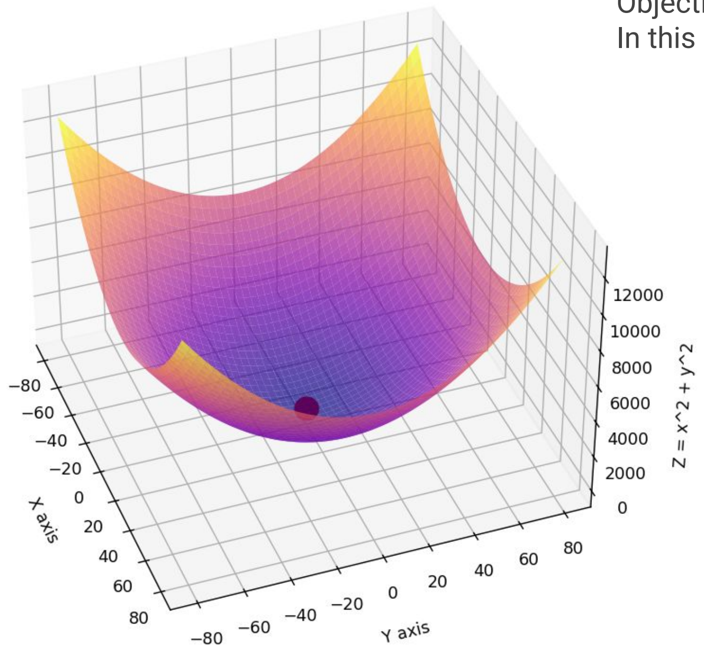
TENSORFLOW BASICS : GradientTape

Gradient Descent Algorithm

Gradient Descent Algorithm is popular algorithm used in Deep Learning model to minimize the loss function iteratively using first-order derivatives.

Let’s see an example using simple function **z(x,y) = x^2 + y^2**; below is the surface plot of this function:

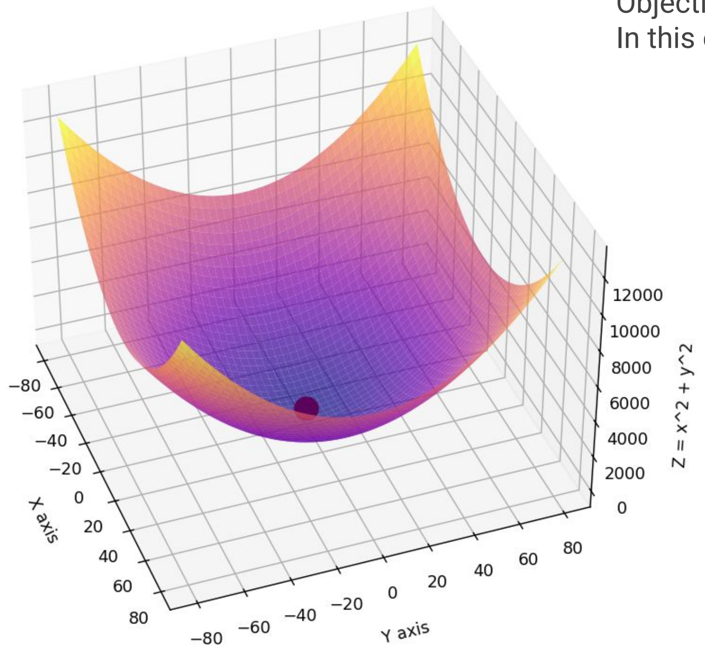


Objective of this algorithm is to find values for variables (x,y) that minimizes the function z.
In this case goal is to identify red ball shown in the figure.

Gradient Descent Algorithm

Gradient Descent Algorithm is popular algorithm used in Deep Learning model to minimize the loss function iteratively using first-order derivatives.

Let's see an example using simple function **z(x,y) = x^2 + y^2**; below is the surface plot of this function:



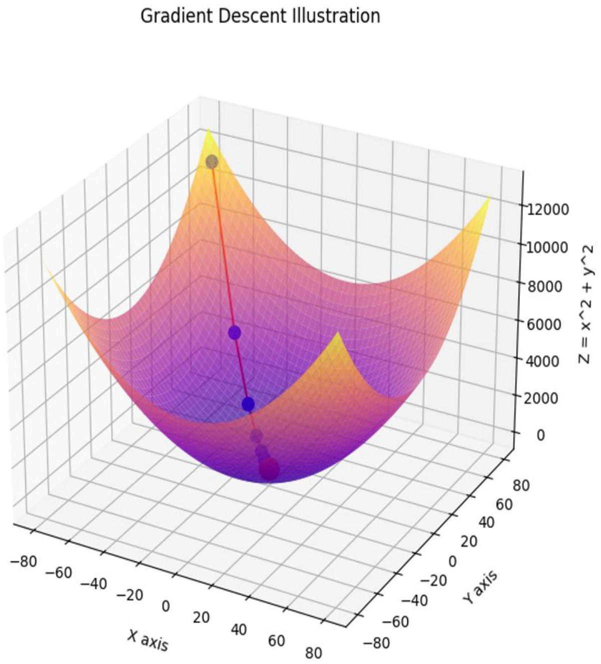
Objective of this algorithm is to find values for variables (x,y) that minimizes the function z. In this case goal is to identify red ball shown in the figure.

Algorithm Steps:

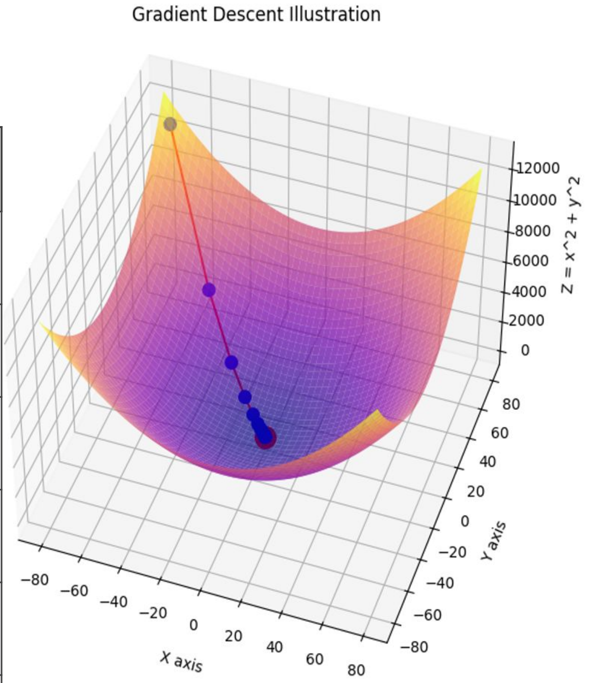
```
step1: randomly pick x and y to start with; ex (-75.0, 75.0)

for  i in range(20): #20 denotes no of iteration
    step2: Calculate the function z = (-75.0)^2 + (75.0)^2 =11250.0
    step3: Calculate the gradient of z w.r.t x ,y
        dz/dx = 2*x = -150.0
        dz/dy = 2*y =   150.0
    step4: Update variables
        $x_i = x_{i-1}$ - lr * dz/dx = -75.0 - 0.2*-150.0 = -45.0
        $Y_i = y_{i-1}$ - lr*dz/dy = 45.0
```

Gradient Descent Visualization



x	y	z	dz_dx	lr*dz_dx	dz_dy	lr*dz_dy
-75.0	75.0	11250.0	-150.0	-30.0	150.0	30.0
-45.0	45.0	4050.0	-90.0	-18.0	90.0	18.0
-27.0	27.0	1458.0	-54.0	-10.8	54.0	10.8
-16.2	16.2	524.9	-32.4	-6.5	32.4	6.5
-9.7	9.7	189.0	-19.4	-3.9	19.4	3.9
-5.8	5.8	68.0	-11.7	-2.3	11.7	2.3



Computation Graph

- We have seen that calculating gradient of a function w.r.t variable is core to gradient Descent Algorithm.
- GradientTape API of tensorflow helps us to do exactly that. i.e. compute gradient of a function w.r.t variable.
- In fact Gradients computed in previous slide was also using gradient tape API. We will review that code as well while going through colab notebook.
- In order to calculate gradients, **GradientTape** uses Reverse mode auto differentiation method by leveraging Computation Graph. Let's understand this point further, as it will bring more clarity while working with GradientTape API.

We will be using our previous equation to illustrate the point further,

$$z = x^2 + y^2$$

Computation Graph Cont..

So, Let’s use computation graph for equation $\mathbf{z = x^2 + y^2}$ to intuitively understand reverse mode Automatic differentiation used by Gradient Tape for computing gradients:

1) In computation graph each variable and operation is represented as a node in the computation graph.
2) Each edge represent the flow of input value from one node to another

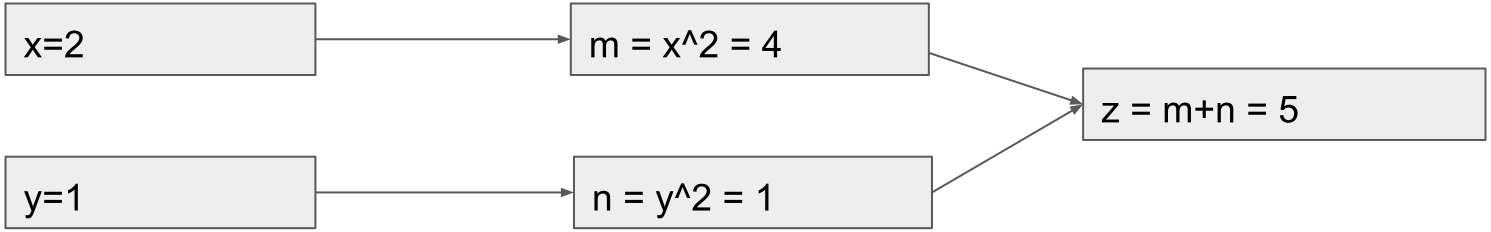
$$\begin{tikzcd}
\boxed{x=2} \arrow[r] & \boxed{m = x^\wedge 2 = 4} \arrow[dr] & \\
 & & \boxed{z = m+n = 5} \\
\boxed{y=1} \arrow[r] & \boxed{n = y^\wedge 2 = 1} \arrow[ur] &
\end{tikzcd}$$

This is called the forward pass where we move from left to right passing the values from variables to finally get the output.

Computation Graph Cont..

So, Let's use computation graph for equation $\mathbf{z = x^2 + y^2}$ to intuitively understand reverse mode Automatic differentiation used by Gradient Tape for computing gradients:

1) In computation graph each variable and operation is represented as a node in the computation graph.
2) Each edge represent the flow of input value from one node to another

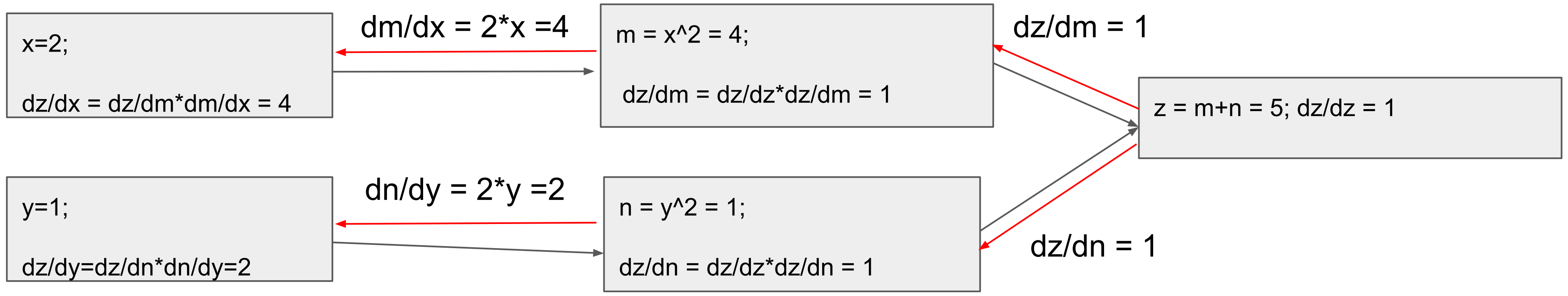


This is called the forward pass where we move from left to right passing the values from variables to finally get the output.

Above, With the help of computation graph we are able to express function z in terms of elementary operations (ex: +, -, /….etc including transcendental function like cos, ln etc..) whose derivatives are well defined and now we can now combine the derivative of these individual operations to arrive at the derivative of output w.r.t any variables. This is the main idea behind Automatic differentiation.

Let's now compute gradient on above Computational Graph as a backward pass:

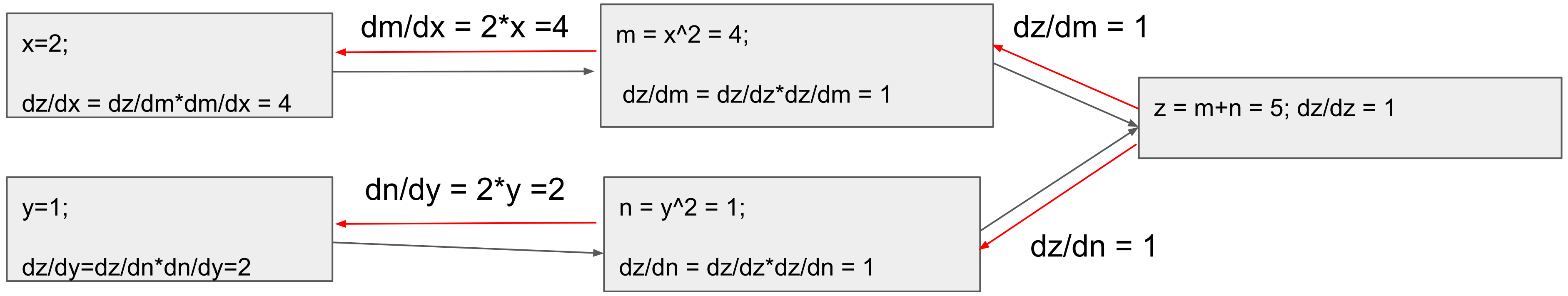
Computation Graph Cont..



Few point to note while calculating gradient during backward pass in above computation graph(denote by red line):

1) Along each edge of the computation graph, the derivative represents the rate of change of the variable in the destination node (where the forward-pass arrow points) with respect to the variable in the source node (where the arrow originates).
2) At each node, the derivative represents the partial derivative of the final output (loss function or objective) with respect to the variable at that node.

Computation Graph Cont..



Few point to note while calculating gradient during backward pass in above computation graph(denote by red line):

1) Along each edge of the computation graph, the derivative represents the rate of change of the variable in the destination node (where the forward-pass arrow points) with respect to the variable in the source node (where the arrow originates).
2) At each node, the derivative represents the partial derivative of the final output (loss function or objective) with respect to the variable at that node.

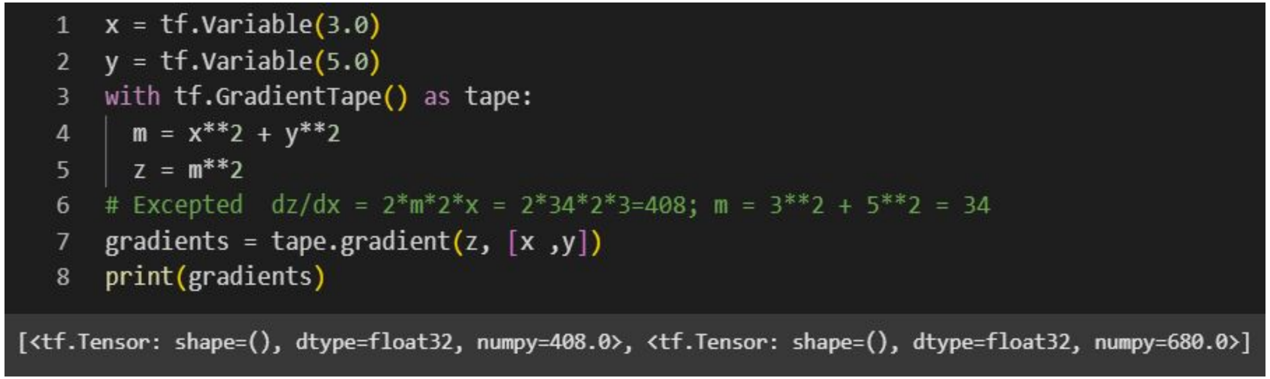
3) Now as part of backward pass, these derivatives are propagated from the output node (where the final computation is made) back to the input nodes, using the chain rule to combine the contributions of the derivatives from subsequent edges and nodes.
ex: dz/dx = dz/dm * dm/dx

4) If there are multiple path originating from input node to output node then gradient across both Path is calculated independently and then summed to get final gradient of output w.r.t input node Variable

Since we are propagating from right to left in reverse order of forward pass in order to compute gradient of final output w.r.t other variables we refer to it as **reverse mode automatic differentiation**.

tf.GradientTape

Let’s introduce tf.GradientTape API through below code snippet



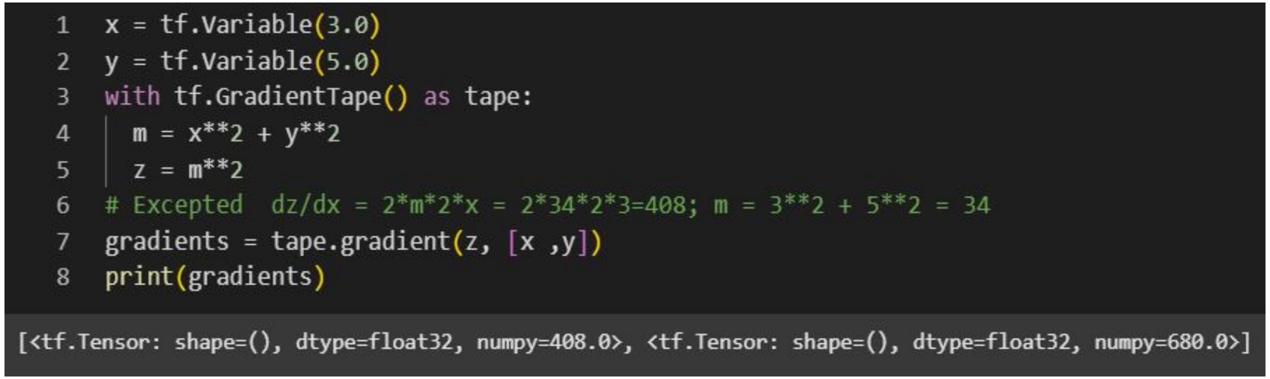
```python
x = tf.Variable(3.0)
y = tf.Variable(5.0)
with tf.GradientTape() as tape:
    m = x**2 + y**2
    z = m**2
# Excepted  dz/dx = 2*m*2*x = 2*34*2*3=408; m = 3**2 + 5**2 = 34
gradients = tape.gradient(z, [x ,y])
print(gradients)
```

```
[<tf.Tensor: shape=(), dtype=float32, numpy=408.0>, <tf.Tensor: shape=(), dtype=float32, numpy=680.0>]
```

1) `GradientTape` is a context manager used along `with` statement. Within its context, it "records" all operations executed during the forward pass, **specifically those involving tensors marked as trainable**. These operations are recorded onto the **"tape object"**, enabling the computation of gradients during the backward pass.

tf.GradientTape

Let’s introduce tf.GradientTape API through below code snippet



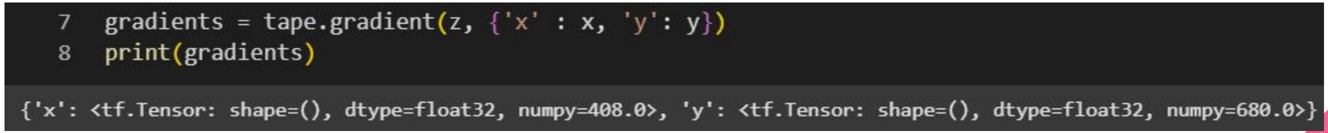
```
x = tf.Variable(3.0)
y = tf.Variable(5.0)
with tf.GradientTape() as tape:
    m = x**2 + y**2
    z = m**2
# Excepted  dz/dx = 2*m*2*x = 2*34*2*3=408; m = 3**2 + 5**2 = 34
gradients = tape.gradient(z, [x ,y])
print(gradients)
```

```
[<tf.Tensor: shape=(), dtype=float32, numpy=408.0>, <tf.Tensor: shape=(), dtype=float32, numpy=680.0>]
```

1) `GradientTape` is a context manager used along `with` statement. Within its context, it "records" all operations executed during the forward pass, **specifically those involving tensors marked as trainable**. These operations are recorded onto the **"tape object"**, enabling the computation of gradients during the backward pass.

2) In order to compute gradient we can use gradient method onto **“tape object”** and pass the output and variable as argument to the method.

3) Tape is flexible in the sense that we can pass variable in different way as we please like list, nested list, dict etc.. gradient output will also have the same structure in which variable is provided. Like above, we provided variable as a list and gradient output is also of the form list

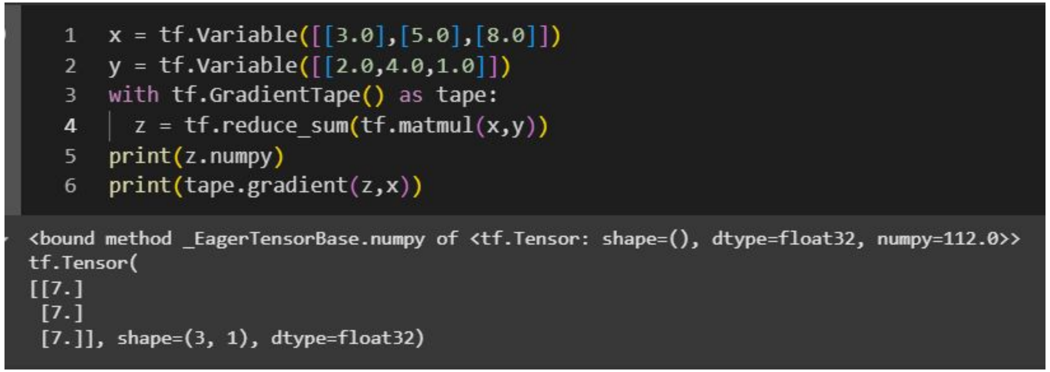


```
gradients = tape.gradient(z, {'x' : x, 'y': y})
print(gradients)
```

```
{'x': <tf.Tensor: shape=(), dtype=float32, numpy=408.0>, 'y': <tf.Tensor: shape=(), dtype=float32, numpy=680.0>}
```

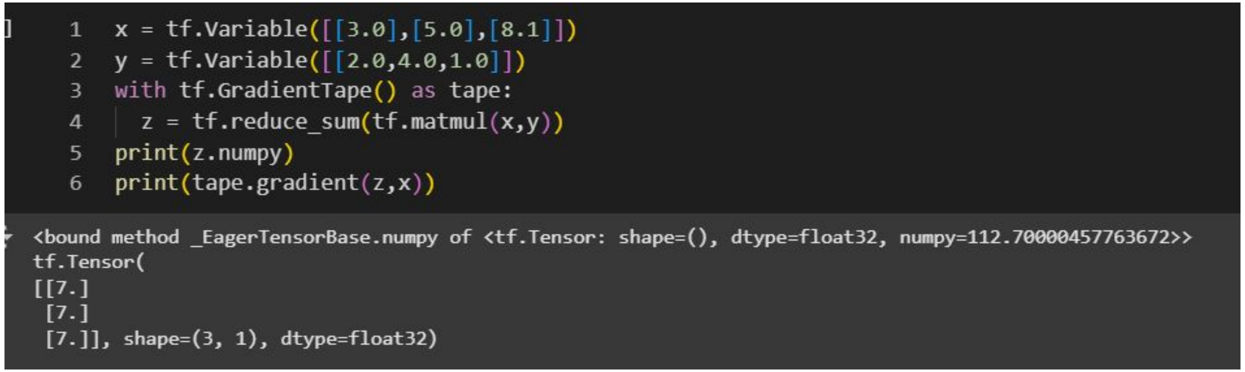
tf.GradientTape Cont..

4) The shape of the gradient w.r.t variable will be of same shape as variable indicating how each element of a variable influences the final output. We see below that x and dz/dx is of same shape (3, 1)



```
x = tf.Variable([[3.0],[5.0],[8.0]])
y = tf.Variable([[2.0,4.0,1.0]])
with tf.GradientTape() as tape:
  z = tf.reduce_sum(tf.matmul(x,y))
print(z.numpy)
print(tape.gradient(z,x))
```

```
<bound method _EagerTensorBase.numpy of <tf.Tensor: shape=(), dtype=float32, numpy=112.0>>
tf.Tensor(
[[7.]
 [7.]
 [7.]], shape=(3, 1), dtype=float32)
```

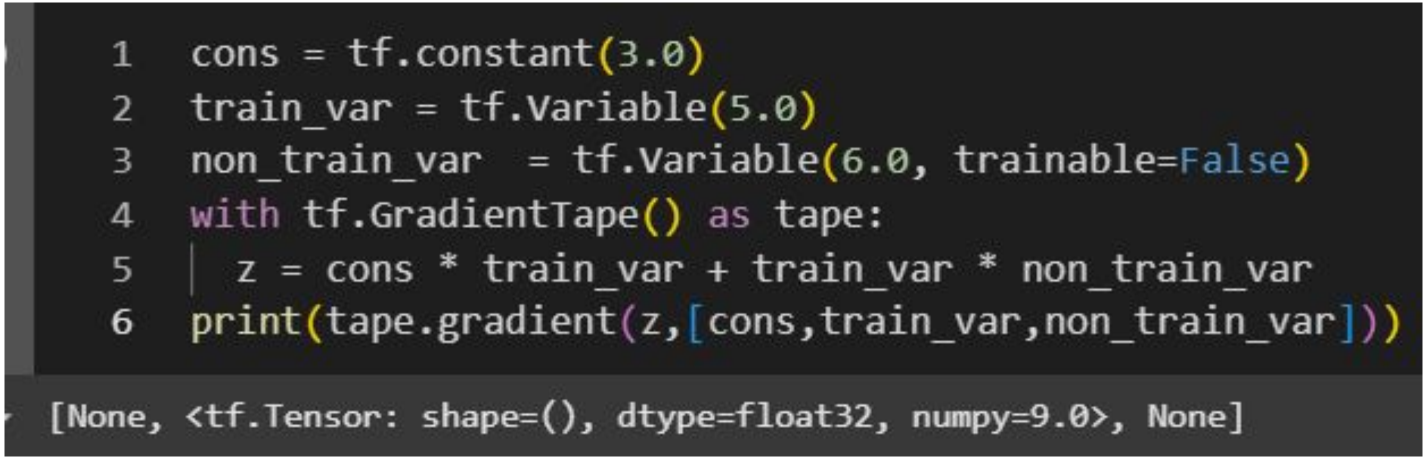


```
x = tf.Variable([[3.0],[5.0],[8.1]])
y = tf.Variable([[2.0,4.0,1.0]])
with tf.GradientTape() as tape:
  z = tf.reduce_sum(tf.matmul(x,y))
print(z.numpy)
print(tape.gradient(z,x))
```

```
<bound method _EagerTensorBase.numpy of <tf.Tensor: shape=(), dtype=float32, numpy=112.70000457763672>>
tf.Tensor(
[[7.]
 [7.]
 [7.]], shape=(3, 1), dtype=float32)
```

tf.GradientTape Cont..

5) By default Tape records operation only w.r.t trainable tf.Variables. Since, tensors like tf.constant are not trainable by default Tape will not record operations and will fail to compute gradient w.r.t such tensors

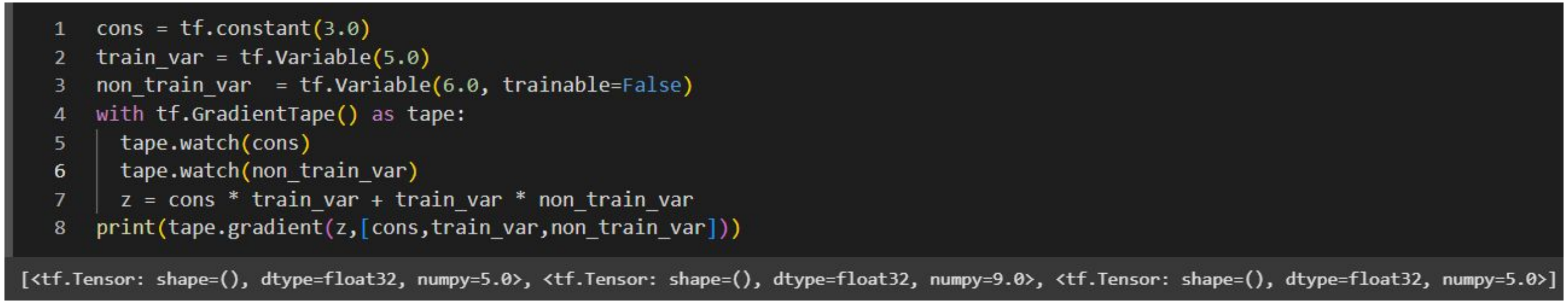


```
cons = tf.constant(3.0)
train_var = tf.Variable(5.0)
non_train_var  = tf.Variable(6.0, trainable=False)
with tf.GradientTape() as tape:
  z = cons * train_var + train_var * non_train_var
print(tape.gradient(z,[cons,train_var,non_train_var]))
```

```
[None, <tf.Tensor: shape=(), dtype=float32, numpy=9.0>, None]
```

tf.GradientTape Cont..

6) We can use **tape.watch( )** to record gradient w.r.t tf.tensor like constant tensor etc..

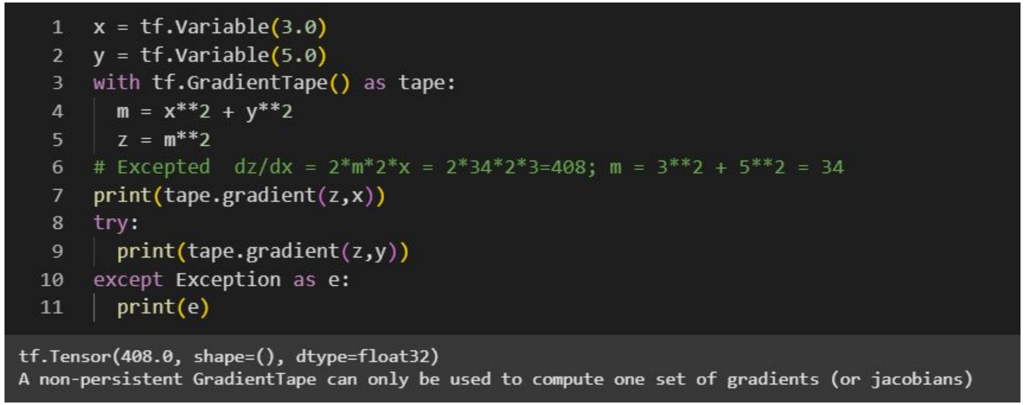


```
cons = tf.constant(3.0)
train_var = tf.Variable(5.0)
non_train_var  = tf.Variable(6.0, trainable=False)
with tf.GradientTape() as tape:
  tape.watch(cons)
  tape.watch(non_train_var)
  z = cons * train_var + train_var * non_train_var
print(tape.gradient(z,[cons,train_var,non_train_var]))
```

```
[<tf.Tensor: shape=(), dtype=float32, numpy=5.0>, <tf.Tensor: shape=(), dtype=float32, numpy=9.0>, <tf.Tensor: shape=(), dtype=float32, numpy=5.0>]
```

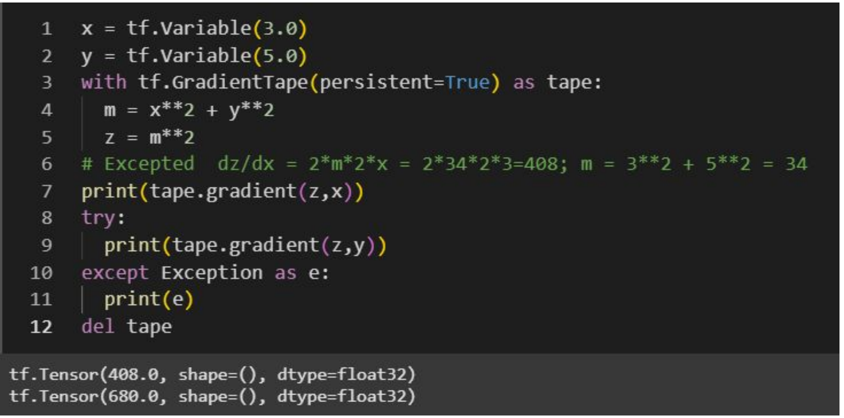
tf.GradientTape, Persistent Tape

7) By default, once **tape.gradient( )** is called, the resource held by tape is released which essentially means you cannot call **tape.gradient( )** again to compute gradient. If we need to compute gradient multiple times we need to create persistent tape that is **tf.GradientTape(persistent = True)**



```python
x = tf.Variable(3.0)
y = tf.Variable(5.0)
with tf.GradientTape() as tape:
  m = x**2 + y**2
  z = m**2
# Excepted  dz/dx = 2*m*2*x = 2*34*2*3=408; m = 3**2 + 5**2 = 34
print(tape.gradient(z,x))
try:
  print(tape.gradient(z,y))
except Exception as e:
  print(e)
```

```
tf.Tensor(408.0, shape=(), dtype=float32)
A non-persistent GradientTape can only be used to compute one set of gradients (or jacobians)
```



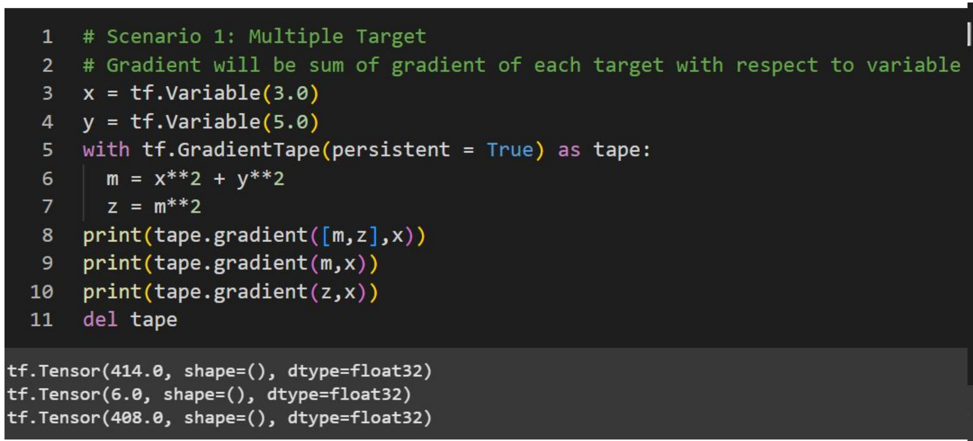
```python
x = tf.Variable(3.0)
y = tf.Variable(5.0)
with tf.GradientTape(persistent=True) as tape:
  m = x**2 + y**2
  z = m**2
# Excepted  dz/dx = 2*m*2*x = 2*34*2*3=408; m = 3**2 + 5**2 = 34
print(tape.gradient(z,x))
try:
  print(tape.gradient(z,y))
except Exception as e:
  print(e)
del tape
```

```
tf.Tensor(408.0, shape=(), dtype=float32)
tf.Tensor(680.0, shape=(), dtype=float32)
```

GradientTape in order to compute gradient during backward pass store intermediate results in the memory, and when we create persistent tape this memory is not automatically released hence we need to use **del tape** , so that memory can be released after tape usage is complete for efficient resource management

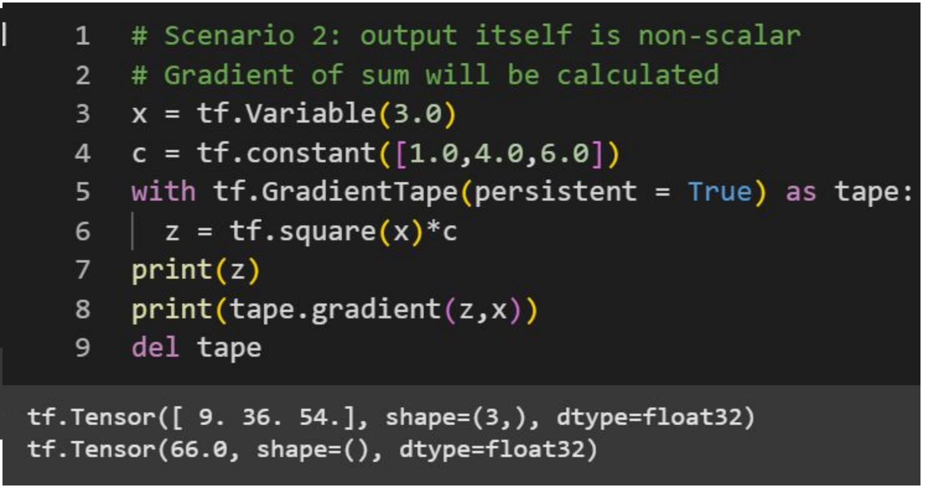
tf.GradientTape, Non-Scalar Output

8) Gradient is generally computed on a scalar output. What if output/target given in **tape.gradient** is not scalar? Let's Explore:



```
# Scenario 1: Multiple Target
# Gradient will be sum of gradient of each target with respect to variable
x = tf.Variable(3.0)
y = tf.Variable(5.0)
with tf.GradientTape(persistent = True) as tape:
    m = x**2 + y**2
    z = m**2
print(tape.gradient([m,z],x))
print(tape.gradient(m,x))
print(tape.gradient(z,x))
del tape
```

```
tf.Tensor(414.0, shape=(), dtype=float32)
tf.Tensor(6.0, shape=(), dtype=float32)
tf.Tensor(408.0, shape=(), dtype=float32)
```

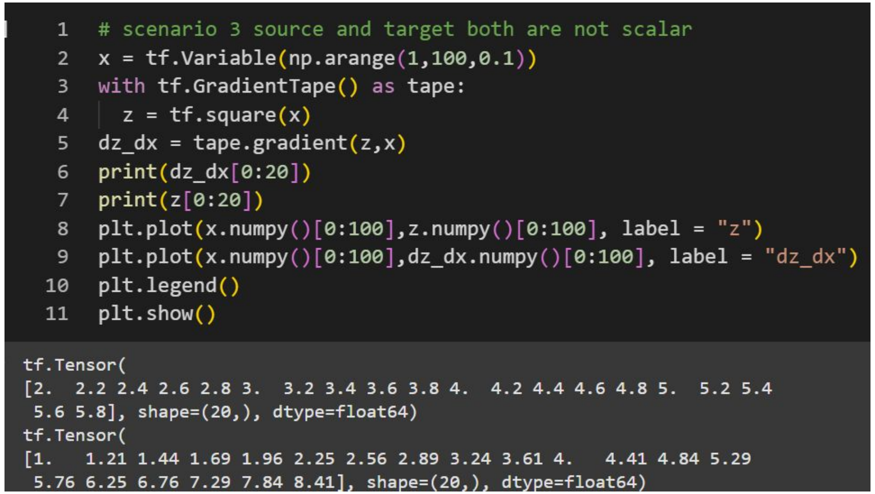


```
# Scenario 2: output itself is non-scalar
# Gradient of sum will be calculated
x = tf.Variable(3.0)
c = tf.constant([1.0,4.0,6.0])
with tf.GradientTape(persistent = True) as tape:
    z = tf.square(x)*c
print(z)
print(tape.gradient(z,x))
del tape
```

```
tf.Tensor([ 9. 36. 54.], shape=(3,), dtype=float32)
tf.Tensor(66.0, shape=(), dtype=float32)
```

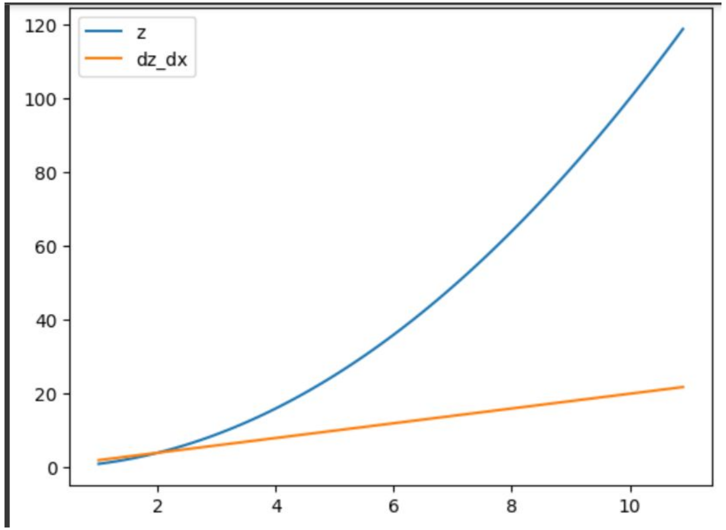
tf.GradientTape, Non-Scalar output

Gradient is generally computed on a scalar output. What if output/target given in **tape.gradient** is not scalar? Let’s Explore:



```
# scenario 3 source and target both are not scalar
x = tf.Variable(np.arange(1,100,0.1))
with tf.GradientTape() as tape:
    z = tf.square(x)
dz_dx = tape.gradient(z,x)
print(dz_dx[0:20])
print(z[0:20])
plt.plot(x.numpy()[0:100],z.numpy()[0:100], label = "z")
plt.plot(x.numpy()[0:100],dz_dx.numpy()[0:100], label = "dz_dx")
plt.legend()
plt.show()
```

```
tf.Tensor(
[2.  2.2 2.4 2.6 2.8 3.  3.2 3.4 3.6 3.8 4.  4.2 4.4 4.6 4.8 5.  5.2 5.4
 5.6 5.8], shape=(20,), dtype=float64)
tf.Tensor(
[1.   1.21 1.44 1.69 1.96 2.25 2.56 2.89 3.24 3.61 4.   4.41 4.84 5.29
 5.76 6.25 6.76 7.29 7.84 8.41], shape=(20,), dtype=float64)
```



Let’s intuitively understand this:

X is of shape [x1=1.0, x2=1.1, x3=1.2 ………], and we have z = [x1**2, x2**2,....]

We have seen earlier dz/dx = [dz/dx1, dz/dx2,dz/dx3,......]

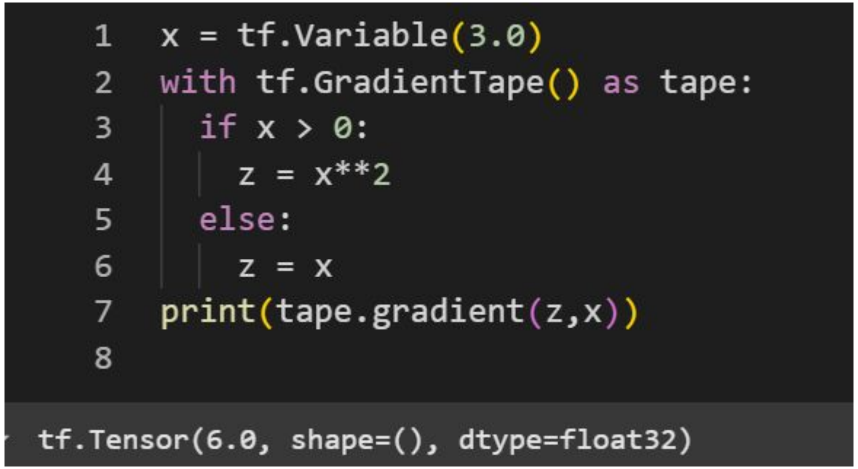
Since z is not scalar gradient of sum will be calculated,

i.e. dz/dx1 = d(x1**2 + x2**2 +...)/dx1 = 2x1 = 2*1 =2

tf.GradientTape, Control Flow statements

9) Control Flow within "**tf.GradientTape**".

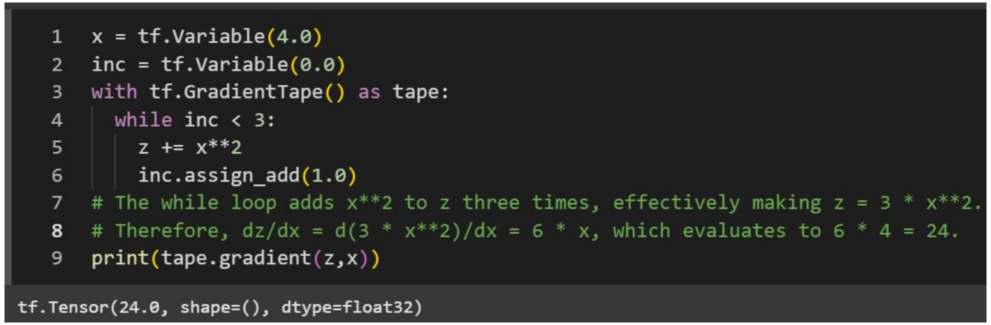
TensorFlow's `GradientTape` is capable of tracking operations performed under control flow constructs (`if`, `while`, etc.) and computes the gradients correctly.



```
x = tf.Variable(3.0)
with tf.GradientTape() as tape:
    if x > 0:
        z = x**2
    else:
        z = x
print(tape.gradient(z,x))

```

```
tf.Tensor(6.0, shape=(), dtype=float32)
```

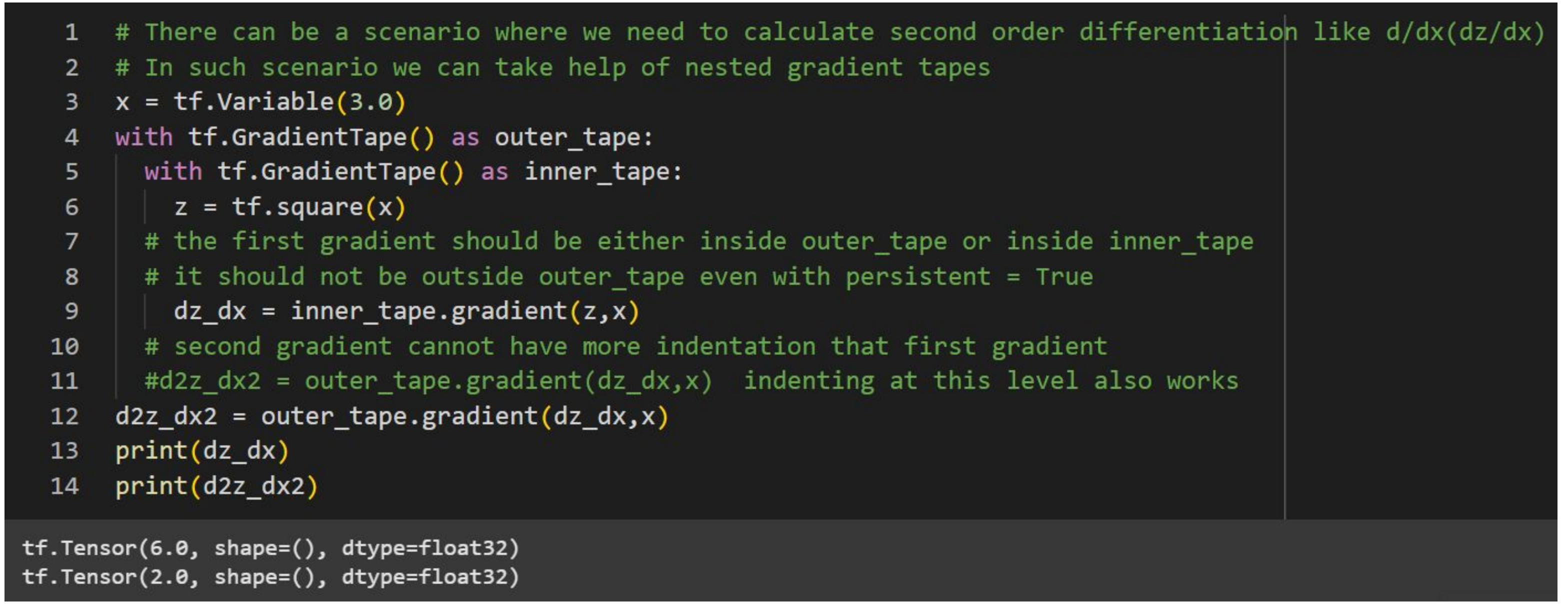


```
x = tf.Variable(4.0)
inc = tf.Variable(0.0)
with tf.GradientTape() as tape:
  while inc < 3:
    z += x**2
    inc.assign_add(1.0)
# The while loop adds x**2 to z three times, effectively making z = 3 * x**2.
# Therefore, dz/dx = d(3 * x**2)/dx = 6 * x, which evaluates to 6 * 4 = 24.
print(tape.gradient(z,x))
```

```
tf.Tensor(24.0, shape=(), dtype=float32)
```

tf.GradientTape, Nested GradientTapes

10) Let’s see how we can compute second order gradients as well using nested GradientTape object

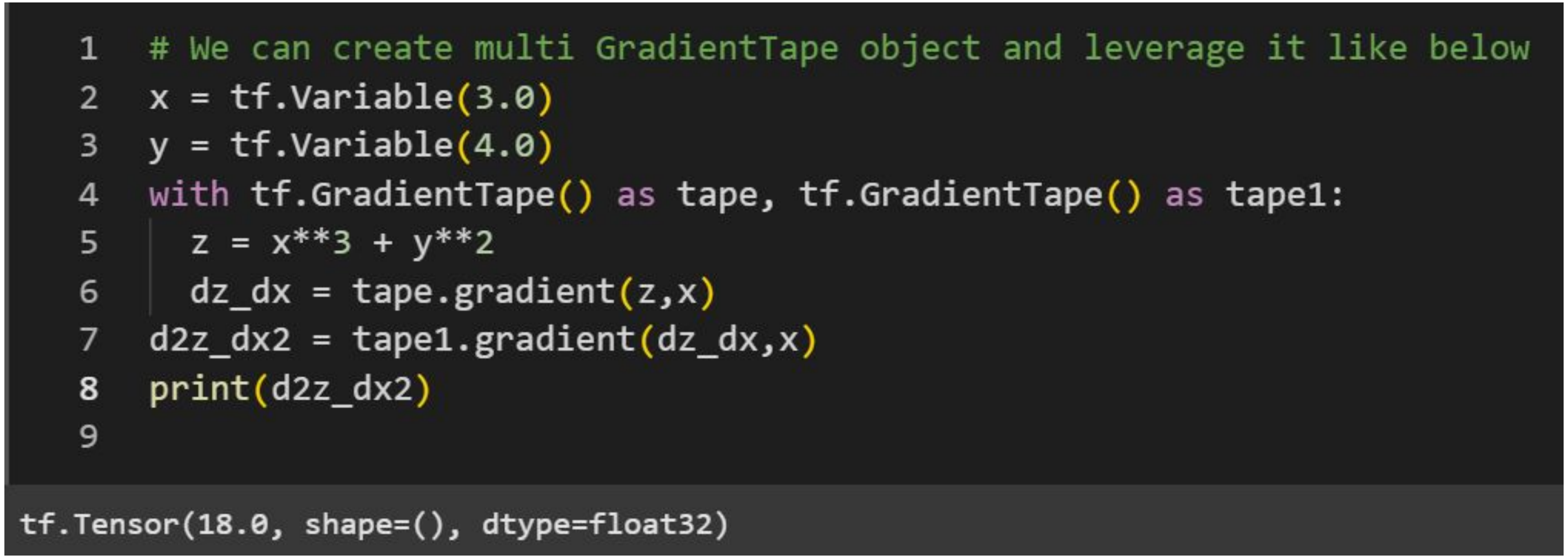


```python
# There can be a scenario where we need to calculate second order differentiation like d/dx(dz/dx)
# In such scenario we can take help of nested gradient tapes
x = tf.Variable(3.0)
with tf.GradientTape() as outer_tape:
  with tf.GradientTape() as inner_tape:
    z = tf.square(x)
  # the first gradient should be either inside outer_tape or inside inner_tape
  # it should not be outside outer_tape even with persistent = True
    dz_dx = inner_tape.gradient(z,x)
  # second gradient cannot have more indentation that first gradient
  #d2z_dx2 = outer_tape.gradient(dz_dx,x)  indenting at this level also works
d2z_dx2 = outer_tape.gradient(dz_dx,x)
print(dz_dx)
print(d2z_dx2)
```

```
tf.Tensor(6.0, shape=(), dtype=float32)
tf.Tensor(2.0, shape=(), dtype=float32)
```

tf.GradientTape, Multiple GradientTape object

11) Let’s explore Multi GradientTape object



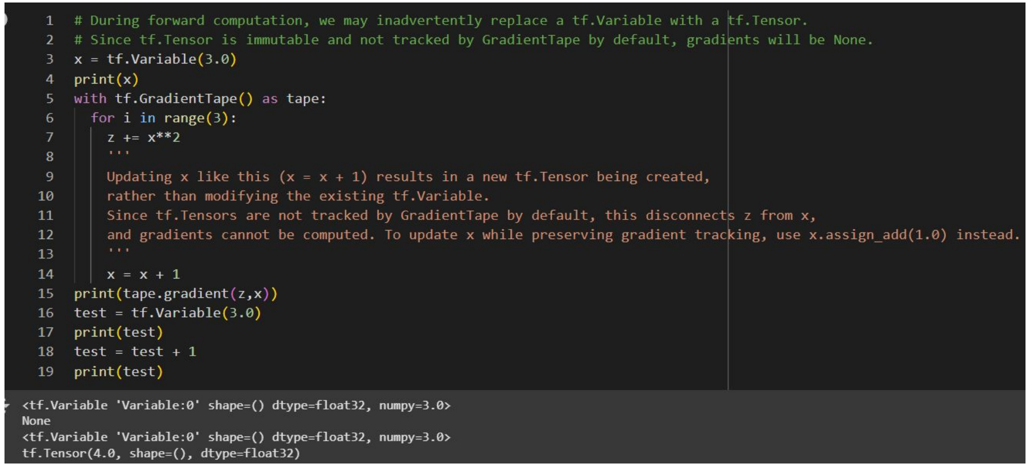
```python
# We can create multi GradientTape object and leverage it like below
x = tf.Variable(3.0)
y = tf.Variable(4.0)
with tf.GradientTape() as tape, tf.GradientTape() as tape1:
  z = x**3 + y**2
  dz_dx = tape.gradient(z,x)
d2z_dx2 = tape1.gradient(dz_dx,x)
print(d2z_dx2)

```

```
tf.Tensor(18.0, shape=(), dtype=float32)
```

tf.GradientTape, Things not to do..

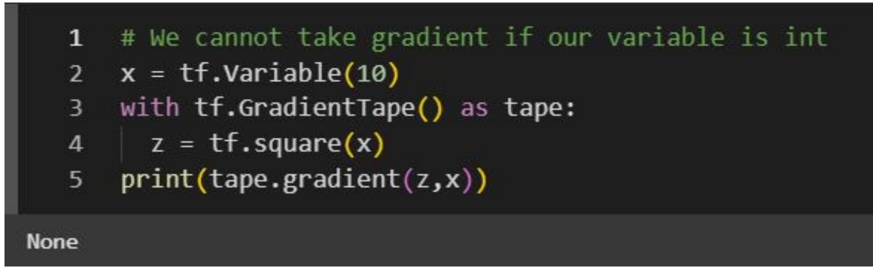
Scenario 1



```
# During forward computation, we may inadvertently replace a tf.Variable with a tf.Tensor.
# Since tf.Tensor is immutable and not tracked by GradientTape by default, gradients will be None.
x = tf.Variable(3.0)
print(x)
with tf.GradientTape() as tape:
  for i in range(3):
    z += x**2
    '''
    Updating x like this (x = x + 1) results in a new tf.Tensor being created,
    rather than modifying the existing tf.Variable.
    Since tf.Tensors are not tracked by GradientTape by default, this disconnects z from x,
    and gradients cannot be computed. To update x while preserving gradient tracking, use x.assign_add(1.0) instead.
    '''
    x = x + 1
print(tape.gradient(z,x))
test = tf.Variable(3.0)
print(test)
test = test + 1
print(test)
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=3.0>
None
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=3.0>
tf.Tensor(4.0, shape=(), dtype=float32)
```

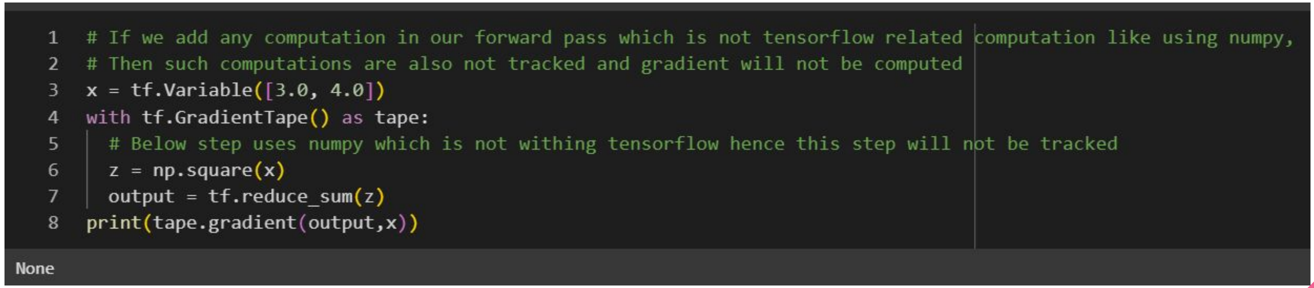
Scenario 2



```
# We cannot take gradient if our variable is int
x = tf.Variable(10)
with tf.GradientTape() as tape:
  z = tf.square(x)
print(tape.gradient(z,x))
```

```
None
```

Scenario 3



```
# If we add any computation in our forward pass which is not tensorflow related computation like using numpy,
# Then such computations are also not tracked and gradient will not be computed
x = tf.Variable([3.0, 4.0])
with tf.GradientTape() as tape:
  # Below step uses numpy which is not withing tensorflow hence this step will not be tracked
  z = np.square(x)
  output = tf.reduce_sum(z)
print(tape.gradient(output,x))
```

```
None
```

Thank You