High Level Code Walk-through – ONNX Tensorflow Converter

Winnie Tsang, Cognitive OpenTech

Data and AI
Open Source Dojo

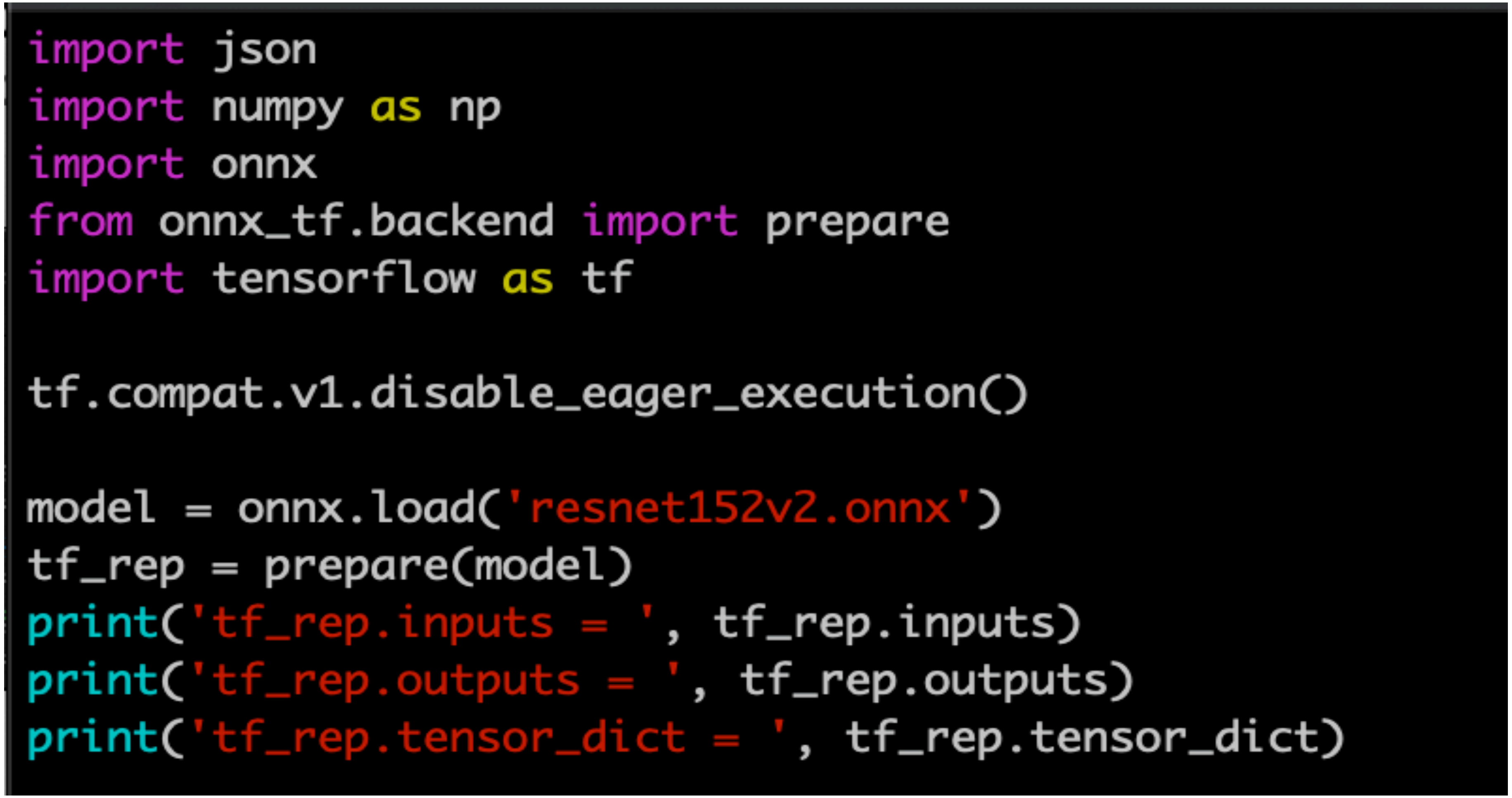
Convert ONNX model to Tensorflow model

1. Download a model from ONNX model zoo, https://github.com/onnx/models
2. Convert the ONNX model into Tensorflow model
3. Run the model in Tensorflow

Download ResNet-152 model

- Download resnetv2.onnx from the following link
  - https://github.com/onnx/models/tree/master/vision/classification/resnet
- Download imagenet_class_index.json
  - https://github.com/USCDataScience/dl4j-kerasimport-examples/blob/master/dl4j-import-example/data/imagenet_class_index.json
- Create some jpeg image for testing
  - Preprocess the data according to the steps in the following link https://github.com/onnx/models/tree/master/vision/classification/resnet#preprocessing

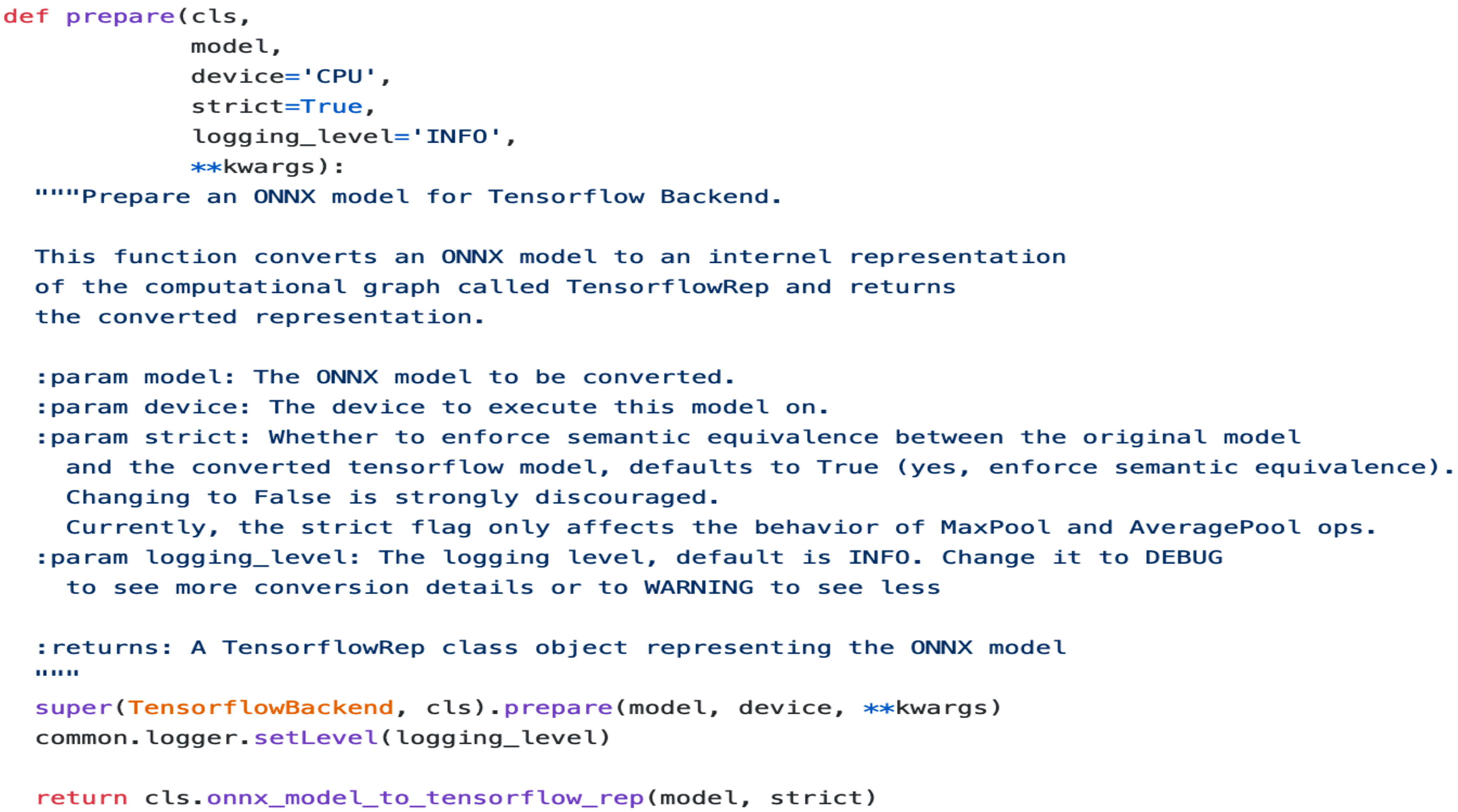
Convert ONNX model to Tensorflow model



```python
import json
import numpy as np
import onnx
from onnx_tf.backend import prepare
import tensorflow as tf

tf.compat.v1.disable_eager_execution()

model = onnx.load('resnet152v2.onnx')
tf_rep = prepare(model)
print('tf_rep.inputs = ', tf_rep.inputs)
print('tf_rep.outputs = ', tf_rep.outputs)
print('tf_rep.tensor_dict = ', tf_rep.tensor_dict)
```

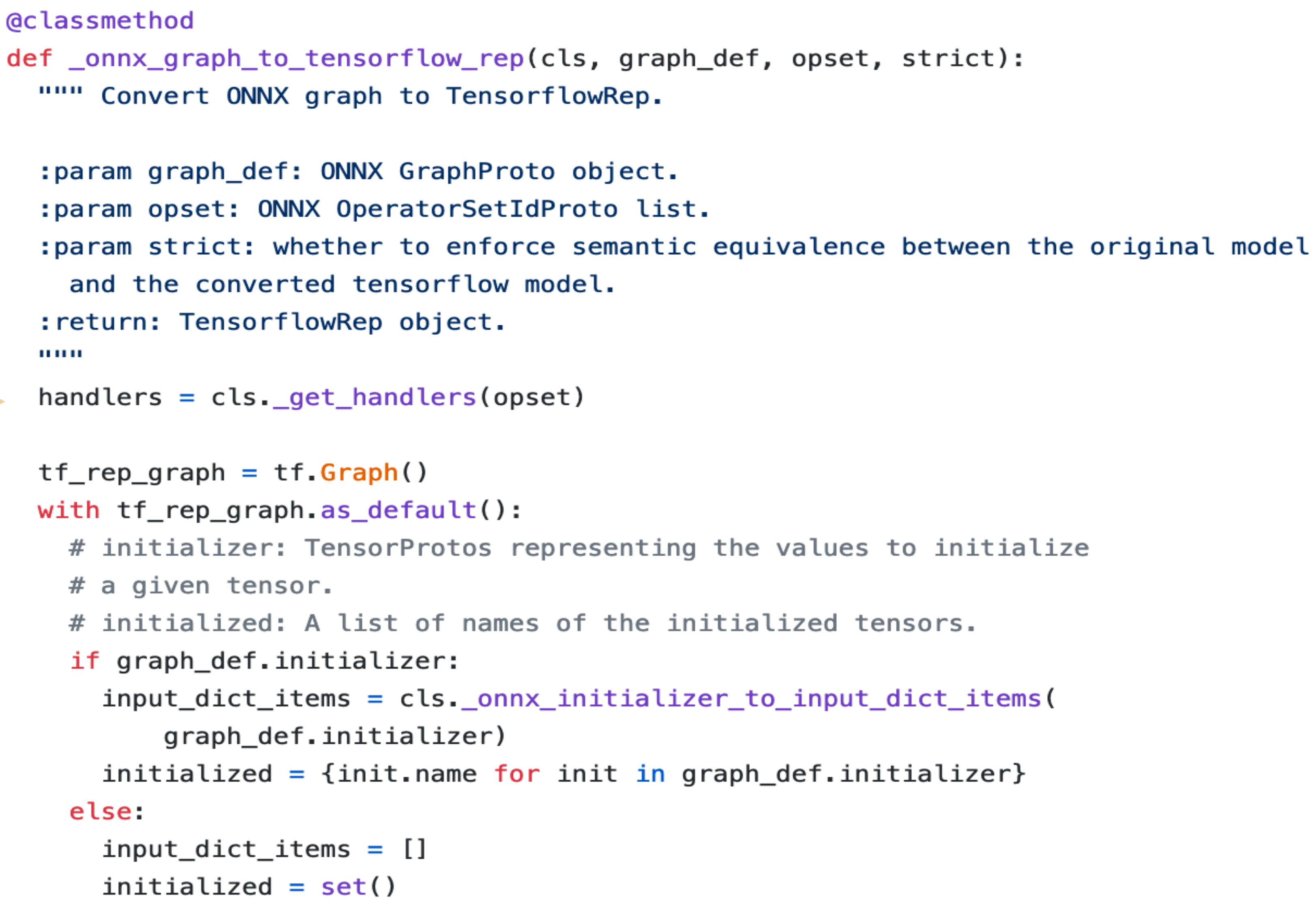
Using prepare to perform the conversion

- **prepare** is a function converts an ONNX model to an internal representation of the computational graph called TensorflowRep and returns the converted representation.
- The required input parameter is an ONNX model file
- The returned TensorflowRep will contain the Tensoflow graph, the inputs, outputs and tensor_dict of the graph.
- This returned TensorflowRep can be used to run the model in Tensorflow



```
def prepare(cls,
            model,
            device='CPU',
            strict=True,
            logging_level='INFO',
            **kwargs):
  """Prepare an ONNX model for Tensorflow Backend.

  This function converts an ONNX model to an internel representation
  of the computational graph called TensorflowRep and returns
  the converted representation.

  :param model: The ONNX model to be converted.
  :param device: The device to execute this model on.
  :param strict: Whether to enforce semantic equivalence between the original model
    and the converted tensorflow model, defaults to True (yes, enforce semantic equivalence).
    Changing to False is strongly discouraged.
    Currently, the strict flag only affects the behavior of MaxPool and AveragePool ops.
  :param logging_level: The logging level, default is INFO. Change it to DEBUG
    to see more conversion details or to WARNING to see less

  :returns: A TensorflowRep class object representing the ONNX model
  """
  super(TensorflowBackend, cls).prepare(model, device, **kwargs)
  common.logger.setLevel(logging_level)

  return cls.onnx_model_to_tensorflow_rep(model, strict)
```

```
@classmethod
def onnx_model_to_tensorflow_rep(cls, model, strict):
  """ Convert ONNX model to TensorflowRep.

  :param model: ONNX ModelProto object.
  :param strict: whether to enforce semantic equivalence between the original model
    and the converted tensorflow model.
  :return: TensorflowRep object.
  """

  # Models with IR_VERSION less than 3 does not have opset_import set.
  # We default to minimum opset, this behavior is consistent with
  # onnx checker.
  # c.f. https://github.com/onnx/onnx/blob/427ac0c1b792363d373e3d7e4eef97fa46458420/onnx/checker.cc#L478
  if model.ir_version < 3:
    opset_import = [make_opsetid(defs.ONNX_DOMAIN, 1)]
  else:
    opset_import = model.opset_import
  return cls._onnx_graph_to_tensorflow_rep(model.graph, opset_import, strict)
```

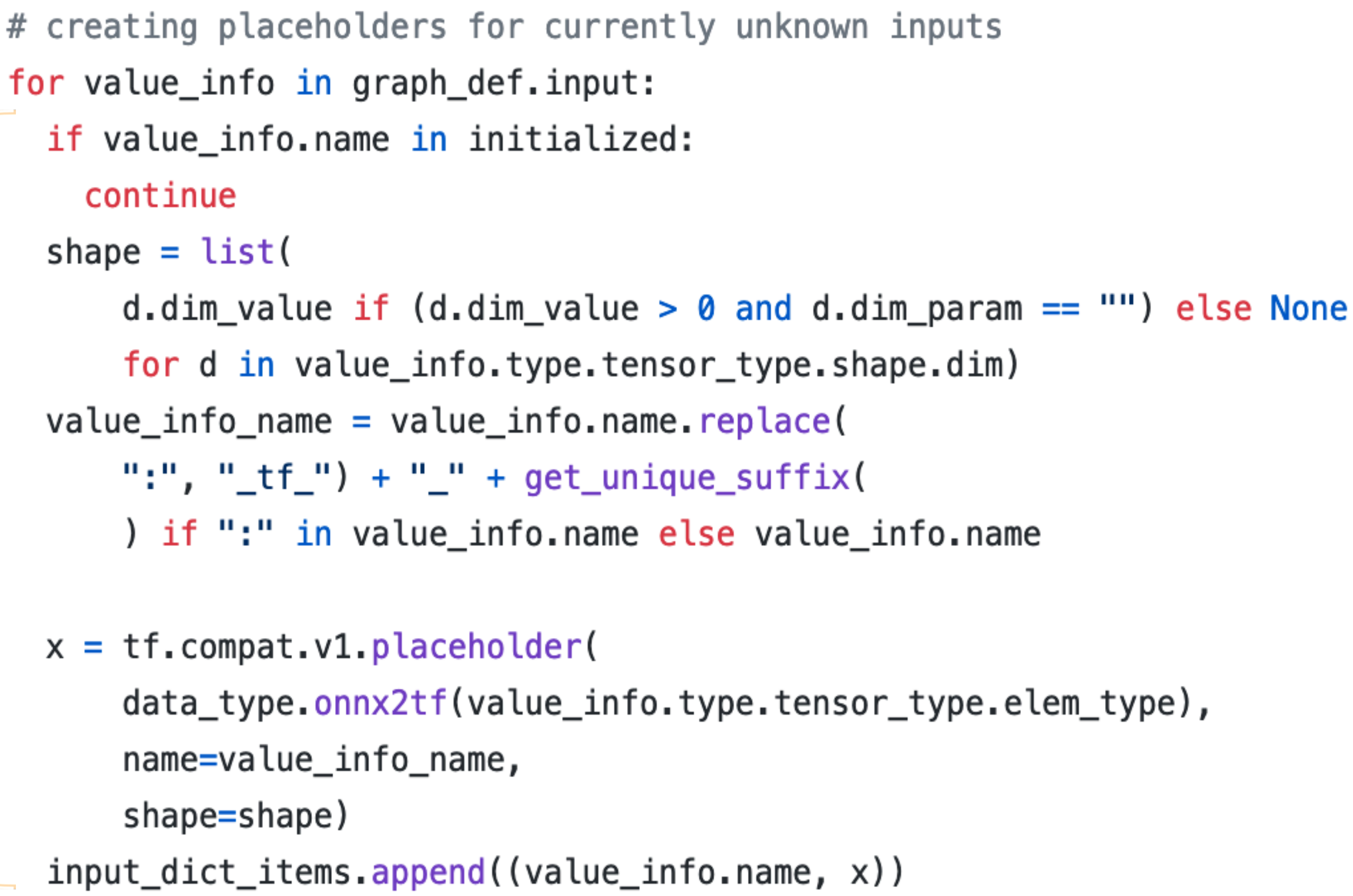


```
@classmethod
def _onnx_graph_to_tensorflow_rep(cls, graph_def, opset, strict):
  """ Convert ONNX graph to TensorflowRep.

  :param graph_def: ONNX GraphProto object.
  :param opset: ONNX OperatorSetIdProto list.
  :param strict: whether to enforce semantic equivalence between the original model
    and the converted tensorflow model.
  :return: TensorflowRep object.
  """
  handlers = cls._get_handlers(opset)

  tf_rep_graph = tf.Graph()
  with tf_rep_graph.as_default():
    # initializer: TensorProtos representing the values to initialize
    # a given tensor.
    # initialized: A list of names of the initialized tensors.
    if graph_def.initializer:
      input_dict_items = cls._onnx_initializer_to_input_dict_items(
          graph_def.initializer)
      initialized = {init.name for init in graph_def.initializer}
    else:
      input_dict_items = []
      initialized = set()
```

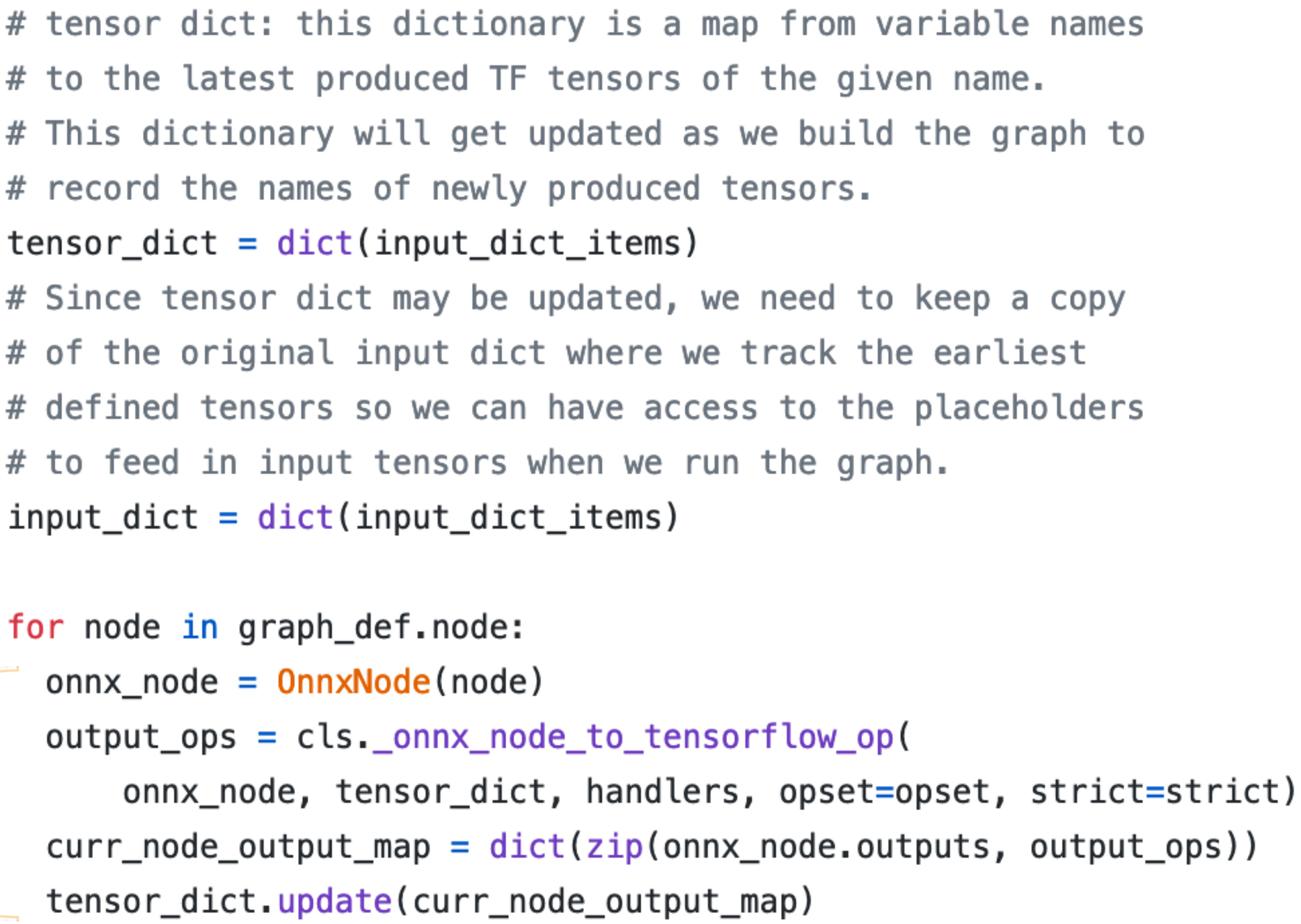
Get all supported handlers for this ONNX opset

Create Constant node to set the initial values for the required initialized tensors in the model.

Create placeholders for all the inputs of the model

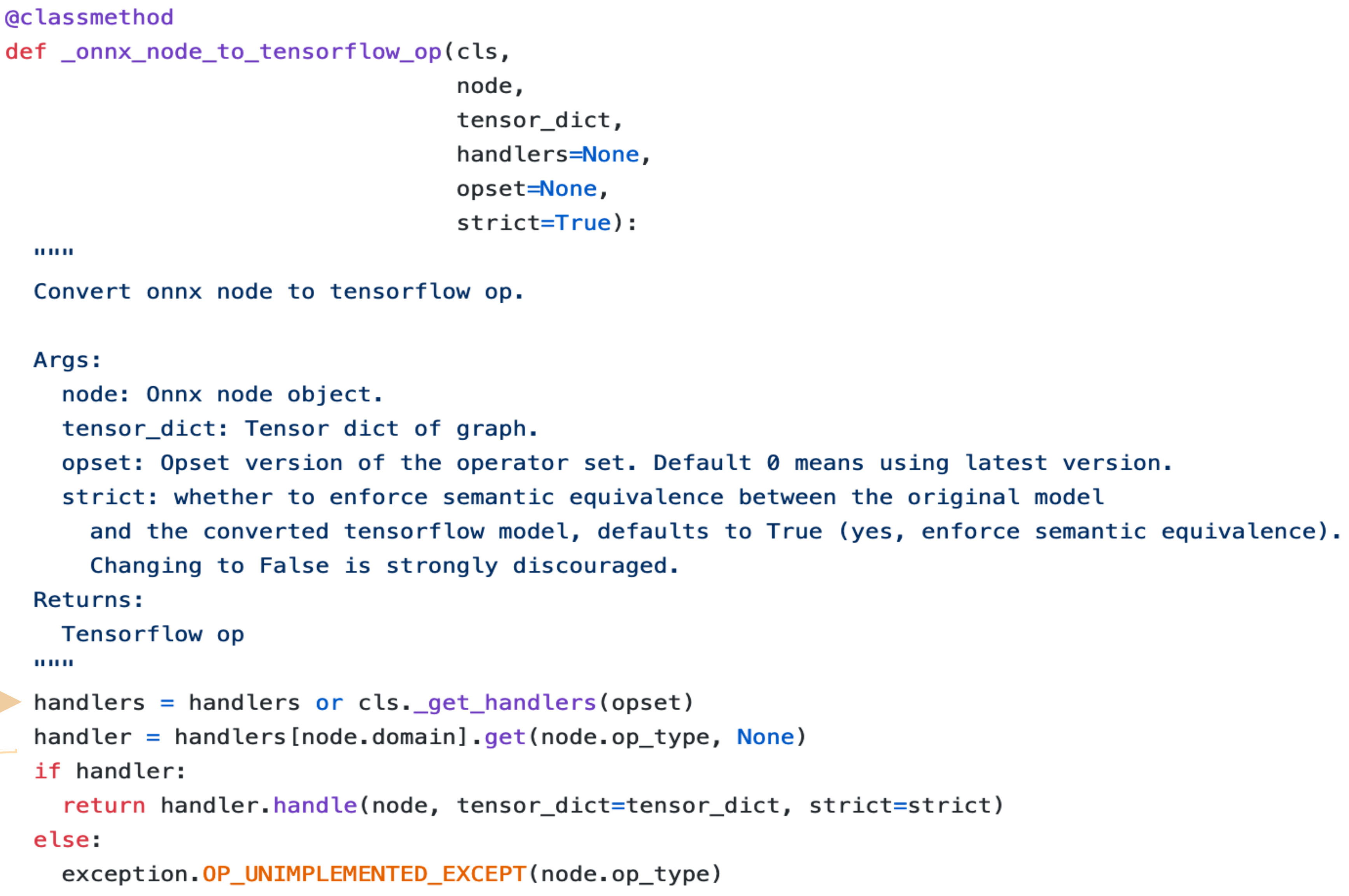


```
# creating placeholders for currently unknown inputs
for value_info in graph_def.input:
  if value_info.name in initialized:
    continue
  shape = list(
      d.dim_value if (d.dim_value > 0 and d.dim_param == "") else None
      for d in value_info.type.tensor_type.shape.dim)
  value_info_name = value_info.name.replace(
      ":", "_tf_") + "_" + get_unique_suffix(
      ) if ":" in value_info.name else value_info.name

  x = tf.compat.v1.placeholder(
      data_type.onnx2tf(value_info.type.tensor_type.elem_type),
      name=value_info_name,
      shape=shape)
  input_dict_items.append((value_info.name, x))
```



```
# tensor dict: this dictionary is a map from variable names
# to the latest produced TF tensors of the given name.
# This dictionary will get updated as we build the graph to
# record the names of newly produced tensors.
tensor_dict = dict(input_dict_items)
# Since tensor dict may be updated, we need to keep a copy
# of the original input dict where we track the earliest
# defined tensors so we can have access to the placeholders
# to feed in input tensors when we run the graph.
input_dict = dict(input_dict_items)

for node in graph_def.node:
  onnx_node = OnnxNode(node)
  output_ops = cls._onnx_node_to_tensorflow_op(
      onnx_node, tensor_dict, handlers, opset=opset, strict=strict)
  curr_node_output_map = dict(zip(onnx_node.outputs, output_ops))
  tensor_dict.update(curr_node_output_map)
```

Each ONNX nodes in the graph will call _onnx_node_to_tensorflow_op to get the corresponding handler to convert the node into the equivalent Tensorflow operator(s)



```
@classmethod
def _onnx_node_to_tensorflow_op(cls,
                                node,
                                tensor_dict,
                                handlers=None,
                                opset=None,
                                strict=True):
  """
  Convert onnx node to tensorflow op.

  Args:
    node: Onnx node object.
    tensor_dict: Tensor dict of graph.
    opset: Opset version of the operator set. Default 0 means using latest version.
    strict: whether to enforce semantic equivalence between the original model
      and the converted tensorflow model, defaults to True (yes, enforce semantic equivalence).
      Changing to False is strongly discouraged.
  Returns:
    Tensorflow op
  """
  handlers = handlers or cls._get_handlers(opset)
  handler = handlers[node.domain].get(node.op_type, None)
  if handler:
    return handler.handle(node, tensor_dict=tensor_dict, strict=strict)
  else:
    exception.OP_UNIMPLEMENTED_EXCEPT(node.op_type)
```

Get all supported handlers for this opset version

Get the specific handler (more detail explanation on the next 2 slides)

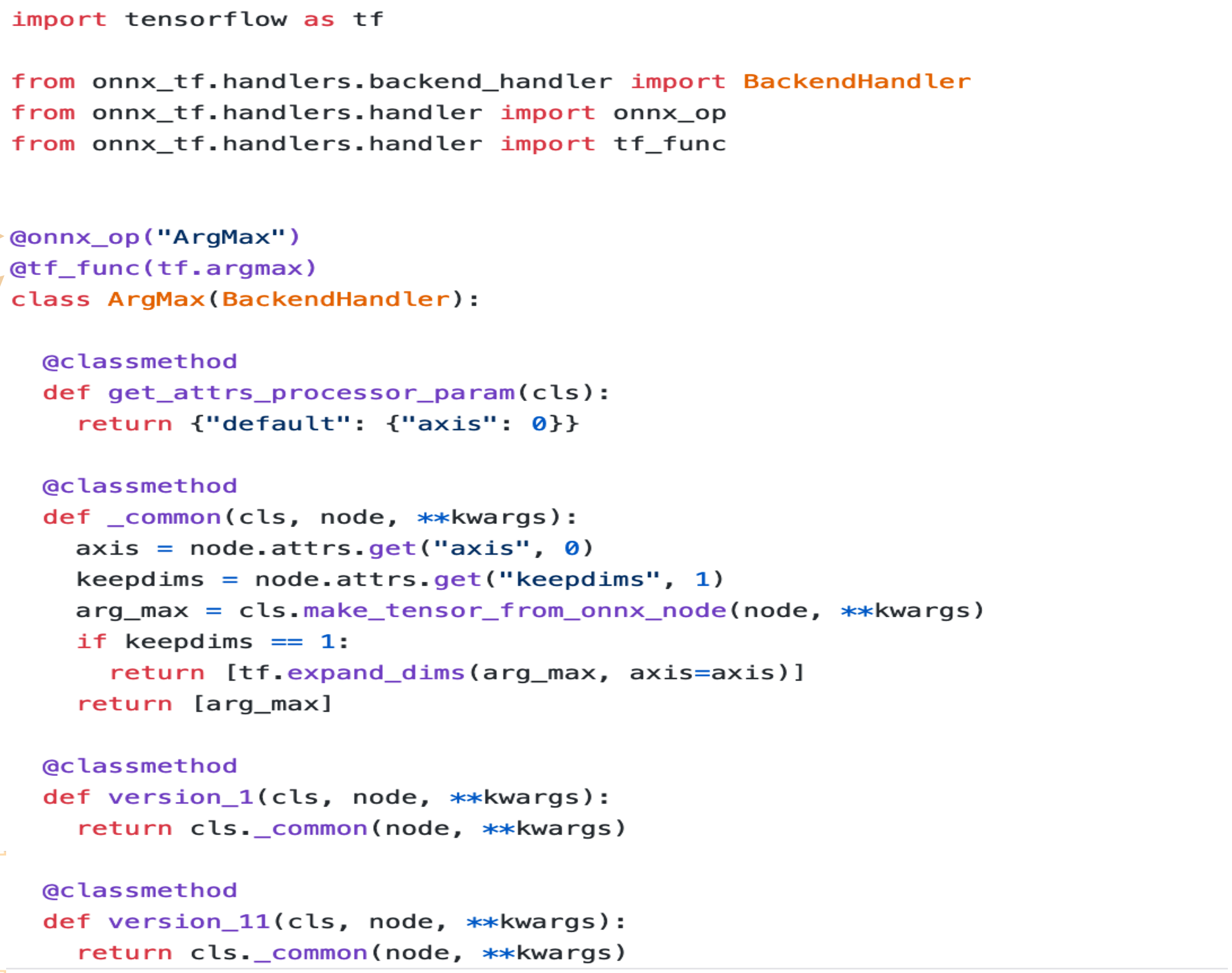
Get the specific handler by opset, domain & op_type

- opset = ONNX Opset version
- node.domain = ONNX_DOMAIN = “” or
  ONNX_ML_DOMAIN = ‘ai.onnx.ml’
- node.op_type = name of the operator in ONNX
- For example:
  - If opset = 11 and node.domain = “” and node.op_type = “ArgMax” then
  - ArgMax version_11 handler will be called to convert the ONNX “ArgMax node” into Tensoflow “ArgMax node” along with the required parameters.
  - https://github.com/onnx/onnx-tensorflow/blob/master/onnx_tf/handlers/backend/arg_max.py

This handler is called by the previous slide example is because it is defined to handle ArgMax ONNX operator

This handler will convert the node into a tf.argmax node

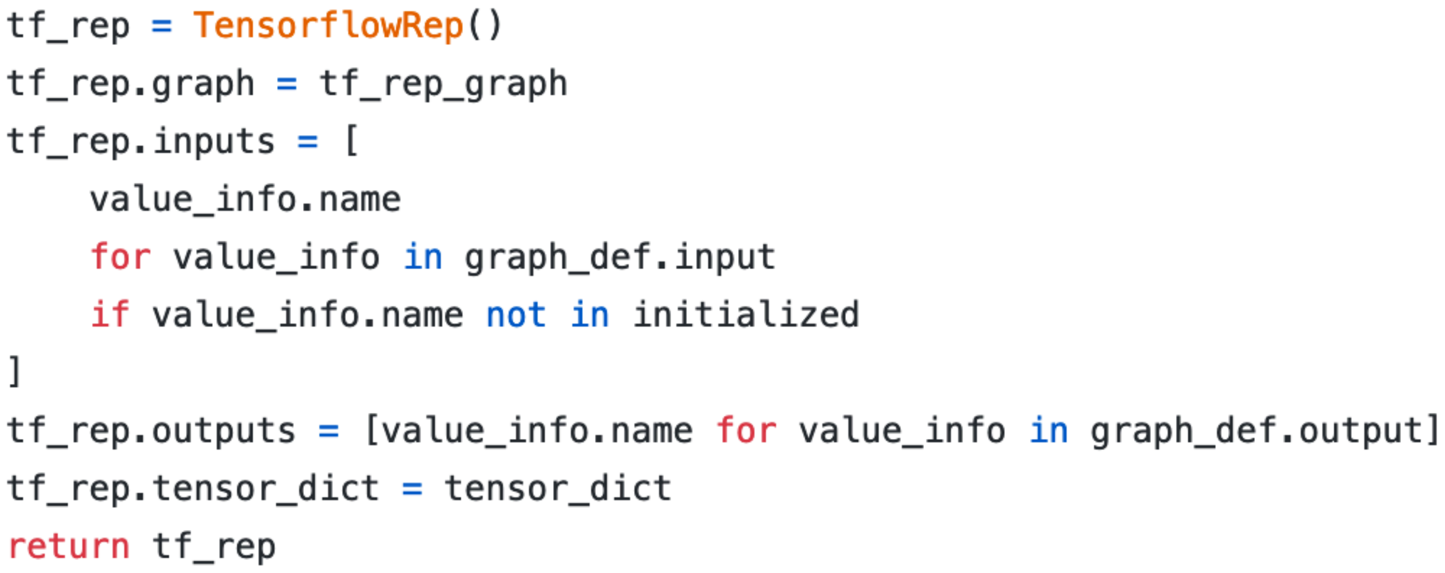
This version_11 function will be called to process the node from the previous slide to convert it to tf.argmax node



```
import tensorflow as tf

from onnx_tf.handlers.backend_handler import BackendHandler
from onnx_tf.handlers.handler import onnx_op
from onnx_tf.handlers.handler import tf_func


@onnx_op("ArgMax")
@tf_func(tf.argmax)
class ArgMax(BackendHandler):

  @classmethod
  def get_attrs_processor_param(cls):
    return {"default": {"axis": 0}}

  @classmethod
  def _common(cls, node, **kwargs):
    axis = node.attrs.get("axis", 0)
    keepdims = node.attrs.get("keepdims", 1)
    arg_max = cls.make_tensor_from_onnx_node(node, **kwargs)
    if keepdims == 1:
      return [tf.expand_dims(arg_max, axis=axis)]
    return [arg_max]

  @classmethod
  def version_1(cls, node, **kwargs):
    return cls._common(node, **kwargs)

  @classmethod
  def version_11(cls, node, **kwargs):
    return cls._common(node, **kwargs)
```

Back to _onnx_graph_to_tensorflow_rep

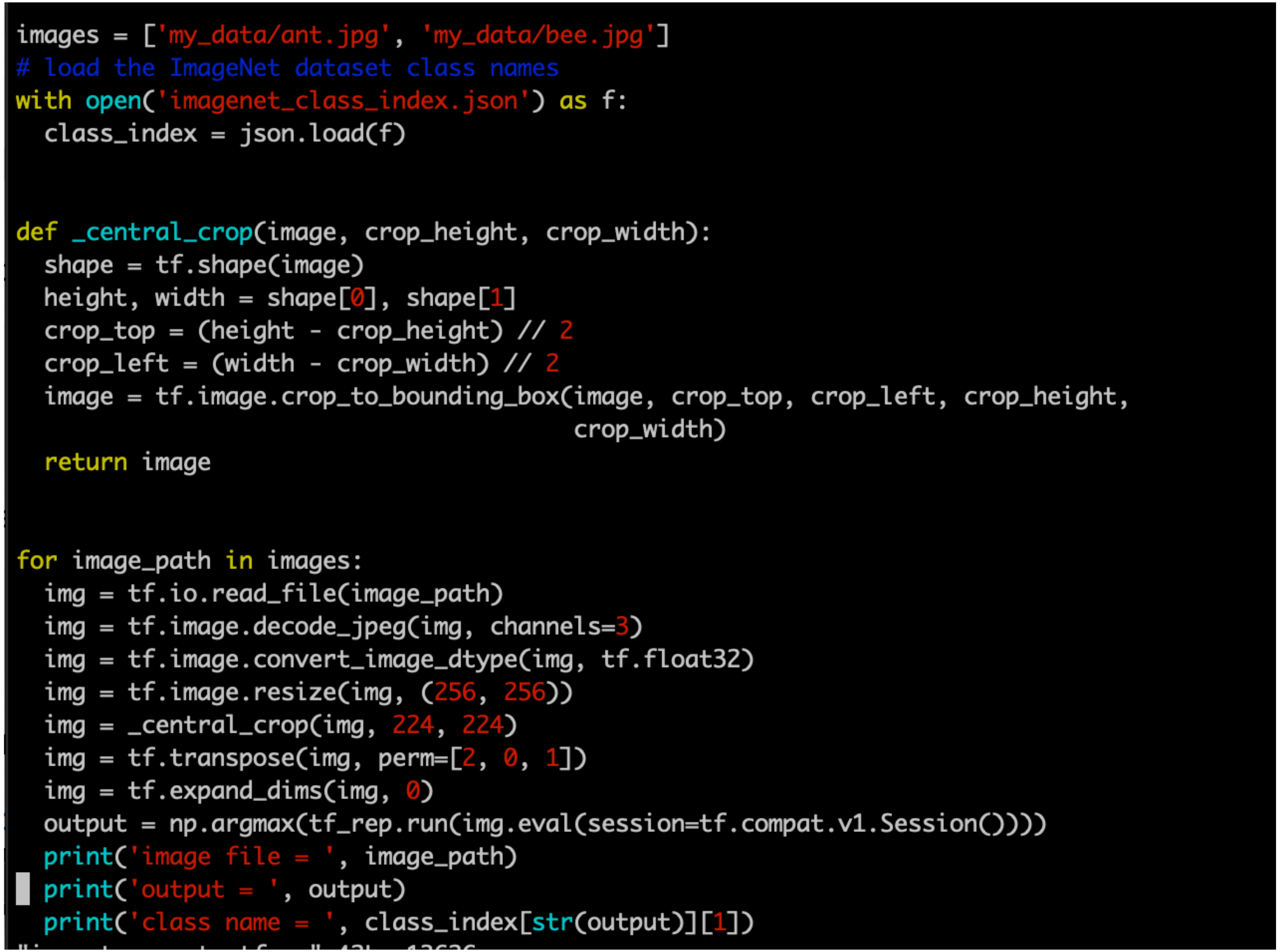
Create TensorflowRep then save the graph, inputs, outputs and tensor_dict in it



```
tf_rep = TensorflowRep()
tf_rep.graph = tf_rep_graph
tf_rep.inputs = [
    value_info.name
    for value_info in graph_def.input
    if value_info.name not in initialized
]
tf_rep.outputs = [value_info.name for value_info in graph_def.output]
tf_rep.tensor_dict = tensor_dict
return tf_rep
```

Run the converted model

- Load the class index json file
- Preprocess the test data
- Run the model
- Decode the result with the class index



Load ImageNet class index json file

```
images = ['my_data/ant.jpg', 'my_data/bee.jpg']
# load the ImageNet dataset class names
with open('imagenet_class_index.json') as f:
  class_index = json.load(f)
```

Preprocess the test images

```
def _central_crop(image, crop_height, crop_width):
  shape = tf.shape(image)
  height, width = shape[0], shape[1]
  crop_top = (height - crop_height) // 2
  crop_left = (width - crop_width) // 2
  image = tf.image.crop_to_bounding_box(image, crop_top, crop_left, crop_height,
                                        crop_width)
  return image


for image_path in images:
  img = tf.io.read_file(image_path)
  img = tf.image.decode_jpeg(img, channels=3)
  img = tf.image.convert_image_dtype(img, tf.float32)
  img = tf.image.resize(img, (256, 256))
  img = _central_crop(img, 224, 224)
```

Transpose NHWC images into NCHW format required by the model

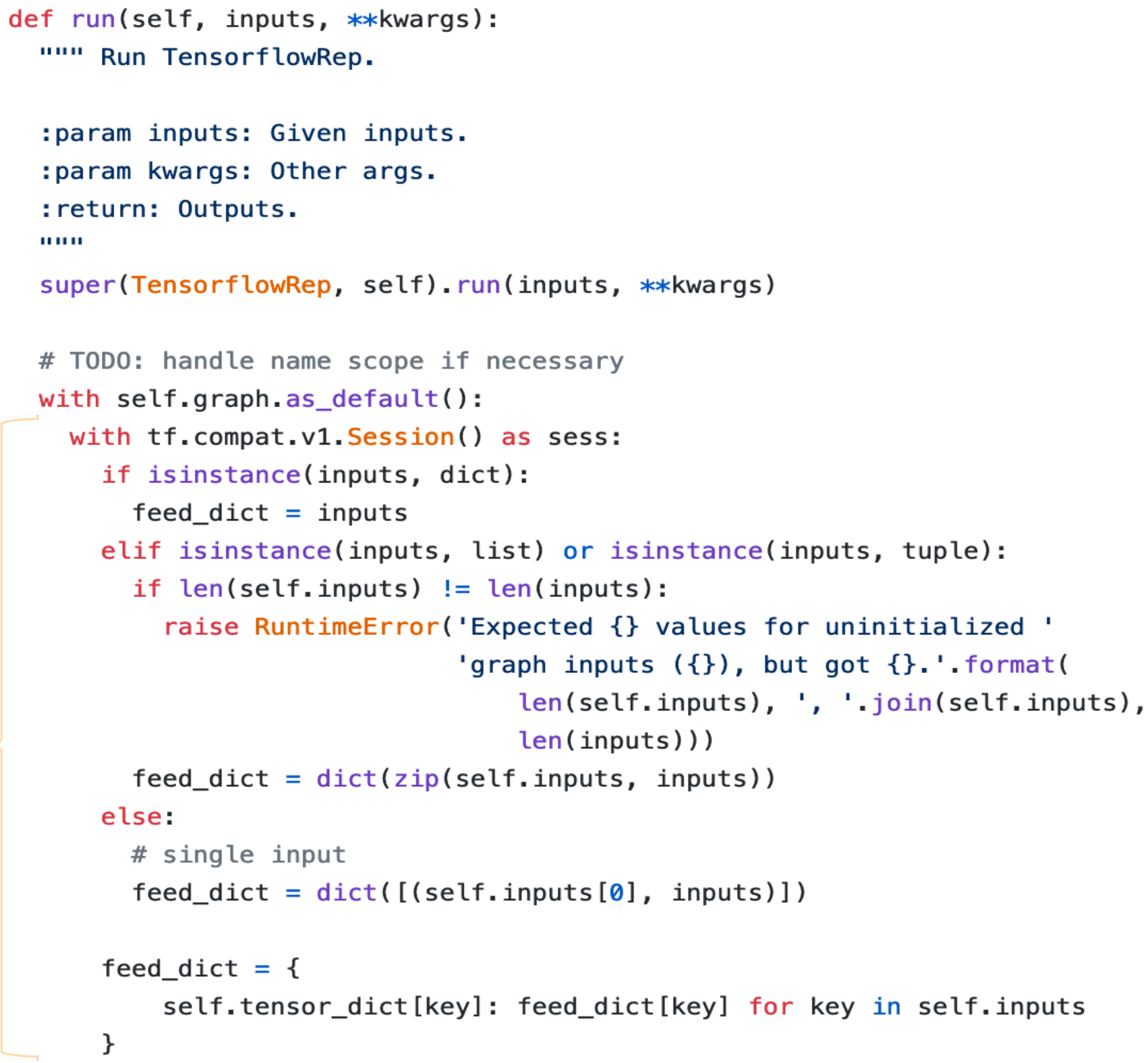
```
  img = tf.transpose(img, perm=[2, 0, 1])
  img = tf.expand_dims(img, 0)
```

Run the model with the test data and decode the result

```
  output = np.argmax(tf_rep.run(img.eval(session=tf.compat.v1.Session())))
  print('image file = ', image_path)
  print('output = ', output)
  print('class name = ', class_index[str(output)][1])
```

Using run to run the converted model

- Run is a function to run the Tensorflow graph in the TensorflowRep object
- The required input parameters are the inputs to the model
- The output is the inference result of the model

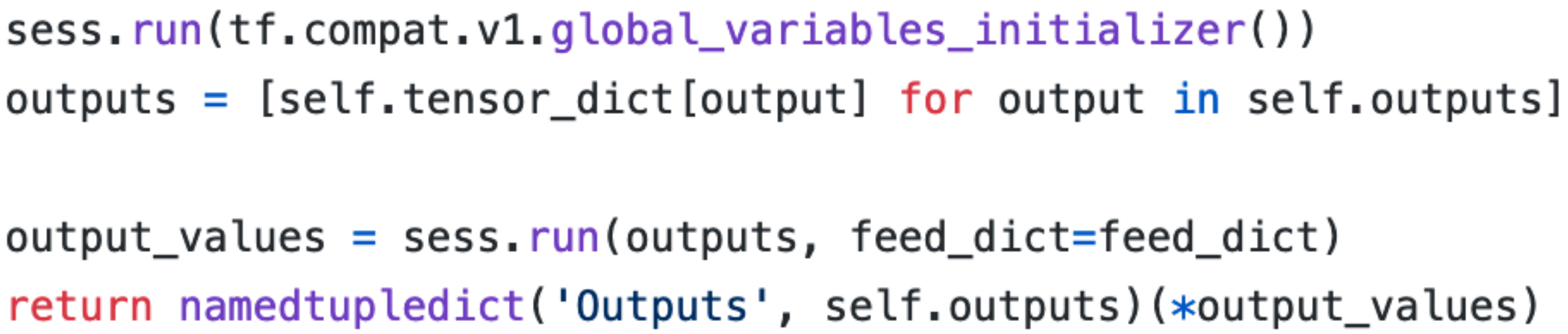


```python
def run(self, inputs, **kwargs):
  """ Run TensorflowRep.

  :param inputs: Given inputs.
  :param kwargs: Other args.
  :return: Outputs.
  """
  super(TensorflowRep, self).run(inputs, **kwargs)

  # TODO: handle name scope if necessary
  with self.graph.as_default():
    with tf.compat.v1.Session() as sess:
      if isinstance(inputs, dict):
        feed_dict = inputs
      elif isinstance(inputs, list) or isinstance(inputs, tuple):
        if len(self.inputs) != len(inputs):
          raise RuntimeError('Expected {} values for uninitialized '
                             'graph inputs ({}), but got {}.'.format(
                                 len(self.inputs), ', '.join(self.inputs),
                                 len(inputs)))
        feed_dict = dict(zip(self.inputs, inputs))
      else:
        # single input
        feed_dict = dict([(self.inputs[0], inputs)])

      feed_dict = {
          self.tensor_dict[key]: feed_dict[key] for key in self.inputs
      }
```

Create the input dictionary for the model

Get all outputs of the model

Run the model with the input dictionary



```
sess.run(tf.compat.v1.global_variables_initializer())
outputs = [self.tensor_dict[output] for output in self.outputs]

output_values = sess.run(outputs, feed_dict=feed_dict)
return namedtupledict('Outputs', self.outputs)(*output_values)
```