# Introduction to Tensors

Run in Google Colab (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/tensor.

```
t tensorflow as tf
t numpy as np
```

Tensors are multi-dimensional arrays with a uniform type (called a **`dtype`**). You can see all supported **`dtypes`** at **`tf.dtypes.DType`** (/api_docs/python/tf/dtypes/DType).

If you're familiar with NumPy (https://numpy.org/devdocs/user/quickstart.html), tensors are (kind of) like **`np.arrays`**.

All tensors are immutable like python numbers and strings: you can never update the contents of a tensor, only create a new one.

## Basics

Let's create some basic tensors.

Here is a "scalar" or "rank-0" tensor . A scalar contains a single value, and no "axes".

```
s will be an int32 tensor by default; see "dtypes" below.
0_tensor = tf.constant(4)
(rank_0_tensor)
```

```
nsor(4, shape=(), dtype=int32)
```



A "vector" or "rank-1" tensor is like a list of values. A vector has 1 axis:

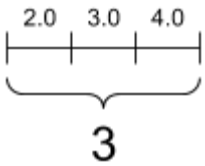
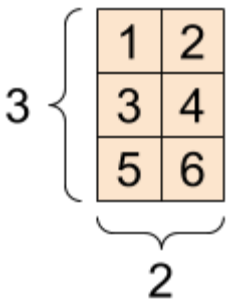
```
's make this a float tensor.
1_tensor = tf.constant([2.0, 3.0, 4.0])
(rank_1_tensor)
```

```
nsor([2. 3. 4.], shape=(3,), dtype=float32)
```

A "matrix" or "rank-2" tensor has 2-axes:

```
we want to be specific, we can set the dtype (see below) at creation time
2_tensor = tf.constant([[1, 2],
                        [3, 4],
                        [5, 6]], dtype=tf.float16)
(rank_2_tensor)
```

```
nsor(
2.]
4.]
6.]], shape=(3, 2), dtype=float16)
```

| A scalar, shape: [ ] | A vector, shape: [3] | A matrix, shape: [3, 2] |
| --- | --- | --- |
|  |  |  |



Tensors may have more axes, here is a tensor with 3-axes:



```
re can be an arbitrary number of
s (sometimes called "dimensions")
3_tensor = tf.constant([
```

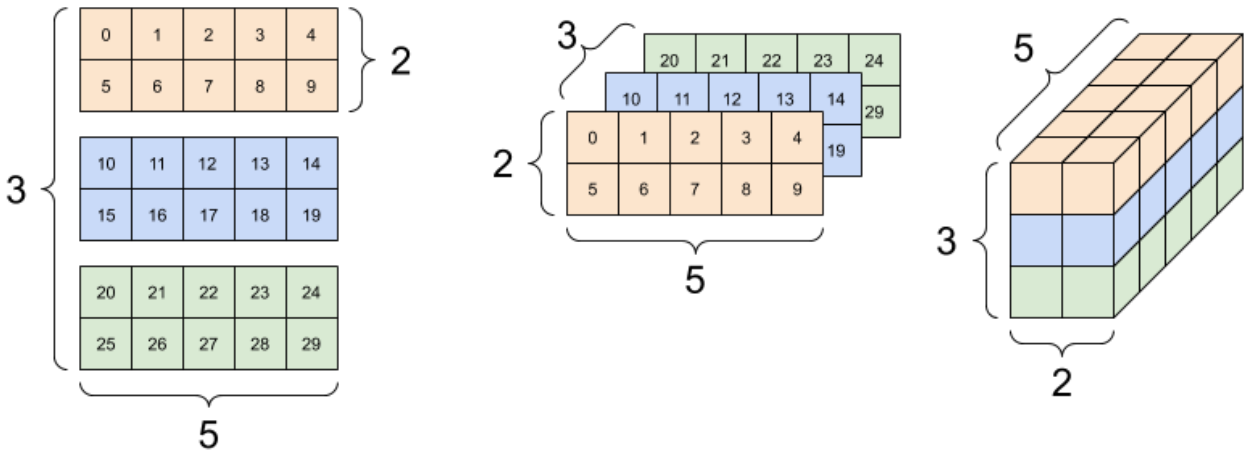
```
, 1, 2, 3, 4],
, 6, 7, 8, 9]],
 0, 11, 12, 13, 14],
 5, 16, 17, 18, 19]],
 0, 21, 22, 23, 24],
 5, 26, 27, 28, 29]],])

(rank_3_tensor)
```

```
nsor(
  1  2  3  4]
  6  7  8  9]]

 11 12 13 14]
 16 17 18 19]]

 21 22 23 24]
 26 27 28 29]]], shape=(3, 2, 5), dtype=int32)
```

There are many ways you might visualize a tensor with more than 2-axes.

---

**A 3-axis tensor, shape: `[3, 2, 5]`**




---

You can convert a tensor to a NumPy array either using `np.array` or the `tensor.numpy` method:



```
ray(rank_2_tensor)
```

```
'([[1., 2.],
  [3., 4.],
  [5., 6.]], dtype=float16)
```

```
.2_tensor.numpy()
```

```
'([[1., 2.],
  [3., 4.],
  [5., 6.]], dtype=float16)
```

Tensors often contain floats and ints, but have many other types, including:

- complex numbers
- strings

The base **`tf.Tensor`** (/api_docs/python/tf/Tensor) class requires tensors to be "rectangular"---that is, along each axis, every element is the same size. However, there are specialized types of Tensors that can handle different shapes:

- ragged (see RaggedTensor (#ragged_tensors) below)
- sparse (see SparseTensor (#sparse_tensors) below)

We can do basic math on tensors, including addition, element-wise multiplication, and matrix multiplication.

```
:f.constant([[1, 2],
             [3, 4]])
:f.constant([[1, 1],
             [1, 1]]) # Could have also said `tf.ones([2,2])`

:(tf.add(a, b), "\n")
:(tf.multiply(a, b), "\n")
:(tf.matmul(a, b), "\n")
```



```
:nsor(
:]
```

```
]], shape=(2, 2), dtype=int32)

nsor(
]
]], shape=(2, 2), dtype=int32)

nsor(
]
]], shape=(2, 2), dtype=int32)
```

```
(a + b, "\n") # element-wise addition
(a * b, "\n") # element-wise multiplication
(a @ b, "\n") # matrix multiplication
```

```
nsor(
]
]], shape=(2, 2), dtype=int32)

nsor(
]
]], shape=(2, 2), dtype=int32)

nsor(
]
]], shape=(2, 2), dtype=int32)
```

Tensors are used in all kinds of operations (ops).

```
f.constant([[4.0, 5.0], [10.0, 1.0]])

d the largest value
(tf.reduce_max(c))
d the index of the largest value
(tf.argmax(c))
pute the softmax
(tf.nn.softmax(c))
```

```
nsor(10.0, shape=(), dtype=float32)
nsor([1 0], shape=(2,), dtype=int64)
nsor(
894143e-01 7.3105860e-01]
987662e-01 1.2339458e-04]], shape=(2, 2), dtype=float32)
```

## About shapes

Tensors have shapes. Some vocabulary:

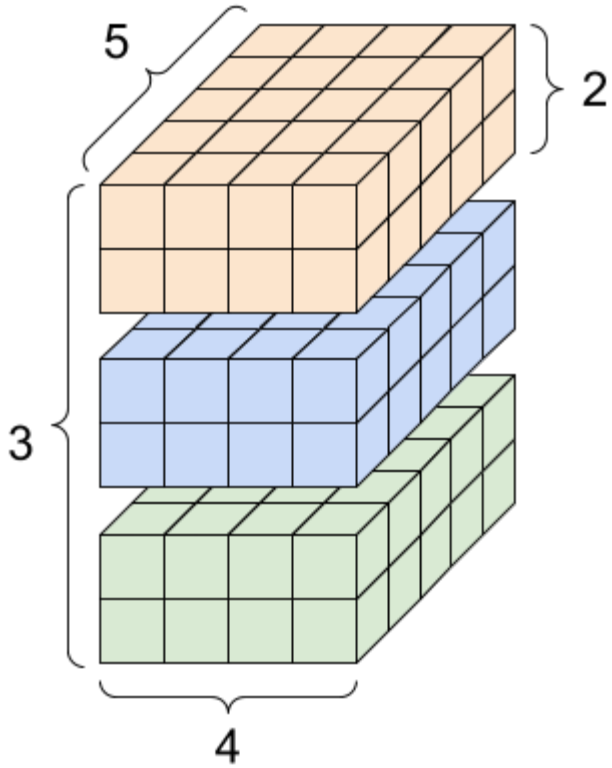
- **Shape**: The length (number of elements) of each of the dimensions of a tensor.
- **Rank**: Number of tensor dimensions. A scalar has rank 0, a vector has rank 1, a matrix is rank 2.
- **Axis** or **Dimension**: A particular dimension of a tensor.
- **Size**: The total number of items in the tensor, the product shape vector

Although you may see reference to a "tensor of two dimensions", a rank-2 tensor does not usually describ

Tensors and `tf.TensorShape` (/api_docs/python/tf/TensorShape) objects have convenient properties for accessing these:

```
4_tensor = tf.zeros([3, 2, 4, 5])
```

**A rank-4 tensor, shape: `[3, 2, 4, 5]`**



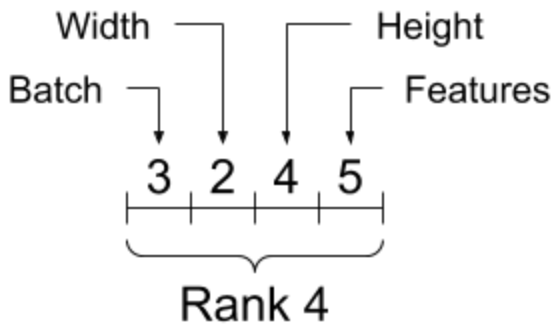

```
("Type of every element:", rank_4_tensor.dtype)
("Number of dimensions:", rank_4_tensor.ndim)
("Shape of tensor:", rank_4_tensor.shape)
("Elements along axis 0 of tensor:", rank_4_tensor.shape[0])
("Elements along the last axis of tensor:", rank_4_tensor.shape[-1])
("Total number of elements (3*2*4*5): ", tf.size(rank_4_tensor).numpy())
```

```
of every element: <dtype: 'float32'>
r of dimensions: 4
 of tensor: (3, 2, 4, 5)
nts along axis 0 of tensor: 3
nts along the last axis of tensor: 5
 number of elements (3*2*4*5):  120
```

While axes are often referred to by their indices, you should always keep track of the meaning of each. Often axes are ordered from global to local: The batch axis first, followed by spatial dimensions, and features for each location last. This way feature vectors are contiguous regions of memory.



**Typical axis order**




# Indexing

## Single-axis indexing

TensorFlow follow standard python indexing rules, similar to indexing a list or a string in python (https://docs.python.org/3/tutorial/introduction.html#strings), and the bacic rules for numpy indexing.

- indexes start at `0`
- negative indices count backwards from the end
- colons, `:`, are used for slices `start:stop:step`

```
1_tensor = tf.constant([0, 1, 1, 2, 3, 5, 8, 13, 21, 34])
(rank_1_tensor.numpy())
```

```
1  1  2  3  5  8 13 21 34]
```

Indexing with a scalar removes the dimension:

```
("First:", rank_1_tensor[0].numpy())
("Second:", rank_1_tensor[1].numpy())
("Last:", rank_1_tensor[-1].numpy())
```



```
: 0
d: 1
```

```
34
```

Indexing with a `:` slice keeps the dimension:

```
("Everything:", rank_1_tensor[:].numpy())
("Before 4:", rank_1_tensor[:4].numpy())
("From 4 to the end:", rank_1_tensor[4:].numpy())
("From 2, before 7:", rank_1_tensor[2:7].numpy())
("Every other item:", rank_1_tensor[::2].numpy())
("Reversed:", rank_1_tensor[::-1].numpy())
```

```
thing: [ 0  1  1  2  3  5  8 13 21 34]
e 4: [0 1 1 2]
4 to the end: [ 3  5  8 13 21 34]
2, before 7: [1 2 3 5 8]
 other item: [ 0  1  3  8 21]
sed: [34 21 13  8  5  3  2  1  1  0]
```

## Multi-axis indexing

Higher rank tensors are indexed by passing multiple indices.

The single-axis exact same rules as in the single-axis case apply to each axis independently.

```
(rank_2_tensor.numpy())
```

```
2.]
4.]
6.]]
```



Passing an integer for each index the result is a scalar.

```
l out a single value from a 2-rank tensor
(rank_2_tensor[1, 1].numpy())
```

You can index using any combination integers and slices:

```
 row and column tensors
("Second row:", rank_2_tensor[1, :].numpy())
("Second column:", rank_2_tensor[:, 1].numpy())
("Last row:", rank_2_tensor[-1, :].numpy())
("First item in last column:", rank_2_tensor[0, -1].numpy())
("Skip the first row:")
(rank_2_tensor[1:, :].numpy(), "\n")
```

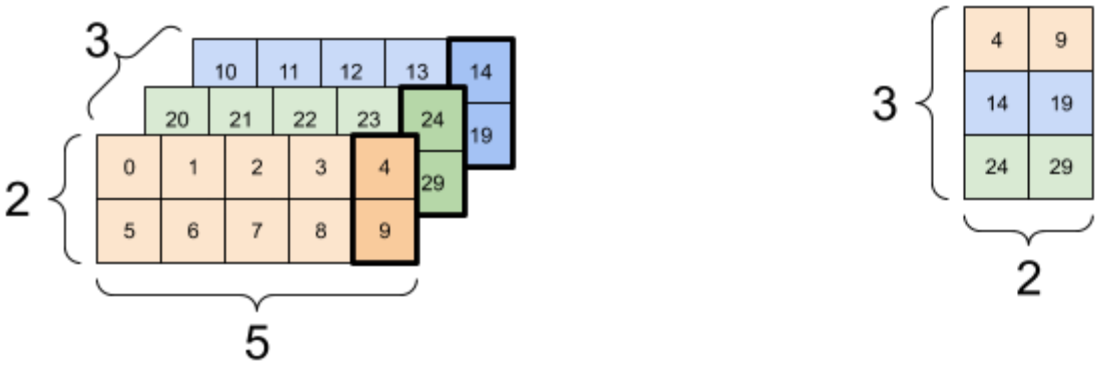
```
d row: [3. 4.]
d column: [2. 4. 6.]
 row: [5. 6.]
 item in last column: 2.0
 the first row:
 4.]
 6.]]
```

Here is an example with a 3-axis tensor:

```
(rank_3_tensor[:, :, 4])
```

```
nsor(
  9]
 19]
 29]], shape=(3, 2), dtype=int32)
```

**Selecting the last feature across all locations in each example in the batch**




# Manipulating Shapes

Reshaping a tensor is of great utility.

The **`tf.reshape`** (/api_docs/python/tf/reshape) operation is fast and cheap as the underlying data does not need to be duplicated.

```
pe returns a `TensorShape` object that shows the size on each dimension
 = tf.Variable(tf.constant([[1], [2], [3]]))
(var_x.shape)


)


 can convert this object into a Python list, too
(var_x.shape.as_list())


]
```



You can reshape a tensor into a new shape. Reshaping is fast and cheap as the underlying data does not need to be duplicated.

```
 can reshape a tensor to a new shape.
e that we're passing in a list
ped = tf.reshape(var_x, [1, 3])



(var_x.shape)
(reshaped.shape)



 )
)
```

The data maintains it's layout in memory and a new tensor is created, with the requested shape, pointing to the same data. TensorFlow uses C-style "row-major" memory ordering, where incrementing the right-most index corresponds to a single step in memory.

```
(rank_3_tensor)



nsor(
  1  2  3  4]
  6  7  8  9]]

 11 12 13 14]
 16 17 18 19]]

 21 22 23 24]
 26 27 28 29]]], shape=(3, 2, 5), dtype=int32)
```

If you flatten a tensor you can see what order it is laid out in memory.

```
-1` passed in the `shape` argument says "Whatever fits".
(tf.reshape(rank_3_tensor, [-1]))
```

```
nsor(
1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
5 26 27 28 29], shape=(30,), dtype=int32)
```

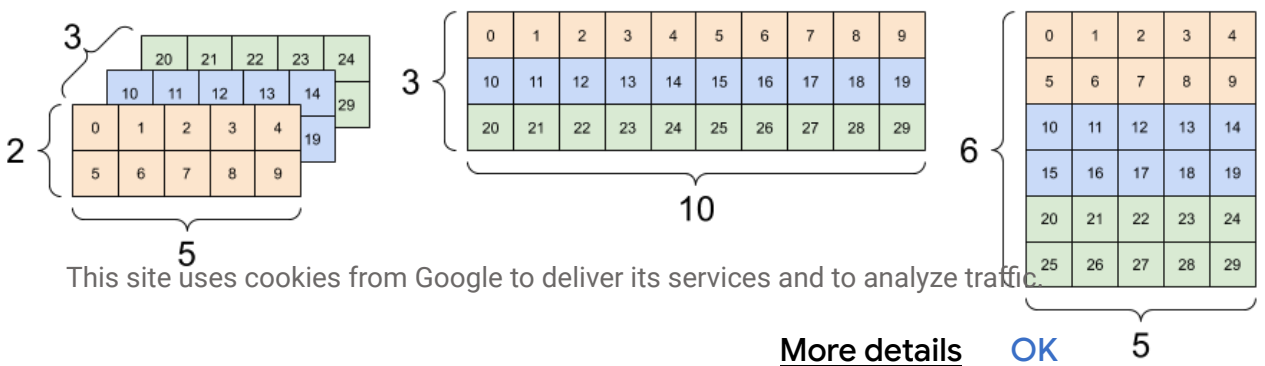
Typically the only reasonable uses of **`tf.reshape`** (/api_docs/python/tf/reshape) are to combine or split adjacent axes (or add/remove `1`s).

For this 3x2x5 tensor, reshaping to (3x2)x5 or 3x(2x5) are both reasonable things to do, as the slices do not mix:

```
(tf.reshape(rank_3_tensor, [3*2, 5]), "\n")
(tf.reshape(rank_3_tensor, [3, -1]))
```

```
nsor(
 1  2  3  4]
 6  7  8  9]
11 12 13 14]
16 17 18 19]
21 22 23 24]
26 27 28 29]], shape=(6, 5), dtype=int32)

nsor(
 1  2  3  4  5  6  7  8  9]
11 12 13 14 15 16 17 18 19]
21 22 23 24 25 26 27 28 29]], shape=(3, 10), dtype=int32)
```

**Some good reshapes.**



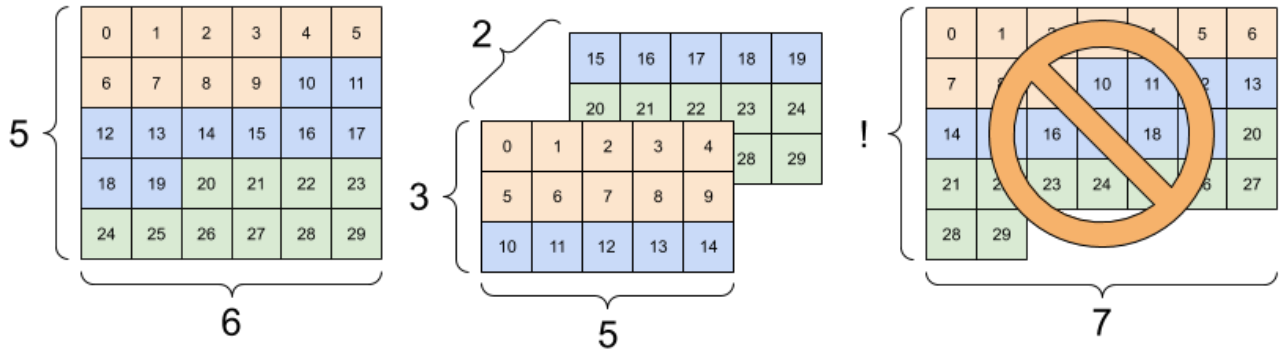
Reshaping will "work" for any new shape with the same total number of elements, but it will not do anything useful if you do not respect the order of the axes.

Swapping axes in **`tf.reshape`** (/api_docs/python/tf/reshape) does not work, you need **`tf.transpose`** (/api_docs/python/tf/transpose) for that.

```
 examples: don't do this

 can't reorder axes with reshape.
(tf.reshape(rank_3_tensor, [2, 3, 5]), "\n")

s is a mess
(tf.reshape(rank_3_tensor, [5, 6]), "\n")

s doesn't work at all

 reshape(rank_3_tensor, [7, -1])
t Exception as e:
nt(f"{type(e).__name__}: {e}")
```

```
nsor(
  1  2  3  4]
  6  7  8  9]
 11 12 13 14]]

 16 17 18 19]
 21 22 23 24]
 26 27 28 29]]], shape=(2, 3, 5), dtype=int32)

nsor(
  1  2  3  4  5]
  7  8  9 10 11]
 13 14 15 16 17]
```

**Some bad reshapes.**




You may run across not-fully-specified shapes. Either the shape contains a **None** (a dimension's length is unknown) or the shape is **None** (the rank of the tensor is unknown).

Except for tf.RaggedTensor (#ragged_tensors), this will only occur in the context of TensorFlow's, symbolic, graph-building APIs:

- tf.function (/guide/function)
- The keras functional API (/guide/keras/functional).

## More on DTypes

To inspect a **`tf.Tensor`** (/api_docs/python/tf/Tensor)'s data type use the **`Tensor.dtype`** (/api_docs/python/tf/Tensor#dtype) property.

When creating a **`tf.Tensor`** (/api_docs/python/tf/Tensor) from a Python object you may optionally specify the datatype.

If you don't, TensorFlow chooses a datatype that can represent your data. TensorFlow converts Python integers to **`tf.int32`** (/api_docs/python/tf#int32) and python floating point numbers to **`tf.float32`** (/api_docs/python/tf#float32). Otherwise TensorFlow uses the same rules NumPy uses when converting to arrays.

You can cast from type to type.

```
64_tensor = tf.constant([2.2, 3.3, 4.4], dtype=tf.float64)
16_tensor = tf.cast(the_f64_tensor, dtype=tf.float16)
, let's cast to an uint8 and lose the decimal precision
8_tensor = tf.cast(the_f16_tensor, dtype=tf.uint8)
(the_u8_tensor)
```

```
nsor([2 3 4], shape=(3,), dtype=uint8)
```

## Broadcasting

Broadcasting is a concept borrowed from the equivalent feature in NumPy (https://numpy.org/doc/stable/user/basics.html). In short, under certain conditions, smaller tensors are "stretched" automatically to fit larger tensors when running combined operations on them.

The simplest and most common case is when you attempt to multiply or add a tensor to a scalar. In that case, the scalar is broadcast to be the same shape as the other argument.

```
f.constant([1, 2, 3])

f.constant(2)
f.constant([2, 2, 2])
. of these are the same computation
(tf.multiply(x, 2))
(x * y)
(x * z)
```

```
nsor([2 4 6], shape=(3,), dtype=int32)
nsor([2 4 6], shape=(3,), dtype=int32)
nsor([2 4 6], shape=(3,), dtype=int32)
```

Likewise, 1-sized dimensions can be stretched out to match the other arguments. Both arguments can be stretched in the same computation.

In this case a 3x1 matrix is element-wise multiplied by a 1x4 matrix to produce a 3x4 matrix. Note how the leading 1 is optional: The shape of y is `[4]`.



```
se are the same computations
f.reshape(x,[3,1])
f.range(1, 5)
(x, "\n")
```

```
(y, "\n")
(tf.multiply(x, y))
```

```
nsor(
```

```
, shape=(3, 1), dtype=int32)
```

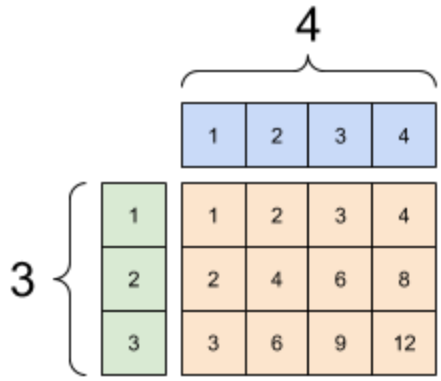
```
nsor([1 2 3 4], shape=(4,), dtype=int32)
```

```
nsor(
 2  3  4]
 4  6  8]
 6  9 12]], shape=(3, 4), dtype=int32)
```

---

**A broadcasted add: a `[3, 1]` times a `[1, 4]` gives a `[3,4]`**

---




---

Here is the same operation without broadcasting:

```
etch = tf.constant([[1, 1, 1, 1],
                    [2, 2, 2, 2],
                    [3, 3, 3, 3]])
```

```
etch = tf.constant([[1, 2, 3, 4],
                    [1, 2, 3, 4],
                    [1, 2, 3, 4]])
```



```
(x_stretch * y_stretch)  # Again, operator overloading
```

```
nsor(
 2  3  4]
 4  6  8]
 6  9 12]], shape=(3, 4), dtype=int32)
```

Most of the time, broadcasting is both time and space efficient, as the broadcast operation never materializes the expanded tensors in memory.

You see what broadcasting looks like using `tf.broadcast_to` (/api_docs/python/tf/broadcast_to).

```
(tf.broadcast_to(tf.constant([1, 2, 3]), [3, 3]))
```

```
nsor(
 3]
 3]
 3]], shape=(3, 3), dtype=int32)
```

Unlike a mathematical op, for example, `broadcast_to` does nothing special to save memory. Here, you are materializing the tensor.

It can get even more complicated. This section (https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html) of Jake VanderPlas's book *Python Data Science Handbook* shows more broadcasting tricks (again in NumPy).

## tf.convert_to_tensor

Most ops, like `tf.matmul` (/api_docs/python/tf/linalg/matmul) and `tf.reshape` (/api_docs/python/tf/reshape) take arguments of class `tf.Tensor` (/api_docs/python/tf/Tensor). However, you'll notice in the above case, we frequently pass Python objects shaped like tensors.

Most, but not all, ops call `convert_to_tensor` on non-tensor arguments. There is a registry of conversions, and most object classes like NumPy's `ndarray`, `TensorShape`, Python lists, and `tf.Variable` (/api_docs/python/tf/Variable) will all convert automatically.

See `tf.register_tensor_conversion_function` (/api_docs/python/tf/register_tensor_conversion_function) for more details, and if you have your own type you'd like to automatically convert to a tensor.

## Ragged Tensors

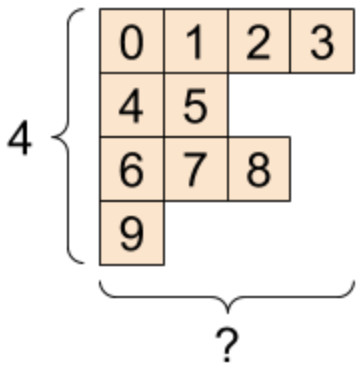
A tensor with variable numbers of elements along some axis is called "ragged". Use `tf.ragged.RaggedTensor` for ragged data.

For example, This cannot be represented as a regular tensor:

---

**A `tf.RaggedTensor` (/api_docs/python/tf/RaggedTensor), shape: `[4, None]`**

---




---

```
d_list = [
0, 1, 2, 3],
4, 5],
6, 7, 8],
9]]
```

```
sor = tf.constant(ragged_list)
t Exception as e:
nt(f"{type(e).__name__}: {e}")
```

```
Error: Can't convert non-rectangular Python sequence to Tensor.
```

Instead create a **`tf.RaggedTensor`** (/api_docs/python/tf/RaggedTensor) using **`tf.ragged.constant`** (/api_docs/python/tf/ragged/constant):

```
d_tensor = tf.ragged.constant(ragged_list)
(ragged_tensor)
```

```
aggedTensor [[0, 1, 2, 3], [4, 5], [6, 7, 8], [9]]>
```

The shape of a **`tf.RaggedTensor`** (/api_docs/python/tf/RaggedTensor) contains unknown dimensions:

```
(ragged_tensor.shape)
```

```
lone)
```

## String tensors

**`tf.string`** (/api_docs/python/tf#string) is a **`dtype`**, which is to say we can represent data as strings (variable-length byte arrays) in tensors.

The strings are atomic and cannot be indexed the way Python strings are. The length of the string is not one of the dimensions of the tensor. See **`tf.strings`** (/api_docs/python/tf/strings) for functions to manipulate them.

Here is a scalar string tensor:



```
sors can be strings, too here is a scalar string.
r_string_tensor = tf.constant("Gray wolf")
```

```
(scalar_string_tensor)
```

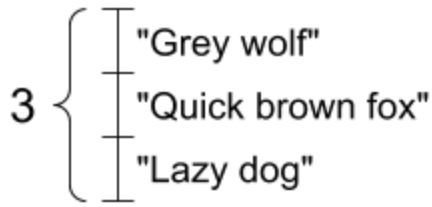
```
nsor(b'Gray wolf', shape=(), dtype=string)
```

And a vector of strings:

---

**A vector of strings, shape: [3,]**

---




---

```
we have two string tensors of different lengths, this is OK.
r_of_strings = tf.constant(["Gray wolf",
                            "Quick brown fox",
                            "Lazy dog"])
e that the shape is (2,), indicating that it is 2 x unknown.
(tensor_of_strings)
```

```
nsor([b'Gray wolf' b'Quick brown fox' b'Lazy dog'], shape=(3,), dtype=string)
```

In the above printout the **b** prefix indicates that **`tf.string`** (/api_docs/python/tf#string) dtype is not a unicode string, but a byte-string. See the Unicode Tutorial (https://www.tensorflow.org/tutorials/load_data/unicode) for more about working with unicode text in TensorFlow.

If you pass unicode characters they are utf-8 encoded.

```
nstant("😅👍")
```



```
ensor: shape=(), dtype=string, numpy=b'\xf0\x9f\xa5\xb3\xf0\x9f\x91\x8d'>
```

Some basic functions with strings can be found in **`tf.strings`** (/api_docs/python/tf/strings), including **`tf.strings.split`** (/api_docs/python/tf/strings/split).

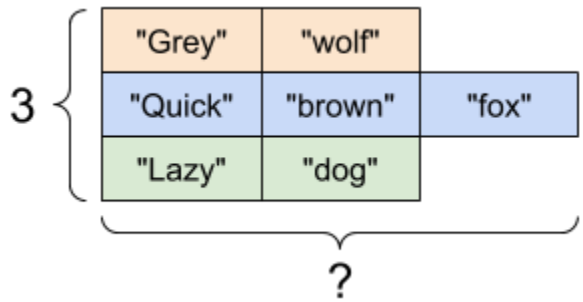
```
can use split to split a string into a set of tensors
(tf.strings.split(scalar_string_tensor, sep=" "))
```

```
nsor([b'Gray' b'wolf'], shape=(2,), dtype=string)
```

```
but it turns into a `RaggedTensor` if we split up a tensor of strings,
each string might be split into a different number of parts.
(tf.strings.split(tensor_of_strings))
```

```
aggedTensor [[b'Gray', b'wolf'], [b'Quick', b'brown', b'fox'], [b'Lazy', b'do
```

**Three strings split, shape: `[3, None]`**




And `tf.string.to_number`:

```
= tf.constant("1 10 100")
(tf.strings.to_number(tf.strings.split(text, " ")))
```

```
nsor([  1.  10. 100.], shape=(3,), dtype=float32)
```

Although you can't use **`tf.cast`** (/api_docs/python/tf/cast) to turn a string tensor into numbers, you can convert it into bytes, and then into numbers.

```
strings = tf.strings.bytes_split(tf.constant("Duck"))
ints = tf.io.decode_raw(tf.constant("Duck"), tf.uint8)
("Byte strings:", byte_strings)
("Bytes:", byte_ints)
```

```
strings: tf.Tensor([b'D' b'u' b'c' b'k'], shape=(4,), dtype=string)
: tf.Tensor([ 68 117  99 107], shape=(4,), dtype=uint8)
```

```
split it up as unicode and then decode it
de_bytes = tf.constant("アヒル 🦆")
de_char_bytes = tf.strings.unicode_split(unicode_bytes, "UTF-8")
de_values = tf.strings.unicode_decode(unicode_bytes, "UTF-8")

("\nUnicode bytes:", unicode_bytes)
("\nUnicode chars:", unicode_char_bytes)
("\nUnicode values:", unicode_values)
```

```
de bytes: tf.Tensor(b'\xe3\x82\xa2\xe3\x83\x92\xe3\x83\xab \xf0\x9f\xa6\x86',

de chars: tf.Tensor([b'\xe3\x82\xa2' b'\xe3\x83\x92' b'\xe3\x83\xab' b' ' b'\

de values: tf.Tensor([ 12450  12498  12523     32 129414], shape=(5,), dtype=
```

The **`tf.string`** (/api_docs/python/tf#string) dtype is used for all raw bytes data in TensorFlow. The **`tf.io`** (/api_docs/python/tf/io) module contains functions for converting data to and from bytes, including decoding images and parsing csv.

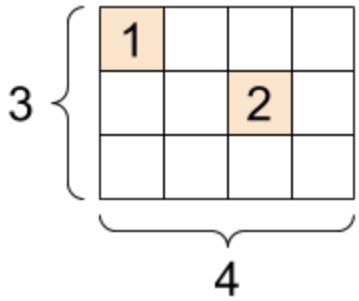
# Sparse tensors

Sometimes, your data is sparse, like a very wide embedding space. TensorFlow supports **`tf.sparse.SparseTensor`** (/api_docs/python/tf/sparse/SparseTensor) and related operations to store sparse data efficiently.

---

**A `tf.SparseTensor`** (/api_docs/python/tf/sparse/SparseTensor)**, shape: `[3, 4]`**

---




---

```
rse tensors store values by index in a memory-efficient manner
e_tensor = tf.sparse.SparseTensor(indices=[[0, 0], [1, 2]],
                                  values=[1, 2],
                                  dense_shape=[3, 4])
(sparse_tensor, "\n")

 can convert sparse tensors to dense
(tf.sparse.to_dense(sparse_tensor))
```

```
eTensor(indices=tf.Tensor(
]
]], shape=(2, 2), dtype=int64), values=tf.Tensor([1 2], shape=(2,), dtype=int

nsor(
 0 0]
 2 0]
 0 0]], shape=(3, 4), dtype=int32)
```

Last updated 2020-05-19.