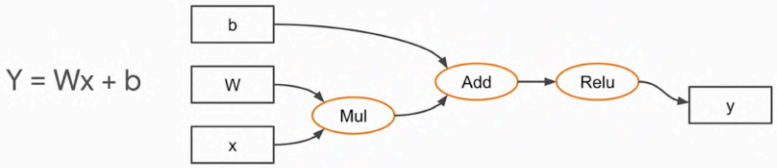
Why graphs?



TensorFlow was originally designed around programming being done in graph mode, where you had to define a graph with all of your operations before you executed it.

For example, if you wanted a formula like ReLu of y equals ReLu of Wx plus b, you'd have a graph like this. You would treat w and x as variables that get loaded into a multiplication operation, or op for short, the results of which will get loaded into an add op along with the variable b, and the results of that were loaded into another op such as ReLu to give us the answer y.

While these may not seem to be as intuitive to you as a Python developer, they do operate really quickly, and using graphs can definitely speed up training and inference time. But they are difficult to code, and because the operations such as multiply, add, and ReLu don't take place until the graph is fully designed, they can be difficult to debug.

Eager mode

- An intuitive interface
- Easier debugging
- Natural control flow

```
def f(x):
  if x > 0:
    x = x * x
  return x
```

Graph mode

- Parallelism
- Distributed execution
- Compilation
- Portability

It goes beyond just variables and ops. Consider for example, control flow. Here we have a function with an "if" statement in it.

If x is greater than zero, return x squared, otherwise return x. With Eager execution in Python, you write very familiar and very simple Pythonic code.

```
@tf.function
def f(x):
    def if_true():
        return x * x
    def if_false():
        return x
    x = tf.cond(
          tf.greater(x, 0),
          if_true,
          if_false)
    return x
```

Eager mode

- An intuitive interface
- Easier debugging
- Natural control flow

```
def f(x):
  if x > 0:
    x = x * x
  return x
```

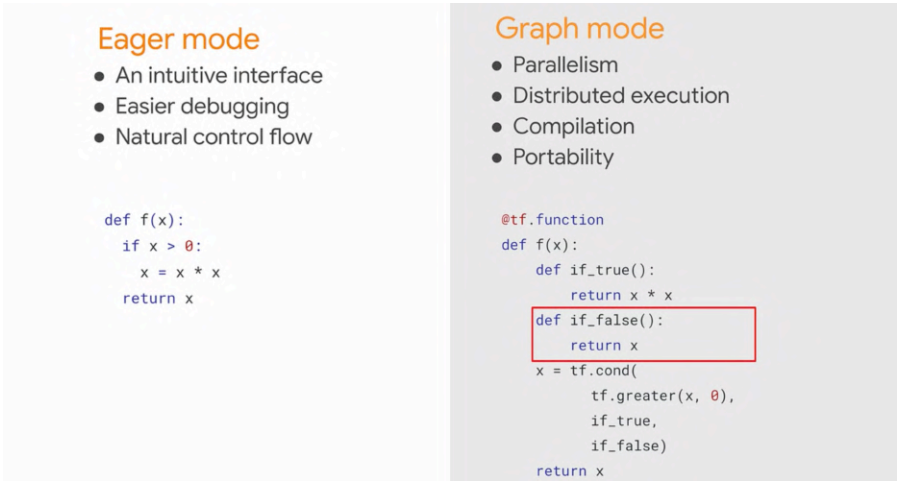
Graph mode

- Parallelism
- Distributed execution
- Compilation
- Portability

But graphs don't support "if" conditionals, so you would have to write code like this using a tf.cond conditional.

Here, you can compare if x is greater than zero using tf.greater, with the next parameter of being a function that's called if it's true, so we'll call that if true, and then parameter after that then names the function to be called if it's false, so we can call that if false. Calling if true will return x squared.

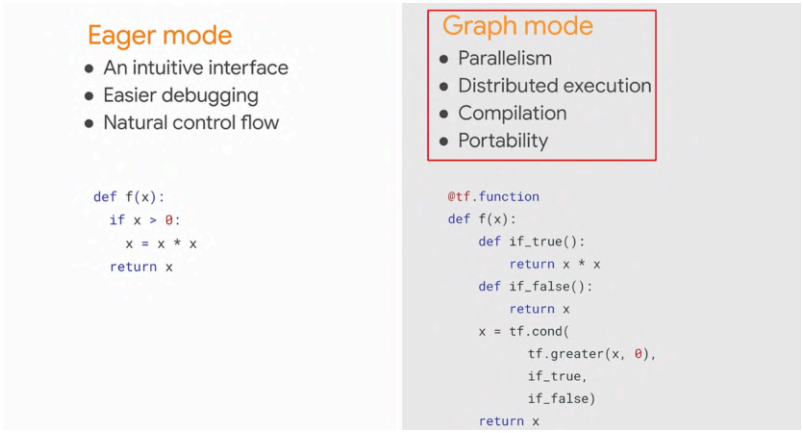
```
@tf.function
def f(x):
    def if_true():
        return x * x
    def if_false():
        return x
    x = tf.cond(
          tf.greater(x, 0),
          if_true,
          if_false)
    return x
```



Calling if true will return x squared. Calling if false will return x. Eager mode lets you write more or less standard Python code with standard control flow syntax.

But you lose some of the benefits that graphs give you. In AutoGraph, graphs have explicit dependencies. By this, I mean that if you look at any node in the graph, you can find out which operations it depends on to execute beforehand by tracing backwards through the graph.

Knowing the dependencies for each operation allow you to look for efficiencies when certain operations, allowing you to run some operations in parallel, for example, or you could distribute them for different machines.



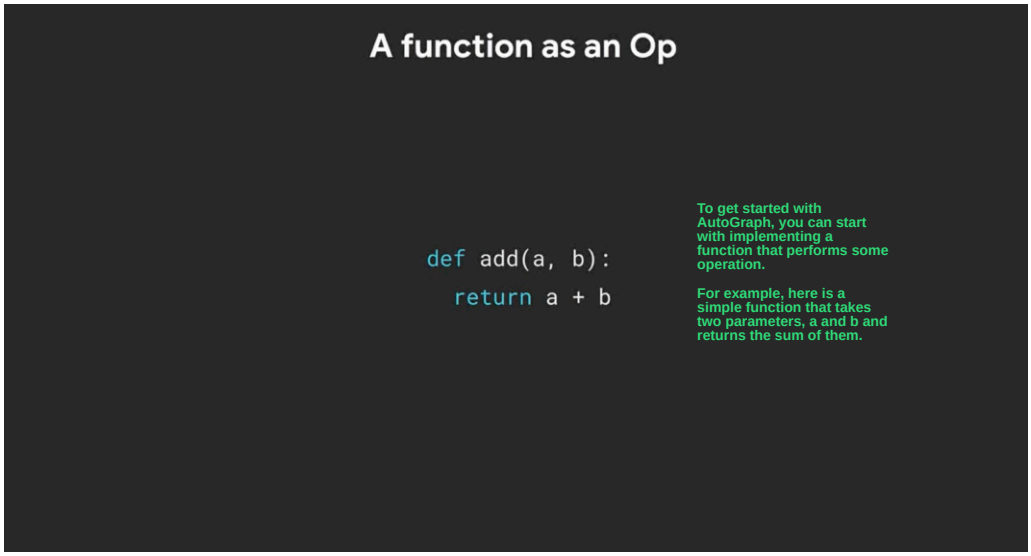
Eager and graph mode do seem to conflict with each other, but we can imagine benefiting from both approaches.

Using Eager mode, for example, if we develop a new model or we're debugging, and then switch to graph mode if we want to squeeze some more performance out of it. Or maybe we're ready to deploy to production and we want the best possible model. You might wonder, why would you write code like this instead of the easy and familiar Pythonic way of doing it?

Graphs do have explicit dependencies, and that makes it relatively easy for you to parallelize and distribute the computation. That also allows for a whole program optimizations like kernel fusion when using GPUs and something in TensorFlow called XLA. But all that is beyond the scope of this course.

You can benefit from using both approaches. One workflow is to use Eager mode when developing and debugging a new model, and then switching to graph mode if you want to speed it up.

Here's where AutoGraph really can help you. It's a technology that allows you to take your Eager-style Pythonic code and automatically turn it into graphs and vice versa.



To get started with AutoGraph, you can start with implementing a function that performs some operation.

For example, here is a simple function that takes two parameters, a and b and returns the sum of them.

A function as an Op

```
@tf.function
def add(a, b):
  return a + b
```

You then decorate your add function by adding @tf.function in the line above your function definition. You can think of this decorator as taking your custom code, a plus b, and wrapping it inside a pre-built tf.function.

So your add function now has the features of tf.function combined with your custom code.

A function as an Op

Your decorated add function now has graph codes that you can take a look at.

If you want to explore what the graph code would look like, you can use tf.autographst_to_code() method and then pass in the add function or whatever function you want that was defined using that tf.function.

```
print(tf.autograph.to_code(add.python_function))
```

```
def tf__add(a, b):
    do_return = False
    retval_ = ag__.UndefinedReturnValue()
    with ag__.FunctionScope('add', 'fscope',
         ag__.ConversionOptions(recursive=True, user_requested=True,
         optional_features=(), internal_convert_user_code=True)) as fscope:
        try:
            do_return = True
            retval_ = fscope.mark_return_value((a + b))
        except:
            do_return = False
            raise
    (do_return,)
    return ag__.retval(retval_)
```

There's a lot of plumbing here that makes this Op within graph mode.

But because it's a very simple Op that just adds a plus b, you can see that in here.

Functions have gradients

```
@tf.function
def add(a, b):
  return a + b
```

```
v = tf.Variable(1.0)

with tf.GradientTape() as tape:
  result = add(v, 1.0)

>>> tape.gradient(result, v).numpy()
1.0
```

You can compute your gradient operations on graph style code too.

Here's an example of computing a gradient on the add function that you just defined. You can define a variable V with a value of one, inside tf.GradientTape width block, you can then call your custom add function, passing it the variable v and one, and storing the result.

To calculate the gradient, you can then call tape.gradient, passing in the result of v. This calculates the gradient of results with respect to v; and this gives us one.

Chain multiple functions

```
def linear_layer(x):
  return 2*x + 1

@tf.function
def deep_net(x):
  return tf.nn.relu(linear_layer(x))

>>> deep_net(tf.constant((1, 2, 3)))
[3, 5, 7]
```

Functions are polymorphic

```
@tf.function
def double(a):
  return a + a
```

```
>>> double(tf.constant(1)).numpy()
2

>>> double(tf.constant(1.1)).numpy()
2.2

>>> double(tf.constant("a")).numpy()
b'aa'
```

Python's Functions are polymorphic, which means that a function which takes in a function of parameter can work whether the parameter passed as an integer or a float, or even a string. This polymorphism applies to graph style code as well.

You can see all of the operations here work like they would when just coded in Python. Here, we defined a decorated a function named double, which takes a parameter and adds it to itself.

They can work if we give it an integer one for which it will return the number 2, it'll work with a float of 1.1 for which it will return 2.2, and it also works with a string, for example, with the letter a for which it will return the string aa.

Note that the prefix b that appears before the string is meant to denote that aa is stored in a byte literal. Don't worry too much about this, but when data is stored as a byte literal, it just means that it's storing numbers instead of strings.

tf.function with Keras

```
class CustomModel(tf.keras.models.Model):
  @tf.function
  def call(self, input_data):
    if tf.reduce_mean(input_data) > 0:
      return input_data
    else:
      return input_data // 2
```

When defining a subclass of Keras classes as you've done earlier in this course, you can also use graphs. Here, I've defined a class called custom model, which inherits from a Keras model class.

Within the class, I've defined the call method decorated with tf.function so it will be converted into graph mode for me too.

FizzBuzz

"Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz"."

- *http://wiki.c2.com/?FizzBuzzTest*

```
def fizzbuzz(max_num):
  counter = 0
  for num in range(max_num):
    if num % 3 == 0 and num % 5 == 0:
      print('FizzBuzz')
    elif num % 3 == 0:
      print('Fizz')
    elif num % 5 == 0:
      print('Buzz')
    else:
      print(num)
    counter += 1
  return counter
```

For quick note on performance, using AutoGraph will have its biggest performance gain for code that uses lots and lots of Ops. That code doesn't have to be inherently complex, and often a very simple piece of pythonic code can use a lot more Ops than you might think.

Consider the simple game of FizzBuzz. It's a simple algorithm looped through seven numbers. If it's a multiple of three, print Fizz, if it's a multiple of five print Buzz, and if it's a multiple of both, print FizzBuzz. But look how many Ops are there to do this, not to mention the conditionals and the control flow. This type of code can look very complex in graph mode, but can execute much quicker and is the best type of scenario for using graph.

Remember that code that uses lots of small Ops tends to have the best performance improvement. If you look at FizzBuzz in graph mode, it will be very hard to code by hand.

Generate AutoGraph code

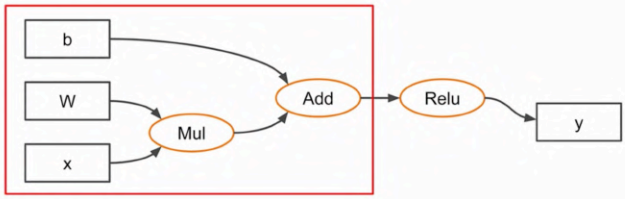
```
@tf.function
def f(x):
  while tf.reduce_sum(x) > 1:
    tf.print(x)
    x = tf.tanh(x)
  return x

>>> tf.autograph.to_code(f.python_function)
```

In the previous section, you had an introduction to autograph, and how by decorating your function with tf.function, TensorFlow could create graph code under the hood for you. As a reminder, if you want to see the generated code, you can just call tf autograph.to_code, passing in your function name with a Python function property.

```
def tf__f(x):
  do_return = False
  retval_ = ag__.UndefinedReturnValue()
  with ag__.FunctionScope(...) as f_scope:
    def get_state():
      return ()

    def set_state(_):
      pass

    def loop_body(x):
      ...
      return x,

    def loop_test(x):
      return ...
    x, = ag__.while_stmt(loop_test,
                         loop_body,
                         get_state,
                         set_state, ...)
    do_return = True
    retval_ = f_scope.mark_return_value(x)
  do_return,
  return ag__.retval(retval_)
```

Order of Execution

$Y = Wx + b$



One thing to keep an eye on when writing code that will be automatically converted to a graph is to make sure that the order of execution is what you intended.

For simple graph like this, you can see that you want to multiply W by x first, and then add the results to b. It's important to remember that when writing Python, that the graph instructions will be implemented in the same order as the original Python code.

Automatic Control Dependencies

```
a = tf.Variable(1.0)
b = tf.Variable(2.0)

@tf.function
def f(x, y):
  a.assign(y * b)
  b.assign_add(x * a)
  return a + b

>>> f(1.0, 2.0)
10.0
```

Let's take a look at an example. Using tf function to automatically generate graphs from regular Python code helps you to avoid designing complicated graphs by yourself.

An example of code that would have complicated graphs is one where the same variable is used as both the input for some calculations, but maybe also used to store the results of other calculations.

For example, consider this code where we're doing lots of reads and writes to a and b. The answer of f of 1, 2 is 10. But how does it get that? We'll start with a being initialized to 1.0, and b to 2.0.

Automatic Control Dependencies

```
a = tf.Variable(1.0)
b = tf.Variable(2.0)

@tf.function
def f(x, y):
  a.assign(y * b)
  b.assign_add(x * a)
  return a + b

>>> f(1.0, 2.0)
10.0
```

```
a = y * b
a = 2.0 * 2.0
a = 4.0
```

The first thing that happens is that the a will get assigned to y times b. B is 2.0, and y is the second parameter to the function, which is also 2.0, so a now gets assigned the value 4.0.

Automatic Control Dependencies

```
a = tf.Variable(1.0)
b = tf.Variable(2.0)

@tf.function
def f(x, y):
  a.assign(y * b)
  b.assign_add(x * a)
  return a + b

>>> f(1.0, 2.0)
10.0
```

```
a = y * b
a = 2.0 * 2.0
a = 4.0

b = 2.0
b = b + x * a
b = 2.0 + 1 * 4.0
b = 6.0
```

B is already 2.0, but by calling a sign add, we're multiplying x, which is one by a, which after the previous line is four, so that the result becomes four. This, when added to b, now makes b equal to 6.

Automatic Control Dependencies

When we return a plus b, we get 10. Now try to imagine how crowded the graph of this would be with all of the changing assignments to a and b, and you could see how inherently complex this might be.

If you try to design this graph yourself, it'll be very hard to avoid an unintentional mistake.

Fortunately, autograph handles that complexity for you, so you don't have to worry about it.

```
a = tf.Variable(1.0)
b = tf.Variable(2.0)

@tf.function
def f(x, y):
  a.assign(y * b)
  b.assign_add(x * a)
  return a + b

>>> f(1.0, 2.0)
10.0
```

```
a = y * b
a = 2.0 * 2.0
a = 4.0

b = 2.0
b = b + x * a
b = 2.0 + 1 * 4.0
b = 6.0

a + b
4.0 + 6.0
10.0
```

```
@tf.function
def sign(x):
  if x > 0:
    return 'Positive'
  else:
    return 'Negative'
```

```
def if_true():
    try:
        do_return = True
        retval_ = fscope.mark_return_value('Positive')
    except:
        do_return = False
        raise
    return (retval_, do_return)

def if_false():
    try:
        do_return = True
        retval_ = fscope.mark_return_value('Negative')
    except:
        do_return = False
        raise
    return (retval_, do_return)

cond = (x > 0)

(retval_, do_return) = ag__.if_stmt(cond, if_true, if_false,
        get_state, set_state, ('retval_', 'do_return'), ())
```

You can also use autograph to generate graphs for conditional control flows.

For example, consider this simple function where if x is greater than 0, we'll return a string that says positive, otherwise we'll return a string that says negative. Let's look at the original Python code that you'd write on the left, and the graph code that autograph will generate for you on the right.

We'll start by seeing how the check, if x is greater than 0 is implemented in graph code. Let's look at the bottom few lines of code on the right. First, the expression, x greater than 0, which evaluates to a true false Boolean, is stored in a variable named cond, and the next line below that, the object ag__ has a function named if statements, which takes in several parameters.

We'll focus on the first three. The first parameter is the Boolean condition, the second parameter is the function to call if the condition is true, and the third parameter is the function to call if the condition is false.

The equivalent of the simple return positive, becomes a function, and this function is called if true, which you can see here. It includes error checking, which is done within a try except block.

Now remember, autograph has generated all of this code for you.

```
@tf.function
def sign(x):
  if x > 0:
    return 'Positive'
  else:
    return 'Negative'
```

```
def if_true():
    try:
        do_return = True
        retval_ = fscope.mark_return_value('Positive')
    except:
        do_return = False
        raise
    return (retval_, do_return)

def if_false():
    try:
        do_return = True
        retval_ = fscope.mark_return_value('Negative')
    except:
        do_return = False
        raise
    return (retval_, do_return)

cond = (x > 0)

(retval_, do_return) = ag__.if_stmt(cond, if_true, if_false,
         get_state, set_state, ('retval_', 'do_return'), ())
```

```
@tf.function
def sign(x):
  if x > 0:
    return 'Positive'
  else:
    return 'Negative'
```

```
def if_true():
    try:
        do_return = True
        retval_ = fscope.mark_return_value('Positive')
    except:
        do_return = False
        raise
    return (retval_, do_return)

def if_false():
    try:
        do_return = True
        retval_ = fscope.mark_return_value('Negative')
    except:
        do_return = False
        raise
    return (retval_, do_return)

cond = (x > 0)

(retval_, do_return) = ag__.if_stmt(cond, if_true, if_false,
         get_state, set_state, ('retval_', 'do_return'), ())
```

Similarly, for the else branch of the if statement on the left, autograph generates a function called if_false.

Remember that the graph code on the rights, allows for some computation of efficiencies even though it's much more work to write by hand.

The code on the left is easy for you to write. You can just write the code on the left, and then let autograph create the code on the right.

Control flows (loops)

```
@tf.function
def f(x):
  while tf.reduce_sum(x) > 1:
    tf.print(x)
    x = tf.tanh(x)
  return x
```

Also consider control flow such as loops. Here's an example of a while loop which is inherently complex in it's operation. Within the loop, tf.print (x) we'll print the value stored in x.

In the next line, x equals tf.tanh(x) will pass the value at x through a tanh function and then store the result of that in x.

Control flows (loops)

```python
@tf.function
def f(x):
  while tf.reduce_sum(x) > 1:
    tf.print(x)
    x = tf.tanh(x)
  return x
```

The loop will continue as long as the reduced sum of x is greater than one. If you add up all of the values inside the Tensor x and the sum is greater than one, then the loop continues.

This function tries to keep changing the tanh operation on the Tensor x as long as the sum of all it's elements is higher than one.

```python
@tf.function
def f(x):
  while tf.reduce_sum(x) > 1:
    tf.print(x)
    x = tf.tanh(x)
  return x
```

```python
def get_state():
    return (x,)

def set_state(loop_vars):
    nonlocal x
    (x,) = loop_vars

def loop_body():
    nonlocal x
    ag__.converted_call(tf.print, (x,), None, fscope)
    x = ag__.converted_call(tf.tanh, (x,), None,
        fscope)

def loop_test():
    return (ag__.converted_call(tf.reduce_sum, (x,),
        None, fscope) > 1)

ag__.while_stmt(loop_test, loop_body, get_state,
    set_state, ('x',), {})

try:
    do_return = True
    retval_ = fscope.mark_return_value(x)
except:
    do_return = False
    raise
```

```python
@tf.function
def f(x):
  while tf.reduce_sum(x) > 1:
    tf.print(x)
    x = tf.tanh(x)
  return x
```

```python
def get_state():
    return (x,)

def set_state(loop_vars):
    nonlocal x
    (x,) = loop_vars

def loop_body():
    nonlocal x
    ag__.converted_call(tf.print, (x,), None, fscope)
    x = ag__.converted_call(tf.tanh, (x,), None,
        fscope)

def loop_test():
    return (ag__.converted_call(tf.reduce_sum, (x,),
        None, fscope) > 1)

ag__.while_stmt(loop_test, loop_body, get_state,
    set_state, ('x',), {})

try:
    do_return = True
    retval_ = fscope.mark_return_value(x)
except:
    do_return = False
    raise
```

```python
@tf.function
def f(x):
  while tf.reduce_sum(x) > 1:
    tf.print(x)
    x = tf.tanh(x)
  return x
```

```python
def get_state():
    return (x,)

def set_state(loop_vars):
    nonlocal x
    (x,) = loop_vars

def loop_body():
    nonlocal x
    ag__.converted_call(tf.print, (x,), None, fscope)
    x = ag__.converted_call(tf.tanh, (x,), None,
        fscope)

def loop_test():
    return (ag__.converted_call(tf.reduce_sum, (x,),
        None, fscope) > 1)


ag__.while_stmt(loop_test, loop_body, get_state,
    set_state, ('x',), {})

try:
    do_return = True
    retval_ = fscope.mark_return_value(x)
except:
    do_return = False
    raise
```

Default behavior of tracing variables (eager mode)

```python
def f(x):
  print("Traced with", x)

for i in range(5):
  f(2)
```

```
Traced with 2
Traced with 2
Traced with 2
Traced with 2
Traced with 2
```

Another thing to consider when using autograph is how moving from eager execution code to graph-based code can impact tracing statements.

Tracing statement is code that's used to help you, the developer, keep track of which function has been executed and variable values as lot while the code is running.

For example, consider the code on the left where we create a function called f, which we then called with a parameter x and it will print the statement that it was traced with x.

Call that function within a loop, like you can see here, the function will be called multiple times, resulting in multiple prints.

Use TensorFlow Ops to trace (graph mode)

```python
@tf.function
def f(x):
  print("Traced with", x)
  tf.print("Executed with", x)

for i in range(5):
  f(2)
```

```
Traced with 2
Executed with 2
Executed with 2
Executed with 2
Executed with 2
Executed with 2
```

But now let's consider what happens if we decorate f with tf.function, meaning it will operate in graph mode.

Also notice that now we've added another line of code after the print statement.

This is now using TensorFlow's print statement, tf.print and giving it the string executed with followed by x.

Let's take a look at how Python's print intensive los tf.print behave differently when the function is decorated with tf.function

Use TensorFlow Ops to trace (graph mode)

```
@tf.function
def f(x):
  print("Traced with", x)
  tf.print("Executed with", x)

for i in range(5):
  f(2)
```

```
Traced with 2
Executed with 2
Executed with 2
Executed with 2
Executed with 2
Executed with 2
```

As before, we'll call f from within a loop five times. Notice that the traced with statement is only printed once.

This is because the Python print statement isn't designed to work within graph or the session that's executed by the graph. It thinks it's only being called once.

In contrast, TensorFlow's tf.print, which is graph aware, will print correctly and as you can see, it executed with two runs five times.

Dangerous variable creation behavior

```
@tf.function
def f(x):
  v = tf.Variable(1.0)
  v.assign_add(x)
  return v
```

```
>>> f(1)

Caught expected exception
  <class 'ValueError'>: in
converted code:
...
```

Another thing to consider is where you create your variables, it's common to think that they can be created within the function scope as local function variables.

But this can lead to errors in graph mode code, were variables and OPS are supposed to be kept and treated separately.

Declare outside the function

```
v = tf.Variable(1.0)
@tf.function
def f(x):
  v.assign_add(x)
  return v
```

Moving the declaration outside the function like this will convert it properly.