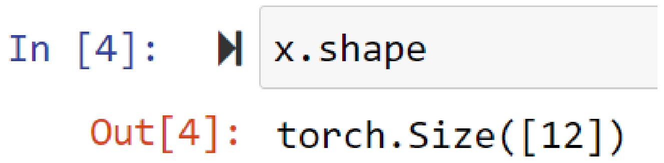
Data Manipulation - Getting Started

- Each of these values is called an *element* of the tensor. The tensor *x* contains 12 elements. We can inspect the total number of elements in a tensor via its *numel* method.



```
In [3]:  x.numel()

Out[3]:  12
```

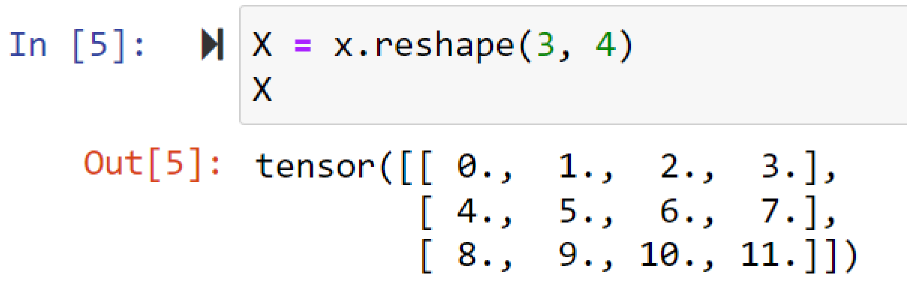
- We can access a tensor's shape (the length along each axis) by inspecting its *shape* attribute. Because we are dealing with a vector here, the shape contains just a single element and is identical to the size.



```
In [4]:  x.shape

Out[4]:  torch.Size([12])
```

Data Manipulation - Getting Started

- We can change the shape of a tensor without altering its size or values, by invoking *reshape*.

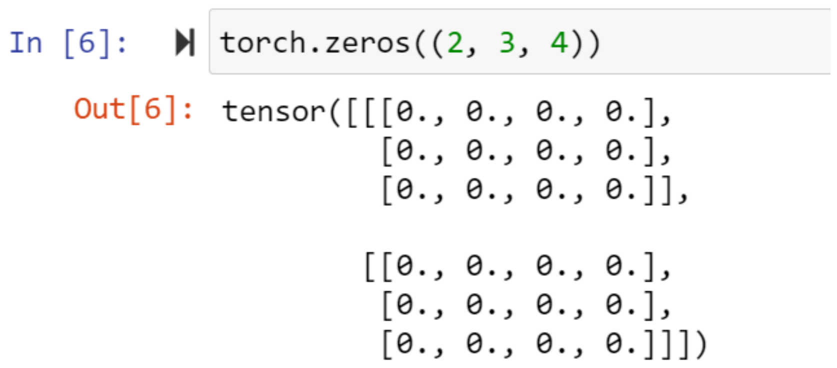


```
In [5]:    X = x.reshape(3, 4)
           X

Out[5]: tensor([[ 0.,  1.,  2.,  3.],
                [ 4.,  5.,  6.,  7.],
                [ 8.,  9., 10., 11.]])
```

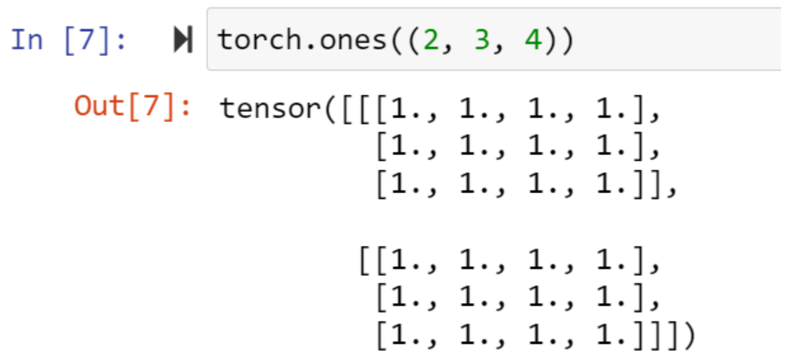
- Given a tensor of size n and target shape (h, w), we know that $w = n/h$. To automatically infer one component of the shape, we can place a *-1* for the shape component that should be inferred automatically. In this case, instead of calling *x.reshape(3, 4)*, we could have equivalently called *x.reshape(-1, 4)* or *x.reshape(3, -1).*

Data Manipulation - Getting Started

- We often need to work with tensors initialized to contain all *zeros* or *ones*. We can construct a tensor with all elements set to zero (or one) and a shape of (2, 3, 4) via the *zeros (or ones)* function.



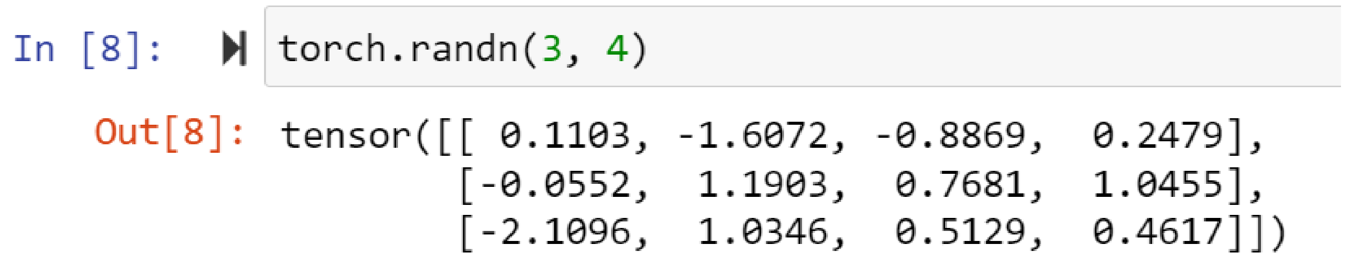
```
In [6]:   torch.zeros((2, 3, 4))

Out[6]: tensor([[[0., 0., 0., 0.],
                 [0., 0., 0., 0.],
                 [0., 0., 0., 0.]],

                [[0., 0., 0., 0.],
                 [0., 0., 0., 0.],
                 [0., 0., 0., 0.]]])
```



```
In [7]:   torch.ones((2, 3, 4))

Out[7]: tensor([[[1., 1., 1., 1.],
                 [1., 1., 1., 1.],
                 [1., 1., 1., 1.]],

                [[1., 1., 1., 1.],
                 [1., 1., 1., 1.],
                 [1., 1., 1., 1.]]])
```

Data Manipulation - Getting Started

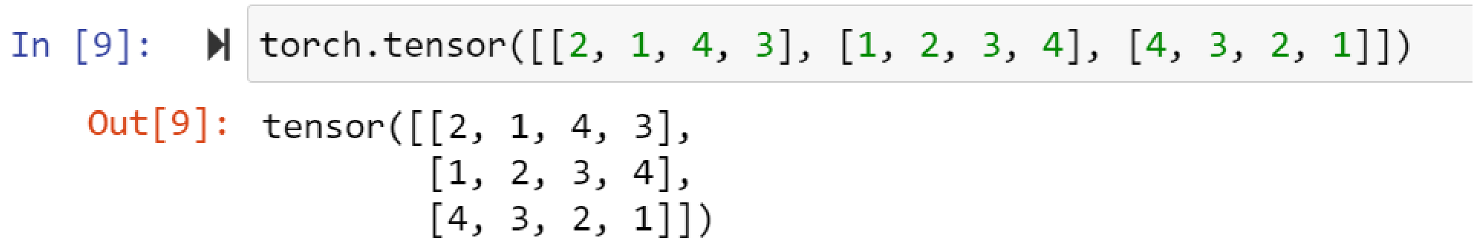
- We often wish to sample each element randomly (and independently) from a given probability distribution. For example, the parameters of neural networks are often initialized randomly.



```
In [8]:    torch.randn(3, 4)

Out[8]: tensor([[ 0.1103, -1.6072, -0.8869,  0.2479],
                [-0.0552,  1.1903,  0.7681,  1.0455],
                [-2.1096,  1.0346,  0.5129,  0.4617]])
```

Data Manipulation - Getting Started

- we can construct tensors by supplying the exact values for each element by supplying (possibly nested) Python list(s) containing numerical literals.



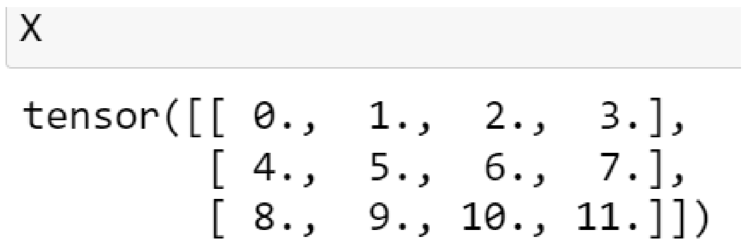
```
In [9]:  torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])

Out[9]: tensor([[2, 1, 4, 3],
                [1, 2, 3, 4],
                [4, 3, 2, 1]])
```

Data Manipulation - Indexing and Slicing

- As with Python lists, we can access tensor elements by indexing (starting with 0).
- To access an element based on its position relative to the end of the list, we can use negative indexing.
- We can access whole ranges of indices via slicing (e.g., *X[start:stop]*), where the returned value includes the first index (start) *but not the last* (stop).

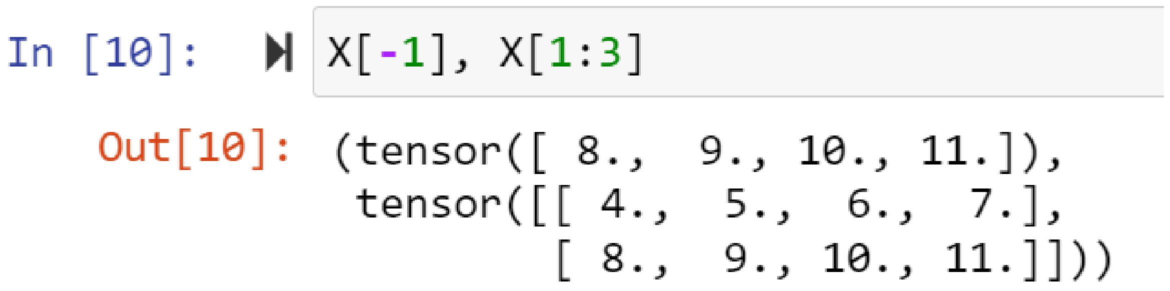
Data Manipulation - Indexing and Slicing

- Finally, when only one index (or slice) is specified for a k^{th}order tensor, it is applied along axis 0. Thus, in the following code, [-1] selects the last row and [1:3] selects the second and third rows.



```
X
```

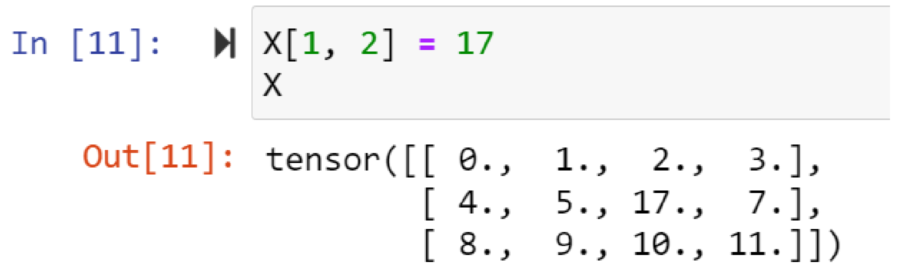
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```



```
In [10]:  X[-1], X[1:3]

Out[10]: (tensor([ 8.,  9., 10., 11.]),
          tensor([[ 4.,  5.,  6.,  7.],
                  [ 8.,  9., 10., 11.]]))
```

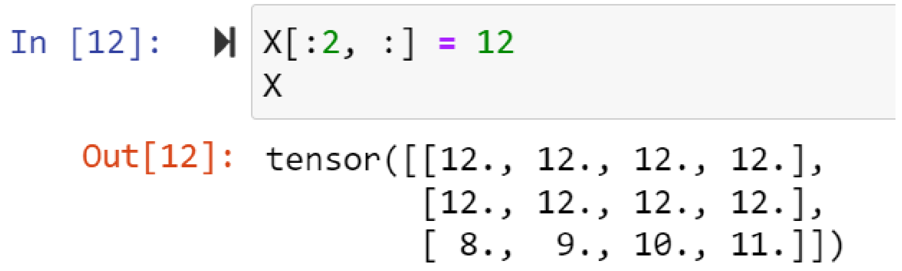
Data Manipulation - Indexing and Slicing

- Beyond reading, we can also write elements of a matrix by specifying indices.



```
In [11]:  X[1, 2] = 17
          X

Out[11]: tensor([[ 0.,  1.,  2.,  3.],
                 [ 4.,  5., 17.,  7.],
                 [ 8.,  9., 10., 11.]])
```

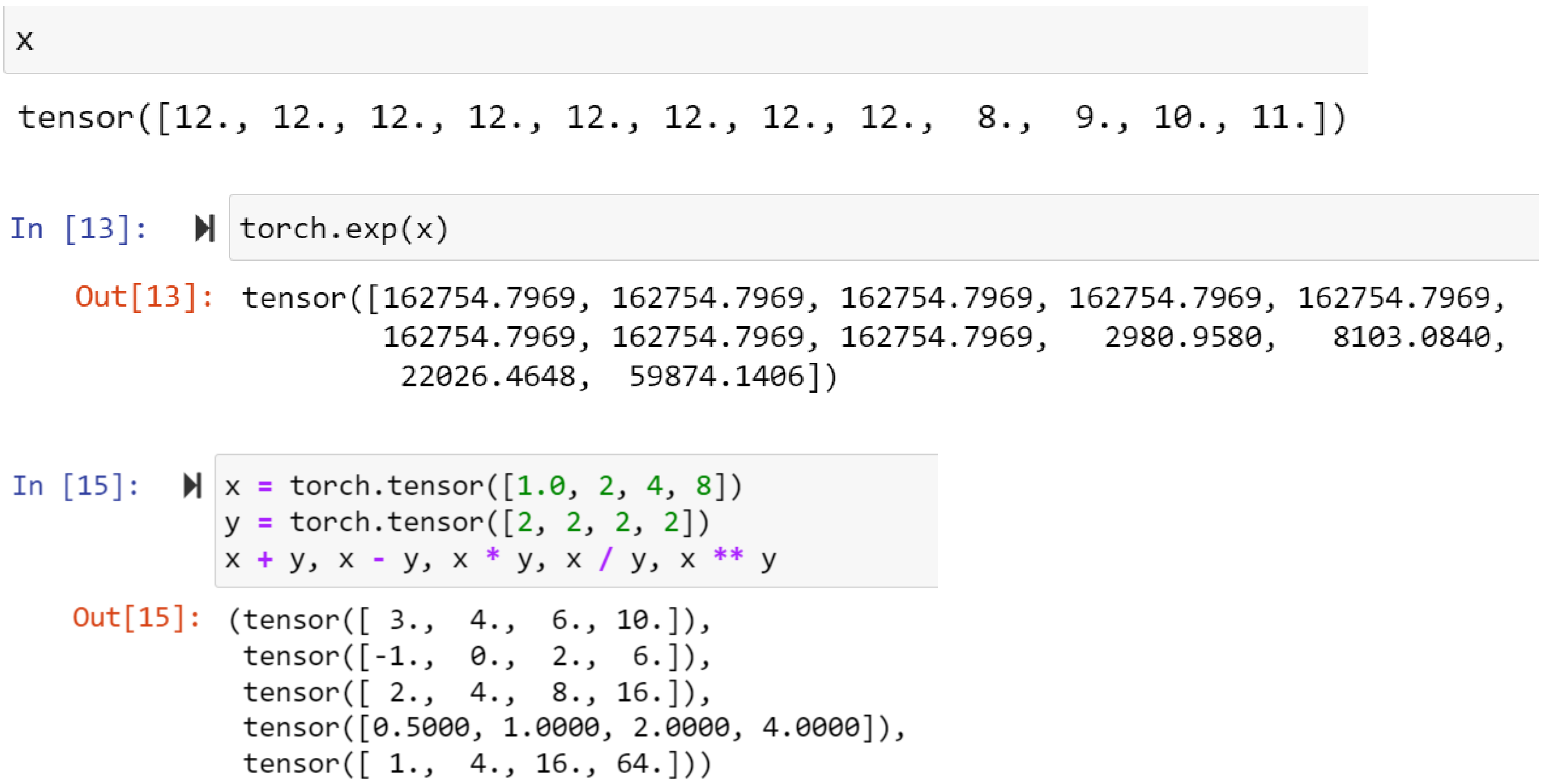
- If we want to assign multiple elements the same value, we apply the indexing on the lefthand side of the assignment operation. For instance, [:2, :] accesses the first and second rows, where : takes all the elements along axis 1 (column).



```
In [12]:  X[:2, :] = 12
          X

Out[12]: tensor([[12., 12., 12., 12.],
                 [12., 12., 12., 12.],
                 [ 8.,  9., 10., 11.]])
```

Data Manipulation - Operations

- *Elementwise* operations



```
x
```

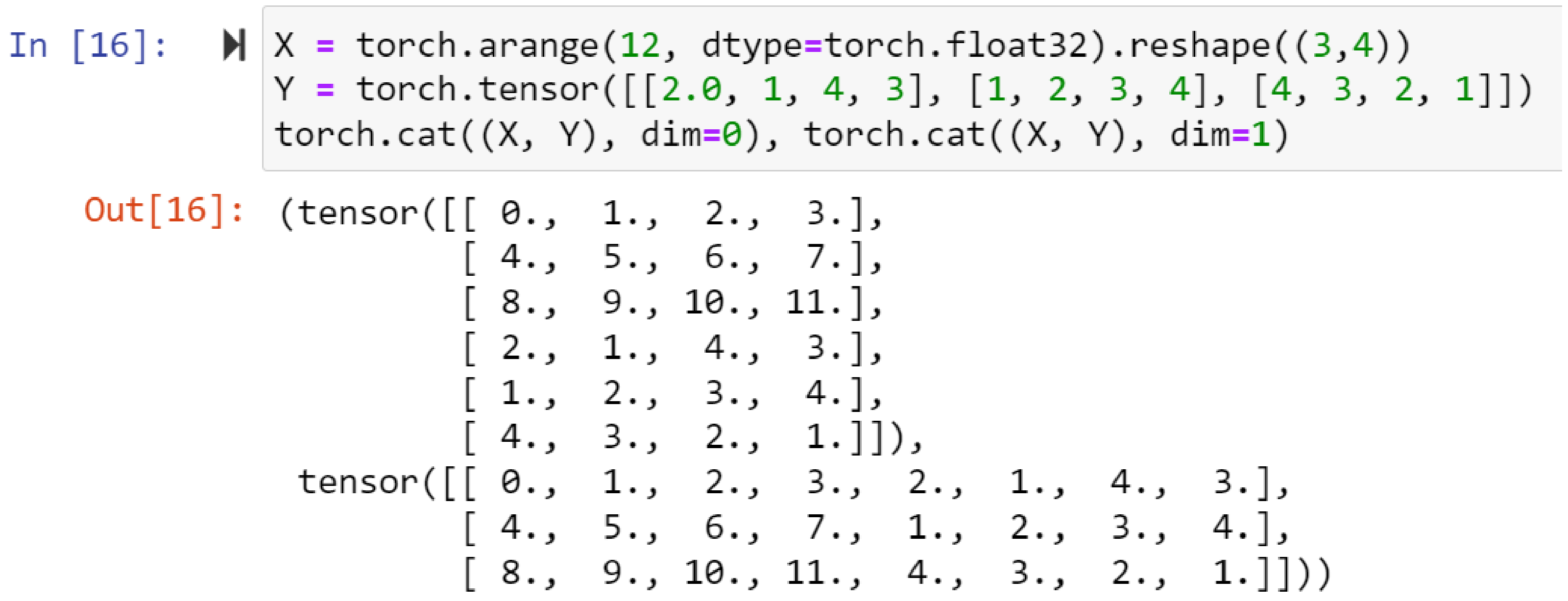
```
tensor([12., 12., 12., 12., 12., 12., 12., 12.,  8.,  9., 10., 11.])
```

```
In [13]:  torch.exp(x)

Out[13]:  tensor([162754.7969, 162754.7969, 162754.7969, 162754.7969, 162754.7969,
                  162754.7969, 162754.7969, 162754.7969,   2980.9580,   8103.0840,
                   22026.4648,  59874.1406])
```

```
In [15]:  x = torch.tensor([1.0, 2, 4, 8])
          y = torch.tensor([2, 2, 2, 2])
          x + y, x - y, x * y, x / y, x ** y

Out[15]:  (tensor([ 3.,  4.,  6., 10.]),
           tensor([-1.,  0.,  2.,  6.]),
           tensor([ 2.,  4.,  8., 16.]),
           tensor([0.5000, 1.0000, 2.0000, 4.0000]),
           tensor([ 1.,  4., 16., 64.]))
```

Data Manipulation - Operations

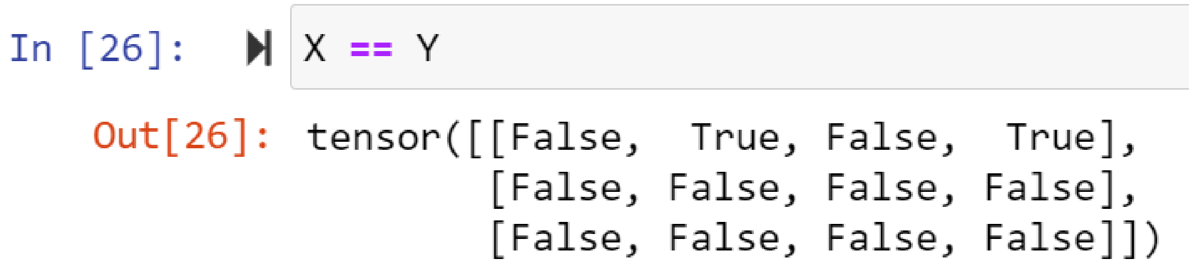
- *Concatenate* multiple tensors



```
In [16]:  X = torch.arange(12, dtype=torch.float32).reshape((3,4))
          Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
          torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1)

Out[16]: (tensor([[ 0.,  1.,  2.,  3.],
                  [ 4.,  5.,  6.,  7.],
                  [ 8.,  9., 10., 11.],
                  [ 2.,  1.,  4.,  3.],
                  [ 1.,  2.,  3.,  4.],
                  [ 4.,  3.,  2.,  1.]]),
          tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
                  [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
                  [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]]))
```

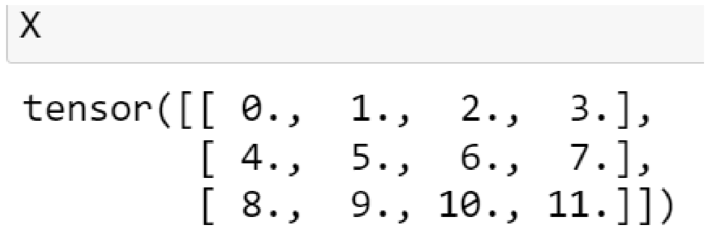
Data Manipulation - Operations

- Sometimes, we want to construct a binary tensor via *logical statements*.

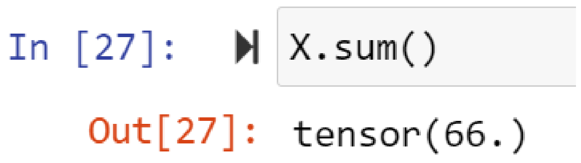


```
In [26]:  X == Y

Out[26]: tensor([[False,  True, False,  True],
                 [False, False, False, False],
                 [False, False, False, False]])
```

- Summing all the elements in the tensor yields a tensor with only one element.



```
X

tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```



```
In [27]:  X.sum()

Out[27]: tensor(66.)
```

Data Manipulation - Broadcasting

- Under certain conditions, even when shapes differ, we can still perform elementwise binary operations by invoking the *broadcasting mechanism*. Broadcasting works according to the following two-step procedure:
  (i) expand one or both arrays by copying elements along axes with length 1 so that after this transformation, the two tensors have the same shape;
  (ii) perform an elementwise operation on the resulting arrays.

Data Manipulation - Broadcasting

- Since *a* and *b* are 3×1 and 1×2 matrices, respectively, their shapes do not match up.
- Broadcasting produces a larger 3×2 matrix by replicating matrix *a* along the columns and matrix *b* along the rows before adding them elementwise.

```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b
```

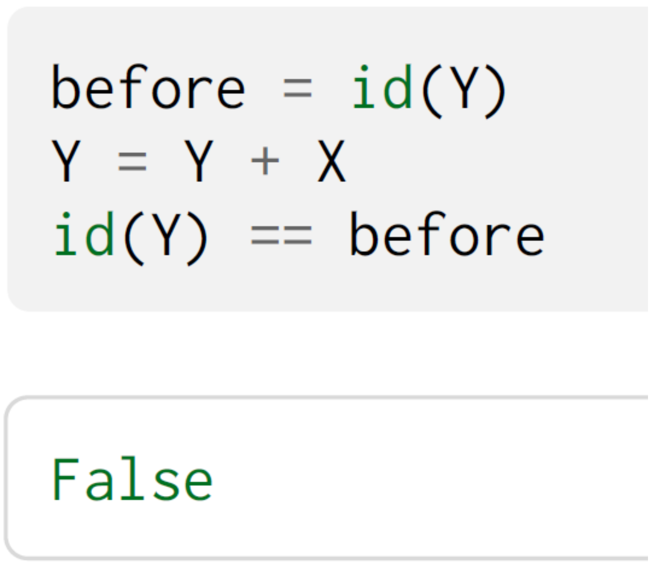
```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

```
a + b
```

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

Data Manipulation - Saving Memory

- Running operations can cause new memory to be allocated to host results. We can demonstrate this issue with Python's *id()* function.



```
before = id(Y)
Y = Y + X
id(Y) == before
```

```
False
```

Data Manipulation - Saving Memory

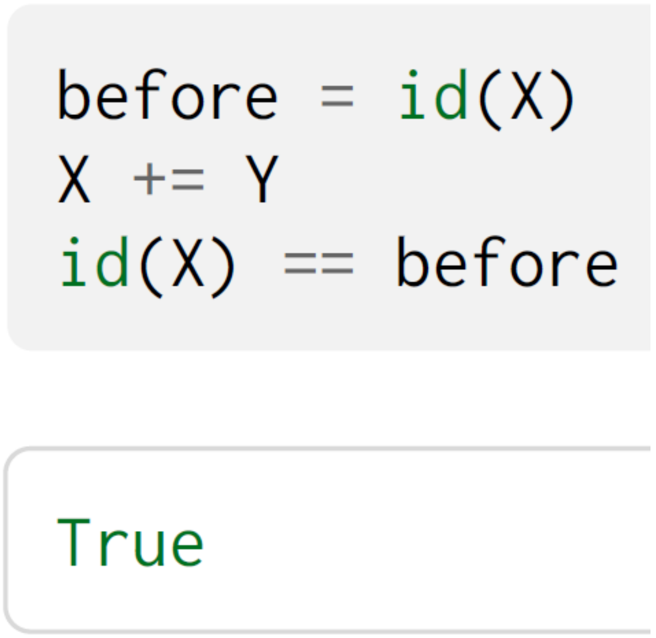
- Whenever possible, we want to perform these updates *in place*. We can assign the result of an operation to a previously allocated array *Z* by using slice notation: *Z[:] = <expression>.*

```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 140125792645008
id(Z): 140125792645008
```

Data Manipulation - Saving Memory

- If the value of *X* is not reused in subsequent computations, we can also use *X[:] = X + Y* or *X += Y* to reduce the memory overhead of the operation.



```
before = id(X)
X += Y
id(X) == before
```

```
True
```

Data Manipulation - Conversion to Other Python Objects

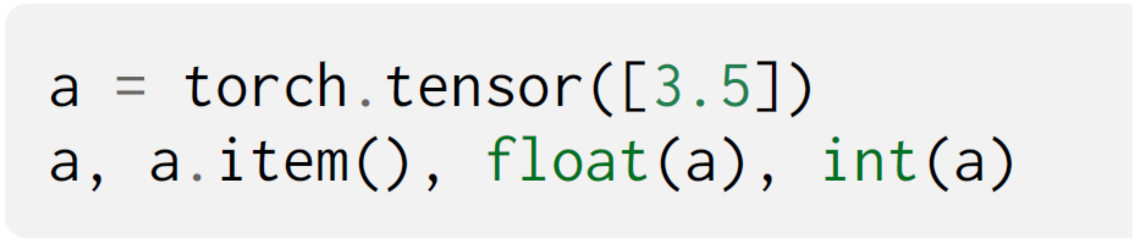
- The torch Tensor and numpy array will share their underlying memory, and changing one through an in-place operation will also change the other.

```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

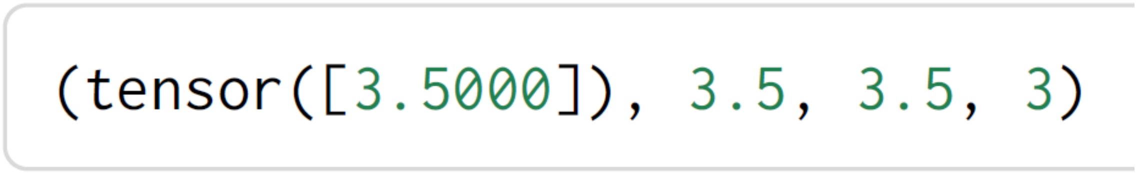
```
(numpy.ndarray, torch.Tensor)
```

Data Manipulation - Conversion to Other Python Objects

- To convert a size-1 tensor to a Python scalar, we can invoke the *item* function or Python’s built-in functions.



```python
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```



```
(tensor([3.5000]), 3.5, 3.5, 3)
```

Data Preprocessing - Reading the Dataset

- Comma-separated values (CSV) files are ubiquitous for storing tabular (spreadsheet-like) data. Here, each line corresponds to one record and consists of several (comma-separated) fields.
- To demonstrate how to load CSV files with *pandas*, we create a CSV file below *../data/house_tiny.csv*. This file represents a dataset of homes, where each row corresponds to a distinct home and the columns correspond to the number of rooms (*NumRooms*), the roof type (*RoofType*), and the price (*Price*).

Data Preprocessing - Reading the Dataset

```python
import os

os.makedirs(os.path.join('..', 'data'), exist_ok=True)
data_file = os.path.join('..', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write('''NumRooms,RoofType,Price
NA,NA,127500
2,NA,106000
4,Slate,178100
NA,NA,140000''')
```

- Now let's import *pandas* and load the dataset with *read_csv*.

```python
import pandas as pd

data = pd.read_csv(data_file)
print(data)
```

```
   NumRooms RoofType   Price
0       NaN      NaN  127500
1       2.0      NaN  106000
2       4.0    Slate  178100
3       NaN      NaN  140000
```

Data Preprocessing - Data Preparation

- In supervised learning, we train models to predict a designated *target* value, given some set of *input* values.
- Our first step in processing the dataset is to separate out columns corresponding to input versus target values. We can select columns either by name or via integer-location based indexing (iloc).

Data Preprocessing - Data Preparation

- *pandas* replaced all CSV entries with value *NA* with a special *NaN (not a number)* value. This can also happen whenever an entry is empty, e.g., "3,,,270000". These are called *missing values.*
- Depending upon the context, missing values might be handled either via *imputation* or *deletion*. Imputation replaces missing values with estimates of their values while deletion simply discards either those rows or those columns that contain missing values.

Data Preprocessing - Data Preparation

- Here are some common imputation heuristics.
- For categorical input fields, we can treat *NaN* as a category. Since the *RoofType* column takes values *Slate* and *NaN*, *pandas* can convert this column into two columns *RoofType_Slate* and *RoofType_nan*.
- A row whose roof type is *Slate* will set values of *RoofType_Slate* and *RoofType_nan* to *1* and *0*, respectively.

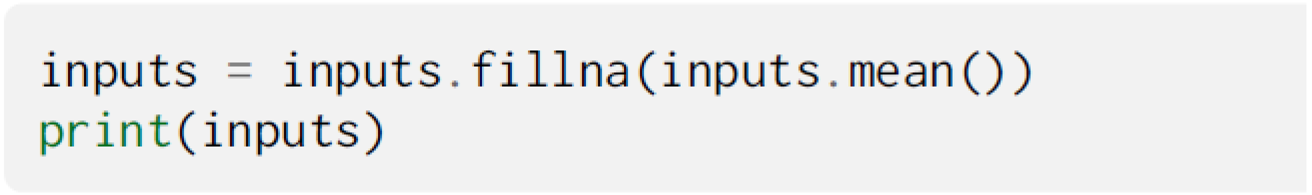
Data Preprocessing - Data Preparation

```
inputs, targets = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```

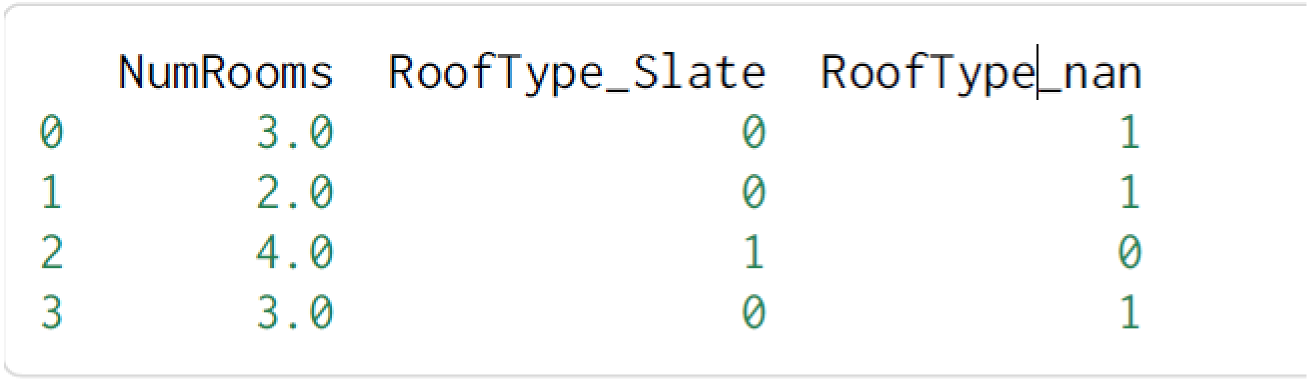
```
   NumRooms  RoofType_Slate  RoofType_nan
0       NaN               0             1
1       2.0               0             1
2       4.0               1             0
3       NaN               0             1
```

Data Preprocessing - Data Preparation

- For missing numerical values, one common heuristic is to replace the *NaN* entries with the mean value of the corresponding column.



```
inputs = inputs.fillna(inputs.mean())
print(inputs)
```



```
   NumRooms  RoofType_Slate  RoofType_nan
0       3.0               0             1
1       2.0               0             1
2       4.0               1             0
3       3.0               0             1
```

Data Preprocessing - Conversion to the Tensor Format

- All the entries in *inputs* and *targets* are numerical, we can load them into a tensor

```
import torch

X, y = torch.tensor(inputs.values), torch.tensor(targets.values)
X, y
```

```
(tensor([[3., 0., 1.],
         [2., 0., 1.],
         [4., 1., 0.],
         [3., 0., 1.]], dtype=torch.float64),
 tensor([127500, 106000, 178100, 140000]))
```

Linear Algebra - Scalars

- We denote scalars by ordinary lower-cased letters (e.g., x, y, and z) and the space of all (continuous) *real-valued* scalars by R.
- The expression $x \in R$ is a formal way to say that x is a real-valued scalar. The symbol $\in$ (pronounced “in”) denotes membership in a set.
- Scalars are implemented as tensors that contain only one element.

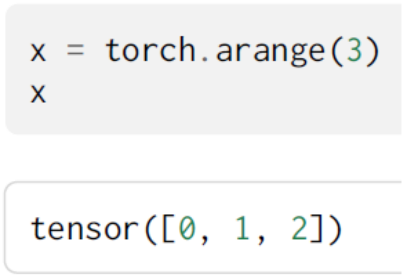
Linear Algebra - Scalars

```python
import torch

x = torch.tensor(3.0)
y = torch.tensor(2.0)

x + y, x * y, x / y, x**y
```

```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

Linear Algebra - Vectors

- You can think of vectors as fixed-length arrays of scalars. We call these values the *elements* of the vector (synonyms include *entries* and *components*).
- We denote vectors by bold lowercase letters, (e.g., **x**, **y**, and **z**).
- Vectors are implemented as 1st-order tensors. Vector indices start at 0, also known as *zero-based indexing.*



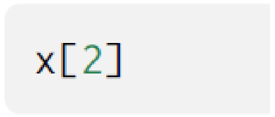
```
x = torch.arange(3)
x
```

```
tensor([0, 1, 2])
```

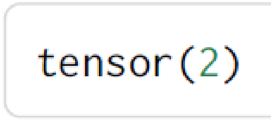
Linear Algebra - Vectors

- We can refer to an element of a vector by using a subscript. For example, x_2 denotes the second element of **x**. Since x_2 is a scalar, we do not bold it.
- By default, we visualize vectors by stacking their elements vertically.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$



```
x[2]
```

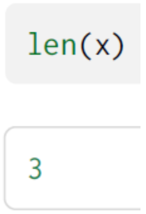


```
tensor(2)
```

Here x_1, …, x_n are elements of the vector.

Linear Algebra - Vectors

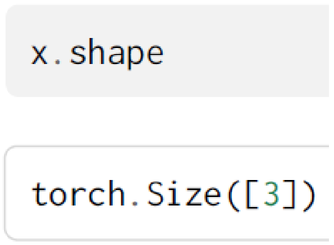
- To indicate that a vector contains n elements, we write $\mathbf{x} \in R^n$. Formally, we call n the *dimensionality* of the vector.



```
len(x)
```

```
3
```

- We can also access the length via the shape attribute. The shape is a tuple that indicates a tensor's length along each axis. Tensors with just one axis have shapes with just one element.



```
x.shape
```

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

Linear Algebra - Matrices

- Just as scalars are 0th-order tensors and vectors are 1st-order tensors, matrices are 2nd-order tensors. We denote matrices by bold capital letters (e.g., **X**, **Y**, and **Z**), and represent them in code by tensors with two axes.
- The expression $\mathbf{A} \in R^{m \times n}$ indicates that a matrix **A** contains $m \times n$ real-valued scalars, arranged as m rows and n columns. When $m = n$, we say that a matrix is *square*.

Linear Algebra - Matrices

- To refer to an individual element, we subscript both the row and column indices, e.g., a_{ij} is the value that belongs to **A**'s i^{th} row and j^{th} column:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

Linear Algebra - Matrices

- We can convert any appropriately sized $m \times n$ tensor into an $m \times n$ matrix by passing the desired shape to *reshape*:

```
A = torch.arange(6).reshape(3, 2)
A
```

```
tensor([[0, 1],
        [2, 3],
        [4, 5]])
```

- When we exchange a matrix’s rows and columns, the result is called its *transpose*. The transpose of an $m \times n$ matrix is an $\mathrm{n} \times m$ matrix:

$$\mathbf{A}^T = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{21} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}$$

```
A.T
```

```
tensor([[0, 2, 4],
        [1, 3, 5]])
```

Linear Algebra - Matrices

- Symmetric matrices are the subset of square matrices that are equal to their own transposes: $\mathbf{A} = \mathbf{A}^T$.

```
A = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
A == A.T
```

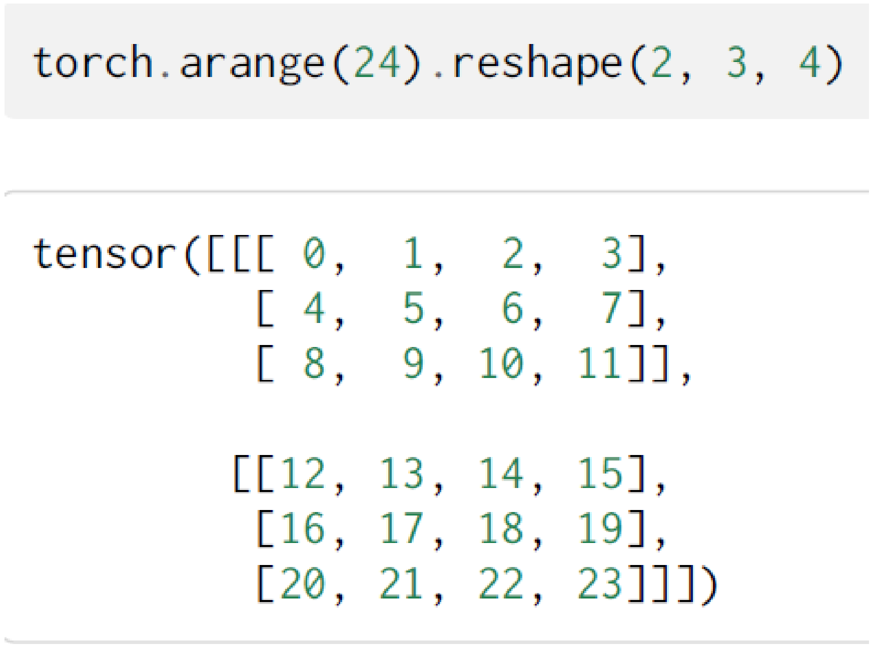
```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

Linear Algebra - Tensors

- We denote general tensors by capital letters with a special font face (e.g., X, Y, and Z) and their indexing mechanism (e.g., x_{ijk} and $[\mathrm{X}]_{1,2i-1,3}$) follows naturally from that of matrices.
- Tensors will become more important when we start working with images. Each image arrives as a 3rd-order tensor with axes corresponding to the *height*, *width*, and *channel*. At each spatial location, the intensities of each color (red, green, and blue) are stacked along the channel.

Linear Algebra - Tensors

- Video (a sequence of images) is represented in code by a 4th-order tensor, where distinct images are indexed along the first axis.



```
torch.arange(24).reshape(2, 3, 4)
```

```
tensor([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],

        [[12, 13, 14, 15],
         [16, 17, 18, 19],
         [20, 21, 22, 23]]])
```

Linear Algebra - Basic Properties of Tensor Arithmetic

- Elementwise operations produce outputs that have the same shape as their operands.

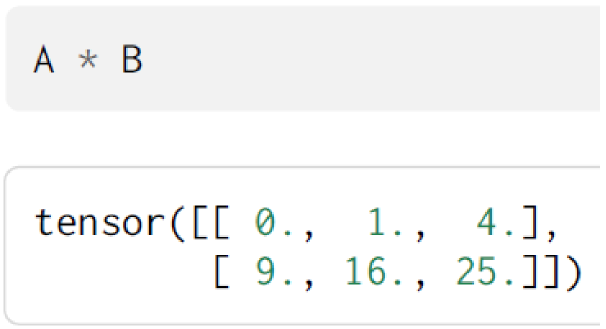
```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
B = A.clone()  # Assign a copy of `A` to `B` by allocating new memory
A, A + B
```

```
(tensor([[0., 1., 2.],
         [3., 4., 5.]]),
 tensor([[ 0.,  2.,  4.],
         [ 6.,  8., 10.]]))
```

Linear Algebra - Basic Properties of Tensor Arithmetic

- The elementwise product of two matrices is called their *Hadamard product* (denoted $\odot$).

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \cdots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \cdots & a_{mn}b_{mn} \end{bmatrix}$$

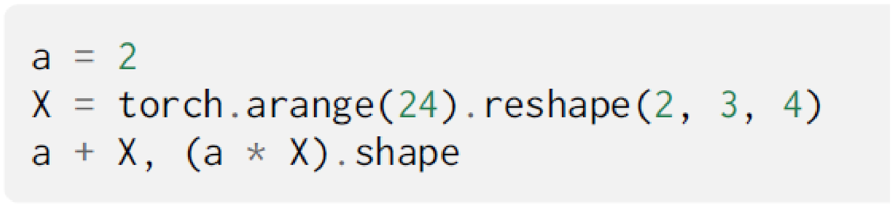


```
A * B
```

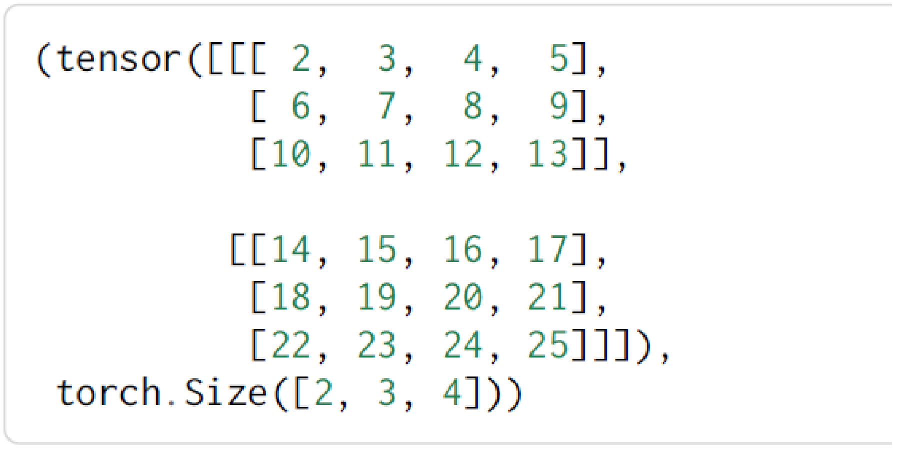
```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

Linear Algebra - Basic Properties of Tensor Arithmetic

- Adding or multiplying a scalar and a tensor produces a result with the same shape as the original tensor. Here, each element of the tensor is added to (or multiplied by) the scalar.



```
a = 2
X = torch.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```



```
(tensor([[[ 2,  3,  4,  5],
          [ 6,  7,  8,  9],
          [10, 11, 12, 13]],

         [[14, 15, 16, 17],
          [18, 19, 20, 21],
          [22, 23, 24, 25]]]),
 torch.Size([2, 3, 4]))
```

Linear Algebra - Reduction

- Often, we wish to calculate the sum of a tensor’s elements. To express the sum of the elements in a vector $\mathbf{x}$ of length n, we write $\sum_{i=1}^{n} x_i$.

```
x = torch.arange(3, dtype=torch.float32)
x, x.sum()
```

```
(tensor([0., 1., 2.]), tensor(3.))
```

```
A.shape, A.sum()
```

```
(torch.Size([2, 3]), tensor(15.))
```

Linear Algebra - Reduction

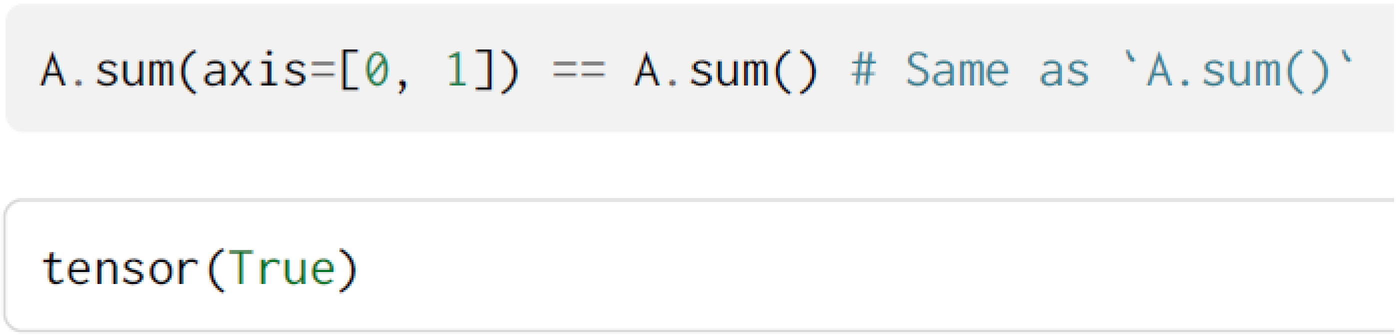
- By default, invoking the *sum* function *reduces* a tensor along all of its axes, eventually producing a scalar.
- Our libraries also allow us to specify the axes along which the tensor should be reduced.
- To sum over all elements along the rows (axis 0), we specify *axis=0* in *sum*. Since the input matrix reduces along axis 0 to generate the output vector, this axis is missing from the shape of the output.

```
A.shape, A.sum(axis=0).shape
```

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

Linear Algebra - Reduction

- Reducing a matrix along both rows and columns via summation is equivalent to summing up all the elements of the matrix.



```
A.sum(axis=[0, 1]) == A.sum() # Same as `A.sum()`
```

```
tensor(True)
```

Linear Algebra - Reduction

- A related quantity is the *mean*, also called the *average*. We calculate the mean by dividing the sum by the total number of elements. Because computing the mean is so common, it gets a dedicated library function that works analogously to *sum*.

```
A.mean(), A.sum() / A.numel()
```

```
(tensor(2.5000), tensor(2.5000))
```

```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

```
(tensor([1.5000, 2.5000, 3.5000]), tensor([1.5000, 2.5000, 3.5000]))
```

Linear Algebra - Non-Reduction Sum

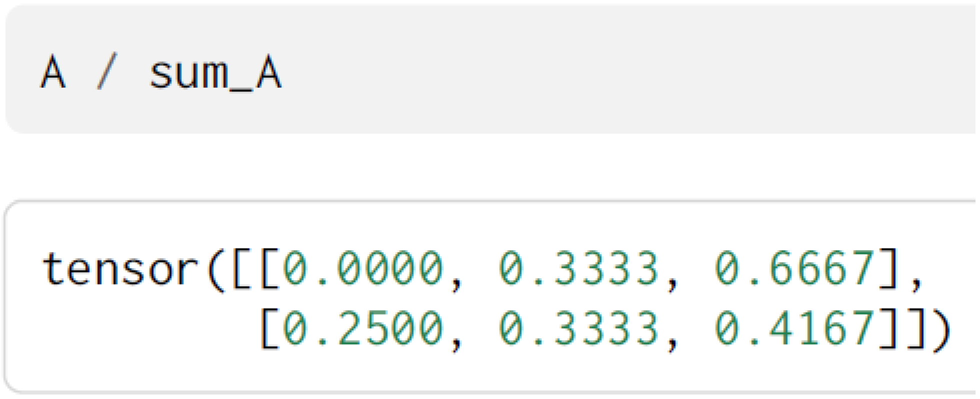
- Sometimes it can be useful to keep the number of axes unchanged when invoking the function for calculating the sum or mean. This matters when we want to use the broadcast mechanism.

```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

```
(tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

Linear Algebra - Non-Reduction Sum

- Since *sum_A* keeps its two axes after summing each row, we can divide *A* by *sum_A* with broadcasting to create a matrix where each row sums up to 1.



```
A / sum_A
```

```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

Linear Algebra - Non-Reduction Sum

- If we want to calculate the cumulative sum of elements of A along some axis, say axis=0 (row by row), we can call the cumsum function. By design, this function does not reduce the input tensor along any axis.

```
A.cumsum(axis=0)
```

```
tensor([[0., 1., 2.],
        [3., 5., 7.]])
```

Linear Algebra - Dot Products

- Given two vectors $\mathbf{x}$, $\mathbf{y} \in R^n$, their dot product $\mathbf{x}^\mathrm{T}\mathbf{y}$ (or $\langle\mathbf{x}, \mathbf{y}\rangle$) is a sum over the products of the elements at the same position: $\mathbf{x}^\mathrm{T}\mathbf{y} = \sum_{i=1}^{n} x_i$.

```
y = torch.ones(3, dtype = torch.float32)
x, y, torch.dot(x, y)
```

```
(tensor([0., 1., 2.]), tensor([1., 1., 1.]), tensor(3.))
```

- Equivalently, we can calculate the dot product of two vectors by performing an elementwise multiplication followed by a sum:

```
torch.sum(x * y)
```

```
tensor(3.)
```

Linear Algebra - Dot Products

- Dot products are useful in a wide range of contexts. For example, given some set of values, denoted by a vector **x** and a set of weights denoted by **w**, the weighted sum could be expressed as the dot product $\mathbf{x}^{\mathrm{T}}\mathbf{w}$.
- When the weights are non-negative and sum to one, the dot product expresses a weighted average.
- After normalizing two vectors to have unit length, the dot products express the cosine of the angle between them.

Linear Algebra - Matrix-Vector Products

- We can visualize our $m{\times}n$ matrix **A** in terms of its row vectors

$$\mathrm{A} = \begin{bmatrix} \mathbf{a}_1^{\mathrm{T}} \\ \mathbf{a}_2^{\mathrm{T}} \\ \vdots \\ \mathbf{a}_m^{\mathrm{T}} \end{bmatrix}$$

where each $\mathbf{a}_i^{\mathrm{T}} \in R^n$ is a row vector representing the i^{th} row of the matrix A.

Linear Algebra - Matrix-Vector Products

- The matrix-vector product $\mathbf{Ax}$ is simply a column vector of length m, whose i^{th}element is the dot product $\mathbf{a}_i^{\mathrm{T}}\mathbf{x}$

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a}_1^{\mathrm{T}}\mathbf{x} \\ \mathbf{a}_2^{\mathrm{T}}\mathbf{x} \\ \vdots \\ \mathbf{a}_m^{\mathrm{T}}\mathbf{x} \end{bmatrix}$$

Linear Algebra - Matrix-Vector Products

- We can think of multiplication with a matrix $\mathbf{A} \in R^{m \times n}$ as a transformation that projects vectors from R^n to R^m.
- These transformations are remarkably useful. For example, we can represent rotations as multiplications by certain square matrices.
- Matrix-vector products also describe the key calculation involved in computing the outputs of each layer in a neural network given the outputs from the previous layer.

Linear Algebra - Matrix-Vector Products

- To express a matrix-vector product in code, we use the *mv* function.
- PyTorch has a convenience operator @ that can execute both matrix-vector and matrix-matrix products (depending on its arguments). Thus we can write A@x.

```
A.shape, x.shape, torch.mv(A, x), A@x
```

```
(torch.Size([2, 3]), torch.Size([3]), tensor([ 5., 14.]), tensor([ 5., 14.]))
```

Linear Algebra - Matrix-Matrix Multiplication

- Say that we have two matrices $\mathbf{A} \in R^{n\times k}$ and $\mathbf{B} \in R^{k\times m}$:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \dots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nk} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \dots & b_{km} \end{bmatrix}$$

- Let $\mathbf{a}_i^{\mathrm{T}} \in R^k$ denote the row vector representing the i^{th} row of the matrix **A** and Let $\boldsymbol{b}_i \in R^k$ denote the column vector from the j^{th} column of the matrix **B**:

Linear Algebra - Matrix-Matrix Multiplication

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^{\mathrm{T}} \\ \mathbf{a}_2^{\mathrm{T}} \\ \vdots \\ \mathbf{a}_n^{\mathrm{T}} \end{bmatrix}, \mathbf{B} = [\mathbf{b}_1 \quad \mathbf{b}_2 \quad \dots \quad \mathbf{b}_m]$$

- To form the matrix product $\mathbf{C} \in R^{n\times m}$, we simply compute each element c_{ij} as the dot product between the i^{th}row of $\mathbf{A}$ and the j^{th} column of $\mathbf{B}$, i.e., $\mathbf{a}_i^{\mathrm{T}}\mathbf{b}_j$

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}_1^{\mathrm{T}} \\ \mathbf{a}_2^{\mathrm{T}} \\ \vdots \\ \mathbf{a}_n^{\mathrm{T}} \end{bmatrix} [\mathbf{b}_1 \quad \mathbf{b}_2 \quad \dots \quad \mathbf{b}_m] = \begin{bmatrix} \mathbf{a}_1^{\mathrm{T}}\mathbf{b}_1 & \mathbf{a}_1^{\mathrm{T}}\mathbf{b}_2 & \cdots & \mathbf{a}_1^{\mathrm{T}}\mathbf{b}_m \\ \mathbf{a}_2^{\mathrm{T}}\mathbf{b}_1 & \mathbf{a}_2^{\mathrm{T}}\mathbf{b}_2 & \dots & \mathbf{a}_2^{\mathrm{T}}\mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^{\mathrm{T}}\mathbf{b}_1 & \mathbf{a}_n^{\mathrm{T}}\mathbf{b}_2 & \dots & \mathbf{a}_n^{\mathrm{T}}\mathbf{b}_m \end{bmatrix}$$

Linear Algebra - Matrix-Matrix Multiplication

```
B = torch.ones(3, 4)
torch.mm(A, B), A@B
```

```
(tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]),
 tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]))
```

Linear Algebra - Norms

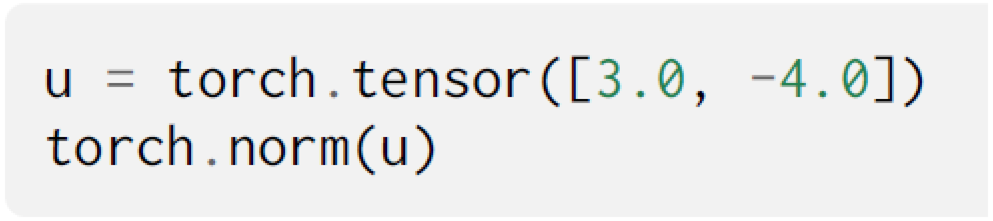
- Some of the most useful operators in linear algebra are *norms*.
- Informally, the norm of a vector tells us how *big* it is. For instance, the ℓ_2 norm measures the (Euclidean) length of a vector.
- Here, we are employing a notion of *size* that concerns the magnitude of a vector's components (not its dimensionality).

Linear Algebra - Norms

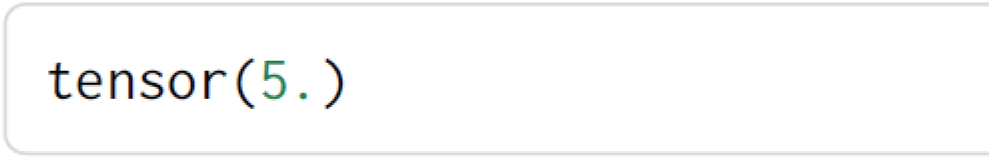
- A norm is a function $\|\cdot\|$ that maps a vector to a scalar and satisfies the following three properties:
  - Given any vector $\mathbf{x}$, if we scale (all elements of) the vector by a scalar $\alpha \in R$, its norm scales accordingly: $\|\alpha\mathbf{x}\| = |\alpha| \, \|\mathbf{x}\|$
  - For any vectors $\mathbf{x}$ and $\mathbf{y}$: norms satisfy the triangle inequality: $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$
  - The norm of a vector is nonnegative and it only vanishes if the vector is zero: $\|\mathbf{x}\| > 0 \ for \ all \ \mathbf{x} \neq \mathbf{0}$

Linear Algebra - Norms

- The ℓ_2 norm is expressed as $\| \mathbf{x} \|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$.
- The method *norm* calculates the ℓ_2 norm.



```
u = torch.tensor([3.0, -4.0])
torch.norm(u)
```



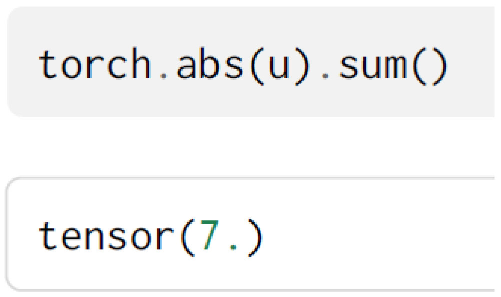
```
tensor(5.)
```

Linear Algebra - Norms

- The ℓ_1 norm is also popular and the associated metric is called the *Manhattan distance*.

$$\| \mathbf{x} \|_1 = \sum_{i=1}^{n} |x_i|$$

- To compute the ℓ_1 norm, we compose the absolute value with the sum operation.



```
torch.abs(u).sum()
```

```
tensor(7.)
```

Linear Algebra - Norms

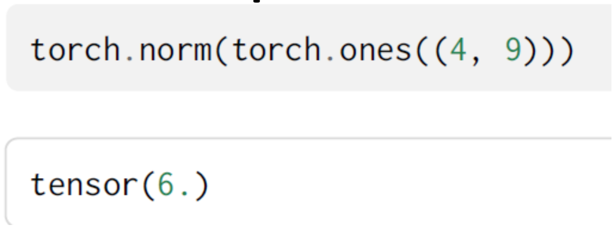
- Both the ℓ_2 and ℓ_1 norms are special cases of the more general ℓ_p norms:

$$\| \mathbf{x} \|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$

- For matrix, we introduce the *Frobenius norm*, which is much easier to compute and defined as the square root of the sum of the squares of a matrix's elements:

Linear Algebra - Norms

- The *Frobenius norm* behaves as if it were an ℓ_2 norm of a matrix-shaped vector.



```
torch.norm(torch.ones((4, 9)))
```

```
tensor(6.)
```

- In deep learning, we are often trying to solve optimization problems: e.g., minimize the distance between predictions and the ground-truth observations. These distances, which constitute the objectives of deep learning algorithms, are often expressed as norms.

Calculus - Derivatives and Differentiation

- A *derivative* is the rate of change in a function with respect to changes in its arguments. Derivatives can tell us how rapidly a loss function would increase or decrease were we to *increase* or *decrease* each parameter by an infinitesimally small amount.
- Formally, for functions $f : R \to R$, that map from scalars to scalars, the derivative of f at a point x is defined as

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Calculus - Derivatives and Differentiation

- This limit tells us what the ratio between a perturbation h and the change in the function value $f(x+h)-f(x)$ converges to as we shrink its size to zero.
- When $f'(x)$ exists, f is said to be *differentiable* at x; and when $f'(x)$ exists for all x on a set, e.g., the interval $[a,b]$, we say that f is differentiable on this set. Not all functions are differentiable. However, because computing the derivative of the loss is a crucial step in nearly all algorithms for training deep neural networks, we often optimize a differentiable *surrogate* instead.

Calculus - Derivatives and Differentiation

```
%matplotlib inline
import numpy as np
from matplotlib_inline import backend_inline
import torch
def f(x):
    return 3 * x ** 2 - 4 * x
```

- Setting $x = 1$, $\frac{f(x+h)-f(x)}{h}$ approaches 2 as h approaches 0.

```
for h in 10.0**np.arange(-1, -6, -1):
    print(f'h={h:.5f}, numerical limit={(f(1+h)-f(1))/h:.5f}')
```

```
h=0.10000, numerical limit=2.30000
h=0.01000, numerical limit=2.03000
h=0.00100, numerical limit=2.00300
h=0.00010, numerical limit=2.00030
h=0.00001, numerical limit=2.00003
```

Calculus - Derivatives and Differentiation

- There are several equivalent notational conventions for derivatives. Given $y = f(x)$, the following expressions are equivalent:
  $f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx}f(x) = Df(x)$
  $= D_x f(x)$

Calculus - Derivatives and Differentiation

- Derivatives of some common functions:

$$\frac{d}{dx}C = 0 \qquad \text{for any constant } C$$

$$\frac{d}{dx}x^n = nx^{n-1} \qquad \text{for } n \neq 0$$

$$\frac{d}{dx}e^x = e^x$$

$$\frac{d}{dx}\ln x = x^{-1}$$

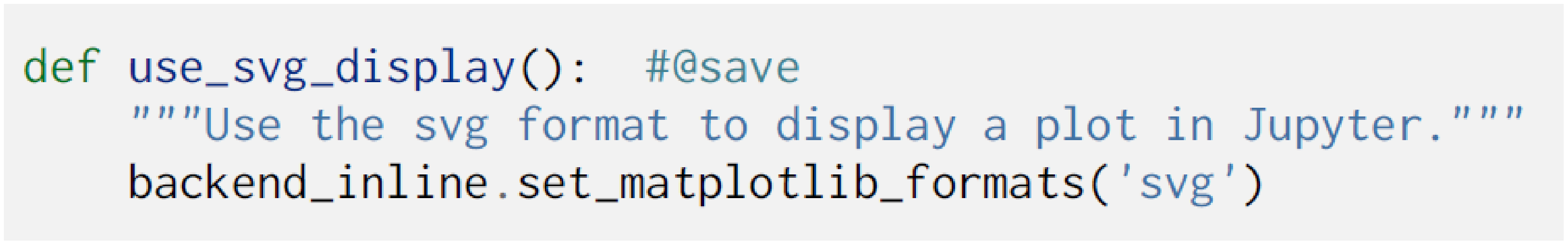
Calculus - Derivatives and Differentiation

- Functions composed from differentiable functions are often themselves differentiable. The following rules come in handy for working with compositions of any differentiable functions f and g, and constant C.

$\frac{d}{dx}[Cf(x)] = C\frac{d}{dx}f(x)$	Constant multiple rule
$\frac{d}{dx}[f(x)+g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x)$	Sum rule
$\frac{d}{dx}[f(x)g(x)] = f(x)\frac{d}{dx}g(x) + g(x)\frac{d}{dx}f(x)$	Product rule
$\frac{d}{dx}\frac{f(x)}{g(x)} = \frac{g(x)\frac{d}{dx}f(x) - f(x)\frac{d}{dx}g(x)}{g^2(x)}$	Quotient rule

Calculus - Visualization Utilities

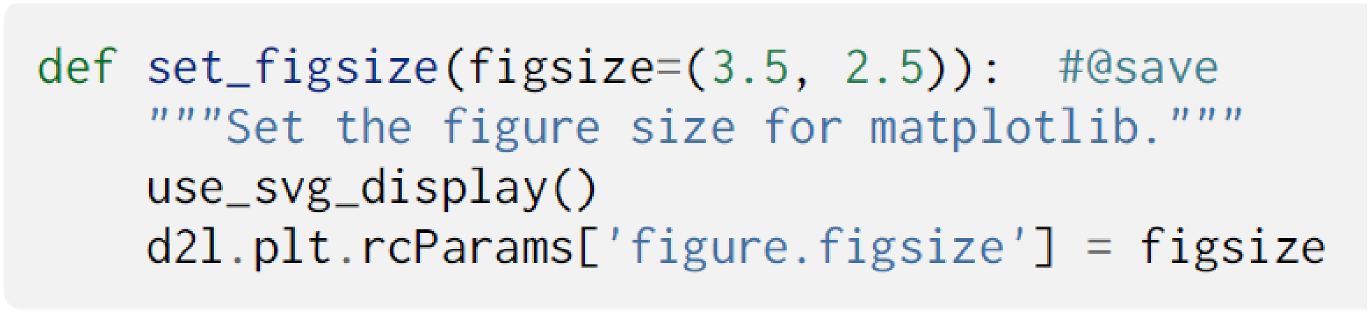
- We can visualize the slopes of functions using the *matplotlib* library. We need to define a few functions.
- *use_svg_display* tells *matplotlib* to output graphics in SVG format for crisper images. The comment *#@save* is a special modifier that allows us to save any function, class, or other code block to the d2l package so that we can invoke it later without repeating the code, e.g., via *d2l.use_svg_display().*



```
def use_svg_display():  #@save
    """Use the svg format to display a plot in Jupyter."""
    backend_inline.set_matplotlib_formats('svg')
```

Calculus - Visualization Utilities

- We can set figure sizes with *set_figsize*. Since the import statement *from matplotlib import pyplot* as *plt* was marked via *#@save* in the d2l package, we can call d2l.plt.



```python
def set_figsize(figsize=(3.5, 2.5)):  #@save
    """Set the figure size for matplotlib."""
    use_svg_display()
    d2l.plt.rcParams['figure.figsize'] = figsize
```

Calculus - Visualization Utilities

- The set_axes function can associate axes with properties, including labels, ranges, and scales.

```
#@save
def set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend):
    """Set the axes for matplotlib."""
    axes.set_xlabel(xlabel), axes.set_ylabel(ylabel)
    axes.set_xscale(xscale), axes.set_yscale(yscale)
    axes.set_xlim(xlim),     axes.set_ylim(ylim)
    if legend:
        axes.legend(legend)
    axes.grid()
```

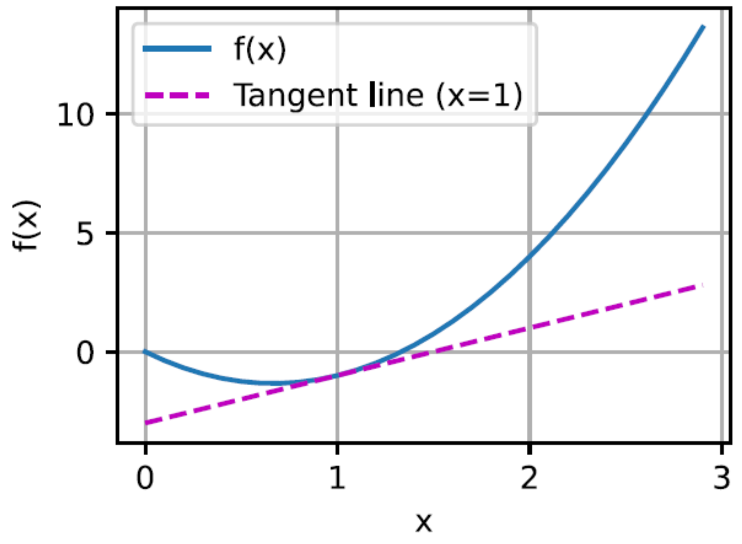
Calculus - Visualization Utilities

- With these three functions, we can define a plot function to overlay multiple curves. Much of the code here is just ensuring that the sizes and shapes of inputs match.

```python
#@save
def plot(X, Y=None, xlabel=None, ylabel=None, legend=[], xlim=None,
         ylim=None, xscale='linear', yscale='linear',
         fmts=('-', 'm--', 'g-.', 'r:'), figsize=(3.5, 2.5), axes=None):
    """Plot data points."""

    def has_one_axis(X):  # True if `X` (tensor or list) has 1 axis
        return (hasattr(X, "ndim") and X.ndim == 1 or isinstance(X, list)
                and not hasattr(X[0], "__len__"))

    if has_one_axis(X): X = [X]
    if Y is None:
        X, Y = [[]] * len(X), X
    elif has_one_axis(Y):
        Y = [Y]
    if len(X) != len(Y):
        X = X * len(Y)

    set_figsize(figsize)
    if axes is None: axes = d2l.plt.gca()
    axes.cla()
    for x, y, fmt in zip(X, Y, fmts):
        axes.plot(x,y,fmt) if len(x) else axes.plot(y,fmt)
    set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
```

Calculus - Visualization Utilities

```
x = np.arange(0, 3, 0.1)
plot(x, [f(x), 2 * x - 3], 'x', 'f(x)', legend=['f(x)', 'Tangent line (x=1)'])
```



Calculus - Partial Derivatives and Gradients

- In deep learning, we need to work with functions of *many* variables. We briefly introduce notions of the derivative that apply to such *multivariate functions*.
- Let $y = f(x_1, x_2, \ldots, x_n)$ be a function with n variables. The *partial derivative* of y with respect to its i^{th}parameter x_i is

$$\frac{\partial y}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}$$

- To calculate $\frac{\partial y}{\partial x_i}$, we can treat $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ as constants and calculate the derivative of y with respect to x_i.

Calculus - Partial Derivatives and Gradients

- The following notation conventions for partial derivatives are all common and all mean the same thing:

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = \partial_{x_i} f = \partial_i f = f_{x_i} = f_i = D_i f = D_{x_i} f$$

- We can concatenate partial derivatives of a multivariate function with respect to all its variables to obtain a vector that is called the *gradient* of the function.

Calculus - Partial Derivatives and Gradients

- Suppose that the input of function $f : R^n \to R$ is an n-dimensional vector $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$ and the output is a scalar. The gradient of the function f with respect to $\mathbf{x}$ is a vector of n partial derivatives:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\partial_{x_1} f(\mathbf{x}), \partial_{x_2} f(\mathbf{x}), \dots, \partial_{x_n} f(\mathbf{x})\right]^T$$

- The following rules come in handy for differentiating multivariate functions:

- For all $\mathbf{A} \in \mathbb{R}^{m \times n}$ we have $\nabla_{\mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A}^\top$ and $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} = \mathbf{A}$.
- For square matrices $\mathbf{A} \in \mathbb{R}^{n \times n}$ we have that $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} \mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$ and in particular $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}$.

Similarly, for any matrix $\mathbf{X}$, we have $\nabla_{\mathbf{X}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}$.

Calculus - Chain Rule

- In deep learning, the gradients of concern are often difficult to calculate because we are working with deeply nested functions (of functions (of functions…)). Fortunately, the *chain rule* takes care of this.
- Returning to functions of a single variable, suppose that $y = f(g(x))$ and that the underlying functions $y = f(u)$ and $u = g(x)$ are both differentiable. The chain rule states that

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

Calculus - Chain Rule

- Turning back to multivariate functions, suppose that $y = f(\mathbf{u})$ has variables $u_1, u_2, \ldots, u_m$, where each $u_i = g_i(\mathbf{x})$ has variables $x_1, x_2, \ldots, x_n$, i.e., $\mathbf{u} = g(\mathbf{x})$. Then the chain rule states that

$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial u_1}\frac{\partial u_1}{\partial x_i} + \frac{\partial y}{\partial u_2}\frac{\partial u_2}{\partial x_i} + \cdots + \frac{\partial y}{\partial u_m}\frac{\partial u_m}{\partial x_i}$$

And thus $\nabla_{\mathbf{x}} y = \mathbf{A} \nabla_u y$ where $\mathbf{A} \in R^{n \times m}$ is a *matrix* that contains the derivative of vector **u** with respect to vector **x**. Thus, evaluating the gradient requires computing a vector-matrix product.

Automatic Differentiation

- All modern deep learning frameworks offer *automatic differentiation* (often shortened to *autograd*).
- As we pass data through each successive function, the framework builds a *computational graph* that tracks how each value depends on others.
- To calculate derivatives, automatic differentiation packages then work backwards through this graph applying the chain rule. The computational algorithm for applying the chain rule this fashion is called *backpropagation*.

A Simple Function

- Let's assume that we are interested in differentiating the function $y = 2\mathbf{x}^\top\mathbf{x}$ with respect to the column vector $\mathbf{x}$. To start, we assign $\mathbf{x}$ an initial value.

```
import torch

x = torch.arange(4.0)
x
```

```
tensor([0., 1., 2., 3.])
```

A Simple Function

- In general, we avoid allocating new memory every time we take a derivative because deep learning requires successively computing derivatives with respect to the same parameters thousands or millions of times.
- Note that the gradient of a scalar-valued function with respect to a vector $\mathbf{x}$ is vector-valued and has the same shape as $\mathbf{x}$.

```
# Better create x = torch.arange(4.0, requires_grad=True)
x.requires_grad_(True)
x.grad  # The default value is None
```

A Simple Function

- We now calculate our function of $\mathbf{x}$ and assign the result to y.

```
y = 2 * torch.dot(x, x)
y
```

```
tensor(28., grad_fn=<MulBackward0>)
```

- We can now take the gradient of y with respect to $\mathbf{x}$ by calling its *backward* method. Next, we can access the gradient via $\mathbf{x}$’s grad attribute.

```
y.backward()
x.grad
```

```
tensor([ 0.,  4.,  8., 12.])
```

A Simple Function

- let's calculate another function of **x** and take its gradient. Note that PyTorch does not automatically reset the gradient buffer when we record a new gradient.
- Instead the new gradient is added to the already stored gradient. This behavior comes in handy when we want to optimize the sum of multiple objective functions.
- To reset the gradient buffer, we can call *x.grad.zero()* as follows:

```
x.grad.zero_()  # Reset the gradient
y = x.sum()
y.backward()
x.grad
```

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

Backward for Non-Scalar Variables

- When $\mathbf{y}$ is a vector, the most natural interpretation of the derivative of $\mathbf{y}$ with respect to a vector $\mathbf{x}$ is a matrix called the *Jacobian* that contains the partial derivatives of each component of $\mathbf{y}$ with respect to each component of $\mathbf{x}$.
- Likewise, for higher-order $\mathbf{y}$ and $\mathbf{x}$, the differentiation result could be an even higher-order tensor.

Backward for Non-Scalar Variables

- While *Jacobians* do show up in some advanced machine learning techniques, more commonly we want to sum up the gradients of each component of $\mathbf{y}$ with respect to the full vector $\mathbf{x}$, yielding a vector of the same shape as $\mathbf{x}$.
- For example, we often have a vector representing the value of our loss function calculated separately for each among a *batch* of training examples. Here, we just want to sum up the gradients computed individually for each example.

Backward for Non-Scalar Variables

- Because deep learning frameworks vary in how they interpret gradients of non-scalar tensors, PyTorch takes some steps to avoid confusion. Invoking *backward* on a non-scalar elicits an error unless we tell PyTorch how to reduce the object to a scalar.

```
x.grad.zero_()
y = x * x
y.backward(gradient=torch.ones(len(y)))  # Faster: y.sum().backward()
x.grad
```

```
tensor([0., 2., 4., 6.])
```

Detaching Computation

- Sometimes, we wish to move some calculations outside of the recorded computational graph. In this case, we need to detach the respective computational influence graph from the final result.
- For example, we have $z = x * y$ and $y = x * x$ but we want to focus on the direct influence of x on z rather than the influence conveyed via y.

Detaching Computation

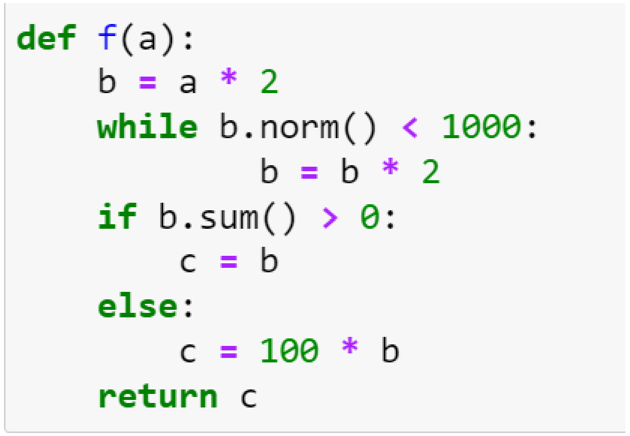
- In this case, we can create a new variable u that takes the same value as y but whose *provenance* (how it was created) has been wiped out.
- Thus u has no ancestors in the graph and gradients to not flow through u to x. For example, taking the gradient of $z = x * u$ will yield the result x, (not $3 * x * x$ as you might have expected since $z = x * x * x$).

```
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x

z.sum().backward()
x.grad == u
```

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

Gradients and Python Control Flow

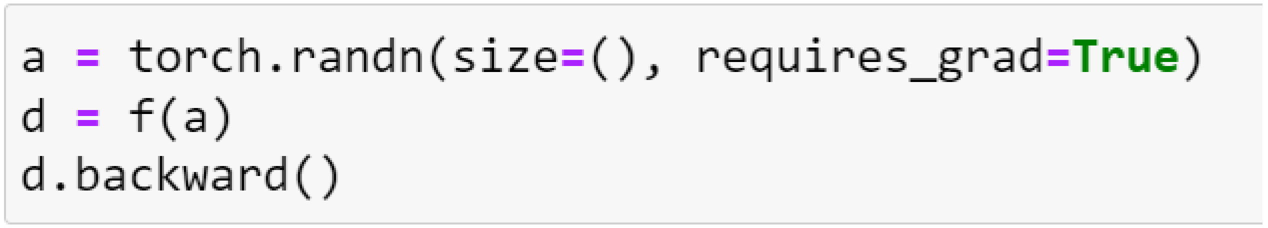
- One benefit of using automatic differentiation is that even if building the computational graph of a function required passing through a maze of Python control flow (e.g., conditionals, loops, and arbitrary function calls), we can still calculate the gradient of the resulting variable.
- For example, we have the following code snippet:



```python
def f(a):
    b = a * 2
    while b.norm() < 1000:
            b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

Gradients and Python Control Flow

- We call this function, passing in a random value as input. Since the input is a random variable, we do not know what form the computational graph will take.
- However, whenever we execute *f(a)* on a specific input, we realize a specific computational graph and can subsequently run *backward*.



```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

Probability and Statistics

- Machine learning is all about uncertainty.
- In supervised learning, we want to predict something unknown (the *target*) given something known (the *features*).
- Depending on our objective, we might attempt to predict the most likely value of the target. Or we might predict the value with the smallest expected distance from the target. And sometimes we wish not only to predict a specific value but to quantify our uncertainty.

Probability and Statistics

- In unsupervised learning, we often care about uncertainty. To determine whether a set of measurements are anomalous, it helps to know how likely one is to observe values in a population of interest.
- Moreover, in reinforcement learning, we wish to develop agents that act intelligently in various environments. This requires reasoning about how an environment might be expected to change and what rewards one might expect to encounter in response to each of the available actions.

Probability and Statistics

- *Probability* is the mathematical field concerned with reasoning under uncertainty.
- Given a probabilistic model of some process, we can reason about the likelihood of various events.
- The use of probabilities to describe the frequencies of repeatable events (like coin tosses) is fairly uncontroversial. In fact, *frequentist* scholars adhere to an interpretation of probability that applies *only* to such repeatable events.
- By contrast *Bayesian* scholars use the language of probability more broadly to formalize our reasoning under uncertainty.

Probability and Statistics

- Bayesian probability is characterized by two unique features:
  - (i) assigning degrees of belief to nonrepeatable events, e.g., what is the *probability* that the moon is made out of cheese?
  - (ii) subjectivity—while Bayesian probability provides unambiguous rules for how one should update their beliefs in light of new evidence, it allows for different individuals to start off with different *prior* beliefs.
- *Statistics* helps us to reason backwards, starting off with collection and organization of data and backing out to what inferences we might draw about the process that generated the data. Whenever we analyze a dataset, hunting for patterns that we hope might characterize a broader population, we are employing statistical thinking.

A Simple Example: Tossing Coins

- Imagine that we plan to toss a coin and want to quantify how likely we are to see heads (vs. tails). If the coin is *fair*, then both outcomes (heads and tails), are equally likely. Moreover, if we plan to toss the coin *n* times then the fraction of heads that we expect to see should exactly match the *expected* fraction of tails.
- Formally, the quantity 1/2 is called a *probability* and here it captures the certainty with which any given toss will come up heads. Probabilities assign scores between 0 and 1 to outcomes of interest, called *events*.
- Here the event of interest is *heads* and we denote the corresponding probability *P*(*heads*).

A Simple Example: Tossing Coins

- Probabilities are *theoretical* quantities that underly the data generating process.
- Statistics are *empirical* quantities that are computed as functions of the observed data. Our interests in probabilistic and statistical quantities are inextricably intertwined.

A Simple Example: Tossing Coins

- We often design special statistics called *estimators* that, given a dataset, produce *estimates* of model parameters like probabilities.
- Moreover, when those estimators satisfy a nice property called *consistency*, our estimates will converge to the corresponding probability.
- In turn, these inferred probabilities tell about the likely statistical properties of data from the same population that we might encounter in the future.

A Simple Example: Tossing Coins

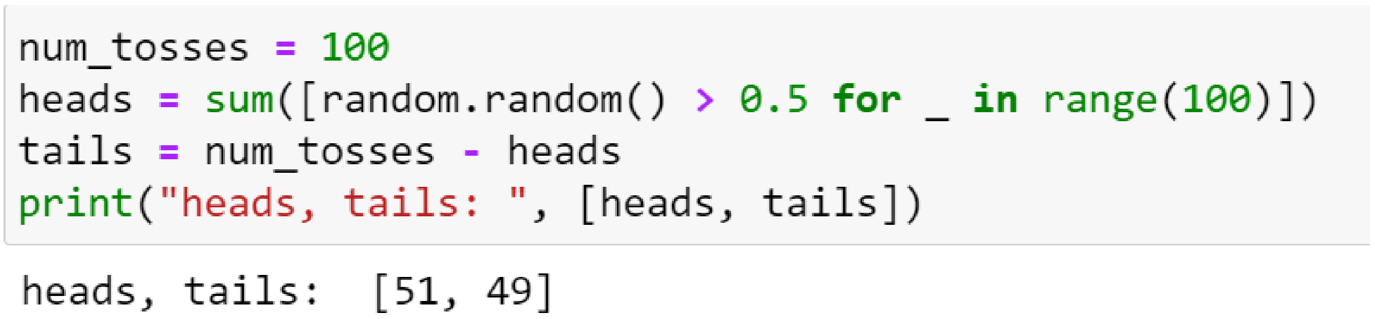
- Suppose that we stumbled upon a real coin for which we did not know the true *P(heads).*
- To investigate this quantity with statistical methods, we need to (i) collect some data; and (ii) design an estimator.
- Data acquisition here is easy; we can toss the coin many times and record all of the outcomes.
- Formally, drawing realizations from some underlying random process is called *sampling*. As you might have guessed, one natural estimator is the fraction between the number of observed *heads* by the total number of tosses.

A Simple Example: Tossing Coins

```
%matplotlib inline
import random
import torch
from torch.distributions.multinomial import Multinomial
from d2l import torch as d2l
```

- Now, suppose that the coin was in fact fair, i.e., *P(heads)* = 0.5. To simulate tosses of a fair coin, we can invoke Python's *random* which yields numbers in the interval [0, 1] where the probability of lying in any sub-interval $[a, b] \subset [0, 1]$ is equal to $b - a$. Thus we can get out 0 and 1 with probability 0.5 each by testing whether the returned float is greater than 0.5.

A Simple Example: Tossing Coins

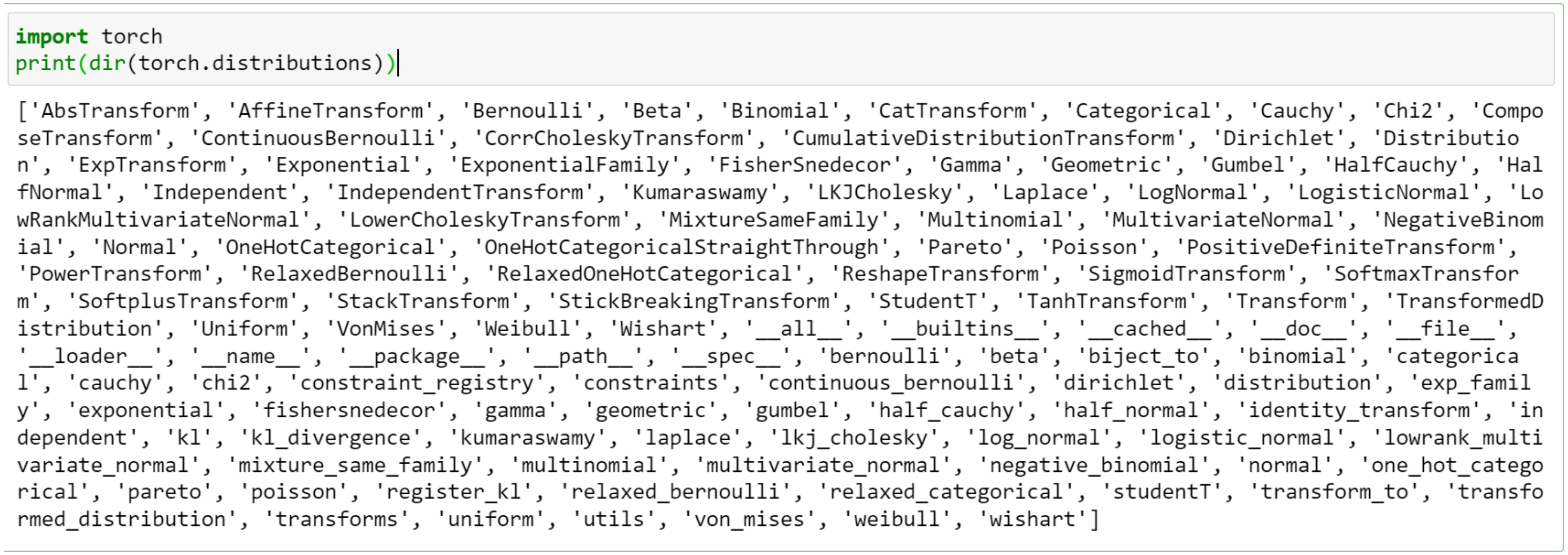


```python
num_tosses = 100
heads = sum([random.random() > 0.5 for _ in range(100)])
tails = num_tosses - heads
print("heads, tails: ", [heads, tails])
```

```
heads, tails:  [51, 49]
```

Documentation

- In order to know which functions and classes can be called in a module, we invoke the *dir* function. For instance, we can query all properties in the module for generating random numbers:



```python
import torch
print(dir(torch.distributions))
```

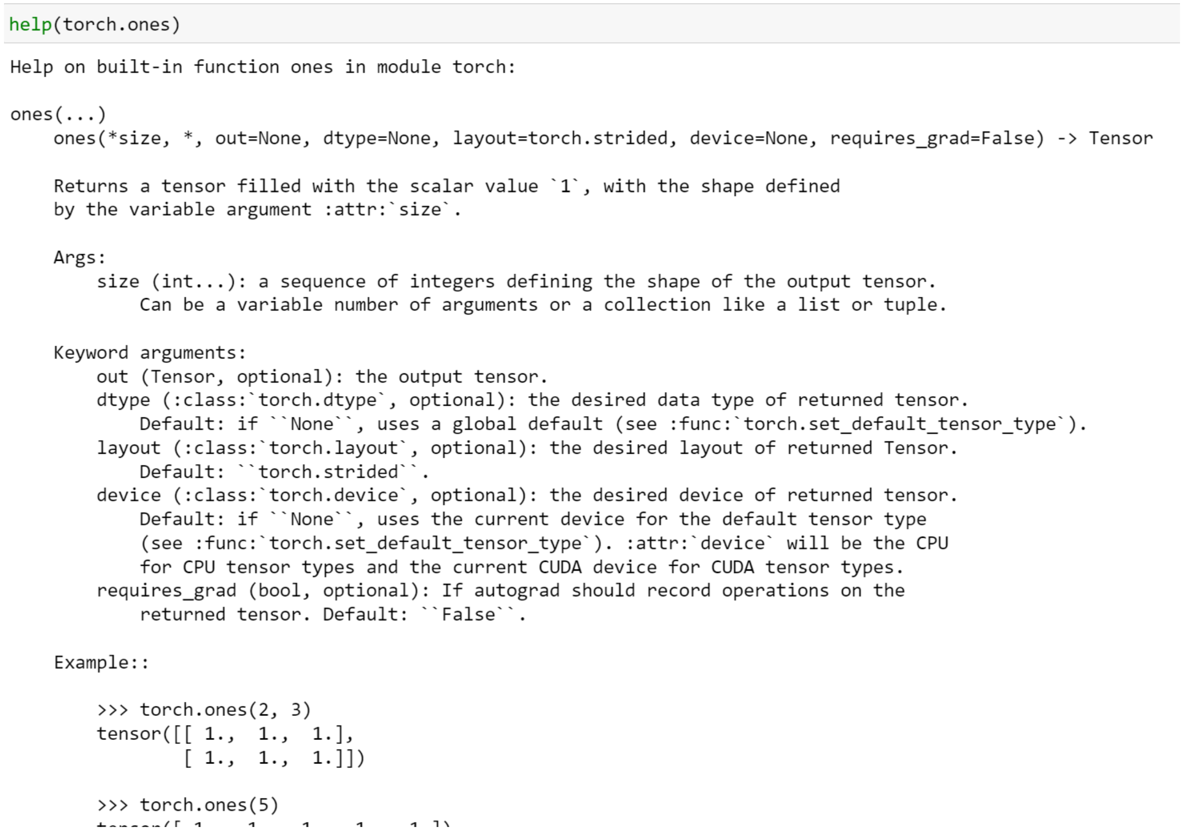
```
['AbsTransform', 'AffineTransform', 'Bernoulli', 'Beta', 'Binomial', 'CatTransform', 'Categorical', 'Cauchy', 'Chi2', 'Compo
seTransform', 'ContinuousBernoulli', 'CorrCholeskyTransform', 'CumulativeDistributionTransform', 'Dirichlet', 'Distributio
n', 'ExpTransform', 'Exponential', 'ExponentialFamily', 'FisherSnedecor', 'Gamma', 'Geometric', 'Gumbel', 'HalfCauchy', 'Hal
fNormal', 'Independent', 'IndependentTransform', 'Kumaraswamy', 'LKJCholesky', 'Laplace', 'LogNormal', 'LogisticNormal', 'Lo
wRankMultivariateNormal', 'LowerCholeskyTransform', 'MixtureSameFamily', 'Multinomial', 'MultivariateNormal', 'NegativeBinom
ial', 'Normal', 'OneHotCategorical', 'OneHotCategoricalStraightThrough', 'Pareto', 'Poisson', 'PositiveDefiniteTransform',
'PowerTransform', 'RelaxedBernoulli', 'RelaxedOneHotCategorical', 'ReshapeTransform', 'SigmoidTransform', 'SoftmaxTransfor
m', 'SoftplusTransform', 'StackTransform', 'StickBreakingTransform', 'StudentT', 'TanhTransform', 'Transform', 'TransformedD
istribution', 'Uniform', 'VonMises', 'Weibull', 'Wishart', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__path__', '__spec__', 'bernoulli', 'beta', 'biject_to', 'binomial', 'categorica
l', 'cauchy', 'chi2', 'constraint_registry', 'constraints', 'continuous_bernoulli', 'dirichlet', 'distribution', 'exp_famil
y', 'exponential', 'fishersnedecor', 'gamma', 'geometric', 'gumbel', 'half_cauchy', 'half_normal', 'identity_transform', 'in
dependent', 'kl', 'kl_divergence', 'kumaraswamy', 'laplace', 'lkj_cholesky', 'log_normal', 'logistic_normal', 'lowrank_multi
variate_normal', 'mixture_same_family', 'multinomial', 'multivariate_normal', 'negative_binomial', 'normal', 'one_hot_catego
rical', 'pareto', 'poisson', 'register_kl', 'relaxed_bernoulli', 'relaxed_categorical', 'studentT', 'transform_to', 'transfo
rmed_distribution', 'transforms', 'uniform', 'utils', 'von_mises', 'weibull', 'wishart']
```

Documentation

- Generally, we can ignore functions that start and end with __ (special objects in Python) or functions that start with a single _ (usually internal functions).
- Based on the remaining function or attribute names, we might hazard a guess that this module offers various methods for generating random numbers, including sampling from the uniform distribution (*uniform*), normal distribution (*normal*), and multinomial distribution (*multinomial*).

Documentation

- For more specific instructions on how to use a given function or class, we can invoke the help function. As an example, let's explore the usage instructions for tensors' *ones* function.



```
help(torch.ones)
```

```
Help on built-in function ones in module torch:

ones(...)
    ones(*size, *, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False) -> Tensor

    Returns a tensor filled with the scalar value `1`, with the shape defined
    by the variable argument :attr:`size`.

    Args:
        size (int...): a sequence of integers defining the shape of the output tensor.
            Can be a variable number of arguments or a collection like a list or tuple.

    Keyword arguments:
        out (Tensor, optional): the output tensor.
        dtype (:class:`torch.dtype`, optional): the desired data type of returned tensor.
            Default: if ``None``, uses a global default (see :func:`torch.set_default_tensor_type`).
        layout (:class:`torch.layout`, optional): the desired layout of returned Tensor.
            Default: ``torch.strided``.
        device (:class:`torch.device`, optional): the desired device of returned tensor.
            Default: if ``None``, uses the current device for the default tensor type
            (see :func:`torch.set_default_tensor_type`). :attr:`device` will be the CPU
            for CPU tensor types and the current CUDA device for CUDA tensor types.
        requires_grad (bool, optional): If autograd should record operations on the
            returned tensor. Default: ``False``.

    Example::

        >>> torch.ones(2, 3)
        tensor([[ 1.,  1.,  1.],
                [ 1.,  1.,  1.]])

        >>> torch.ones(5)
```

Documentation

- In the Jupyter notebook, we can use ? to display the document in another window. For example, list? will create content that is almost identical to help(list), displaying it in a new browser window. In addition, if we use two question marks, such as list??, the Python code implementing the function will also be displayed.