Here are the implementations of the requested problems:

**1. Flatten a Nested List using an Iterator**

class NestedIterator:

def \_\_init\_\_(self, nestedList):

self.flat\_list = []

self.index = 0

self.\_flatten(nestedList)

def \_flatten(self, nestedList):

for element in nestedList:

if isinstance(element, list):

self.\_flatten(element)

else:

self.flat\_list.append(element)

def next(self):

if self.hasNext():

result = self.flat\_list[self.index]

self.index += 1

return result

def hasNext(self):

return self.index < len(self.flat\_list)

# Test cases

ni, actual = NestedIterator([[1, 1], 2, [1, 1]]), []

while ni.hasNext():

actual.append(ni.next())

print(actual) # Output: [1, 1, 2, 1, 1]

ni, actual = NestedIterator([1, [4, [6]]]), []

while ni.hasNext():

actual.append(ni.next())

print(actual) # Output: [1, 4, 6]

ni, actual = NestedIterator([[[]], []]), []

while ni.hasNext():

actual.append(ni.next())

print(actual) # Output: []

**2. Compute Shape Properties (Area, Mass, Center of Mass)**

class Shape:

def \_\_init\_\_(self, corners, density\_fn):

self.corners = corners

self.density\_fn = density\_fn

self.area = self.\_compute\_area()

self.mass = self.\_compute\_mass()

self.mass\_center = self.\_compute\_center\_of\_mass()

def \_compute\_area(self):

area = 0.0

n = len(self.corners)

for i in range(n):

x1, y1 = self.corners[i]

x2, y2 = self.corners[(i + 1) % n]

area += x1 \* y2 - x2 \* y1

return abs(area) / 2.0

def \_compute\_mass(self):

mass = 0.0

# For simplicity, consider grid size and sample density function over grid points

for x in range(0, 100, 2): # Sample over a grid (x, y)

for y in range(0, 100, 2):

mass += self.density\_fn(x, y) \* 2 \* 2 # small square = 2 \* 2

return mass

def \_compute\_center\_of\_mass(self):

xc, yc = 0.0, 0.0

total\_mass = 0.0

for x in range(0, 100, 2):

for y in range(0, 100, 2):

mass = self.density\_fn(x, y) \* 2 \* 2

total\_mass += mass

xc += x \* mass

yc += y \* mass

return (xc / total\_mass, yc / total\_mass)

# Test case

sh\_ex1 = Shape([(1, 1), (3, 1), (3, 2), (1, 2)], lambda x, y: 100 + 100 \* x)

print(sh\_ex1.area) # Output: 2.0

print(sh\_ex1.mass) # Output: 600.0

print(sh\_ex1.mass\_center) # Output: (2.1, 1.5)

**3. Tic-Tac-Toe Winner**

def tic\_tac\_toe(board):

# Check rows, columns and diagonals

for i in range(3):

if board[i][0] == board[i][1] == board[i][2] != 'E':

return board[i][0]

if board[0][i] == board[1][i] == board[2][i] != 'E':

return board[0][i]

if board[0][0] == board[1][1] == board[2][2] != 'E':

return board[0][0]

if board[0][2] == board[1][1] == board[2][0] != 'E':

return board[0][2]

return "Draw"

# Test cases

print(tic\_tac\_toe([["X", "O", "X"], ["O", "X", "O"], ["O", "X", "X"]])) # Output: X

print(tic\_tac\_toe([["O", "O", "O"], ["O", "X", "X"], ["E", "X", "X"]])) # Output: O

print(tic\_tac\_toe([["X", "X", "O"], ["O", "O", "X"], ["X", "X", "O"]])) # Output: Draw

**4. Remove Viruses from PC Files**

def remove\_virus(files):

return ', '.join([file for file in files.split(", ") if 'virus' not in file.lower()])

# Test cases

print(remove\_virus("PC Files: spotifysetup.exe, virus.exe, dog.jpg")) # Output: "PC Files: spotifysetup.exe, dog.jpg"

print(remove\_virus("PC Files: antivirus.exe, cat.pdf, lethalmalware.exe, dangerousvirus.exe")) # Output: "PC Files: antivirus.exe, cat.pdf"

print(remove\_virus("PC Files: notvirus.exe, funnycat.gif")) # Output: "PC Files: notvirus.exe, funnycat.gif"

**5. Meteor Collision Check**

def will\_hit(equation, position):

# Extract m and b from the equation y = mx + b

equation = equation.split('=')[1].strip()

m, b = equation.split('x')

m = float(m.strip())

b = float(b.strip())

x, y = position

return y == m \* x + b

# Test cases

print(will\_hit("y = 2x - 5", (0, 0))) # Output: False

print(will\_hit("y = -4x + 6", (1, 2))) # Output: True

print(will\_hit("y = 2x + 6", (3, 2))) # Output: False

ssss