tf.Graph

Contents

Class `Graph`

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

See the guide: Building Graphs > Core graph data structures

A TensorFlow computation, represented as a dataflow graph.

A `Graph` contains a set of `tf.Operation` objects, which represent units of computation; and `tf.Tensor` objects, which represent the units of data that flow between operations.

A default `Graph` is always registered, and accessible by calling `tf.get_default_graph`. To add an operation to the default graph, simply call one of the functions that defines a new `Operation`:

```
c = tf.constant(4.0)
assert c.graph is tf.get_default_graph()
```

Another typical usage involves the `tf.Graph.as_default` context manager, which overrides the current default graph for the lifetime of the context:

```
g = tf.Graph()
with g.as_default():
  # Define operations and tensors in `g`.
  c = tf.constant(30.0)
  assert c.graph is g
```

Important note: This class *is not* thread-safe for graph construction. All operations should be created from a single thread, or external synchronization must be provided. Unless otherwise specified, all methods are not thread-safe.

A `Graph` instance supports an arbitrary number of "collections" that are identified by name. For convenience when building a large graph, collections can store groups of related objects: for example, the `tf.Variable` uses a collection (named `tf.GraphKeys.GLOBAL_VARIABLES`) for all variables that are created during the construction of a graph. The caller may define additional collections by specifying a new name.

Properties

`building_function`

Returns True iff this graph represents a function.

`collections`

Returns the names of the collections known to this graph.

`finalized`

True if this graph has been finalized.

`graph_def_versions`

The GraphDef version information of this graph.

For details on the meaning of each version, see `GraphDef`.

Returns:

A `VersionDef`.

`seed`

The graph-level random seed of this graph.

`version`

Returns a version number that increases as ops are added to the graph.

Note that this is unrelated to the `tf.Graph.graph_def_versions`.

Returns:

An integer version that increases as ops are added to the graph.

Methods

`__init__`

```
__init__()
```

Creates a new, empty Graph.

`add_to_collection`

```
add_to_collection(
    name,
    value
)
```

Stores `value` in the collection with the given `name`.

Note that collections are not sets, so it is possible to add a value to a collection several times.

Args:

- `name`: The key for the collection. The `GraphKeys` class contains many standard names for collections.
- `value`: The value to add to the collection.

add_to_collections

```
add_to_collections(
    names,
    value
)
```

Stores `value` in the collections given by `names`.

Note that collections are not sets, so it is possible to add a value to a collection several times. This function makes sure that duplicates in `names` are ignored, but it will not check for pre-existing membership of `value` in any of the collections in `names`.

`names` can be any iterable, but if `names` is a string, it is treated as a single collection name.

Args:

- `names`: The keys for the collections to add to. The `GraphKeys` class contains many standard names for collections.
- `value`: The value to add to the collections.

as_default

```
as_default()
```

Returns a context manager that makes this `Graph` the default graph.

This method should be used if you want to create multiple graphs in the same process. For convenience, a global default graph is provided, and all ops will be added to this graph if you do not create a new graph explicitly. Use this method with the `with` keyword to specify that ops created within the scope of a block should be added to this graph.

The default graph is a property of the current thread. If you create a new thread, and wish to use the default graph in that thread, you must explicitly add a `with g.as_default():` in that thread's function.

The following code examples are equivalent:

```
# 1. Using Graph.as_default():
g = tf.Graph()
with g.as_default():
  c = tf.constant(5.0)
  assert c.graph is g

# 2. Constructing and making default:
with tf.Graph().as_default() as g:
  c = tf.constant(5.0)
  assert c.graph is g
```

Returns:

A context manager for using this graph as the default graph.

as_graph_def

```
as_graph_def(
    from_version=None,
    add_shapes=False
)
```

Returns a serialized `GraphDef` representation of this graph.

The serialized `GraphDef` can be imported into another `Graph` (using `tf.import_graph_def`) or used with the C++ Session API.

This method is thread-safe.

Args:

- `from_version`: Optional. If this is set, returns a `GraphDef` containing only the nodes that were added to this graph since its `version` property had the given value.
- `add_shapes`: If true, adds an "_output_shapes" list attr to each node with the inferred shapes of each of its outputs.

Returns:

A `GraphDef` protocol buffer.

Raises:

- `ValueError`: If the `graph_def` would be too large.

as_graph_element

```
as_graph_element(
    obj,
    allow_tensor=True,
    allow_operation=True
)
```

Returns the object referred to by `obj`, as an `Operation` or `Tensor`.

This function validates that `obj` represents an element of this graph, and gives an informative error message if it is not.

This function is the canonical way to get/validate an object of one of the allowed types from an external argument reference in the Session API.

This method may be called concurrently from multiple threads.

Args:

- `obj`: A `Tensor`, an `Operation`, or the name of a tensor or operation. Can also be any object with an `_as_graph_element()` method that returns a value of one of these types.
- `allow_tensor`: If true, `obj` may refer to a `Tensor`.
- `allow_operation`: If true, `obj` may refer to an `Operation`.

Returns:

The `Tensor` or `Operation` in the Graph corresponding to `obj`.

Raises:

- `TypeError`: If `obj` is not a type we support attempting to convert to types.
- `ValueError`: If `obj` is of an appropriate type but invalid. For example, an invalid string.
- `KeyError`: If `obj` is not an object in the graph.

clear_collection

```
clear_collection(name)
```

Clears all values in a collection.

Args:

- `name`: The key for the collection. The `GraphKeys` class contains many standard names for collections.

colocate_with

```
colocate_with(
    op,
    ignore_existing=False
)
```

Returns a context manager that specifies an op to colocate with.

> ★ **Note:** this function is not for public use, only for internal libraries.

For example:

```
a = tf.Variable([1.0])
with g.colocate_with(a):
  b = tf.constant(1.0)
  c = tf.add(a, b)
```

`b` and `c` will always be colocated with `a`, no matter where `a` is eventually placed.

NOTE Using a colocation scope resets any existing device constraints.

If `op` is `None` then `ignore_existing` must be `True` and the new scope resets all colocation and device constraints.

Args:

- `op`: The op to colocate all created ops with, or `None`.
- `ignore_existing`: If true, only applies colocation of this op within the context, rather than applying all colocation properties on the stack. If `op` is `None`, this value must be `True`.

Raises:

- `ValueError`: if op is None but ignore_existing is False.

Yields:

A context manager that specifies the op with which to colocate newly created ops.

`container`

```
container(container_name)
```

Returns a context manager that specifies the resource container to use.

Stateful operations, such as variables and queues, can maintain their states on devices so that they can be shared by multiple processes. A resource container is a string name under which these stateful operations are tracked. These resources can be released or cleared with `tf.Session.reset()`.

For example:

```
with g.container('experiment0'):
  # All stateful Operations constructed in this context will be placed
  # in resource container "experiment0".
  v1 = tf.Variable([1.0])
  v2 = tf.Variable([2.0])
  with g.container("experiment1"):
    # All stateful Operations constructed in this context will be
    # placed in resource container "experiment1".
    v3 = tf.Variable([3.0])
    q1 = tf.FIFOQueue(10, tf.float32)
  # All stateful Operations constructed in this context will be
  # be created in the "experiment0".
  v4 = tf.Variable([4.0])
  q1 = tf.FIFOQueue(20, tf.float32)
  with g.container(""):
    # All stateful Operations constructed in this context will be
    # be placed in the default resource container.
    v5 = tf.Variable([5.0])
    q3 = tf.FIFOQueue(30, tf.float32)

# Resets container "experiment0", after which the state of v1, v2, v4, q1
# will become undefined (such as uninitialized).
tf.Session.reset(target, ["experiment0"])
```

Args:

- `container_name`: container name string.

Returns:

A context manager for defining resource containers for stateful ops, yields the container name.

`control_dependencies`

```
control_dependencies(control_inputs)
```

Returns a context manager that specifies control dependencies.

Use with the `with` keyword to specify that all operations constructed within the context should have control dependencies on `control_inputs`. For example:

```
with g.control_dependencies([a, b, c]):
  # `d` and `e` will only run after `a`, `b`, and `c` have executed.
  d = ...
  e = ...
```

Multiple calls to `control_dependencies()` can be nested, and in that case a new `Operation` will have control dependencies on the union of `control_inputs` from all active contexts.

```
with g.control_dependencies([a, b]):
  # Ops constructed here run after `a` and `b`.
  with g.control_dependencies([c, d]):
    # Ops constructed here run after `a`, `b`, `c`, and `d`.
```

You can pass None to clear the control dependencies:

```
with g.control_dependencies([a, b]):
  # Ops constructed here run after `a` and `b`.
  with g.control_dependencies(None):
    # Ops constructed here run normally, not waiting for either `a` or `b`.
    with g.control_dependencies([c, d]):
      # Ops constructed here run after `c` and `d`, also not waiting
      # for either `a` or `b`.
```

N.B. The control dependencies context applies *only* to ops that are constructed within the context. Merely using an op or tensor in the context does not add a control dependency. The following example illustrates this point:

```
# WRONG
def my_func(pred, tensor):
  t = tf.matmul(tensor, tensor)
  with tf.control_dependencies([pred]):
    # The matmul op is created outside the context, so no control
    # dependency will be added.
    return t

# RIGHT
def my_func(pred, tensor):
  with tf.control_dependencies([pred]):
    # The matmul op is created in the context, so a control dependency
    # will be added.
    return tf.matmul(tensor, tensor)
```

Args:

- `control_inputs`: A list of `Operation` or `Tensor` objects which must be executed or computed before running the operations defined in the context. Can also be `None` to clear the control dependencies.

Returns:

A context manager that specifies control dependencies for all operations constructed within the context.

Raises:

- `TypeError`: If `control_inputs` is not a list of `Operation` or `Tensor` objects.

`create_op`

```
create_op(
    op_type,
    inputs,
    dtypes,
    input_types=None,
    name=None,
    attrs=None,
    op_def=None,
    compute_shapes=True,
    compute_device=True
)
```

Creates an `Operation` in this graph.

This is a low-level interface for creating an `Operation`. Most programs will not call this method directly, and instead use the Python op constructors, such as `tf.constant()`, which add ops to the default graph.

Args:

- `op_type`: The `Operation` type to create. This corresponds to the `OpDef.name` field for the proto that defines the operation.
- `inputs`: A list of `Tensor` objects that will be inputs to the `Operation`.
- `dtypes`: A list of `DType` objects that will be the types of the tensors that the operation produces.
- `input_types`: (Optional.) A list of `DType`s that will be the types of the tensors that the operation consumes. By default, uses the base `DType` of each input in `inputs`. Operations that expect reference-typed inputs must specify `input_types` explicitly.
- `name`: (Optional.) A string name for the operation. If not specified, a name is generated based on `op_type`.
- `attrs`: (Optional.) A dictionary where the key is the attribute name (a string) and the value is the respective `attr` attribute of the `NodeDef` proto that will represent the operation (an `AttrValue` proto).
- `op_def`: (Optional.) The `OpDef` proto that describes the `op_type` that the operation will have.
- `compute_shapes`: (Optional.) If True, shape inference will be performed to compute the shapes of the outputs.
- `compute_device`: (Optional.) If True, device functions will be executed to compute the device property of the Operation.

Raises:

- `TypeError`: if any of the inputs is not a `Tensor`.
- `ValueError`: if colocation conflicts with existing device assignment.

Returns:

An `Operation` object.

`device`

```
device(device_name_or_function)
```

Returns a context manager that specifies the default device to use.

The `device_name_or_function` argument may either be a device name string, a device function, or None:

- If it is a device name string, all operations constructed in this context will be assigned to the device with that name,

unless overridden by a nested `device()` context.

- If it is a function, it will be treated as a function from Operation objects to device name strings, and invoked each time a new Operation is created. The Operation will be assigned to the device with the returned name.
- If it is None, all `device()` invocations from the enclosing context will be ignored.

For information about the valid syntax of device name strings, see the documentation in `DeviceNameUtils`.

For example:

```
with g.device('/device:GPU:0'):
  # All operations constructed in this context will be placed
  # on GPU 0.
  with g.device(None):
    # All operations constructed in this context will have no
    # assigned device.

# Defines a function from `Operation` to device string.
def matmul_on_gpu(n):
  if n.type == "MatMul":
    return "/device:GPU:0"
  else:
    return "/cpu:0"

with g.device(matmul_on_gpu):
  # All operations of type "MatMul" constructed in this context
  # will be placed on GPU 0; all other operations will be placed
  # on CPU 0.
```

N.B. The device scope may be overridden by op wrappers or other library code. For example, a variable assignment op `v.assign()` must be colocated with the `tf.Variable` `v`, and incompatible device scopes will be ignored.

Args:

- `device_name_or_function`: The device name or function to use in the context.

Yields:

A context manager that specifies the default device to use for newly created ops.

finalize

```
finalize()
```

Finalizes this graph, making it read-only.

After calling `g.finalize()`, no new operations can be added to `g`. This method is used to ensure that no operations are added to a graph when it is shared between multiple threads, for example when using a `tf.train.QueueRunner`.

get_all_collection_keys

```
get_all_collection_keys()
```

Returns a list of collections used in this graph.

get_collection

```
get_collection(
    name,
    scope=None
)
```

Returns a list of values in the collection with the given `name`.

This is different from `get_collection_ref()` which always returns the actual collection list if it exists in that it returns a new list each time it is called.

Args:

- `name`: The key for the collection. For example, the `GraphKeys` class contains many standard names for collections.
- `scope`: (Optional.) A string. If supplied, the resulting list is filtered to include only items whose `name` attribute matches `scope` using `re.match`. Items without a `name` attribute are never returned if a scope is supplied. The choice of `re.match` means that a `scope` without special tokens filters by prefix.

Returns:

The list of values in the collection with the given `name`, or an empty list if no value has been added to that collection. The list contains the values in the order under which they were collected.

get_collection_ref

```
get_collection_ref(name)
```

Returns a list of values in the collection with the given `name`.

If the collection exists, this returns the list itself, which can be modified in place to change the collection. If the collection does not exist, it is created as an empty list and the list is returned.

This is different from `get_collection()` which always returns a copy of the collection list if it exists and never creates an empty collection.

Args:

- `name`: The key for the collection. For example, the `GraphKeys` class contains many standard names for collections.

Returns:

The list of values in the collection with the given `name`, or an empty list if no value has been added to that collection.

get_name_scope

```
get_name_scope()
```

Returns the current name scope.

For example:

```
with tf.name_scope('scope1'):
  with tf.name_scope('scope2'):
    print(tf.get_default_graph().get_name_scope())
```

would print the string `scope1/scope2`.

Returns:

A string representing the current name scope.

get_operation_by_name

```
get_operation_by_name(name)
```

Returns the `Operation` with the given `name`.

This method may be called concurrently from multiple threads.

Args:

- `name`: The name of the `Operation` to return.

Returns:

The `Operation` with the given `name`.

Raises:

- `TypeError`: If `name` is not a string.
- `KeyError`: If `name` does not correspond to an operation in this graph.

get_operations

```
get_operations()
```

Return the list of operations in the graph.

You can modify the operations in place, but modifications to the list such as inserts/delete have no effect on the list of operations known to the graph.

This method may be called concurrently from multiple threads.

Returns:

A list of Operations.

get_tensor_by_name

```
get_tensor_by_name(name)
```

Returns the `Tensor` with the given `name`.

This method may be called concurrently from multiple threads.

Args:

- `name`: The name of the `Tensor` to return.

Returns:

The `Tensor` with the given `name`.

Raises:

- `TypeError`: If `name` is not a string.
- `KeyError`: If `name` does not correspond to a tensor in this graph.

`gradient_override_map`

```
gradient_override_map(op_type_map)
```

EXPERIMENTAL: A context manager for overriding gradient functions.

This context manager can be used to override the gradient function that will be used for ops within the scope of the context.

For example:

```
@tf.RegisterGradient("CustomSquare")
def _custom_square_grad(op, grad):
  # ...

with tf.Graph().as_default() as g:
  c = tf.constant(5.0)
  s_1 = tf.square(c)  # Uses the default gradient for tf.square.
  with g.gradient_override_map({"Square": "CustomSquare"}):
    s_2 = tf.square(s_2)  # Uses _custom_square_grad to compute the
                          # gradient of s_2.
```

Args:

- `op_type_map`: A dictionary mapping op type strings to alternative op type strings.

Returns:

A context manager that sets the alternative op type to be used for one or more ops created in that context.

Raises:

- `TypeError`: If `op_type_map` is not a dictionary mapping strings to strings.

`is_feedable`

```
is_feedable(tensor)
```

Returns `True` if and only if `tensor` is feedable.

`is_fetchable`

```
is_fetchable(tensor_or_op)
```

Returns `True` if and only if `tensor_or_op` is fetchable.

name_scope

```
name_scope(name)
```

Returns a context manager that creates hierarchical names for operations.

A graph maintains a stack of name scopes. A `with name_scope(...):` statement pushes a new name onto the stack for the lifetime of the context.

The `name` argument will be interpreted as follows:

- A string (not ending with '/') will create a new name scope, in which `name` is appended to the prefix of all operations created in the context. If `name` has been used before, it will be made unique by calling `self.unique_name(name)`.
- A scope previously captured from a `with g.name_scope(...) as scope:` statement will be treated as an "absolute" name scope, which makes it possible to re-enter existing scopes.
- A value of `None` or the empty string will reset the current name scope to the top-level (empty) name scope.

For example:

```
with tf.Graph().as_default() as g:
  c = tf.constant(5.0, name="c")
  assert c.op.name == "c"
  c_1 = tf.constant(6.0, name="c")
  assert c_1.op.name == "c_1"

  # Creates a scope called "nested"
  with g.name_scope("nested") as scope:
    nested_c = tf.constant(10.0, name="c")
    assert nested_c.op.name == "nested/c"

    # Creates a nested scope called "inner".
    with g.name_scope("inner"):
      nested_inner_c = tf.constant(20.0, name="c")
      assert nested_inner_c.op.name == "nested/inner/c"

    # Create a nested scope called "inner_1".
    with g.name_scope("inner"):
      nested_inner_1_c = tf.constant(30.0, name="c")
      assert nested_inner_1_c.op.name == "nested/inner_1/c"

      # Treats `scope` as an absolute name scope, and
      # switches to the "nested/" scope.
      with g.name_scope(scope):
        nested_d = tf.constant(40.0, name="d")
        assert nested_d.op.name == "nested/d"

        with g.name_scope(""):
          e = tf.constant(50.0, name="e")
          assert e.op.name == "e"
```

The name of the scope itself can be captured by `with g.name_scope(...) as scope:`, which stores the name of the scope in the variable `scope`. This value can be used to name an operation that represents the overall result of executing the ops in a scope. For example:

```
inputs = tf.constant(...)
with g.name_scope('my_layer') as scope:
  weights = tf.Variable(..., name="weights")
  biases = tf.Variable(..., name="biases")
  affine = tf.matmul(inputs, weights) + biases
  output = tf.nn.relu(affine, name=scope)
```

NOTE: This constructor validates the given `name`. Valid scope names match one of the following regular expressions:

```
[A-Za-z0-9.][A-Za-z0-9_.\\-/]* (for scopes at the root)
[A-Za-z0-9_.\\-/]* (for other scopes)
```

Args:

- `name`: A name for the scope.

Returns:

A context manager that installs `name` as a new name scope.

Raises:

- `ValueError`: If `name` is not a valid scope name, according to the rules above.

`prevent_feeding`

```
prevent_feeding(tensor)
```

Marks the given `tensor` as unfeedable in this graph.

`prevent_fetching`

```
prevent_fetching(op)
```

Marks the given `op` as unfetchable in this graph.

`unique_name`

```
unique_name(
    name,
    mark_as_used=True
)
```

Return a unique operation name for `name`.

> ★ **Note:** You rarely need to call `unique_name()` directly. Most of the time you just need to create `with g.name_scope()` blocks to generate structured names.

`unique_name` is used to generate structured names, separated by `"/"`, to help identify operations when debugging a graph. Operation names are displayed in error messages reported by the TensorFlow runtime, and in various visualization tools such as TensorBoard.

If `mark_as_used` is set to `True`, which is the default, a new unique name is created and marked as in use. If it's set to

`False`, the unique name is returned without actually being marked as used. This is useful when the caller simply wants to know what the name to be created will be.

Args:

- `name`: The name for an operation.
- `mark_as_used`: Whether to mark this name as being used.

Returns:

A string to be passed to `create_op()` that will be used to name the operation being created.

Last updated November 2, 2017.