TANGLECV GROUP PROJECT

DAG SecureVote Data App

Luka Brown, Victoria Jordan, Harpaz Aviv, Serhat Oncu

DAG Benefits

Verification

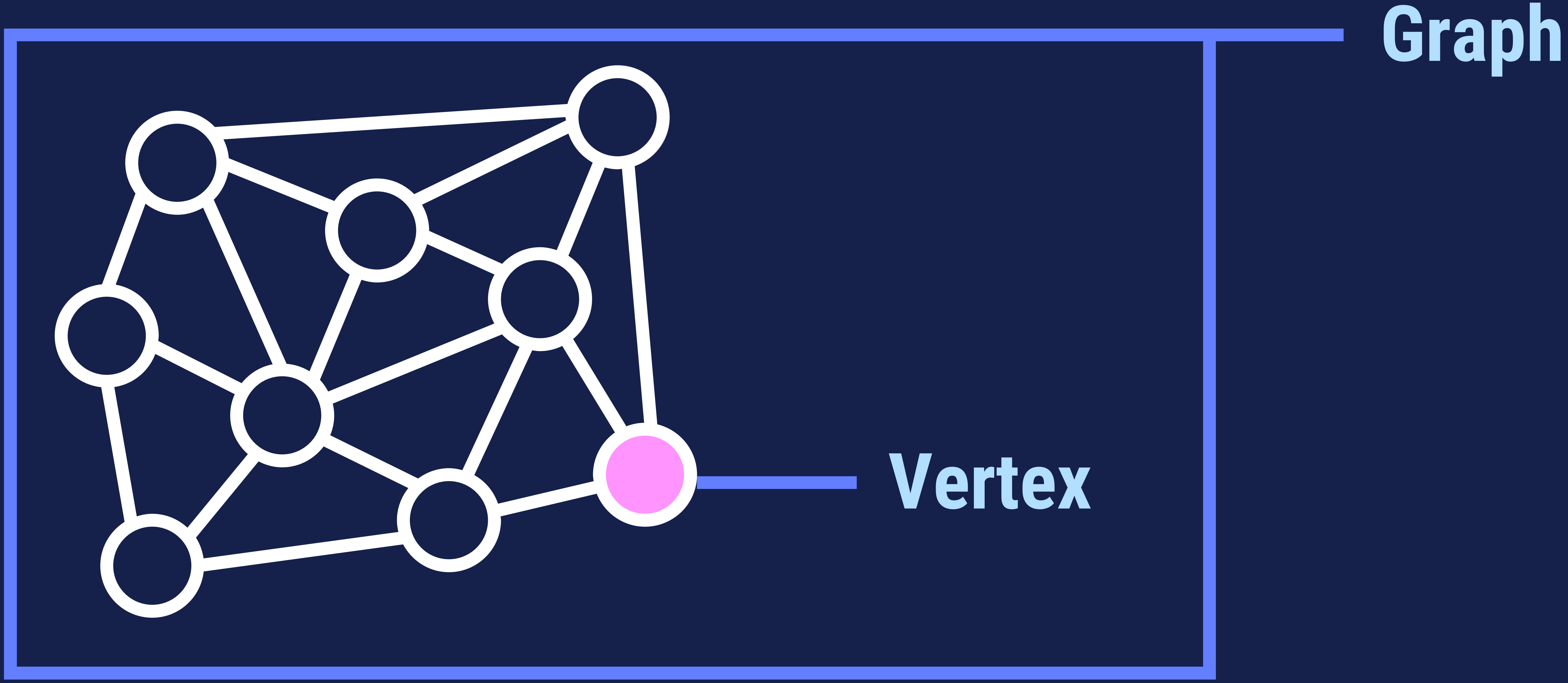
Fast

Security

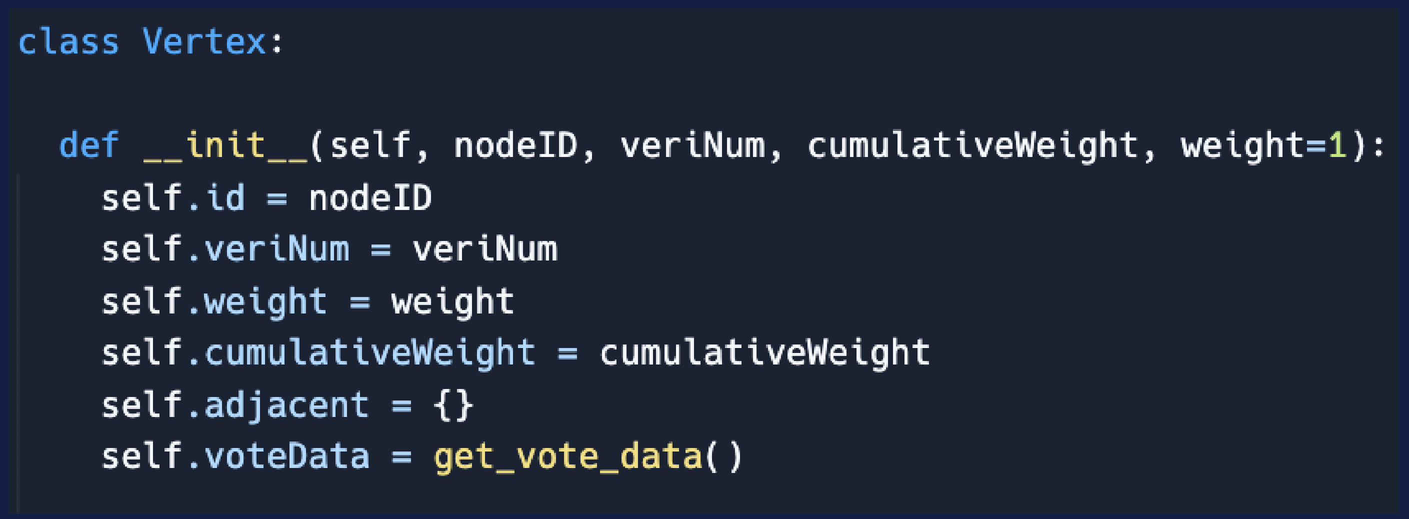
Scalable

Code Overview

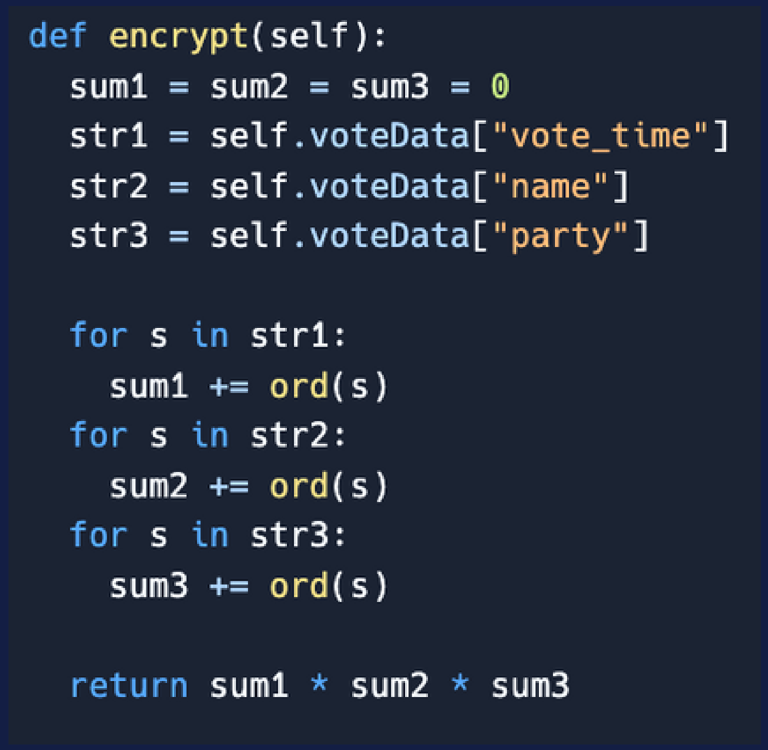
Our code utilizes two interconnected classes:
Vertex & Graph



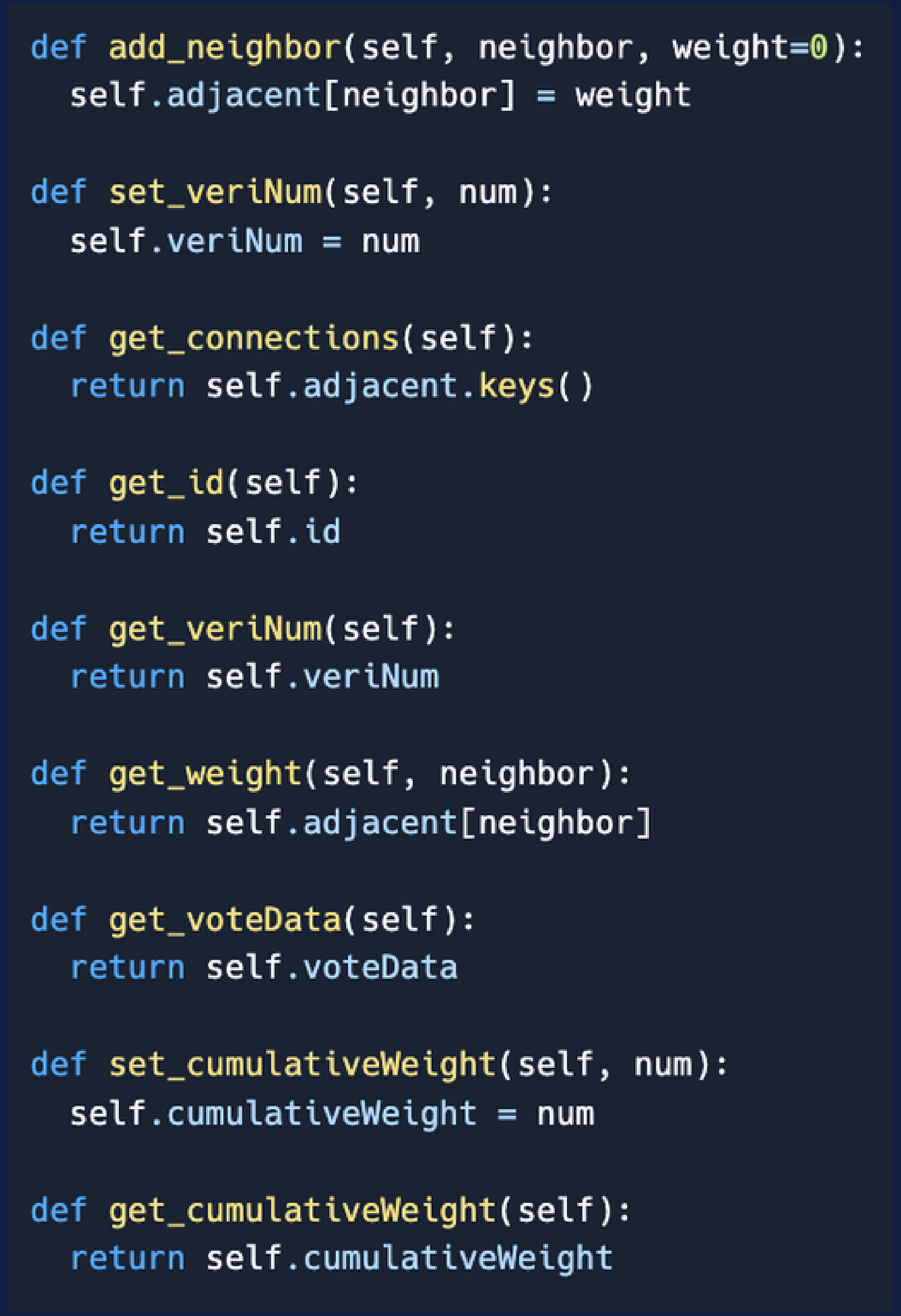
Vertex Class



```python
class Vertex:

  def __init__(self, nodeID, veriNum, cumulativeWeight, weight=1):
    self.id = nodeID
    self.veriNum = veriNum
    self.weight = weight
    self.cumulativeWeight = cumulativeWeight
    self.adjacent = {}
    self.voteData = get_vote_data()
```

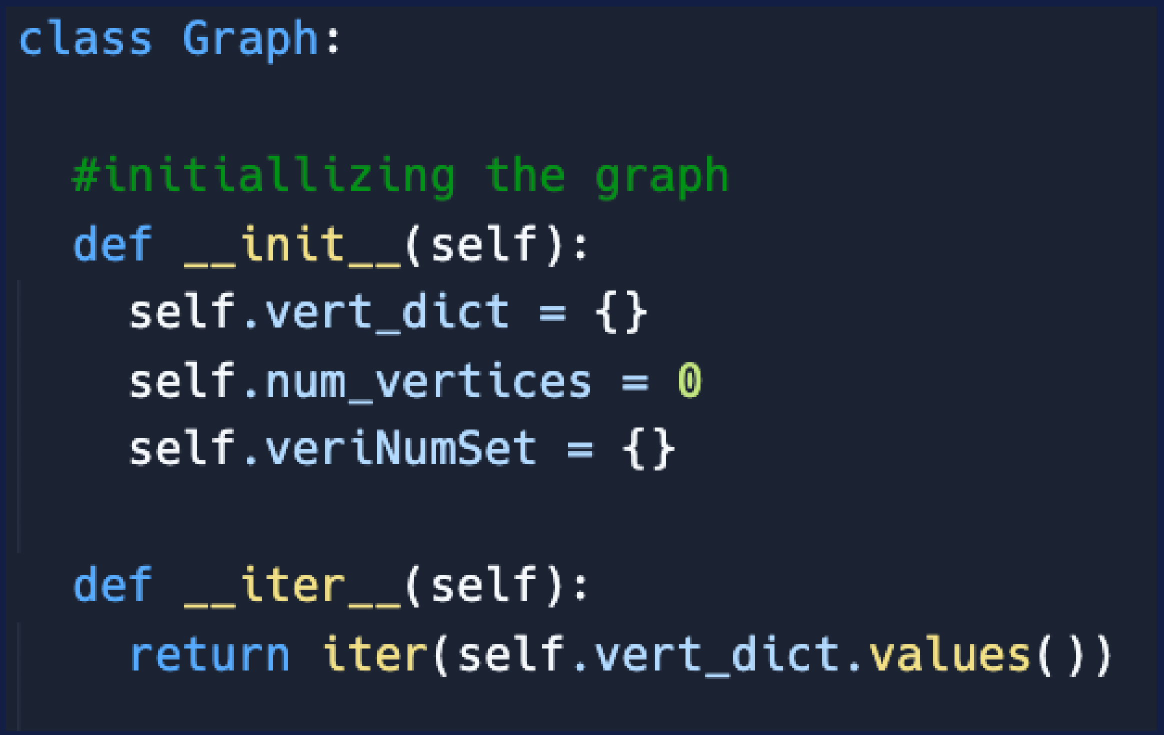


```python
def encrypt(self):
  sum1 = sum2 = sum3 = 0
  str1 = self.voteData["vote_time"]
  str2 = self.voteData["name"]
  str3 = self.voteData["party"]

  for s in str1:
    sum1 += ord(s)
  for s in str2:
    sum2 += ord(s)
  for s in str3:
    sum3 += ord(s)

  return sum1 * sum2 * sum3
```



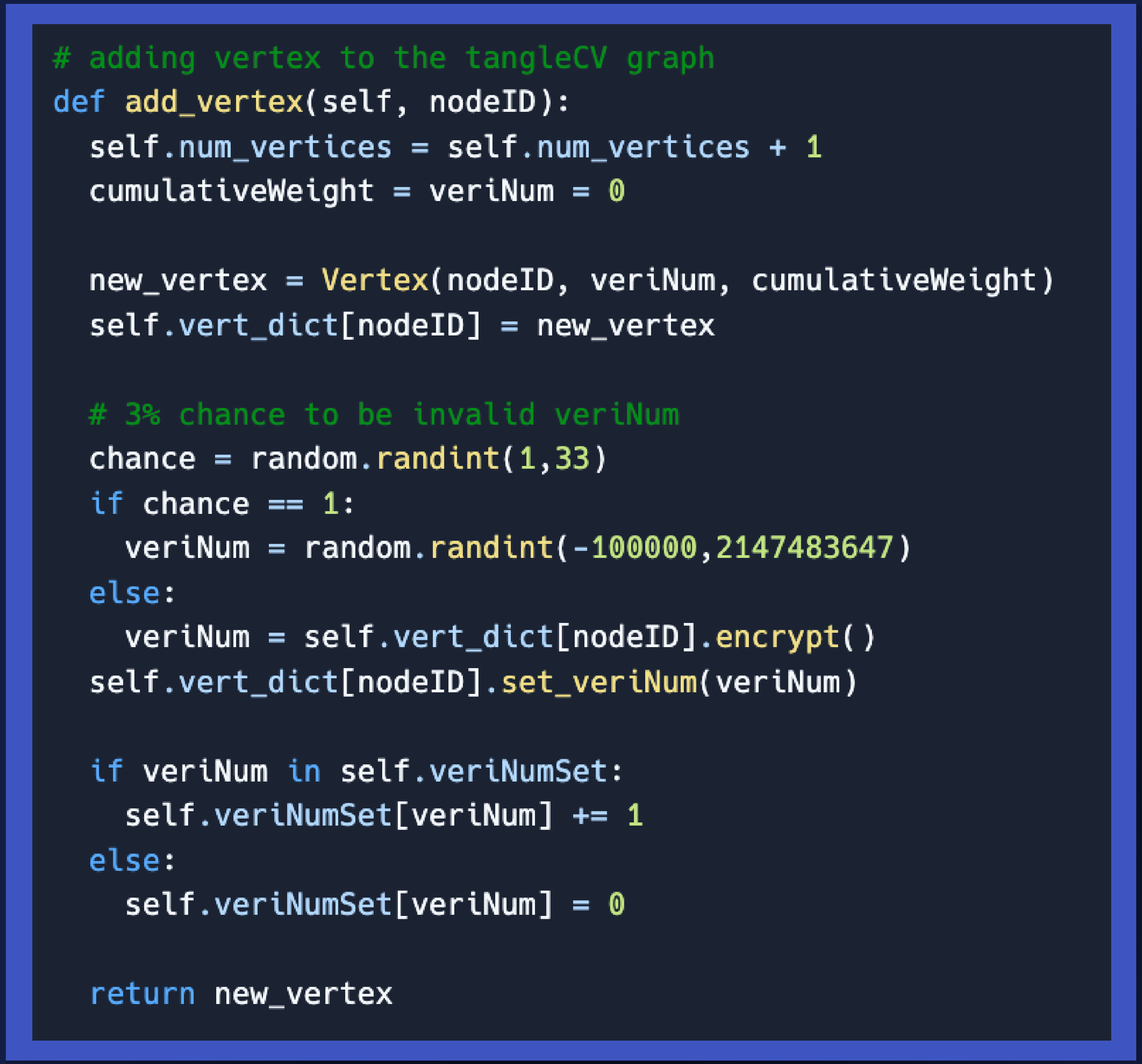
```python
def add_neighbor(self, neighbor, weight=0):
  self.adjacent[neighbor] = weight

def set_veriNum(self, num):
  self.veriNum = num

def get_connections(self):
  return self.adjacent.keys()

def get_id(self):
  return self.id

def get_veriNum(self):
  return self.veriNum

def get_weight(self, neighbor):
  return self.adjacent[neighbor]

def get_voteData(self):
  return self.voteData

def set_cumulativeWeight(self, num):
  self.cumulativeWeight = num

def get_cumulativeWeight(self):
  return self.cumulativeWeight
```

Graph Class



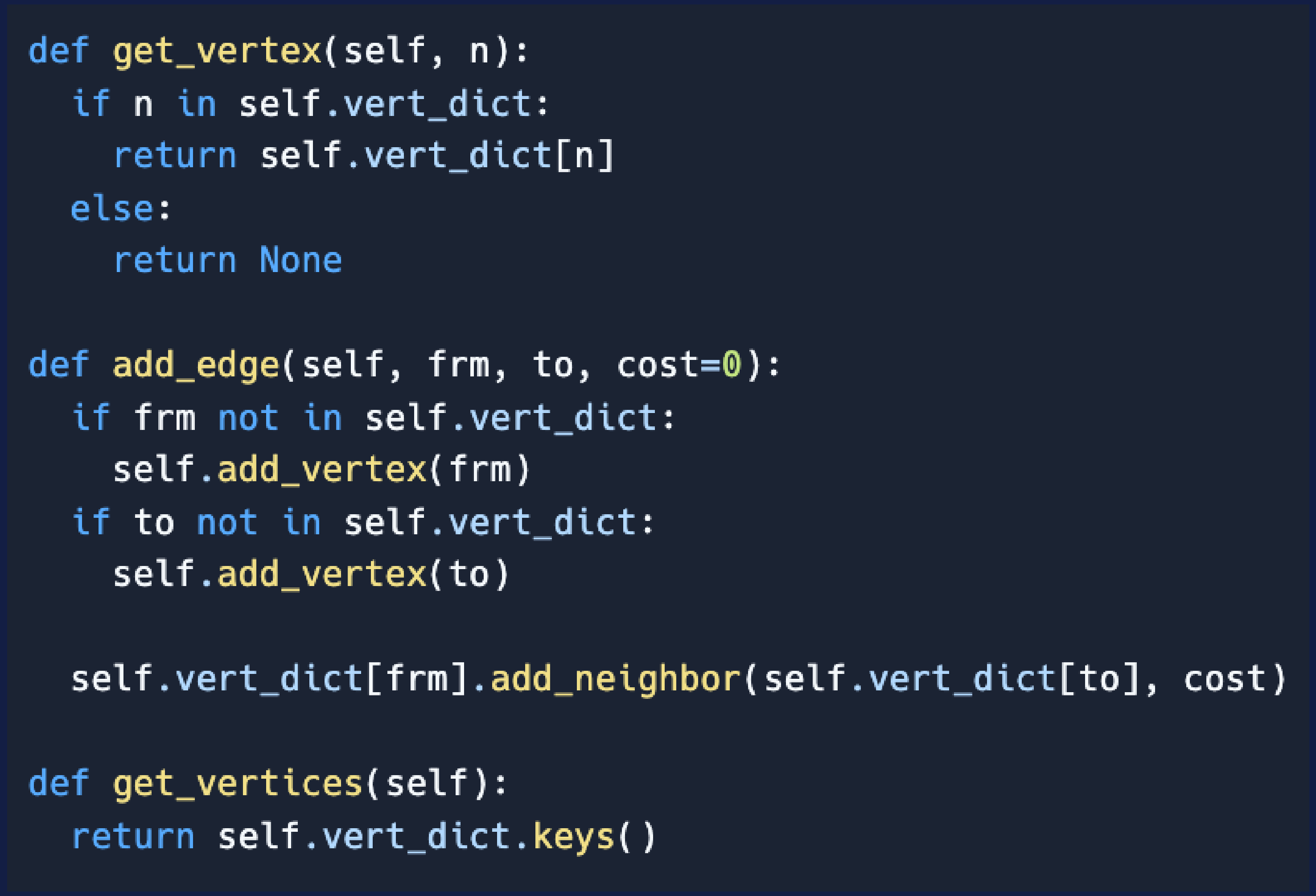
```python
class Graph:

  #initiallizing the graph
  def __init__(self):
    self.vert_dict = {}
    self.num_vertices = 0
    self.veriNumSet = {}

  def __iter__(self):
    return iter(self.vert_dict.values())
```

This is where our classes connect!

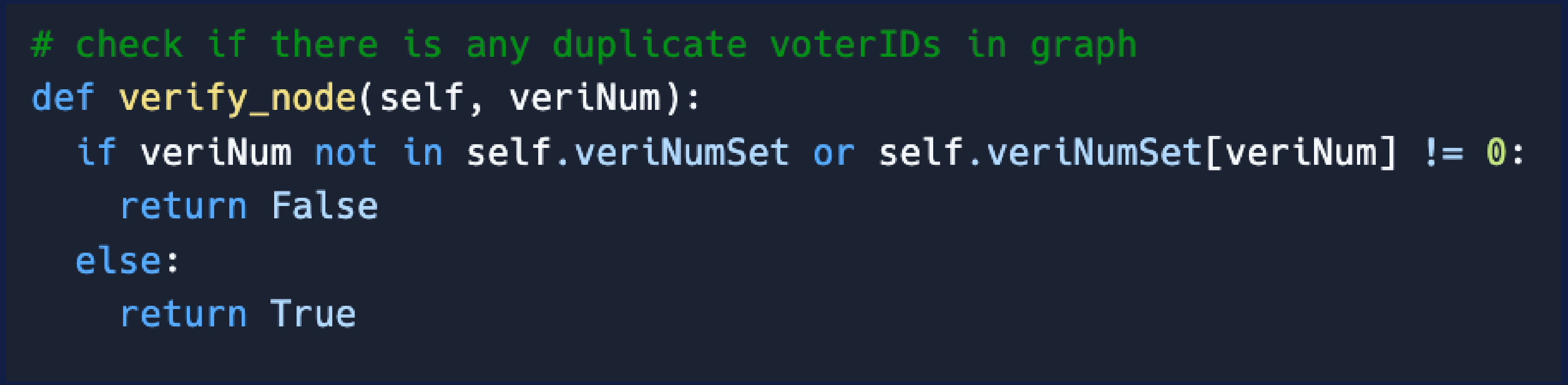


```python
# adding vertex to the tangleCV graph
def add_vertex(self, nodeID):
  self.num_vertices = self.num_vertices + 1
  cumulativeWeight = veriNum = 0

  new_vertex = Vertex(nodeID, veriNum, cumulativeWeight)
  self.vert_dict[nodeID] = new_vertex

  # 3% chance to be invalid veriNum
  chance = random.randint(1,33)
  if chance == 1:
    veriNum = random.randint(-100000,2147483647)
  else:
    veriNum = self.vert_dict[nodeID].encrypt()
  self.vert_dict[nodeID].set_veriNum(veriNum)

  if veriNum in self.veriNumSet:
    self.veriNumSet[veriNum] += 1
  else:
    self.veriNumSet[veriNum] = 0

  return new_vertex
```

Graph Class

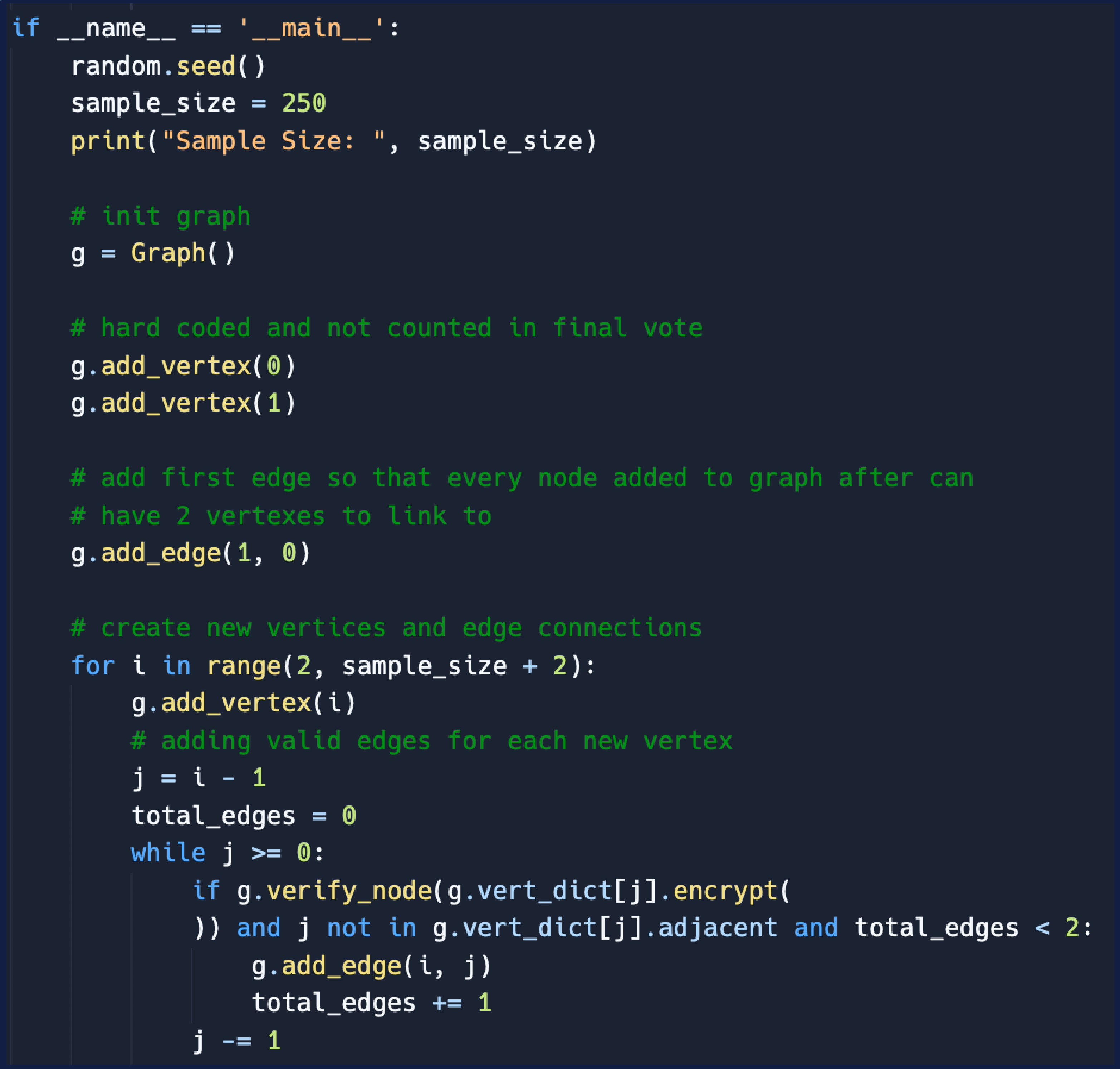


```python
def get_vertex(self, n):
  if n in self.vert_dict:
    return self.vert_dict[n]
  else:
    return None

def add_edge(self, frm, to, cost=0):
  if frm not in self.vert_dict:
    self.add_vertex(frm)
  if to not in self.vert_dict:
    self.add_vertex(to)

  self.vert_dict[frm].add_neighbor(self.vert_dict[to], cost)

def get_vertices(self):
  return self.vert_dict.keys()
```



```python
# check if there is any duplicate voterIDs in graph
def verify_node(self, veriNum):
  if veriNum not in self.veriNumSet or self.veriNumSet[veriNum] != 0:
    return False
  else:
    return True
```

Generating Our Graph



```
if __name__ == '__main__':
    random.seed()
    sample_size = 250
    print("Sample Size: ", sample_size)

    # init graph
    g = Graph()

    # hard coded and not counted in final vote
    g.add_vertex(0)
    g.add_vertex(1)

    # add first edge so that every node added to graph after can
    # have 2 vertexes to link to
    g.add_edge(1, 0)

    # create new vertices and edge connections
    for i in range(2, sample_size + 2):
        g.add_vertex(i)
        # adding valid edges for each new vertex
        j = i - 1
        total_edges = 0
        while j >= 0:
            if g.verify_node(g.vert_dict[j].encrypt(
            )) and j not in g.vert_dict[j].adjacent and total_edges < 2:
                g.add_edge(i, j)
                total_edges += 1
            j -= 1
```

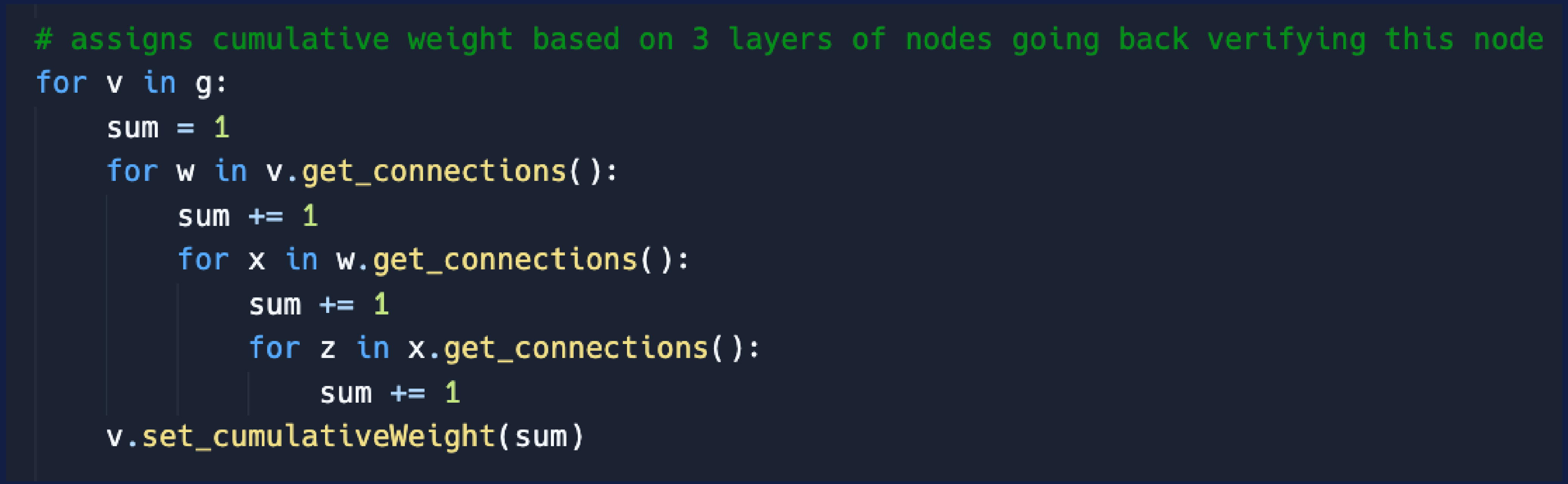
Outline

- Graph initialization
- Create 2 genesis nodes with an edge connecting them
- For loop creating 'sample_size' number of vertices with edges connected only to **valid** nodes

Vertex Weights

We wanted to have cumulatively weighted nodes, like our TangleCV example, but we ran into trouble when trying to iterate through all the vertices

After deciding to pivot, we created a cumulative weight based on the 3 layers above any given node

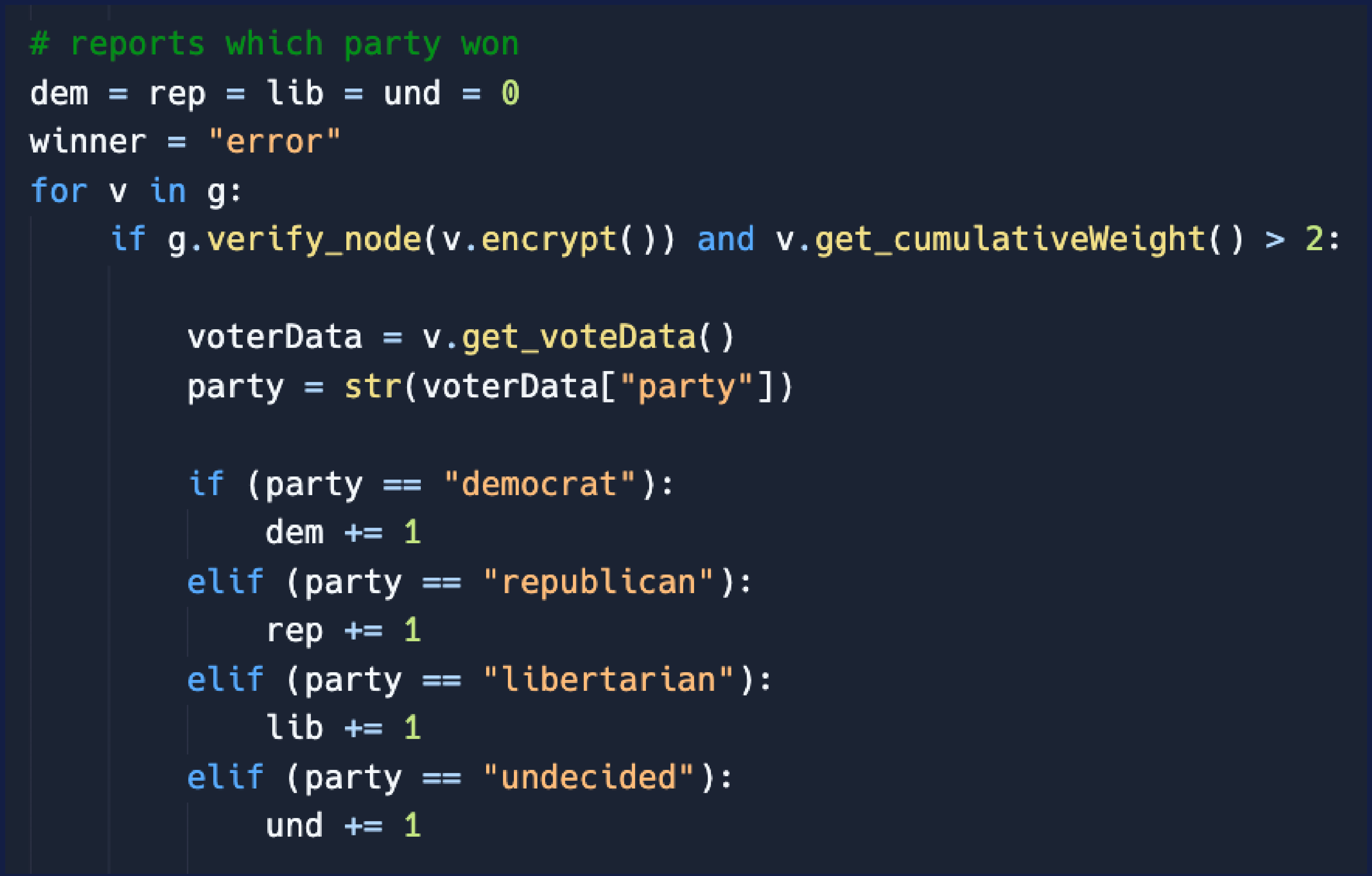


```
# assigns cumulative weight based on 3 layers of nodes going back verifying this node
for v in g:
    sum = 1
    for w in v.get_connections():
        sum += 1
        for x in w.get_connections():
            sum += 1
            for z in x.get_connections():
                sum += 1
    v.set_cumulativeWeight(sum)
```

Election Results

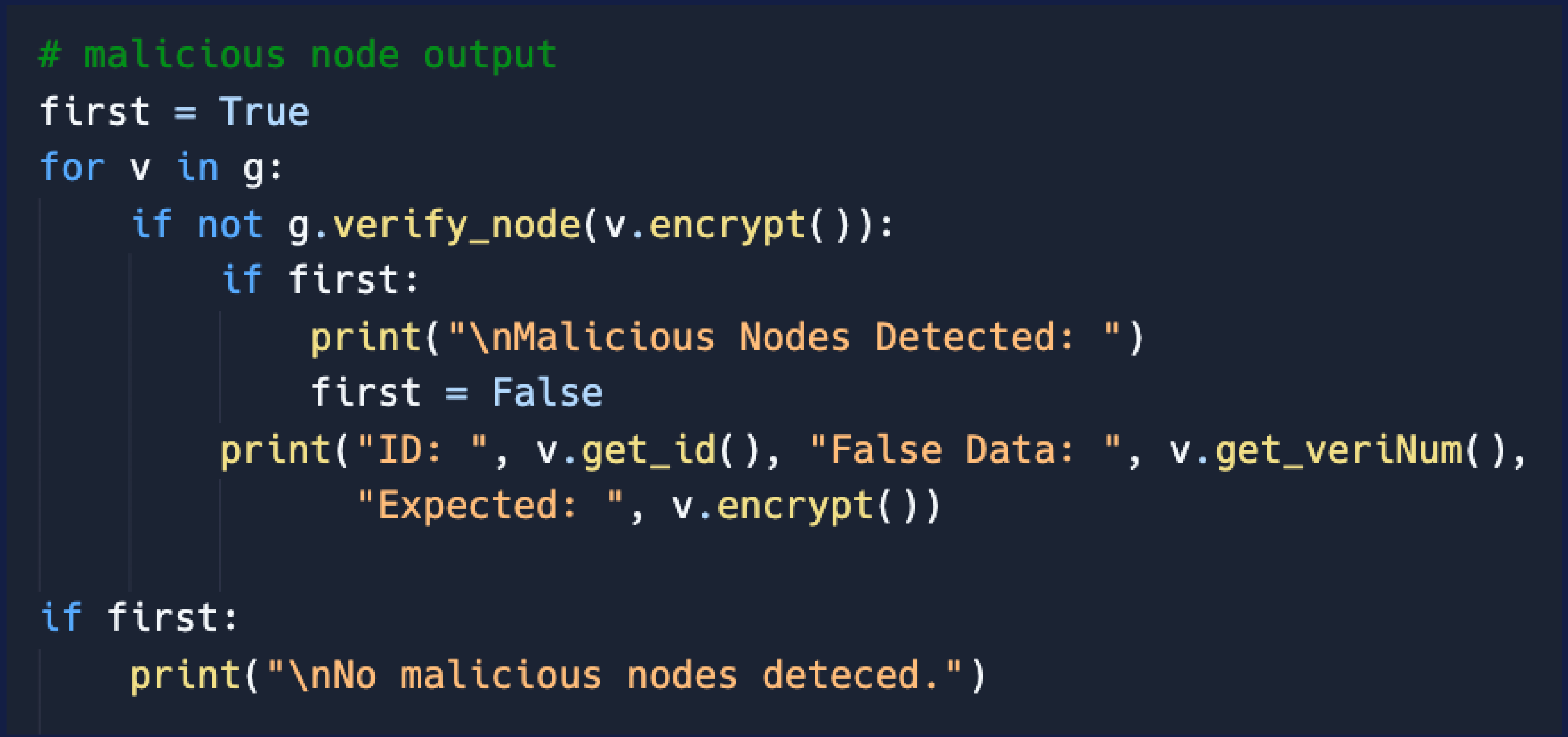
Of course, results must be accessible!

Only valid votes are counted.



```
# reports which party won
dem = rep = lib = und = 0
winner = "error"
for v in g:
    if g.verify_node(v.encrypt()) and v.get_cumulativeWeight() > 2:

        voterData = v.get_voteData()
        party = str(voterData["party"])

        if (party == "democrat"):
            dem += 1
        elif (party == "republican"):
            rep += 1
        elif (party == "libertarian"):
            lib += 1
        elif (party == "undecided"):
            und += 1
```

Malicious Nodes



```
# malicious node output
first = True
for v in g:
    if not g.verify_node(v.encrypt()):
        if first:
            print("\nMalicious Nodes Detected: ")
            first = False
        print("ID: ", v.get_id(), "False Data: ", v.get_veriNum(),
              "Expected: ", v.encrypt())

if first:
    print("\nNo malicious nodes deteced.")
```

Nodes flagged as malicious are printed, along with the conflicting data information

Areas of Improvement

- Cumulative vertex weights
- Edge connections based on weight
- More robust vote data analysis

THANK YOU FOR VOTING WITH

SecureVote!™

Enjoy our live demo →