Einops:
Easy tensor manipulation

Alberto Albiol

UNIVERSIDAD
POLITECNICA
DE VALENCIA

Contents

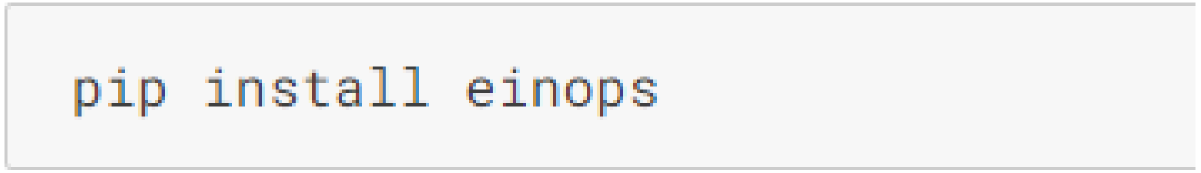
Common tensor operations

- Reshape:
  - Ej. Flatten tensor before fully connected layer
- Reduction:
  - Ej. Take max, mean,min… in Pooling layers
- Dot product (einsum)
  - Linear layers

EINOPS library makes all these operations easy and INTUITIVE-> Reduce errors

Installation

- Simple installation with pip:



```
pip install einops
```

- Can be used for numpy, pytorch, tensorflow
- Same behaviour in all platforms!

Rearrange

- Example: convert *channels first* to *channels last*
- Traditional method uses **transpose**:

```
y = x.transpose(0, 2, 3, 1)
```

- Not very intuitive!

Rearrange

- Using einops:

```
from einops import rearrange
```

```
z = torch.randn([10,3,224,224]) # batch of 10 224x224 color images (channels first)
print(z.shape)
```

```
torch.Size([10, 3, 224, 224])
```

```
z1 = rearrange(z,'b c w h -> b h w c')
print(z1.shape)
```

```
torch.Size([10, 224, 224, 3])
```

- You can use longer names for dimensions: Ej. color, RBG, batch...

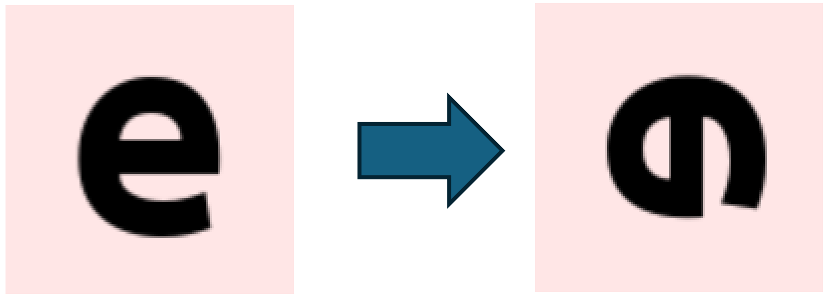
Rearrange examples

```
ims = numpy.load('./resources/test_images.npy', allow_pickle=False)
# There are 6 images of shape 96x96 with 3 color channels packed into tensor
print(ims.shape, ims.dtype)
```

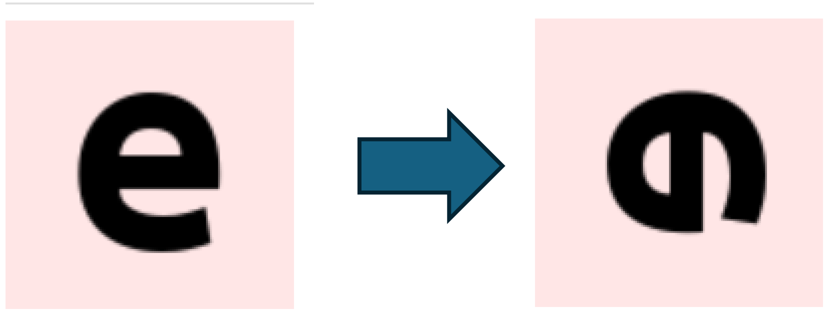
```
(6, 96, 96, 3) float64
```

```
# display the first image (whole 4d tensor can't be rendered)
ims[0]
```

Transpose images

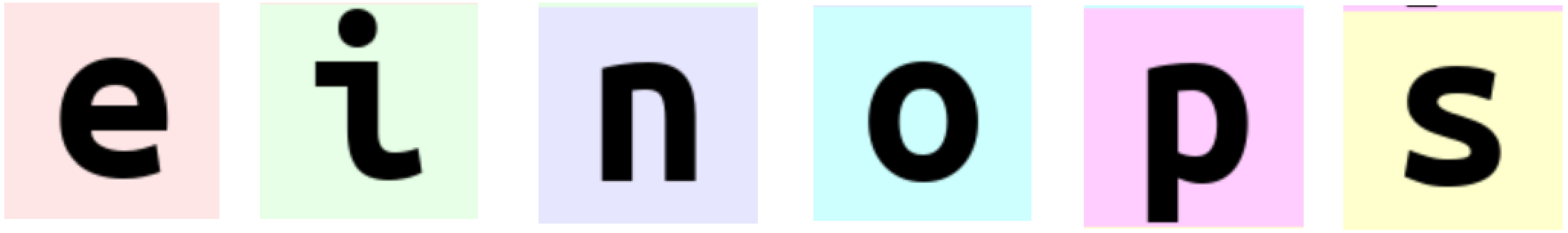


Transpose images

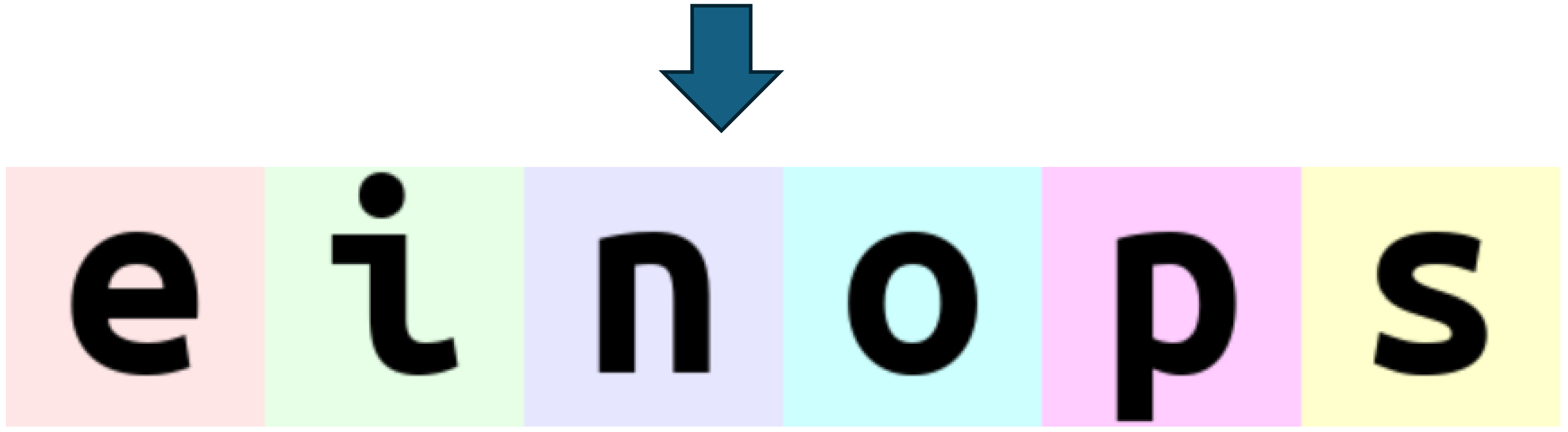


```
# rearrange, as its name suggests, rearranges elements
# below we swapped height and width.
# In other words, transposed first two axes (dimensions)
rearrange(ims[0], 'h w c -> w h c')
```

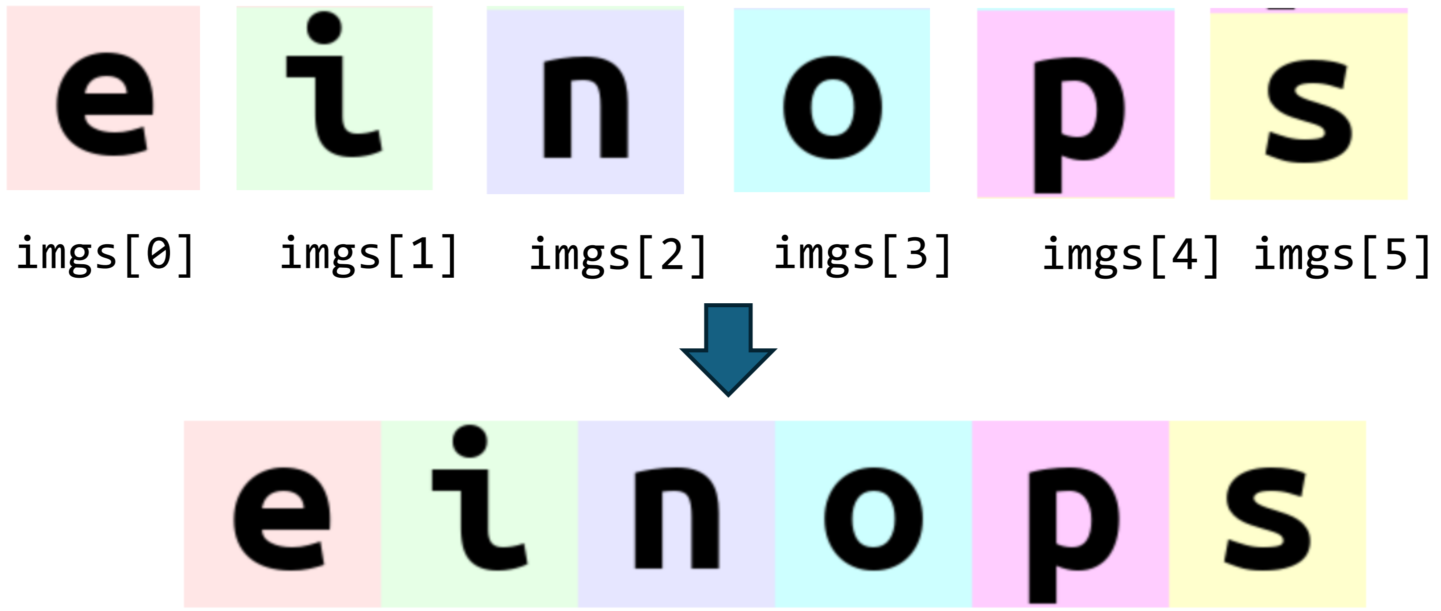
Tile images



imgs[0] imgs[1] imgs[2] imgs[3] imgs[4] imgs[5]

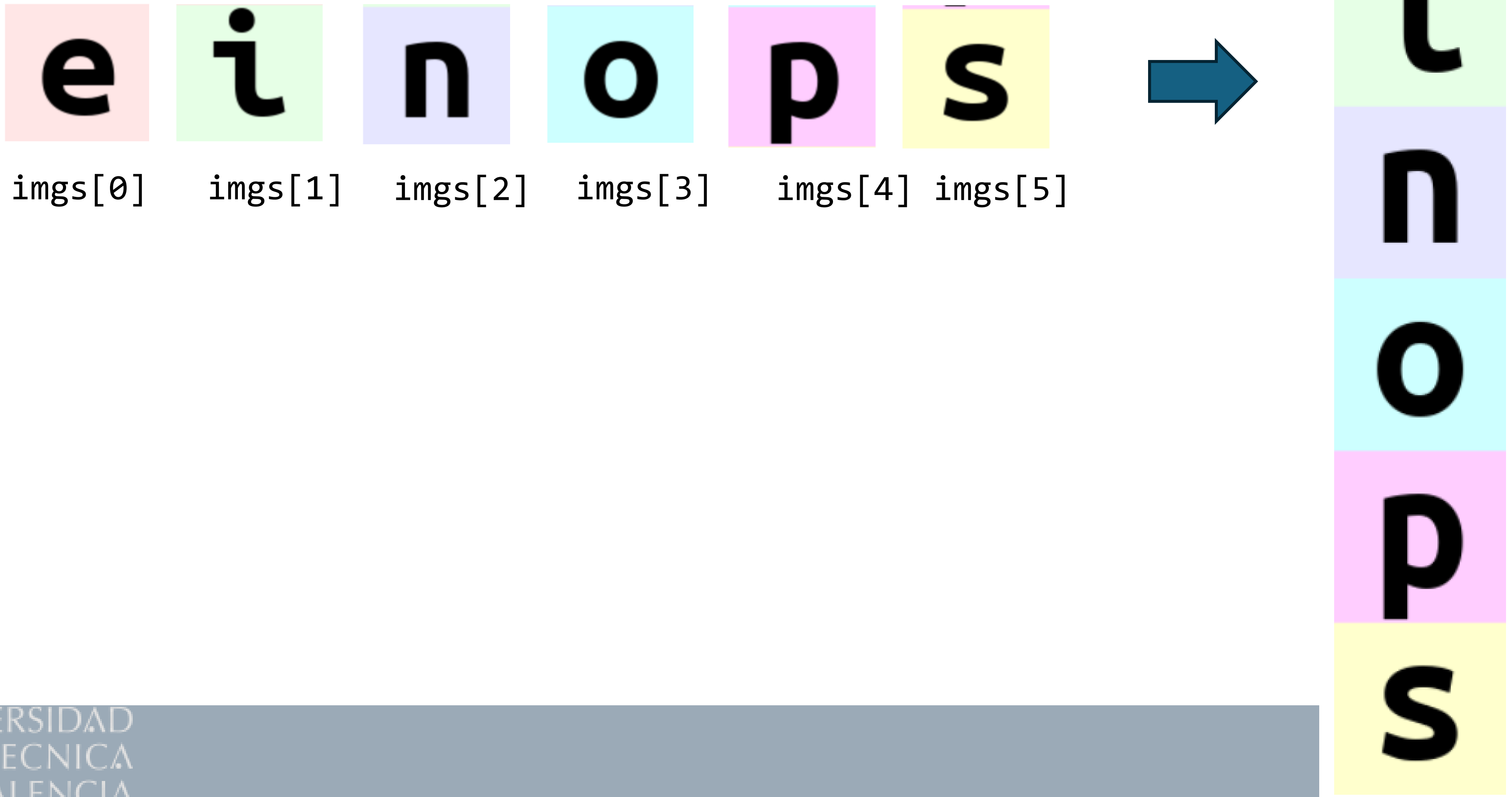


Tile images

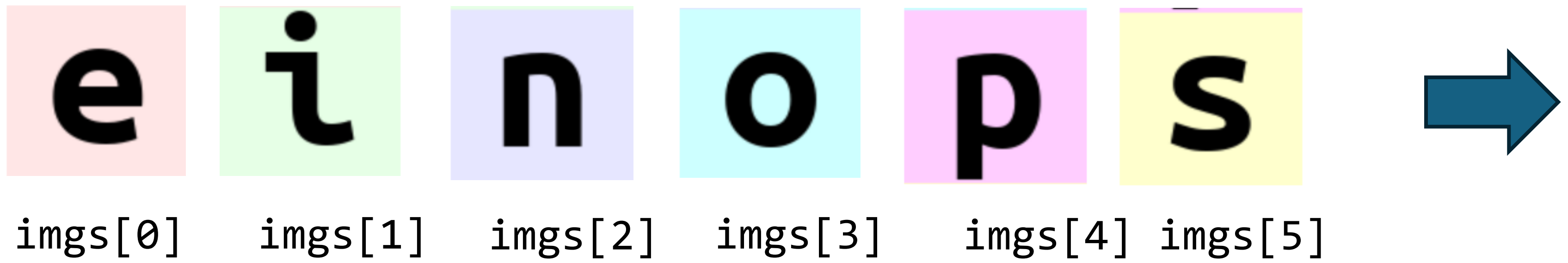


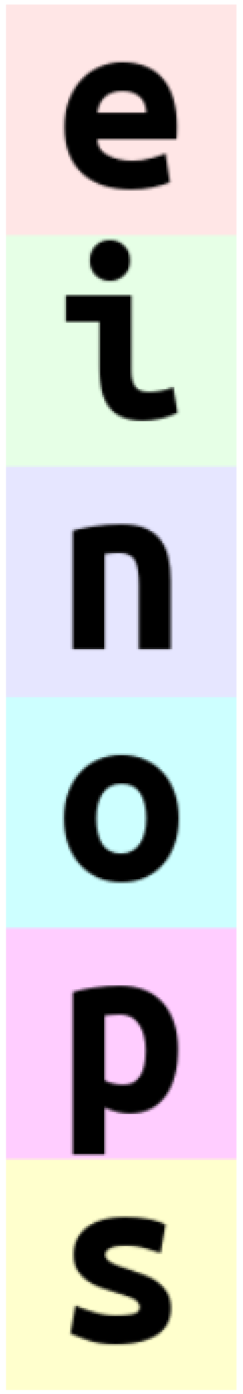
```
# resulting dimensions are computed very simply
# length of newly composed axis is a product of components
# [6, 96, 96, 3] -> [96, (6 * 96), 3]
rearrange(ims, 'b h w c -> h (b w) c').shape
```

Tile images



Tile images





```
# einops allows seamlessly composing batch and height to a new height dimension
# We just rendered all images by collapsing to 3d tensor!
rearrange(ims, 'b h w c -> (b h) w c')
```

Advanced tiling

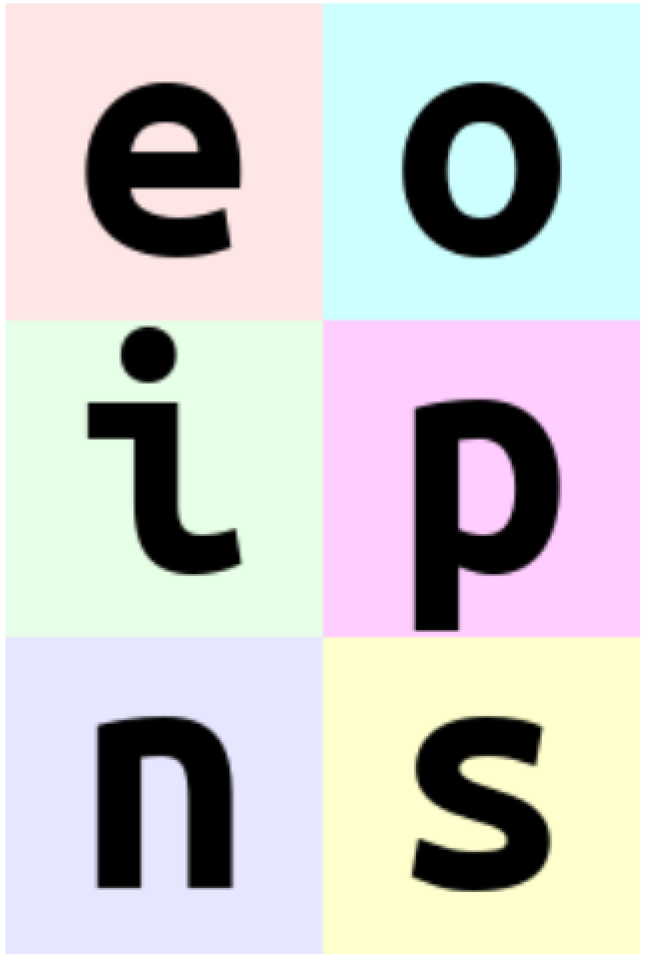
Advanced tiling

```
# decomposition is the inverse process - represent an axis as a combination of new axes
# several decompositions possible, so b1=2 is to decompose 6 to b1=2 and b2=3
rearrange(ims, '(b1 b2) h w c -> b1 b2 h w c ', b1=2).shape
```

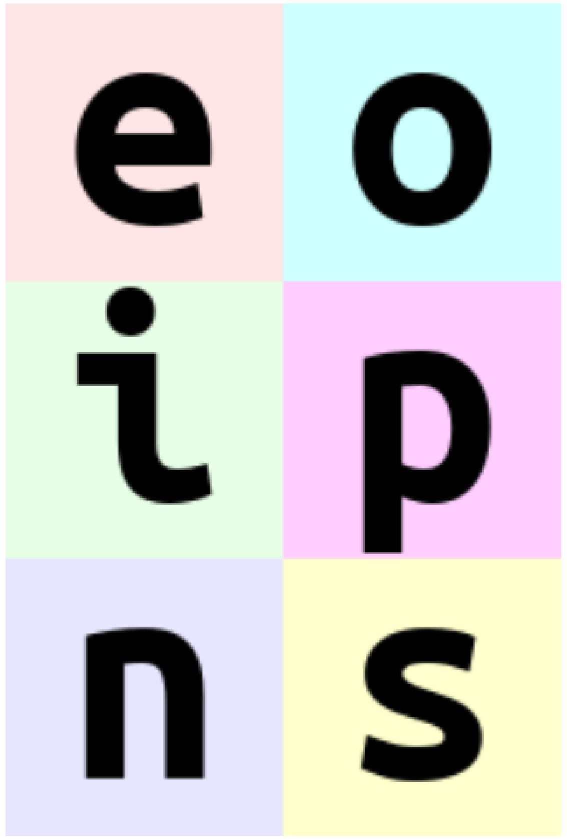
```
(2, 3, 96, 96, 3)
```

```
# finally, combine composition and decomposition:
rearrange(ims, '(b1 b2) h w c -> (b1 h) (b2 w) c ', b1=2)
```

Advanced tiling



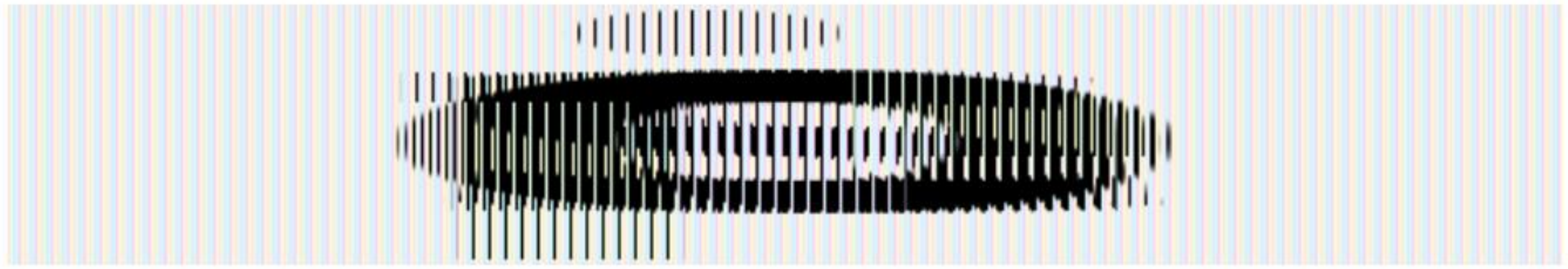
Advanced tiling



```
# slightly different composition: b1 is merged with width, b2 with height
# ... so letters are ordered by w then by h
rearrange(ims, '(b1 b2) h w c -> (b2 h) (b1 w) c ', b1=2)
```

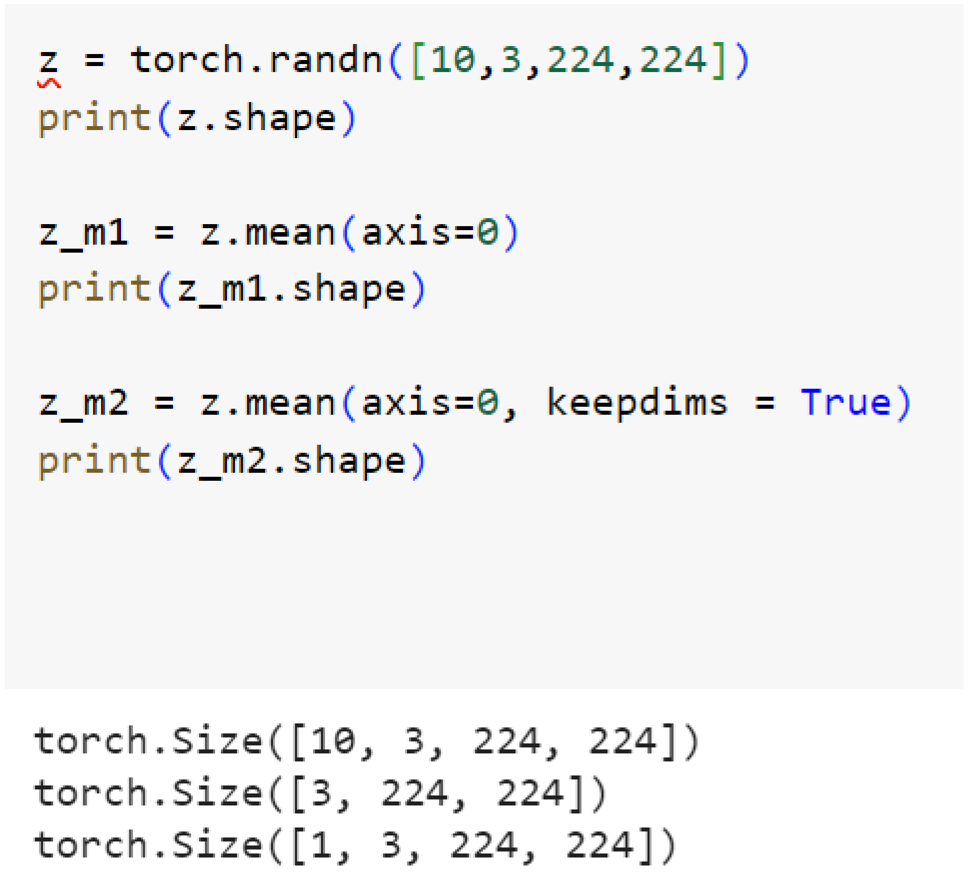
Order of axes

```
# order of axes in composition is different
# rule is just as for digits in the number: leftmost digit is the most significant,
# while neighboring numbers differ in the rightmost axis.

# you can also think of this as lexicographic sort
rearrange(ims, 'b h w c -> h (w b) c')
```



Reduce operations

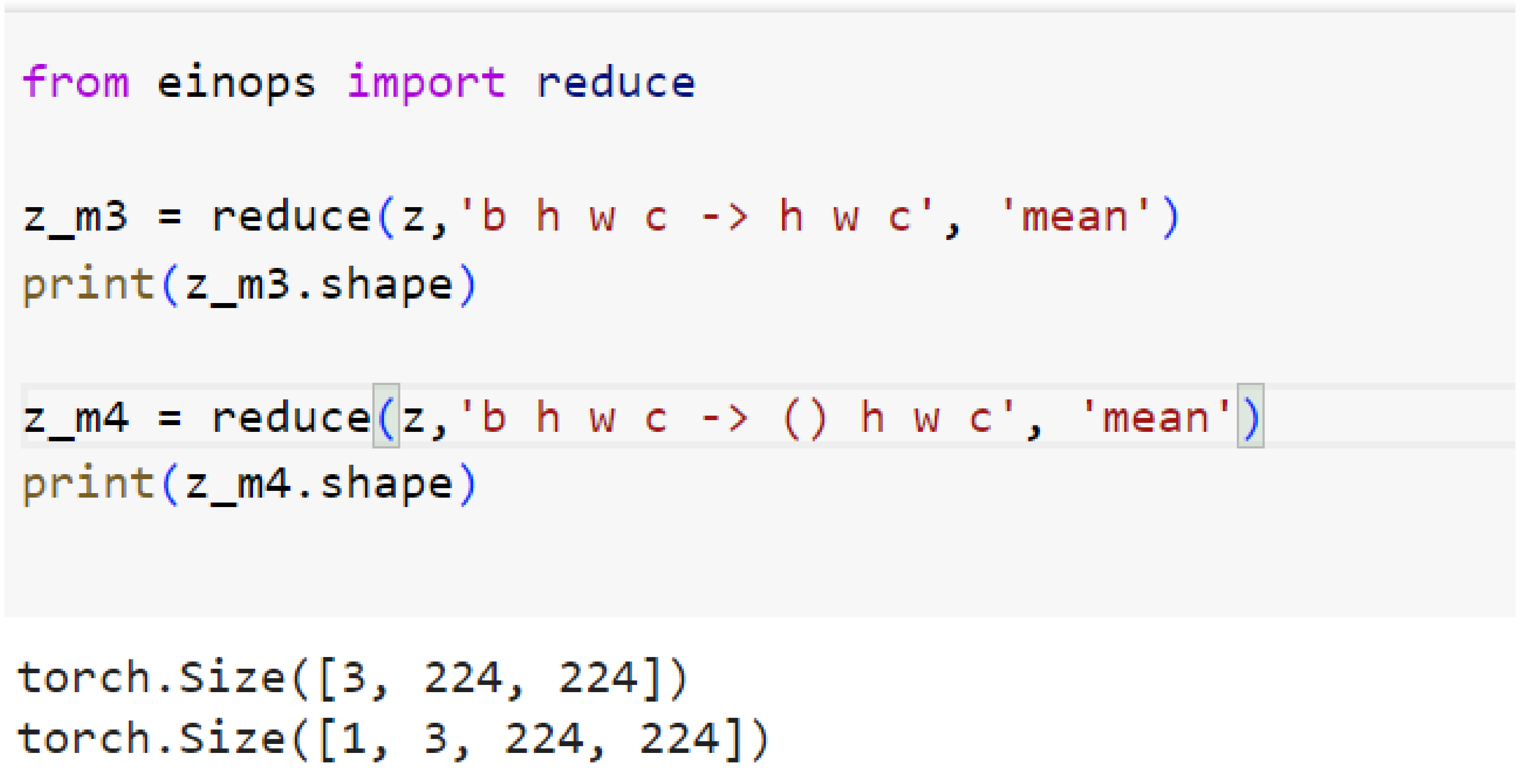
- Typical reduce operations are:
  - mean, min, max, median…
- If not forced these operation remove axes



```
z = torch.randn([10,3,224,224])
print(z.shape)

z_m1 = z.mean(axis=0)
print(z_m1.shape)

z_m2 = z.mean(axis=0, keepdims = True)
print(z_m2.shape)
```

```
torch.Size([10, 3, 224, 224])
torch.Size([3, 224, 224])
torch.Size([1, 3, 224, 224])
```

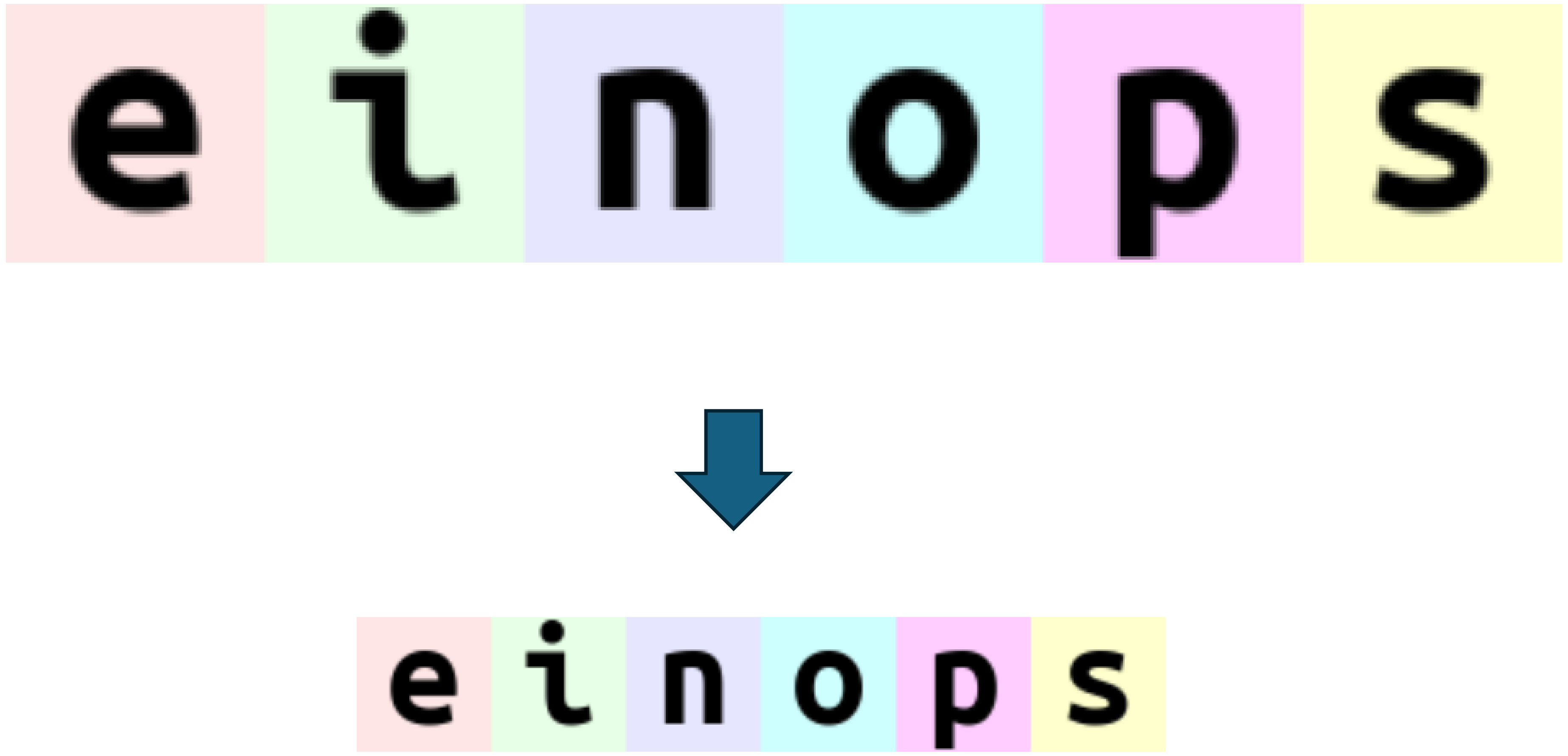
Reduce with einops



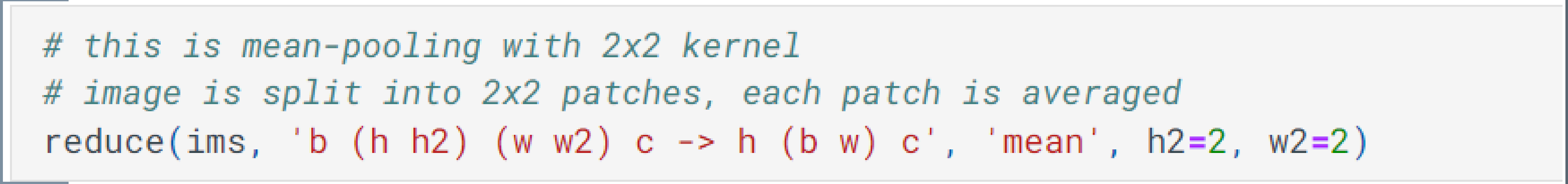
```
from einops import reduce

z_m3 = reduce(z,'b h w c -> h w c', 'mean')
print(z_m3.shape)

z_m4 = reduce(z,'b h w c -> () h w c', 'mean')
print(z_m4.shape)
```

```
torch.Size([3, 224, 224])
torch.Size([1, 3, 224, 224])
```

Pooling with reduce



Pooling with reduce


```
# this is mean-pooling with 2x2 kernel
# image is split into 2x2 patches, each patch is averaged
reduce(ims, 'b (h h2) (w w2) c -> h (b w) c', 'mean', h2=2, w2=2)
```

Einsum

- A fundamental operation in deep learning is dot product (between vectors, matrices)

$$(A \cdot B)_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

- Einstein notation simplifies writing:

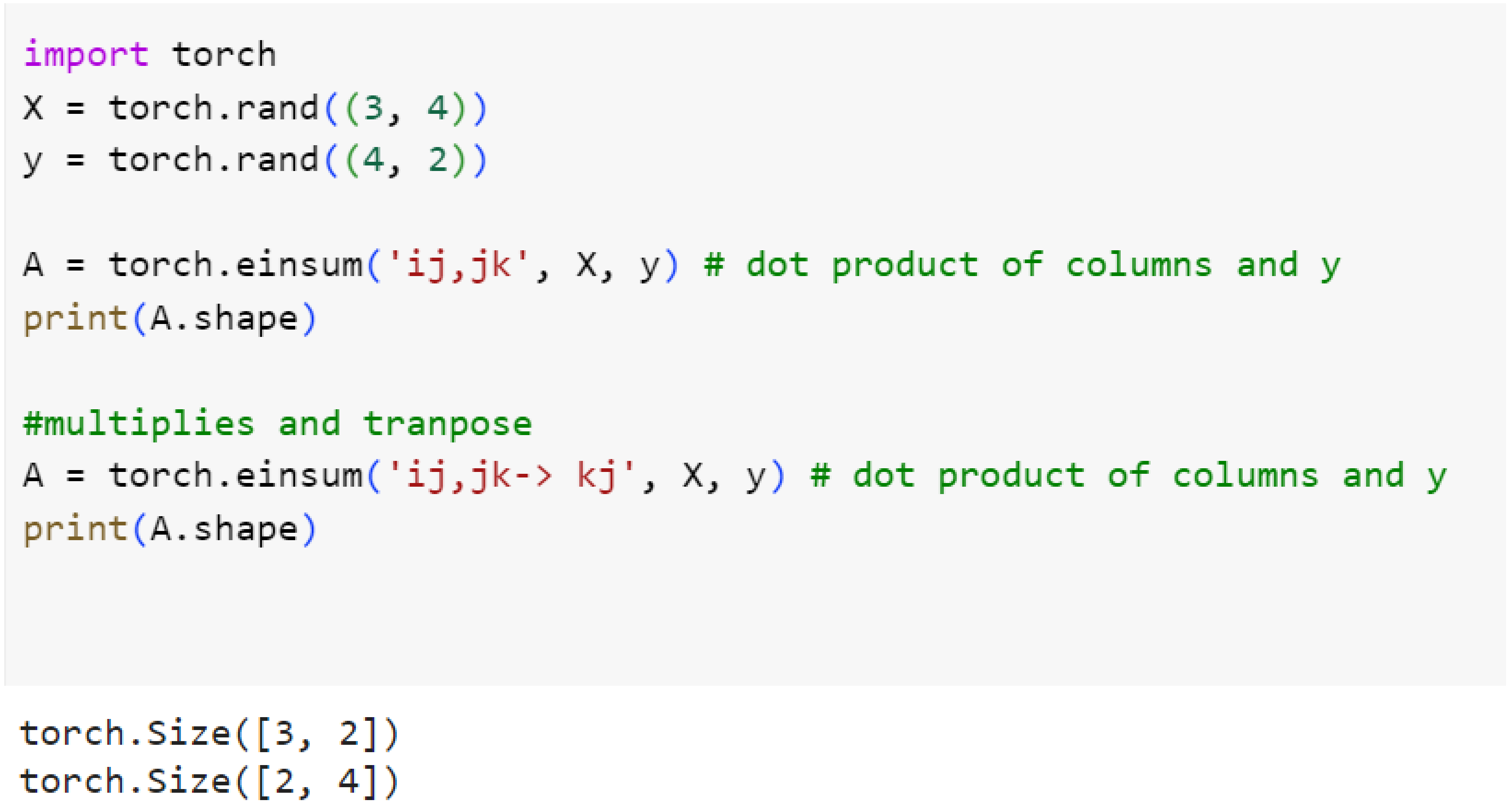
$$(A \cdot B)_{ij} = A_{ik} \cdot B_{kj}$$

- Missing dimensions are summed up

Matrix-vector multiplication

```
import torch
X = torch.rand((3, 3))
y = torch.rand( 3)

A = torch.einsum('ij,i -> j', X, y) # dot product of columns and y
A = torch.einsum('ij,j -> i', X, y) # dot product of rows and y

#simplified version
A = torch.einsum('ij,i', X, y) # dot product of columns and y
A = torch.einsum('ij,j', X, y) # dot product of rows and y
```

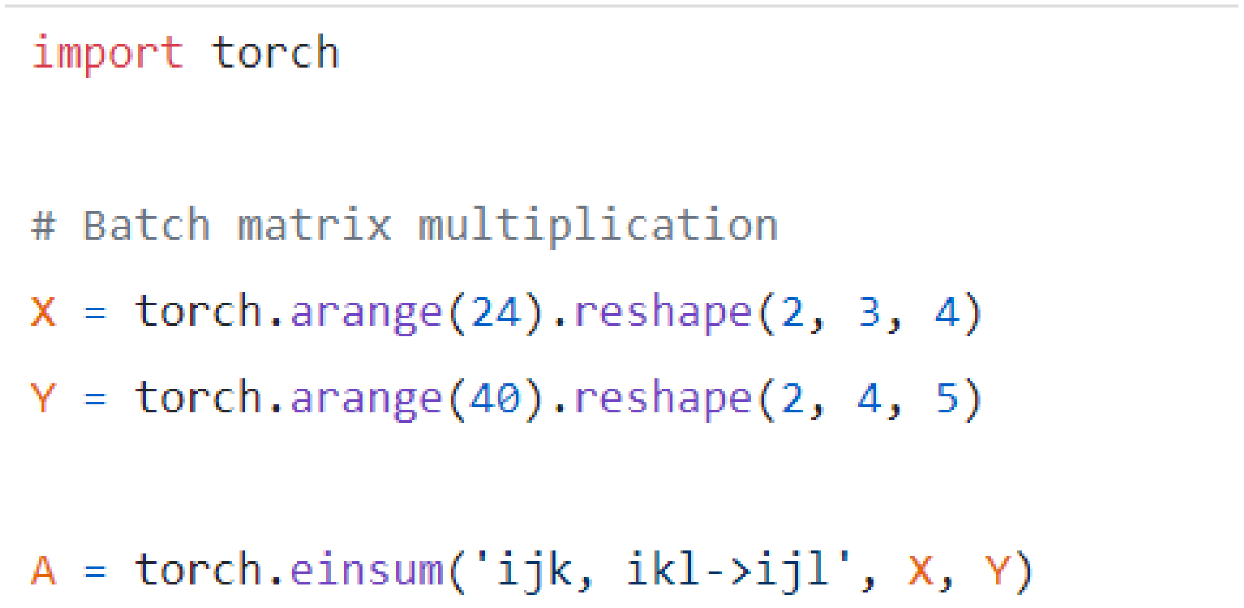
Matrix-matrix multiplication



```python
import torch
X = torch.rand((3, 4))
y = torch.rand((4, 2))

A = torch.einsum('ij,jk', X, y) # dot product of columns and y
print(A.shape)

#multiplies and tranpose
A = torch.einsum('ij,jk-> kj', X, y) # dot product of columns and y
print(A.shape)
```

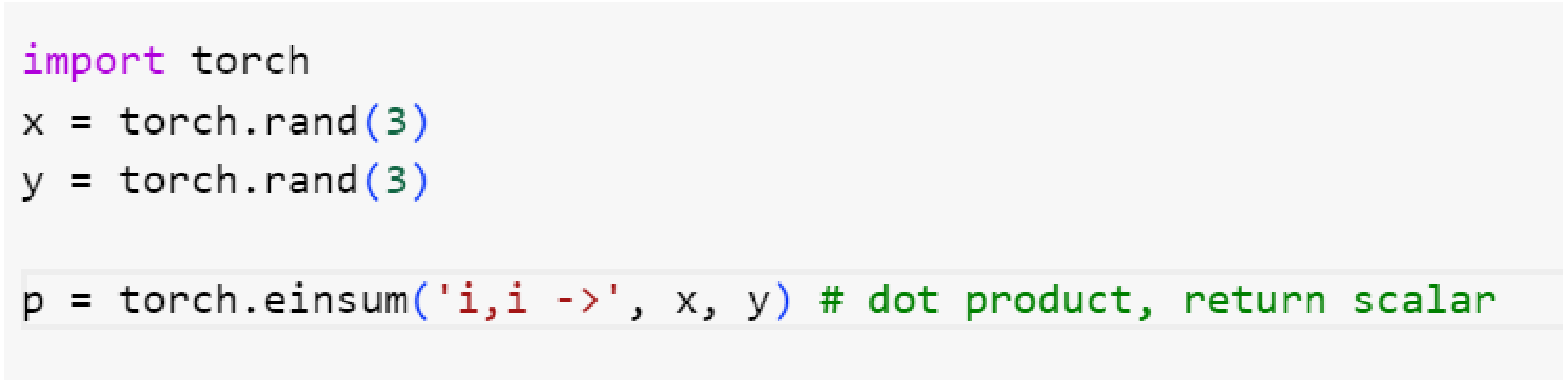
```
torch.Size([3, 2])
torch.Size([2, 4])
```

Batch matrix multiplication



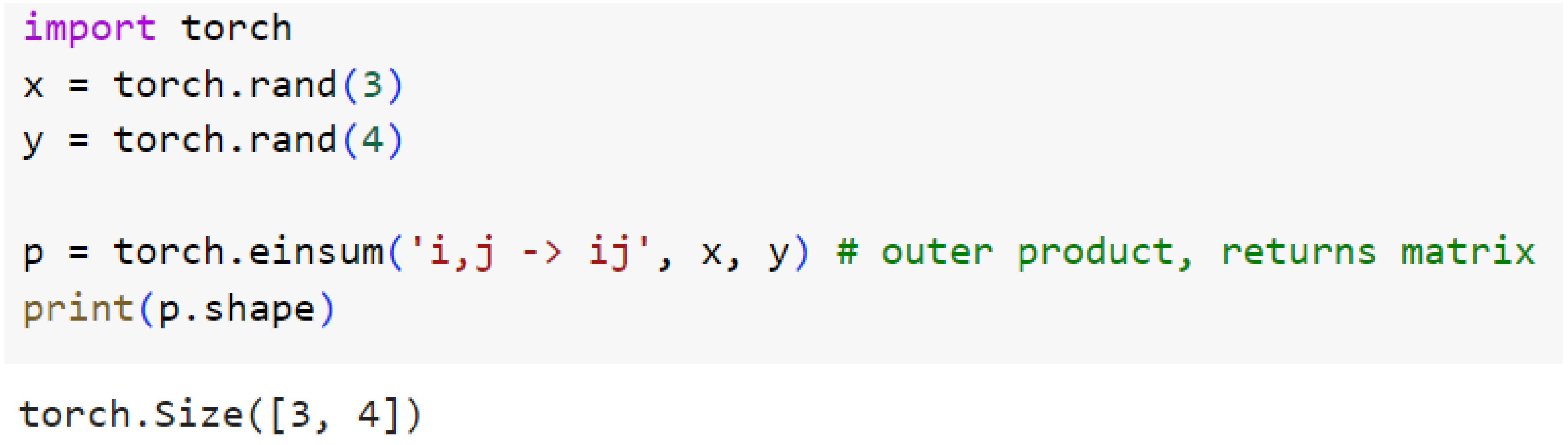
```python
import torch


# Batch matrix multiplication
X = torch.arange(24).reshape(2, 3, 4)
Y = torch.arange(40).reshape(2, 4, 5)

A = torch.einsum('ijk, ikl->ijl', X, Y)
```

Dot product



```python
import torch
x = torch.rand(3)
y = torch.rand(3)

p = torch.einsum('i,i ->', x, y) # dot product, return scalar
```

Outer product

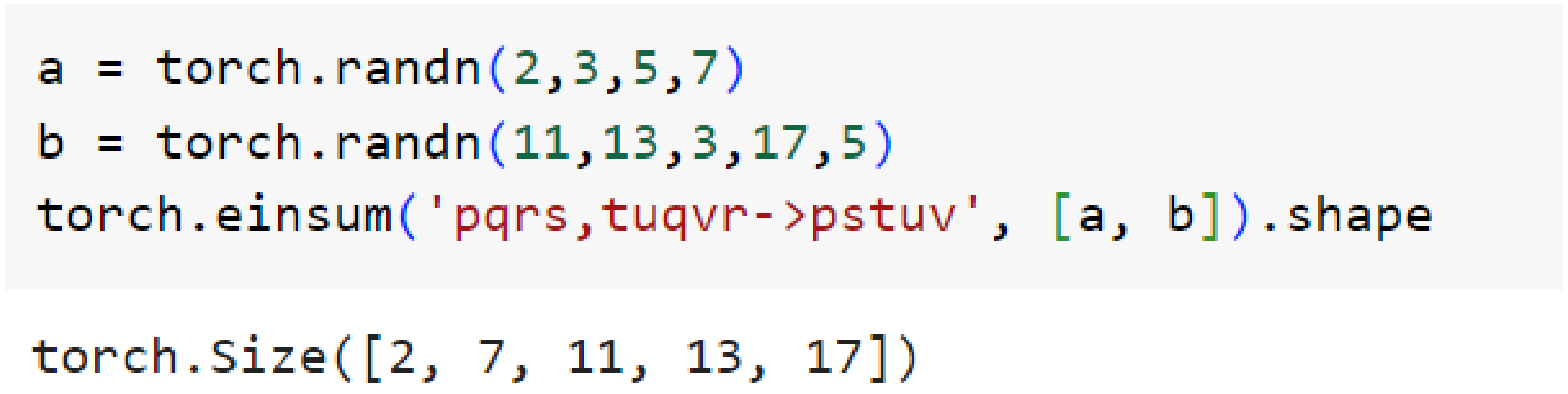


```python
import torch
x = torch.rand(3)
y = torch.rand(4)

p = torch.einsum('i,j -> ij', x, y) # outer product, returns matrix
print(p.shape)
```

```
torch.Size([3, 4])
```

Advanced einsum: tensor contractor

$$C_{pstuv} = \sum_q \sum_r A_{pqrs} B_{tuqvr} = A_{pqrs} B_{tuqvr}$$



```
a = torch.randn(2,3,5,7)
b = torch.randn(11,13,3,17,5)
torch.einsum('pqrs,tuqvr->pstuv', [a, b]).shape
```

```
torch.Size([2, 7, 11, 13, 17])
```