tf.Tensor

Contents

Class **Tensor**

Defined in `tensorflow/python/framework/ops.py`.

See the guide: Building Graphs > Core graph data structures

Represents one of the outputs of an `Operation`.

A `Tensor` is a symbolic handle to one of the outputs of an `Operation`. It does not hold the values of that operation's output, but instead provides a means of computing those values in a TensorFlow `tf.Session`.

This class has two primary purposes:

1. A `Tensor` can be passed as an input to another `Operation`. This builds a dataflow connection between operations, which enables TensorFlow to execute an entire `Graph` that represents a large, multi-step computation.
2. After the graph has been launched in a session, the value of the `Tensor` can be computed by passing it to `tf.Session.run`. `t.eval()` is a shortcut for calling `tf.get_default_session().run(t)`.

In the following example, `c`, `d`, and `e` are symbolic `Tensor` objects, whereas `result` is a numpy array that stores a concrete value:

```
# Build a dataflow graph.
c = tf.constant([[1.0, 2.0], [3.0, 4.0]])
d = tf.constant([[1.0, 1.0], [0.0, 1.0]])
e = tf.matmul(c, d)

# Construct a `Session` to execute the graph.
sess = tf.Session()

# Execute the graph and store the value that `e` represents in `result`.
result = sess.run(e)
```

Properties

`device`

The name of the device on which this tensor will be produced, or None.

`dtype`

The `DType` of elements in this tensor.

graph

The `Graph` that contains this tensor.

name

The string name of this tensor.

op

The `Operation` that produces this tensor as an output.

shape

Returns the `TensorShape` that represents the shape of this tensor.

The shape is computed using shape inference functions that are registered in the Op for each `Operation`. See `tf.TensorShape` for more details of what a shape represents.

The inferred shape of a tensor is used to provide shape information without having to launch the graph in a session. This can be used for debugging, and providing early error messages. For example:

```
c = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])

print(c.shape)
==> TensorShape([Dimension(2), Dimension(3)])

d = tf.constant([[1.0, 0.0], [0.0, 1.0], [1.0, 0.0], [0.0, 1.0]])

print(d.shape)
==> TensorShape([Dimension(4), Dimension(2)])

# Raises a ValueError, because `c` and `d` do not have compatible
# inner dimensions.
e = tf.matmul(c, d)

f = tf.matmul(c, d, transpose_a=True, transpose_b=True)

print(f.shape)
==> TensorShape([Dimension(3), Dimension(4)])
```

In some cases, the inferred shape may have unknown dimensions. If the caller has additional information about the values of these dimensions, `Tensor.set_shape()` can be used to augment the inferred shape.

Returns:

A `TensorShape` representing the shape of this tensor.

value_index

The index of this tensor in the outputs of its `Operation`.

Methods

__init__

```
__init__(
    op,
    value_index,
    dtype
)
```

Creates a new `Tensor`.

Args:

- `op`: An `Operation`. `Operation` that computes this tensor.
- `value_index`: An `int`. Index of the operation's endpoint that produces this tensor.
- `dtype`: A `DType`. Type of elements stored in this tensor.

Raises:

- `TypeError`: If the op is not an `Operation`.

__abs__

```
__abs__(
    x,
    name=None
)
```

Computes the absolute value of a tensor.

Given a tensor `x` of complex numbers, this operation returns a tensor of type `float32` or `float64` that is the absolute value of each element in `x`. All elements in `x` must be complex numbers of the form *a* + *bj*. The absolute value is computed as $\sqrt{a^2 + b^2}$. For example:

```
x = tf.constant([[-2.25 + 4.75j], [-3.25 + 5.75j]])
tf.abs(x)  # [5.25594902, 6.60492229]
```

Args:

- `x`: A `Tensor` or `SparseTensor` of type `float32`, `float64`, `int32`, `int64`, `complex64` or `complex128`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` or `SparseTensor` the same size and type as `x` with absolute values. Note, for `complex64` or `complex128` `input, the returned` Tensor `will be of type` float32 `or` float64`, respectively.

__add__

```
__add__(
    x,
    y
)
```

Returns x + y element-wise.

NOTE: `Add` supports broadcasting. `AddN` does not. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`, `complex64`, `complex128`, `string`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`.

__and__

```
__and__(
    x,
    y
)
```

Returns the truth value of x AND y element-wise.

NOTE: `LogicalAnd` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor` of type `bool`.
- `y`: A `Tensor` of type `bool`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

__bool__

```
__bool__()
```

Dummy method to prevent a tensor from being used as a Python `bool`.

This overload raises a `TypeError` when the user inadvertently treats a `Tensor` as a boolean (e.g. in an `if` statement). For example:

```
if tf.constant(True):  # Will raise.
  # ...

if tf.constant(5) < tf.constant(7):  # Will raise.
  # ...
```

This disallows ambiguities between testing the Python value vs testing the dynamic condition of the `Tensor`.

Raises:

`TypeError`.

`__div__`

```
__div__(
    x,
    y
)
```

Divide two values using Python 2 semantics. Used for Tensor.**div**.

Args:

- `x`: `Tensor` numerator of real numeric type.
- `y`: `Tensor` denominator of real numeric type.
- `name`: A name for the operation (optional).

Returns:

`x / y` returns the quotient of x and y.

`__eq__`

```
__eq__(other)
```

`__floordiv__`

```
__floordiv__(
    x,
    y
)
```

Divides `x / y` elementwise, rounding toward the most negative integer.

The same as `tf.div(x,y)` for integers, but uses `tf.floor(tf.div(x,y))` for floating point arguments so that the result is always an integer (though possibly an integer represented as floating point). This op is generated by `x // y` floor division in Python 3 and in Python 2.7 with `from __future__ import division`.

Note that for efficiency, `floordiv` uses C semantics for negative numbers (unlike Python and Numpy).

`x` and `y` must have the same type, and the result will have the same type as well.

Args:

- `x`: `Tensor` numerator of real numeric type.
- `y`: `Tensor` denominator of real numeric type.
- `name`: A name for the operation (optional).

Returns:

`x / y` rounded down (except possibly towards zero for negative integers).

Raises:

- `TypeError`: If the inputs are complex.

`__ge__`

```
__ge__(
    x,
    y,
    name=None
)
```

Returns the truth value of (x >= y) element-wise.

NOTE: `GreaterEqual` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`__getitem__`

```
__getitem__(
    tensor,
    slice_spec,
    var=None
)
```

Overload for Tensor.**getitem**.

This operation extracts the specified region from the tensor. The notation is similar to NumPy with the restriction that currently only support basic indexing. That means that using a tensor as input is not currently allowed

Some useful examples:

```
# strip leading and trailing 2 elements
foo = tf.constant([1,2,3,4,5,6])
print(foo[2:-2].eval())  # [3,4]

# skip every row and reverse every column
foo = tf.constant([[1,2,3], [4,5,6], [7,8,9]])
print(foo[::2,::-1].eval())  # [[3,2,1], [9,8,7]]

# Insert another dimension
foo = tf.constant([[1,2,3], [4,5,6], [7,8,9]])
print(foo[tf.newaxis, :, :].eval()) # => [[[1,2,3], [4,5,6], [7,8,9]]]
print(foo[:, tf.newaxis, :].eval()) # => [[[1,2,3]], [[4,5,6]], [[7,8,9]]]
print(foo[:, :, tf.newaxis].eval()) # => [[[1],[2],[3]], [[4],[5],[6]],
[[7],[8],[9]]]

# Ellipses (3 equivalent operations)
foo = tf.constant([[1,2,3], [4,5,6], [7,8,9]])
print(foo[tf.newaxis, :, :].eval())  # [[[1,2,3], [4,5,6], [7,8,9]]]
print(foo[tf.newaxis, ...].eval())  # [[[1,2,3], [4,5,6], [7,8,9]]]
print(foo[tf.newaxis].eval())  # [[[1,2,3], [4,5,6], [7,8,9]]]
```

Notes: - `tf.newaxis` is `None` as in NumPy. - An implicit ellipsis is placed at the end of the `slice_spec` - NumPy advanced indexing is currently not supported.

Args:

- `tensor`: An ops.Tensor object.
- `slice_spec`: The arguments to Tensor.**getitem**.
- `var`: In the case of variable slice assignment, the Variable object to slice (i.e. tensor is the read-only view of this variable).

Returns:

The appropriate slice of "tensor", based on "slice_spec".

Raises:

- `ValueError`: If a slice range is negative size.
- `TypeError`: If the slice indices aren't int, slice, or Ellipsis.

`__gt__`

```
__gt__(
    x,
    y,
    name=None
)
```

Returns the truth value of (x > y) element-wise.

NOTE: `Greater` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`,

`half`.

- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`__invert__`

```
__invert__(
    x,
    name=None
)
```

Returns the truth value of NOT x element-wise.

Args:

- `x`: A `Tensor` of type `bool`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`__iter__`

```
__iter__()
```

Dummy method to prevent iteration. Do not call.

NOTE(mrry): If we register **getitem** as an overloaded operator, Python will valiantly attempt to iterate over the Tensor from 0 to infinity. Declaring this method prevents this unintended behavior.

Raises:

- `TypeError`: when invoked.

`__le__`

```
__le__(
    x,
    y,
    name=None
)
```

Returns the truth value of (x <= y) element-wise.

NOTE: `LessEqual` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

__lt__

```
__lt__(
    x,
    y,
    name=None
)
```

Returns the truth value of (x < y) element-wise.

NOTE: `Less` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

__matmul__

```
__matmul__(
    x,
    y
)
```

Multiplies matrix `a` by matrix `b`, producing `a` * `b`.

The inputs must, following any transpositions, be tensors of rank >= 2 where the inner 2 dimensions specify valid matrix multiplication arguments, and any further outer dimensions match.

Both matrices must be of the same type. The supported types are: `float16`, `float32`, `float64`, `int32`, `complex64`, `complex128`.

Either matrix can be transposed or adjointed (conjugated and transposed) on the fly by setting one of the corresponding flag to `True`. These are `False` by default.

If one or both of the matrices contain a lot of zeros, a more efficient multiplication algorithm can be used by setting the corresponding `a_is_sparse` or `b_is_sparse` flag to `True`. These are `False` by default. This optimization is only available for plain matrices (rank-2 tensors) with datatypes `bfloat16` or `float32`.

For example:

```
# 2-D tensor `a`
# [[1, 2, 3],
#  [4, 5, 6]]
a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3])

# 2-D tensor `b`
# [[ 7,  8],
#  [ 9, 10],
#  [11, 12]]
b = tf.constant([7, 8, 9, 10, 11, 12], shape=[3, 2])

# `a` * `b`
# [[ 58,  64],
#  [139, 154]]
c = tf.matmul(a, b)

# 3-D tensor `a`
# [[[ 1,  2,  3],
#   [ 4,  5,  6]],
#  [[ 7,  8,  9],
#   [10, 11, 12]]]
a = tf.constant(np.arange(1, 13, dtype=np.int32),
                shape=[2, 2, 3])

# 3-D tensor `b`
# [[[13, 14],
#   [15, 16],
#   [17, 18]],
#  [[19, 20],
#   [21, 22],
#   [23, 24]]]
b = tf.constant(np.arange(13, 25, dtype=np.int32),
                shape=[2, 3, 2])

# `a` * `b`
# [[[ 94, 100],
#   [229, 244]],
#  [[508, 532],
#   [697, 730]]]
c = tf.matmul(a, b)

# Since python >= 3.5 the @ operator is supported (see PEP 465).
# In TensorFlow, it simply calls the `tf.matmul()` function, so the
# following lines are equivalent:
d = a @ b @ [[10.], [11.]]
d = tf.matmul(tf.matmul(a, b), [[10.], [11.]])
```

Args:

- `a`: **`Tensor`** of type **`float16`**, **`float32`**, **`float64`**, **`int32`**, **`complex64`**, **`complex128`** and rank > 1.
- `b`: **`Tensor`** with same type and rank as **`a`**.
- `transpose_a`: If **`True`**, **`a`** is transposed before multiplication.
- `transpose_b`: If **`True`**, **`b`** is transposed before multiplication.
- `adjoint_a`: If **`True`**, **`a`** is conjugated and transposed before multiplication.
- `adjoint_b`: If **`True`**, **`b`** is conjugated and transposed before multiplication.
- `a_is_sparse`: If **`True`**, **`a`** is treated as a sparse matrix.
- `b_is_sparse`: If **`True`**, **`b`** is treated as a sparse matrix.

- `name`: Name for the operation (optional).

Returns:

A `Tensor` of the same type as `a` and `b` where each inner-most matrix is the product of the corresponding matrices in `a` and `b`, e.g. if all transpose or adjoint attributes are `False`:

`output`[..., i, j] = sum_k (`a`[..., i, k] * `b`[..., k, j]), for all indices i, j.

- `Note`: This is matrix product, not element-wise product.

Raises:

- `ValueError`: If transpose_a and adjoint_a, or transpose_b and adjoint_b are both set to True.

__mod__

```
__mod__(
    x,
    y
)
```

Returns element-wise remainder of division. When `x < 0` xor `y < 0` is

true, this follows Python semantics in that the result here is consistent with a flooring divide. E.g. `floor(x / y) * y + mod(x, y) = x`.

NOTE: `FloorMod` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `int32`, `int64`, `float32`, `float64`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`.

__mul__

```
__mul__(
    x,
    y
)
```

Dispatches cwise mul for "Dense*Dense*" *and* "*Dense*Sparse".

__neg__

```
__neg__(
    x,
    name=None
)
```

Computes numerical negative value element-wise.

I.e., $y = -x$.

Args:

- `x`: A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`.

`__nonzero__`

```
__nonzero__()
```

Dummy method to prevent a tensor from being used as a Python `bool`.

This is the Python 2.x counterpart to `__bool__()` above.

Raises:

`TypeError`.

`__or__`

```
__or__(
    x,
    y
)
```

Returns the truth value of x OR y element-wise.

NOTE: `LogicalOr` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor` of type `bool`.
- `y`: A `Tensor` of type `bool`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

__pow__

```
__pow__(
    x,
    y
)
```

Computes the power of one value to another.

Given a tensor `x` and a tensor `y`, this operation computes x^y for corresponding elements in `x` and `y`. For example:

```
x = tf.constant([[2, 2], [3, 3]])
y = tf.constant([[8, 16], [2, 3]])
tf.pow(x, y)  # [[256, 65536], [9, 27]]
```

Args:

- `x`: A `Tensor` of type `float32`, `float64`, `int32`, `int64`, `complex64`, or `complex128`.
- `y`: A `Tensor` of type `float32`, `float64`, `int32`, `int64`, `complex64`, or `complex128`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`.

__radd__

```
__radd__(
    y,
    x
)
```

Returns x + y element-wise.

NOTE: `Add` supports broadcasting. `AddN` does not. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`, `complex64`, `complex128`, `string`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`.

__rand__

```
__rand__(
    y,
    x
)
```

Returns the truth value of x AND y element-wise.

NOTE: `LogicalAnd` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor` of type `bool`.
- `y`: A `Tensor` of type `bool`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`__rdiv__`

```
__rdiv__(
    y,
    x
)
```

Divide two values using Python 2 semantics. Used for Tensor.**div**.

Args:

- `x`: `Tensor` numerator of real numeric type.
- `y`: `Tensor` denominator of real numeric type.
- `name`: A name for the operation (optional).

Returns:

`x / y` returns the quotient of x and y.

`__rfloordiv__`

```
__rfloordiv__(
    y,
    x
)
```

Divides `x / y` elementwise, rounding toward the most negative integer.

The same as `tf.div(x,y)` for integers, but uses `tf.floor(tf.div(x,y))` for floating point arguments so that the result is always an integer (though possibly an integer represented as floating point). This op is generated by `x // y` floor division in Python 3 and in Python 2.7 with `from __future__ import division`.

Note that for efficiency, `floordiv` uses C semantics for negative numbers (unlike Python and Numpy).

`x` and `y` must have the same type, and the result will have the same type as well.

Args:

- `x`: `Tensor` numerator of real numeric type.
- `y`: `Tensor` denominator of real numeric type.
- `name`: A name for the operation (optional).

Returns:

`x / y` rounded down (except possibly towards zero for negative integers).

Raises:

- `TypeError`: If the inputs are complex.

`__rmatmul__`

```
__rmatmul__(
    y,
    x
)
```

Multiplies matrix `a` by matrix `b`, producing `a` * `b`.

The inputs must, following any transpositions, be tensors of rank >= 2 where the inner 2 dimensions specify valid matrix multiplication arguments, and any further outer dimensions match.

Both matrices must be of the same type. The supported types are: `float16`, `float32`, `float64`, `int32`, `complex64`, `complex128`.

Either matrix can be transposed or adjointed (conjugated and transposed) on the fly by setting one of the corresponding flag to `True`. These are `False` by default.

If one or both of the matrices contain a lot of zeros, a more efficient multiplication algorithm can be used by setting the corresponding `a_is_sparse` or `b_is_sparse` flag to `True`. These are `False` by default. This optimization is only available for plain matrices (rank-2 tensors) with datatypes `bfloat16` or `float32`.

For example:

```
# 2-D tensor `a`
# [[1, 2, 3],
#  [4, 5, 6]]
a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3])

# 2-D tensor `b`
# [[ 7,  8],
#  [ 9, 10],
#  [11, 12]]
b = tf.constant([7, 8, 9, 10, 11, 12], shape=[3, 2])

# `a` * `b`
# [[ 58,  64],
#  [139, 154]]
c = tf.matmul(a, b)

# 3-D tensor `a`
# [[[ 1,  2,  3],
#   [ 4,  5,  6]],
#  [[ 7,  8,  9],
#   [10, 11, 12]]]
a = tf.constant(np.arange(1, 13, dtype=np.int32),
                shape=[2, 2, 3])

# 3-D tensor `b`
# [[[13, 14],
#   [15, 16],
#   [17, 18]],
#  [[19, 20],
#   [21, 22],
#   [23, 24]]]
b = tf.constant(np.arange(13, 25, dtype=np.int32),
                shape=[2, 3, 2])

# `a` * `b`
# [[[ 94, 100],
#   [229, 244]],
#  [[508, 532],
#   [697, 730]]]
c = tf.matmul(a, b)

# Since python >= 3.5 the @ operator is supported (see PEP 465).
# In TensorFlow, it simply calls the `tf.matmul()` function, so the
# following lines are equivalent:
d = a @ b @ [[10.], [11.]]
d = tf.matmul(tf.matmul(a, b), [[10.], [11.]])
```

Args:

- `a`: **`Tensor`** of type **`float16`**, **`float32`**, **`float64`**, **`int32`**, **`complex64`**, **`complex128`** and rank > 1.
- `b`: **`Tensor`** with same type and rank as **`a`**.
- `transpose_a`: If **`True`**, **`a`** is transposed before multiplication.
- `transpose_b`: If **`True`**, **`b`** is transposed before multiplication.
- `adjoint_a`: If **`True`**, **`a`** is conjugated and transposed before multiplication.
- `adjoint_b`: If **`True`**, **`b`** is conjugated and transposed before multiplication.
- `a_is_sparse`: If **`True`**, **`a`** is treated as a sparse matrix.
- `b_is_sparse`: If **`True`**, **`b`** is treated as a sparse matrix.
- `name`: Name for the operation (optional).

Returns:

A `Tensor` of the same type as `a` and `b` where each inner-most matrix is the product of the corresponding matrices in `a` and `b`, e.g. if all transpose or adjoint attributes are `False`:

`output`[..., i, j] = sum_k (`a`[..., i, k] * `b`[..., k, j]), for all indices i, j.

- `Note`: This is matrix product, not element-wise product.

Raises:

- `ValueError`: If transpose_a and adjoint_a, or transpose_b and adjoint_b are both set to True.

__rmod__

```
__rmod__(
    y,
    x
)
```

Returns element-wise remainder of division. When `x < 0` xor `y < 0` is

true, this follows Python semantics in that the result here is consistent with a flooring divide. E.g. `floor(x / y) * y + mod(x, y) = x`.

NOTE: `FloorMod` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `int32`, `int64`, `float32`, `float64`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`.

__rmul__

```
__rmul__(
    y,
    x
)
```

Dispatches cwise mul for "Dense*Dense*" *and* "*Dense*Sparse".

__ror__

```
__ror__(
    y,
    x
)
```

Returns the truth value of x OR y element-wise.

NOTE: `LogicalOr` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor` of type `bool`.
- `y`: A `Tensor` of type `bool`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`__rpow__`

```
__rpow__(
    y,
    x
)
```

Computes the power of one value to another.

Given a tensor `x` and a tensor `y`, this operation computes x^y for corresponding elements in `x` and `y`. For example:

```
x = tf.constant([[2, 2], [3, 3]])
y = tf.constant([[8, 16], [2, 3]])
tf.pow(x, y)  # [[256, 65536], [9, 27]]
```

Args:

- `x`: A `Tensor` of type `float32`, `float64`, `int32`, `int64`, `complex64`, or `complex128`.
- `y`: A `Tensor` of type `float32`, `float64`, `int32`, `int64`, `complex64`, or `complex128`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`.

`__rsub__`

```
__rsub__(
    y,
    x
)
```

Returns x - y element-wise.

NOTE: `Sub` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `uint8`, `int8`, `uint16`, `int16`, `int32`, `int64`, `complex64`, `complex128`.

- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`.

`__rtruediv__`

```
__rtruediv__(
    y,
    x
)
```

`__rxor__`

```
__rxor__(
    y,
    x
)
```

x ^ y = (x | y) & ~(x & y).

`__sub__`

```
__sub__(
    x,
    y
)
```

Returns x - y element-wise.

NOTE: `Sub` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `uint8`, `int8`, `uint16`, `int16`, `int32`, `int64`, `complex64`, `complex128`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`.

`__truediv__`

```
__truediv__(
    x,
    y
)
```

__xor__

```
__xor__(
    x,
    y
)
```

x ^ y = (x | y) & ~(x & y).

consumers

```
consumers()
```

Returns a list of `Operation` s that consume this tensor.

Returns:

A list of `Operation` s.

eval

```
eval(
    feed_dict=None,
    session=None
)
```

Evaluates this tensor in a `Session` .

Calling this method will execute all preceding operations that produce the inputs needed for the operation that produces this tensor.

N.B. Before invoking `Tensor.eval()` , its graph must have been launched in a session, and either a default session must be available, or `session` must be specified explicitly.

Args:

- `feed_dict` : A dictionary that maps `Tensor` objects to feed values. See `tf.Session.run` for a description of the valid feed values.
- `session` : (Optional.) The `Session` to be used to evaluate this tensor. If none, the default session will be used.

Returns:

A numpy array corresponding to the value of this tensor.

get_shape

```
get_shape()
```

Alias of Tensor.shape.

set_shape

```
set_shape(shape)
```

Updates the shape of this tensor.

This method can be called multiple times, and will merge the given **shape** with the current shape of this tensor. It can be used to provide additional information about the shape of this tensor that cannot be inferred from the graph alone. For example, this can be used to provide additional information about the shapes of images:

```
_, image_data = tf.TFRecordReader(...).read(...)
image = tf.image.decode_png(image_data, channels=3)

# The height and width dimensions of `image` are data dependent, and
# cannot be computed without executing the op.
print(image.shape)
==> TensorShape([Dimension(None), Dimension(None), Dimension(3)])

# We know that each image in this dataset is 28 x 28 pixels.
image.set_shape([28, 28, 3])
print(image.shape)
==> TensorShape([Dimension(28), Dimension(28), Dimension(3)])
```

Args:

- shape: A **TensorShape** representing the shape of this tensor.

Raises:

- ValueError: If **shape** is not compatible with the current shape of this tensor.

Class Members

OVERLOADABLE_OPERATORS

__array_priority__