Solutions to Homework 3

Question 1: 10 points.

Problem Statement: As shown in Figure 1 below, rectangles may be defined by the (x,y) coordinates of corner points that are diagonally opposite.

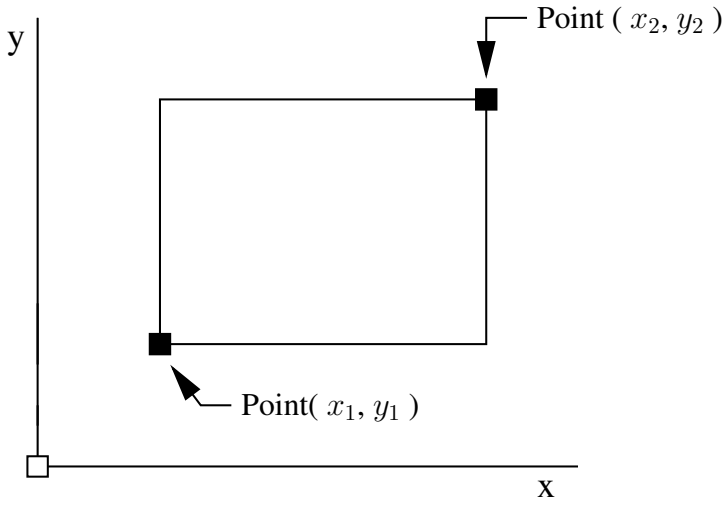

Figure 1: Definition of a rectangle via diagonally opposite corner points.

With this definition in place, the following script of code is a basic implementation of a class for creating and working with rectangle objects.

```
# ========================================================================
# Rectangle01.py: Very basic implementation of rectangle objects, where
# corner points are defined by varibles (x1, y1) and (x2, y2).
#
# Written by: Mark Austin                                    October 2025
# ========================================================================

class Rectangle:

  def __init__(self, x1, y1, x2, y2 ):
    self.x1 = x1;
    self.y1 = y1;
```

```
    self.x2 = x2;
    self.y2 = y2;
    self.name = "";

  # Set rectangle name ...

  def setName(self, name ):
    self.name = name;

  # Compute perimeter of rectangle ..

  def getPerimeter (self):
     perimeter = 2*(abs(self.x2-self.x1) + abs(self.y2-self.y1))
     return perimeter

  # Compute area of rectangle ..

  def getArea (self):
     area = abs(self.x2-self.x1)*abs(self.y2-self.y1);
     return area

  # String represention of rectangle ...

  def __str__(self):
     rectangleinfo = [];

     ... details removed ...

     return "".join(rectangleinfo);
```

The Rectangle class uses variables x1, y1, x2, and y2 to define the corner points, and has a method to create rectangle objects (i.e, __init__), convert the details of a rectangle object into a string format (i.e., __str__), and compute the rectangle area and perimeter (i.e., getArea() and getPerimeter()).

A script of test program usage is as follows:

```
prompt >>
prompt >> python3 TestRectangle01.py
--- Enter TestRectangle01.main()    ...
--- ============================== ...
--- Part 1: Create and print rectangle A ...

--- Rectangle: A ...
--- -----------------------------------------------
---   Corner Point (x1,y1) = (  1.00,   2.00) ...
---   Corner Point (x2,y2) = (  3.00,   4.00) ...
---   Perimeter =   8.00 ...
---   Area      =   4.00 ...
--- -----------------------------------------------

--- Part 2: Create and print rectangle B ...
```

```
--- Rectangle: B ...
--- -------------------------------------------------
---   Corner Point (x1,y1) = (  0.00,   0.00) ...
---   Corner Point (x2,y2) = (  6.00,   5.00) ...
---   Perimeter =  22.00 ...
---   Area      =  30.00 ...
--- -------------------------------------------------

--- ============================== ...
--- Finished TestRectangle01.main()    ...
prompt >> exit
```

Source Code: Full details of the rectangle code and test program can be found in: python-code.d/classes/ ...

Question: Now suppose that instead of using the variables x_1, y_1, x_2 and y_2 to define the corner points, we use the class Point:

```
import math

class Point:

    def __init__(self, xCoord=0, yCoord=0):
        self.__xCoord = xCoord
        self.__yCoord = yCoord

    # get x coordinate

    def get_xCoord(self):
        return self.__xCoord

    ..... details of other functions removed ...
```

The appropriate modification for Rectangle is:

```
from Point import Point

class Rectangle:

   def __init__(self, x1, y1, x2, y2 ):
       self.pt1 = Point(x1,y1)     # <-- create lower corner point ...
       self.pt2 = Point(x2,y2)     # <-- create upper corner point ...

   ..... details rectangle removed ....
}
```

The arrangement of Rectangle and Point classes can be visualized as follows:

Figure 2: Test program script and classes in a rectangle system.

What to do? Fill in the missing details (i.e., constructors and __str__ method) of class Point. Modify the code in Rectangle to use the Point class. The resulting program should have essentially the same functionality as the original implementation (v1) of Rectangle.

Python Source Code: The source code is comprised of three Python files: Point.py, Rectangle.py and TestRectangle.py.

Abbreviated Object Source Code: Python.py

```
# ======================================================
# Point class that demonstrates overloading of operators
# for arithmetic and relational expressions.
#
# Modified by: Mark Austin                    October 2025
# ======================================================

import math

class Point:

    def __init__(self, xCoord=0, yCoord=0):
        self.xCoord = xCoord
        self.yCoord = yCoord

    # Get/set X, Y coordinates

    def getX(self):
        return self.xCoord

    def setX(self, xCoord):
        self.xCoord = xCoord

    # Get/set Y coordinate

    def getY(self):
        return self.yCoord

    def setY(self, yCoord):
        self.yCoord = yCoord

    # Get current position

    def get_position(self):
        return self.__xCoord, self.__yCoord
```

```
    # Move x & y coordinates by p & q

    def move(self, p, q):
        self.xCoord += p
        self.yCoord += q

    # Compute distance between two points ...

    def distance(self, second):
        x_d = self.xCoord - second.xCoord
        y_d = self.yCoord - second.yCoord
        return (x_d**2 + y_d**2)**0.5

    # Return string represention of object ...

    def __str__(self):
        return "( %6.2f, %6.2f )" % ( self.xCoord, self.yCoord )
```

Object Source Code: Rectangle02.py

```
# ============================================================================
# Rectangle02.py: Very basic implementation of rectangle objects, where
# corner points are defined by varibles (x1, y1) and (x2, y2).
#
# Written by: Mark Austin                                          October 2025
# ============================================================================

import math

from Point import Point

class Rectangle:

  def __init__(self, x1, y1, x2, y2 ):
    self.pt1 = Point(x1,y1)
    self.pt2 = Point(x2,y2)
    self.name = "";

  # Set rectangle name ...

  def setName(self, name ):
    self.name = name;

  # Compute perimeter of rectangle ..

  def getPerimeter (self):
     x1 = self.pt1.getX(); y1 = self.pt1.getY();
     x2 = self.pt2.getX(); y2 = self.pt2.getY();
     perimeter = 2*((x2 - x1) + abs(y2 - y1))
     return perimeter

  # Compute area of rectangle ..
```

```
  def getArea (self):
     x1 = self.pt1.getX(); y1 = self.pt1.getY();
     x2 = self.pt2.getX(); y2 = self.pt2.getY();
     area = abs( x2 - x1 )*abs( y2 - y1)
     return area

  # String represention of rectangle ...

  def __str__(self):
     rectangleinfo = [];
     rectangleinfo.append("--- Rectangle: {:s} ...  \n".format( self.name ));
     rectangleinfo.append("--- ------------------------------------------------ \n");
     rectangleinfo.append("---   Corner Point (x1,y1) = {:s} ... \n".format( self.pt1.__str__() ));
     rectangleinfo.append("---   Corner Point (x2,y2) = {:s} ... \n".format( self.pt2.__str__() ));
     rectangleinfo.append("---   Perimeter = {:6.2f} ... \n".format( self.getPerimeter()));
     rectangleinfo.append("---   Area      = {:6.2f} ... \n".format( self.getArea()));
     rectangleinfo.append("--- ------------------------------------------------ \n");
     return "".join(rectangleinfo);
```

Test Program Source Code: TestRectangle02.py

```
# ================================================================
# TestRectangle02.py: Exercise point() version of Rectangle.
#
# Written by: Mark Austin                                October 2025
# ================================================================

from Rectangle02 import Rectangle;

# main method ...

def main():
    print("--- Enter TestRectangle02.main()       ... ");
    print("--- ================================== ... \n");

    print("--- Part 1: Create and print rectangle A ... \n");

    rectangleA = Rectangle( 1.0, 2.0, 3.0, 4.0 )
    rectangleA.setName("A")
    print(rectangleA)

    print("--- Part 2: Create and print rectangle B ... \n");

    rectangleB = Rectangle( 0.0, 0.0, 6.0, 5.0 )
    rectangleB.setName("B")
    print(rectangleB)

    print("--- ================================== ... ");
    print("--- Finished TestRectangle02.main()       ... ");

# call the main method ...
```

```
if __name__ == "__main__":
    main()
```

Abbreviated Program Output:

```
--- Enter TestRectangle02.main()       ...
--- ================================== ...

--- Part 1: Create and print rectangle A ...

--- Rectangle: A ...
--- ------------------------------------------------
---   Corner Point (x1,y1) = (   1.00,   2.00 ) ...
---   Corner Point (x2,y2) = (   3.00,   4.00 ) ...
---   Perimeter =   8.00 ...
---   Area      =   4.00 ...
--- ------------------------------------------------

--- Part 2: Create and print rectangle B ...

--- Rectangle: B ...
--- ------------------------------------------------
---   Corner Point (x1,y1) = (   0.00,   0.00 ) ...
---   Corner Point (x2,y2) = (   6.00,   5.00 ) ...
---   Perimeter =  22.00 ...
---   Area      =  30.00 ...
--- ------------------------------------------------

--- ================================== ...
--- Finished TestRectangle02.main()       ...
```

Question 2: 20 points.

Problem Statement: The left-hand side of Figure 3 shows the essential details of a domain familiar to many children. One by one, rectangular blocks are stacked as high as possible until they come tumbling down – the goal, afterall, is to create a spectacular crash!!

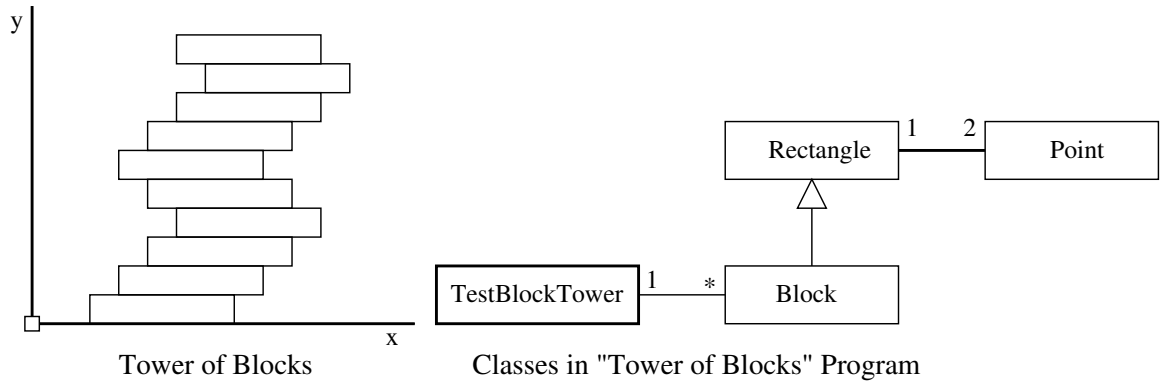

Figure 3: Schematic and classes for tower of blocks problem.

Suppose that we wanted to model this process and use engineering principles to predict incipient instability of the block tower. Consider the following observations:

1. Rather than start from scratch, it would make sense to create a Block class that inherits the properties of Rectangle (previous question), and adds details relevant to engineering analysis (e.g., the density of the block).

2. Then we could develop a BlockTower class that systematically assembles the tower, starting at the base and working upwards. At each step of the tower assembly, analysis procedures should make sure that the tower is still stable.

The right-hand side of Figure 3 shows the relatioship among the classes. One TestBlockTower program (1) will employ many blocks, as indicated by the asterik (*).

Develop a Python program that builds upon the Rectangle class written in the previous questions. The class Block should store the depth and density of the block – this will be important in determining the mass and centroid of each block. The TestBlockTower class will use block objects to build the tower. A straight forward way of modeling the block tower is with a List. After each block is added, the program should conduct a stability check. If the system is still stable, then add another block should be added. The simulation should cease when the tower of blocks eventually becomes unstable.

Note. To simplify the analysis, assume that adjacent blocks are firmly connected.

Stability Considerations. If the blocks are stacked perfectly on top of each other, then from a mathematical

standpoint the tower will never become unstable. In practice, this never happens. There is always a small offset and, eventually, it's the accumulation of offsets that leads to spectacular disaster.

For the purposes of this question, assume that blocks are six units wide, one unit high, and depth of one unit. When a new block is added, the block offset should be one unit. To make the question interesting, assume that four blocks are stacked with an offset to the right, then three blocks are added with an offset to the left, then four to the right, three to the left, and so forth. This sequence can be accomplished with the looping construct:

```
# Compute incremental offset for i-th block .....

offset = math.floor((BlockNo - 1)/5.0) + (BlockNo-1)%5;
if ((BlockNo-1)%5 == 4 ):
   offset = offset - 2;
```

The tower will become unstable when the center of gravity of blocks above a particular level falls outside the edge of the supporting block.

What to do? Write a Python program that will:

1. Determine how many blocks can be added to the stack before it crashes.
2. Create a figure of the block configuration and centroid position immediately before collapse.

Python Source Code: Four files: Point.py, Vertex01.py, Block01.py, TestBlockTower01.py.

Abbreviated Point Object Code: Point02.py

```
# ======================================================
# Point class that demonstrates overloading of operators
# for arithmetic and relational expressions.
#
# Modified by: Mark Austin                   October 2025
# ======================================================

import math

class Point:

    def __init__(self, xCoord=0, yCoord=0):
        self.xCoord = xCoord
        self.yCoord = yCoord

    # Get/set X coordinate
```

```
    def getX(self):
        return self.xCoord

    def setX(self, xCoord):
        self.xCoord = xCoord

    # Get/set Y coordinate

    def getY(self):
        return self.yCoord

    def setY(self, yCoord):
        self.yCoord = yCoord

    # Get current position

    def get_position(self):
        return self.__xCoord, self.__yCoord

    # change x & y coordinates by p & q

    def move(self, p, q):
        self.xCoord += p
        self.yCoord += q

    # compute distance between two points ...

    def distance(self, second):
        x_d = self.xCoord - second.xCoord
        y_d = self.yCoord - second.yCoord
        return (x_d**2 + y_d**2)**0.5

    # Overload Arithmetic operators ...
    # ---------------------------------------

    ... details removed ...

    # Overload Relational Operators ...
    # ---------------------------------------

    ... details removed ...

    # return string represention of object ...
    # ---------------------------------------

    def __str__(self):
        return "( %6.2f, %6.2f )" % ( self.xCoord, self.yCoord )
```

Vertex Object Code: Vertex01.py

```
# ============================================================
# Vertex.py: A vertex is a point with a label ...
```

```
# ============================================================

from Point import Point

class Vertex(Point):
   label = ""

   # Constructor method ...

   def __init__(self, x, y) :
     Point.__init__(self, x, y)
     self.label = ""

   # -----------------------------------------
   # Set/get label ...
   # -----------------------------------------

   def setLabel(self, label ):
     self.label = label

   def getLabel(self):
     return self.label

   # -----------------------------------------
   # Assemble string represention of Vertex ...
   # -----------------------------------------

   def __str__(self):
      vertexinfo = [];
      vertexinfo.append("\n");
      vertexinfo.append("--- Vertex: {:s} ... \n".format( self.getLabel()));
      vertexinfo.append("--- ------------------------------------------------ \n");
      vertexinfo.append("---   Coordinate: (x,y) = {:s} ... \n".format( Point.__self__()));
      vertexinfo.append("--- ------------------------------------------------ ");
      return "".join(vertexinfo);
```

Block Object Code: Block01.py

```
# ====================================================================
# Block01.py: A block is rectangle that has mass (density + thickness).
#
# Written by: Mark Austin                                  October 2025
# ====================================================================

import math

from Rectangle02 import Rectangle
from Point import Point

from matplotlib.patches import Circle
from matplotlib.lines import Line2D

class Block (Rectangle):
```

```
density   = 0.0;
thickness = 0.0;

def __init__(self, x1, y1, x2, y2 ):
   Rectangle.__init__(self, x1, y1, x2, y2 );

# Set block density ...

def setDensity (self, density ):
   self.density = density;

# Set block thickness density ...

def setThickness (self, thickness ):
   self.thickness = thickness;

# Compute block mass and centroid ...

def getMass(self):
   volume = self.thickness * self.getArea();
   return self.density*volume;

def getCentroid(self):
    centroid = Point();

    centroid.setX( 1.0/2.0 * ( self.pt1.getX() + self.pt2.getX() ));
    centroid.setY( 1.0/2.0 * ( self.pt1.getY() + self.pt2.getY() ));
    return centroid;

# Draw block ...

def draw(self, ax):
    width = 0.1;

    x1 = self.pt1.getX();
    y1 = self.pt1.getY();
    x2 = self.pt2.getX();
    y2 = self.pt2.getY();

    # Draw block edges ...

    ax.add_line( Line2D( [x1, x1], [y1, y2] ) )
    ax.add_line( Line2D( [x1, x2], [y2, y2] ) )
    ax.add_line( Line2D( [x2, x2], [y2, y1] ) )
    ax.add_line( Line2D( [x2, x1], [y1, y1] ) )

    # Draw block vertices as small circles ...

    ax.add_patch( Circle( (x1, y1), width, facecolor='red') )
    ax.add_patch( Circle( (x2, y1), width, facecolor='red') )
    ax.add_patch( Circle( (x1, y2), width, facecolor='red') )
    ax.add_patch( Circle( (x2, y2), width, facecolor='red') )

# String represention of block ...
```

```
   def __str__(self):
      blockinfo = [];
      blockinfo.append("--- Block: {:s} ...  \n".format( self.name ));
      blockinfo.append("--- Density: {:f} ...  \n".format( self.density ));
      # blockinfo.append("--- Centroid: {:s} ...  \n".format( self.getCentroid().__str()__ ));
      blockinfo.append("--- ------------------------------------------------ \n");
      blockinfo.append("---   Corner Point (x1,y1) = {:s} ... \n".format( self.pt1.__str__() ));
      blockinfo.append("---   Corner Point (x2,y2) = {:s} ... \n".format( self.pt2.__str__() ));
      blockinfo.append("--- ------------------------------------------------ \n");
      return "".join(blockinfo);
```

Test Program: TestBlockTower01.py

```
# ======================================================================
# TestObjectBlockTower01.py: Assemble tower of blocks until they become
# unstable.
#
# Written by: Mark Austin                                    October 2025
# ======================================================================

import math;
import matplotlib.pyplot as plt

from matplotlib.patches import Circle
from matplotlib.lines import Line2D

from Block01 import Block;

# main method ...

def main():
    print("--- Enter TestObjectBlockTower01.main() ... ");
    print("--- ================================== ... ");

    print("--- \n");
    print("--- Part 1: Create and print test block A ...");
    print("--- ===================================== ...");

    blockA = Block( 0.0, 0.0, 6.0, 1.0 )
    blockA.setName("A")
    blockA.setThickness( 1.0 )
    blockA.setDensity( 1.0 )

    print(blockA)
    print("--- Block mass  = {:6.2f} ...".format( blockA.getMass() ));
    print("--- CentroidX() = {:6.2f} ...".format( blockA.getCentroid().getX() ));
    print("--- CentroidY() = {:6.2f} ...".format( blockA.getCentroid().getY() ));

    print("--- ");
    print("--- Part 2: Simulate Block Tower ... ");
    print("--- ===================================== ... ");
    print("--- ");
```

```
tower = [];

blockTowerStable = True;
maxIterations = 20;
BlockNo       = 0;
towerCentroidX = 0.0;
towerCentroidY = 0.0;
while( blockTowerStable == True and BlockNo < maxIterations ):
   BlockNo = BlockNo + 1;

   print("--- ");
   print("--- Add Block No: {:d} ...".format( BlockNo ));
   print("--- ======================================== ... ");

   # Compute incremental offset for i-th block .....

   offset = math.floor((BlockNo - 1)/5.0) + (BlockNo-1)%5;
   if ((BlockNo-1)%5 == 4 ):
      offset = offset - 2;

   print("--- offset = {:d} ...".format( offset ));

   # Compute (x,y) coordinates of block vertices...

   x1 =  0.0 + offset;
   x2 =  6.0 + offset;
   y1 =  0.0 + BlockNo - 1;
   y2 =  1.0 + BlockNo - 1;

   # Create new block ...

   b = Block ( x1, y1, x2, y2 );
   b.setDensity( 1.0 );
   b.setThickness( 1.0 )
   print(b)

   # Add block to tower ....

   tower.append( b );

   # Compute (x,y) coordinates of tower centroid ...

   TotalMass = 0.0;
   FirstMomentX = 0.0;
   FirstMomentY = 0.0;

   for item in tower:
       TotalMass += item.getMass();
       FirstMomentX = FirstMomentX + item.getMass() * item.getCentroid().getX();
       FirstMomentY = FirstMomentY + item.getMass() * item.getCentroid().getY();

   print("--- Block Tower Analytics                 ...");
   print("--- ------------------------------------ ...");
   print("--- Total Mass     = {:6.2f} ...".format(TotalMass) );
   print("--- FirstMoment(X) = {:6.2f} ...".format( FirstMomentX ));
```

```
        print("--- FirstMoment(Y) = {:6.2f} ...".format( FirstMomentY ));
        print("--- Tower Centroid(X) = {:6.2f} ...".format( FirstMomentX/TotalMass ));
        print("--- Tower Centroid(Y) = {:6.2f} ...".format( FirstMomentY/TotalMass ));

        # Save tower centroid ...

        towerCentroidX = FirstMomentX/TotalMass;
        towerCentroidY = FirstMomentY/TotalMass;

        # Test for stability of tower ...

        print("--- ------------------------------------ ...");
        if ( FirstMomentX/TotalMass < 6.0 ):
           print("--- Tower of blocks is stable ...");
        else:
           blockTowerStable = False;
           print("");
           print("--- Crash!!");
           print("--- Tower of {:d} blocks is unstable".format(BlockNo) );

    print("--- ");
    print("--- Part 3: Draw Block Tower ... ");
    print("--- ==================================== ... ");

    # Define Matplotlib figure and axis

    fig, ax = plt.subplots()

    # Draw individual blocks in tower ...

    for block in tower:
        block.draw(ax);

    # Draw tower centroid ...

    width = 0.2;
    ax.add_patch( Circle( (towerCentroidX, towerCentroidY), width, facecolor='green') )

    # Create and show matplotlib graphic ...

    plt.title('Block Tower at Collapse')
    plt.ylabel('y')
    plt.xlabel('x')
    plt.ylim( -1, 20)
    plt.xlim( -1, 14)
    plt.grid(True)
    plt.show()

    print("--- ======================================= ... ");
    print("--- Finished TestObjectBlockTower01.main() ... ");

# call the main method ...

if __name__ == "__main__":
    main()
```

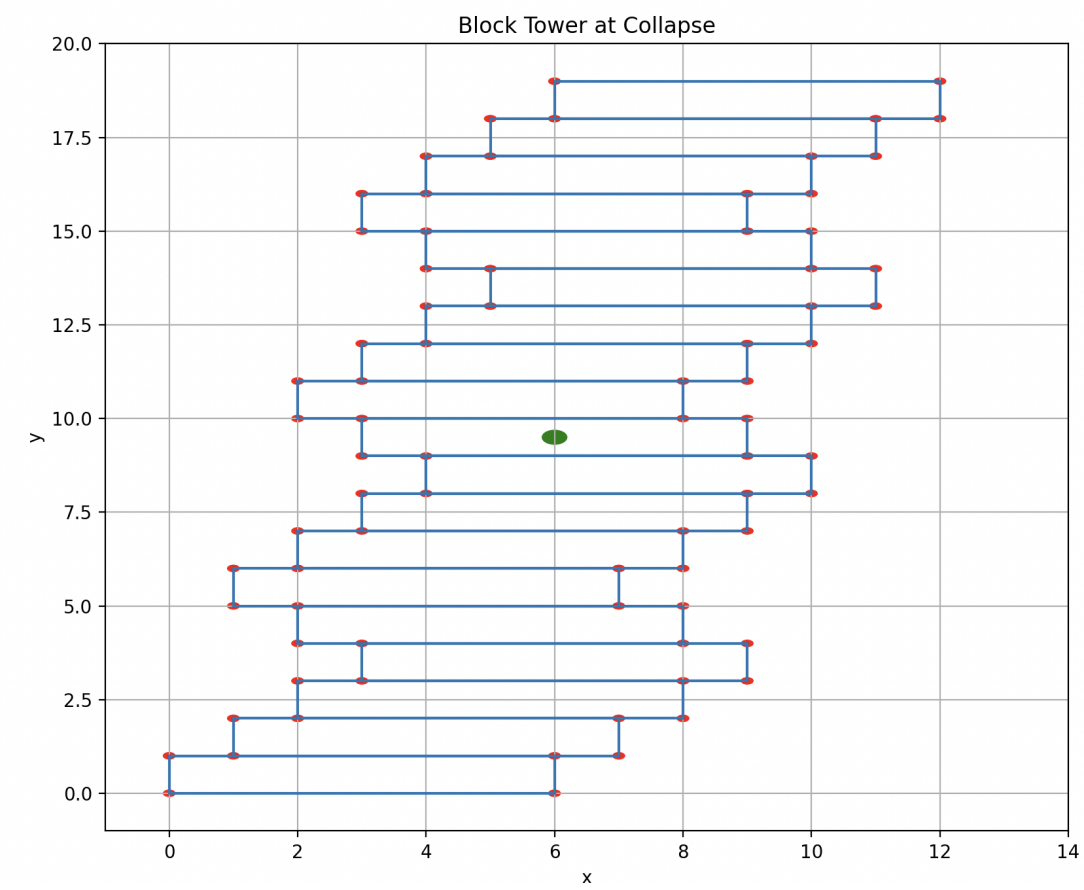

Figure 4: Block tower at collapse.

Program Output: The abbreviated textual output is:

```
--- Enter TestObjectBlockTower01.main() ...
--- ================================== ...

--- Part 1: Create and print test block A ...
--- ===================================== ...
--- Block: A ...
--- Density: 1.000000 ...
--- ------------------------------------------------
---   Corner Point (x1,y1) = (   0.00,   0.00 ) ...
---   Corner Point (x2,y2) = (   6.00,   1.00 ) ...
--- ------------------------------------------------

--- Block mass  =   6.00 ...
--- CentroidX() =   3.00 ...
--- CentroidY() =   0.50 ...
```

```
---
--- Part 2: Simulate Block Tower ...
--- ================================== ...
---
--- Add Block No: 1 ...
--- ======================================== ...
--- offset = 0 ...
--- Block:  ...
--- Density: 1.000000 ...
--- ------------------------------------------------
---   Corner Point (x1,y1) = (   0.00,   0.00 ) ...
---   Corner Point (x2,y2) = (   6.00,   1.00 ) ...
--- ------------------------------------------------

--- Block Tower Analytics                ...
--- ---------------------------------- ...
--- Total Mass     =   6.00 ...
--- FirstMoment(X) =  18.00 ...
--- FirstMoment(Y) =   3.00 ...
--- Tower Centroid(X) =   3.00 ...
--- Tower Centroid(Y) =   0.50 ...
--- ---------------------------------- ...
--- Tower of blocks is stable ...
---
--- Add Block No: 2 ...
--- ======================================== ...
--- offset = 1 ...
--- Block:  ...
--- Density: 1.000000 ...
--- ------------------------------------------------
---   Corner Point (x1,y1) = (   1.00,   1.00 ) ...
---   Corner Point (x2,y2) = (   7.00,   2.00 ) ...
--- ------------------------------------------------

--- Block Tower Analytics                ...
--- ---------------------------------- ...
--- Total Mass     =  12.00 ...
--- FirstMoment(X) =  42.00 ...
--- FirstMoment(Y) =  12.00 ...
--- Tower Centroid(X) =   3.50 ...
--- Tower Centroid(Y) =   1.00 ...
--- ---------------------------------- ...
--- Tower of blocks is stable ...

... lines of output removed ...

--- Add Block No: 18 ...
--- ======================================== ...
--- offset = 5 ...
--- Block:  ...
--- Density: 1.000000 ...
--- ------------------------------------------------
---   Corner Point (x1,y1) = (   5.00,  17.00 ) ...
---   Corner Point (x2,y2) = (  11.00,  18.00 ) ...
--- ------------------------------------------------
```

```
--- Block Tower Analytics                ...
--- ------------------------------------ ...
--- Total Mass     = 108.00 ...
--- FirstMoment(X) = 630.00 ...
--- FirstMoment(Y) = 972.00 ...
--- Tower Centroid(X) =   5.83 ...
--- Tower Centroid(Y) =   9.00 ...
--- ------------------------------------ ...
--- Tower of blocks is stable ...
---
--- Add Block No: 19 ...
--- ======================================== ...
--- offset = 6 ...
--- Block:  ...
--- Density: 1.000000 ...
--- ------------------------------------------------
---   Corner Point (x1,y1) = (   6.00,  18.00 ) ...
---   Corner Point (x2,y2) = (  12.00,  19.00 ) ...
--- ------------------------------------------------

--- Block Tower Analytics                ...
--- ------------------------------------ ...
--- Total Mass     = 114.00 ...
--- FirstMoment(X) = 684.00 ...
--- FirstMoment(Y) = 1083.00 ...
--- Tower Centroid(X) =   6.00 ...
--- Tower Centroid(Y) =   9.50 ...
--- ------------------------------------ ...

--- Crash!!
--- Tower of 19 blocks is unstable
---
--- Part 3: Draw Block Tower ...

--- ===================================== ...
--- Finished TestObjectBlockTower01.main() ...
```

Question 3: 10 points.

Problem Statement: Figure 5 plots the function

$$f(x) = \left[x - \frac{1}{x}\right]^3 + \left[x - \frac{1}{x}\right] - 30. \tag{1}$$

over the range $[-4, 4]$.

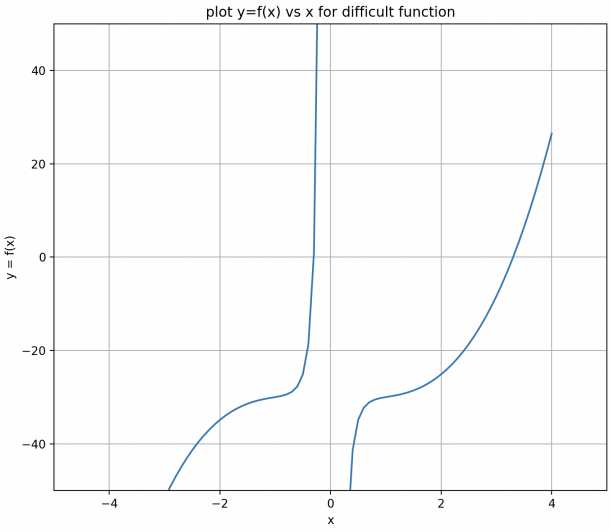

Figure 5: Plot y = f(x) vs x.

Theoretical considerations indicate that equation has two real roots:

$$[r_1, r_2] = \frac{3}{2} \pm \frac{\sqrt{13}}{2} \tag{2}$$

Now suppose that we have Figure 5, and can see that the two roots lie in the intervals $[-1, 0]$ and $[3, 4]$, but for some reason don't know about equation 2.

Write a Python program that:

1. Uses the **method of bisection** to compute the lower and upper roots to equation
2. Computes the roots with the **method of Newton Raphson** iteration.
3. Investigates behavior of **Newton Raphson iteration** when we seek solutions to the lower root and the starting value x_o = -2.

Python Source Code:

```
# ============================================================================
# TestNewtonRaphson04.py: Use methods of bisection and newton raphson to
# compute roots to:
#
#            f(x) = (x - 1/x)**3 + (x - 1/x) - 30 = 0.0
#
# Written By: Mark Austin                                          October 2025
# ============================================================================

import math;
import Solutions;
import numpy as np
import matplotlib.pyplot as plt

# Function: f(x) = (x - 1/x)**3 + (x - 1/x) - 30

def f1(x):
    u = x - 1/x;
    return u**3 + u - 30;

def df1(x):
    u  = x - 1/x;
    du = 1 + 1/x**2;
    return 3*u*u*du + du;

# main method ...

def main():
    print("--- Enter TestNewtonRaphson04.main()          ... ");
    print("--- ========================================= ... ");

    print("--- ");
    print("--- Part 1: Method of Bisection to solve f1(x) = 0 for upper root ... ");
    print("--- ============================================================= ... ");

    # Initialize problem setup ...

    a =  3.0; b =  4.0;
    tolerance     = 0.00000001
    maxiterations = 100

    print("--- Inputs:")
    print("---   a = {:5.2f} ...".format(a) )
    print("---   b = {:5.2f} ...".format(b) )
    print("---   tolerance      = {:16.8f} ...".format(tolerance) )
    print("---   max iterations = {:16.8f} ...".format(maxiterations) )

    # Compute roots to equation ...

    print("--- Execution:")
    root, i, converged = Solutions.bisection(f1, a, b, tolerance, maxiterations )

    # Summary of computations ...
```

```
print("--- Output:")
print("---   root = {:10.5f} ...".format(root) )
print("---   f(root) --> {:12.5e} ...".format( f1(root)) )
print("---   no iterations = {:d} ...".format(i) )
print("---   converged: {:s} ...".format( str(converged) ) )

print("--- ");
print("--- Part 1: Method of Bisection to solve f1(x) = 0 for lower root ... ");
print("--- ============================================================ ... ");

# Initialize problem setup ...

a =  -1.0; b =  -0.1;
tolerance     = 0.00000001
maxiterations = 100

print("--- Inputs:")
print("---   a = {:5.2f} ...".format(a) )
print("---   b = {:5.2f} ...".format(b) )
print("---   tolerance      = {:16.8f} ...".format(tolerance) )
print("---   max iterations = {:16.8f} ...".format(maxiterations) )

# Compute roots to equation ...

print("--- Execution:")
root, i, converged = Solutions.bisection(f1, a, b, tolerance, maxiterations )

# Summary of computations ...

print("--- Output:")
print("---   root = {:10.5f} ...".format(root) )
print("---   f(root) --> {:12.5e} ...".format( f1(root)) )
print("---   no iterations = {:d} ...".format(i) )
print("---   converged: {:s} ...".format( str(converged) ) )

print("--- ");
print("--- Part 2: Solve f1(x) = 0 for upper root: x0 = 10 ... ");
print("--- ============================================== ... ");

# Initialize problem setup ...

x0 = 10.0;
tolerance     = 0.00000001
maxiterations = 100

print("--- Inputs:")
print("---   x0 = {:5.2f} ...".format(x0) )
print("---   tolerance      = {:16.8f} ...".format(tolerance) )
print("---   max iterations = {:16.8f} ...".format(maxiterations) )

# Compute roots to equation ...

print("--- Execution:")
root, i, converged = Solutions.newtonraphson(f1, df1, x0, tolerance, maxiterations )
```

```
# Summary of computations ...

print("--- Output:")
print("---   root = {:16.8f} ...".format(root) )
print("---   f(root)   --> {:16.8e} ...".format(   f1(root)) )
print("---   df(root)  --> {:16.8e} ...".format(  df1(root)) )
print("---   no iterations = {:d} ...".format(i) )
print("---   converged: {:s} ...".format( str(converged) ) )

print("--- ");
print("--- Part 2: Solve f1(x) = 0 for lower root: x0 = -0.1 ... ");
print("--- ================================================ ... ");

x0 = -0.10;
print("--- Inputs:")
print("---   x0 = {:5.2f} ...".format(x0) )
print("---   tolerance      = {:8.5f} ...".format(tolerance) )
print("---   max iterations = {:8.2f} ...".format(maxiterations) )

# Compute roots to equation ...

print("--- Execution:")
root, i, converged = Solutions.newtonraphson(f1, df1, x0, tolerance, maxiterations )

# Summary of computations ...

print("--- Output:")
print("---   root = {:16.8f} ...".format(root) )
print("---   f(root)   --> {:16.8e} ...".format(   f1(root)) )
print("---   df(root)  --> {:16.8e} ...".format(  df1(root)) )
print("---   no iterations = {:d} ...".format(i) )
print("---   converged: {:s} ...".format( str(converged) ) )

print("--- ");
print("--- Part 3: Solve f1(x) = 0 for lower root: x0 = -2.0 ... ");
print("--- ================================================ ... ");

x0 = -2.00; # <-- This could converge to the wrong root ...

print("--- Inputs:")
print("---   x0 = {:5.2f} ...".format(x0) )
print("---   tolerance      = {:8.5f} ...".format(tolerance) )
print("---   max iterations = {:8.2f} ...".format(maxiterations) )

# Compute roots to equation ...

print("--- Execution:")
root, i, converged = Solutions.newtonraphson(f1, df1, x0, tolerance, maxiterations )

# Summary of computations ...

print("--- Output:")
print("---   root = {:16.8f} ...".format(root) )
print("---   f(root)   --> {:16.8e} ...".format(   f1(root)) )
```

```
    print("---   df(root)  --> {:16.8e} ...".format(  df1(root)) )
    print("---   no iterations = {:d} ...".format(i) )
    print("---   converged: {:s} ...".format( str(converged) ) )


    print("--- ======================================== ... ");
    print("--- Leave TestNewtonRaphson04.main()          ... ");

# call the main method ...

main()
```

Program Output: The abbreviated textual output for Part 1:

```
--- Part 1: Method of Bisection to solve f1(x) = 0 for upper root ...
--- ============================================================ ...

--- Inputs:
---   a =  3.00 ...
---   b =  4.00 ...
---   tolerance      =       0.00000001 ...
---   max iterations =     100.00000000 ...
--- Execution:
---   Initial Conditions:
---   f(a) --> -8.37037e+00 ...
---   f(b) -->  2.64844e+01 ...
---   Main Loop for Root Computation:
---   Iteration 000: dx = 5.00000e-01, x =  3.5000000e+00, f(x) -->  6.4231050e+00 ...
---   Iteration 001: dx = 2.50000e-01, x =  3.2500000e+00, f(x) --> -1.5856210e+00 ...
---   Iteration 002: dx = 1.25000e-01, x =  3.3750000e+00, f(x) -->  2.2599397e+00 ...
---   Iteration 003: dx = 6.25000e-02, x =  3.3125000e+00, f(x) -->  2.9818477e-01 ...

... lines of output removed ...

---   Iteration 028: dx = 1.86265e-09, x =  3.3027756e+00, f(x) -->  1.3632697e-08 ...
---   Iteration 029: dx = 9.31323e-10, x =  3.3027756e+00, f(x) --> -1.4834889e-08 ...
---   Iteration 030: dx = 4.65661e-10, x =  3.3027756e+00, f(x) --> -6.0109784e-10 ...
--- Output:
---   root =    3.30278 ...
---   f(root) --> -6.01098e-10 ...
---   no iterations = 30 ...
---   converged: True ...
---
--- Part 1: Method of Bisection to solve f1(x) = 0 for lower root ...
--- ============================================================ ...
--- Inputs:
---   a = -1.00 ...
---   b = -0.10 ...
---   tolerance      =       0.00000001 ...
---   max iterations =     100.00000000 ...
--- Execution:
---   Initial Conditions:
```

```
---   f(a) --> -3.00000e+01 ...
---   f(b) -->  9.50199e+02 ...
---   Main Loop for Root Computation:
---   Iteration 000: dx = 4.50000e-01, x = -5.5000000e-01, f(x) --> -2.6692220e+01 ...
---   Iteration 001: dx = 2.25000e-01, x = -3.2500000e-01, f(x) --> -6.4075416e+00 ...
---   Iteration 002: dx = 1.12500e-01, x = -2.1250000e-01, f(x) -->  6.5216951e+01 ...

... lines of output removed ...

---   Iteration 027: dx = 3.35276e-09, x = -3.0277564e-01, f(x) -->  8.4499833e-07 ...
---   Iteration 028: dx = 1.67638e-09, x = -3.0277564e-01, f(x) -->  2.8603744e-07 ...
---   Iteration 029: dx = 8.38190e-10, x = -3.0277564e-01, f(x) -->  6.5570092e-09 ...
--- Output:
---   root =   -0.30278 ...
---   f(root) -->  6.55701e-09 ...
---   no iterations = 29 ...
---   converged: True ...
```

For both problems, and with good starting conditions, the Bisection algorithm works, convergence to the nearby root is slow.

And for Part 2:

```
--- Part 2: Solve f1(x) = 0 for upper root: x0 = 10 ...
--- ============================================== ...

--- Inputs:
---   x0 = 10.00 ...
---   tolerance      =       0.00000001 ...
---   max iterations =     100.00000000 ...
--- Execution:
---   Initial Conditions:
---     x0     -->  1.00000e+01 ...
---     f(x0)  -->  9.50199e+02 ...
---     df(x0) -->  2.97980e+02 ...
---   Main Loop for Newton Raphson Iteration:
---   Iteration 001: dx = -3.18880e+00, x =  6.81120e+00, f(x) -->  2.72657e+02 ...
---   Iteration 002: dx = -1.98822e+00, x =  4.82298e+00, f(x) -->  7.29476e+01 ...
---   Iteration 003: dx = -1.07747e+00, x =  3.74551e+00, f(x) -->  1.55691e+01 ...
---   Iteration 004: dx = -3.89625e-01, x =  3.35589e+00, f(x) -->  1.65161e+00 ...
---   Iteration 005: dx = -5.22132e-02, x =  3.30367e+00, f(x) -->  2.74114e-02 ...
---   Iteration 006: dx = -8.96245e-04, x =  3.30278e+00, f(x) -->  7.99266e-06 ...
---   Iteration 007: dx = -2.61481e-07, x =  3.30278e+00, f(x) -->  6.85674e-13 ...
---   Iteration 008: dx = -2.24319e-14, x =  3.30278e+00, f(x) --> -1.06581e-14 ...
--- Output:
---   root =       3.30277564 ...
---   f(root)   -->  -1.06581410e-14 ...
---   df(root)  -->   3.05668464e+01 ...
---   no iterations = 8 ...
---   converged: True ...
---
--- Part 2: Solve f1(x) = 0 for lower root: x0 = -0.1 ...
```

```
--- ================================================ ...

--- Inputs:
---   x0 = -0.10 ...
---   tolerance      =  0.00000 ...
---   max iterations =   100.00 ...
--- Execution:
---   Initial Conditions:
---     x0     --> -1.00000e-01 ...
---     f(x0)  -->  9.50199e+02 ...
---     df(x0) -->  2.97980e+04 ...
---   Main Loop for Newton Raphson Iteration:
---   Iteration 001: dx = -3.18880e-02, x = -1.31888e-01, f(x) -->  3.90995e+02 ...
---   Iteration 002: dx = -3.99045e-02, x = -1.71793e-01, f(x) -->  1.55933e+02 ...
---   Iteration 003: dx = -4.62072e-02, x = -2.18000e-01, f(x) -->  5.77746e+01 ...
---   Iteration 004: dx = -4.49831e-02, x = -2.62983e-01, f(x) -->  1.78844e+01 ...
---   Iteration 005: dx = -2.99823e-02, x = -2.92965e-01, f(x) -->  3.50371e+00 ...
---   Iteration 006: dx = -9.16718e-03, x = -3.02132e-01, f(x) -->  2.15445e-01 ...
---   Iteration 007: dx = -6.40463e-04, x = -3.02773e-01, f(x) -->  9.44084e-04 ...
---   Iteration 008: dx = -2.83129e-06, x = -3.02776e-01, f(x) -->  1.83181e-08 ...
---   Iteration 009: dx = -5.49378e-11, x = -3.02776e-01, f(x) -->  0.00000e+00 ...
--- Output:
---   root =      -0.30277564 ...
---   f(root)   -->   0.00000000e+00 ...
---   df(root)  -->   3.33433154e+02 ...
---   no iterations = 9 ...
---   converged: True ...
```

Now let's look at the output for Part 3:

```
--- Part 3: Solve f1(x) = 0 for lower root: x0 = -2.0 ...
--- ================================================ ...

--- Inputs:
---   x0 = -2.00 ...
---   tolerance      =  0.00000 ...
---   max iterations =   100.00 ...
--- Execution:
---   Initial Conditions:
---     x0     --> -2.00000e+00 ...
---     f(x0)  --> -3.48750e+01 ...
---     df(x0) -->  9.68750e+00 ...
---   Main Loop for Newton Raphson Iteration:
---   Iteration 001: dx =  3.60000e+00, x =  1.60000e+00, f(x) --> -2.80981e+01 ...
---   Iteration 002: dx =  5.24560e+00, x =  6.84560e+00, f(x) -->  2.77398e+02 ...
---   Iteration 003: dx = -2.00222e+00, x =  4.84338e+00, f(x) -->  7.43353e+01 ...
---   Iteration 004: dx = -1.08844e+00, x =  3.75495e+00, f(x) -->  1.59471e+01 ...
---   Iteration 005: dx = -3.96971e-01, x =  3.35797e+00, f(x) -->  1.71774e+00 ...
---   Iteration 006: dx = -5.42316e-02, x =  3.30374e+00, f(x) -->  2.95841e-02 ...
---   Iteration 007: dx = -9.67239e-04, x =  3.30278e+00, f(x) -->  9.30919e-06 ...
---   Iteration 008: dx = -3.04552e-07, x =  3.30278e+00, f(x) -->  9.30811e-13 ...
---   Iteration 009: dx = -3.04517e-14, x =  3.30278e+00, f(x) --> -1.06581e-14 ...
--- Output:
```

```
---   root =       3.30277564 ...
---   f(root)   -->  -1.06581410e-14 ...
---   df(root)  -->   3.05668464e+01 ...
---   no iterations = 9 ...
---   converged: True ...
```

With a starting value x_0 = -2, the first iteration jumps to x_1 = 3.6, and then the algorithm converges to the wrong root !!!

Question 4: 10 points.

Problem Statement: For values $x > 0$, the nonlinear function

$$f(x) = 1 + \sin(x) \tag{3}$$

has roots at x = $3\pi/2$, $7\pi/2$, $11\pi/2$, and so forth.

Write a Python program to plot equation 3 over the range [0, 4π], and then demonstrate that while the **method of Newton Raphson** struggles to find value(s) of x for which f(x) = 0 (why?), the **method of Modified Newton Raphson** computes the same roots with ease.

Python Source Code:

```
# ================================================================================
# TestNewtonRaphson03.py: Use newton raphson/modified newton raphson algorithms
# to compute roots of equations:
#
#            f(x) = 1 + sin(x) = 0.0
#
# Written By: Mark Austin                                          October 2025
# ================================================================================

import math;
import Solutions;
import numpy as np
import matplotlib.pyplot as plt

# Functions: f(x) = 1 + sin(x) = 0.

def f1(x):
    return 1 + np.sin(x);

def df1(x):
    return np.cos(x);

def ddf1(x):
    return -np.sin(x);

# main method ...

def main():
    print("--- Enter TestNewtonRaphson03.main()          ... ");
    print("--- ======================================== ... ");

    print("--- ");
    print("--- Plot f1(x) over range [0,4pi] ... ");
    print("--- ======================================== ... ");
```

```
# Generate x and y coordinate arrays ....

xcoords = np.linspace(0, 4*math.pi, 61, endpoint=True);

ycoords = [];
for i in range( len(xcoords) ):
    ycoords.append( f1( xcoords[i] ) );

print(xcoords)
print(ycoords)

# Plot f(x) ...

plt.plot( xcoords, ycoords )
plt.title('f(x) = 1 + sin(x)');
plt.xlabel('x');
plt.ylabel('f(x)');
plt.grid(True)
plt.show()

print("--- ");
print("--- Case Study 1: Use Newton Raphson with initial guess: x0 = 6 ... ");
print("--- ============================================================ ... ");

# Initialize problem setup ...

x0 = 6.0;
tolerance     = 0.00000001
maxiterations = 100

print("--- Inputs:")
print("---   x0 = {:5.2f} ...".format(x0) )
print("---   tolerance      = {:16.8f} ...".format(tolerance) )
print("---   max iterations = {:16.8f} ...".format(maxiterations) )

# Compute roots to equation ...

print("--- Execution:")
root, i, converged = Solutions.newtonraphson(f1, df1, x0, tolerance, maxiterations )

# Summary of computations ...

print("--- Output:")
print("---   root = {:12.7f} ...".format(root) )
print("---   f(root)   --> {:16.8e} ...".format(   f1(root)) )
print("---   df(root)  --> {:16.8e} ...".format(  df1(root)) )
print("---   no iterations = {:d} ...".format(i) )
print("---   converged: {:s} ...".format( str(converged) ) )

print("--- ");
print("--- Case Study 2: Use Modified Newton Raphson with initial guess: x0 = 6 ... ");
print("--- ================================================================== ... ");

print("--- Inputs:")
print("---   x0 = {:5.2f} ...".format(x0) )
```

```
    print("---   tolerance      = {:16.8f} ...".format(tolerance) )
    print("---   max iterations = {:16.8f} ...".format(maxiterations) )

    # Compute roots to equation ...

    print("--- Execution:")
    root, i, converged = Solutions.modifiednewtonraphson(f1, df1, ddf1, x0, tolerance, maxiterations

    # Summary of computations ...

    print("--- Output:")
    print("---   root = {:12.7f} ...".format(root) )
    print("---   f(root)   --> {:16.8e} ...".format(   f1(root)) )
    print("---   df(root)  --> {:16.8e} ...".format(  df1(root)) )
    print("---   ddf(root) --> {:16.8e} ...".format( ddf1(root)) )
    print("---   no iterations = {:d} ...".format(i) )
    print("---   converged: {:s} ...".format( str(converged) ) )

    print("--- ======================================== ... ");
    print("--- Leave TestNewtonRaphson03.main()          ... ");

# call the main method ...

main()
```

Program Output: The graphical output is:

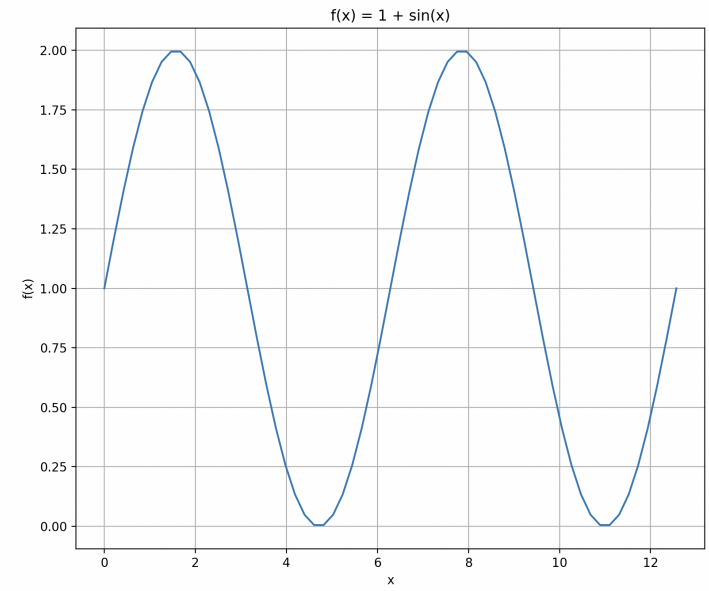

Figure 6: Plot y = 1 + sin(x) vs x. Roots at $3\pi/2$ and $7\pi/2$.

The abbreviated textual output is:

```
--- Case Study 1: Use Newton Raphson with initial guess: x0 = 6 ...
--- ========================================================== ...

--- Inputs:
---   x0 =  6.00 ...
---   tolerance      =       0.00000001 ...
---   max iterations =     100.00000000 ...
--- Execution:
---   Initial Conditions:
---     x0     -->  6.00000e+00 ...
---     f(x0)  -->  7.20585e-01 ...
---     df(x0) -->  9.60170e-01 ...
---   Main Loop for Newton Raphson Iteration:
---   Iteration 001: dx = -7.50476e-01, x =  5.24952e+00, f(x) -->  1.40822e-01 ...
---   Iteration 002: dx = -2.75217e-01, x =  4.97431e+00, f(x) -->  3.41050e-02 ...

... lines of output removed ...

---   Iteration 026: dx = -1.41960e-08, x =  4.71239e+00, f(x) -->  1.11022e-16 ...
---   Iteration 027: dx = -6.49759e-09, x =  4.71239e+00, f(x) -->  1.11022e-16 ...
--- Output:
---   root =    4.7123890 ...
---   f(root)   -->   1.11022302e-16 ...
---   df(root)  -->   1.05890903e-08 ...
---   no iterations = 27 ...
---   converged: True ...
---
--- Case Study 2: Use Modified Newton Raphson with initial guess: x0 = 6 ...
--- ================================================================== ...
--- Inputs:
---   x0 =  6.00 ...
---   tolerance      =       0.00000001 ...
---   max iterations =     100.00000000 ...
--- Execution:
---   Initial Conditions:
---     x0     -->  6.00000e+00 ...
---     f(x0)  -->  7.20585e-01 ...
---   Main Loop for Modified Newton Raphson Iteration:
---   Iteration 001: dx = -9.60170e-01, x =  5.03983e+00, f(x) -->  5.31314e-02 ...
---   Iteration 002: dx = -3.21621e-01, x =  4.71821e+00, f(x) -->  1.69358e-05 ...
---   Iteration 003: dx = -5.81991e-03, x =  4.71239e+00, f(x) -->  5.55112e-16 ...
---   Iteration 004: dx = -3.47827e-08, x =  4.71239e+00, f(x) -->  0.00000e+00 ...
---   Iteration 005: dx =  0.00000e+00, x =  4.71239e+00, f(x) -->  0.00000e+00 ...
--- Output:
---   root =    4.7123890 ...
---   f(root)   -->   0.00000000e+00 ...
---   df(root)  -->  -1.92750456e-09 ...
---   ddf(root) -->   1.00000000e+00 ...
---   no iterations = 5 ...
---   converged: True ...
```

Summary: From a numerical standpoint, Newton Raphson requires 27 iterations to satisfy the convergence criteria. Modified Newton Raphson converges in only 5 iterations.

Question 5: 10 points.

Problem Statement: Consider the family of matrix equations AX = B defined by

$$\begin{bmatrix} 1 & 2 & -3 \\ 3 & -1 & 5 \\ 4 & 1 & a^2 - 14 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ a + 2 \end{bmatrix}. \tag{4}$$

Determine the values of 'a' for which matrix A will be singular. Then,

1. Develop a program that uses NumPy to store matrices A and B, and then systematically evaluates the determinant and rank of A, and the rank of augmented matrix [A | B] for the values of 'a' that make A singular.

1. Develop a second program that uses SymPy to store matrices A and B symbolically, and then computes symbolic solution to the matrix equations 4. For the values of 'a' that make A singular, evaluate the determinant and rank of A, and the rank of augmented matrix [A | B].

You should find that the Python module SymPy is considerably more powerful than its numerical counterpart NumPy.

Hand Calculation: We begin with:

$$det(A) = 1 \cdot det \begin{bmatrix} -1 & 5 \\ 1 & a^2 - 14 \end{bmatrix} - 2 \cdot det \begin{bmatrix} 3 & 5 \\ 4 & a^2 - 14 \end{bmatrix} - 3 \cdot det \begin{bmatrix} 3 & -1 \\ 4 & 1 \end{bmatrix} = 112 - 7a^2 \tag{5}$$

Setting det(A) = $112 - 7a^2$ = 0 gives a = -4 or a = 4.

Program 1: Python Source Code: Implementation with NumPy.

```
# ============================================================
# TestMatrixOperationsNumPy01.py: Compute symbolic solutions to
# linear matrix equations ...
#
# ============================================================

import numpy as np
from numpy.linalg import matrix_rank

# ============================================================
# Function to print two-dimensional matrices ...
# ============================================================
```

```
def PrintMatrix(name, a):
    print("Matrix: {:s} ".format(name) );
    for row in a:
        for col in row:
            print("{:8.2f}".format(col), end=" ")
        print("")

# ============================================================
# main method ...
# ============================================================

def main():
    print("--- Case 1: Set a = 4 in matrices A and B ...");

    a = 4
    A = np.array( [ [ 1,  2, -3],
                    [ 3, -1,  5],
                    [ 4,  1, a*a - 14 ] ])
    B = np.array( [ [4], [2], [a+2] ])

    PrintMatrix("A", A)
    PrintMatrix("B", B)

    print("");
    print("--- determinant(A)  --> {:f} ...".format( np.linalg.det(A) ))
    print("--- rank [A]        --> {:f} ...".format( matrix_rank(A) ))

    print("");
    print("--- Create augmented matrix [ A | B ] ...\n");

    C = np.concatenate((A, B), axis=1)
    PrintMatrix("Matrix [A|B]", C)

    print("--- rank [A|B]      --> {:f} ...".format( matrix_rank(C) ))

    print("");
    print("--- Case 2: Set a = -4 in matrices A and B ...");

    a = -4
    A = np.array( [ [ 1,  2, -3],
                    [ 3, -1,  5],
                    [ 4,  1, a*a - 14 ] ])
    B = np.array( [ [4], [2], [a+2] ])

    PrintMatrix("A", A)
    PrintMatrix("B", B)

    print("");
    print("--- determinant(A)  --> {:f} ...".format( np.linalg.det(A) ))
    print("--- rank [A]        --> {:f} ...".format( matrix_rank(A) ))

    print("");
    print("--- Create augmented matrix [ A | B ] ...\n");
```

```
    C = np.concatenate((A, B), axis=1)
    PrintMatrix("Matrix [A|B]", C)

    print("");
    print("--- rank [A|B]      --> {:f} ...".format( matrix_rank(C) ))

# call the main method ...

main()
```

Program 1 Output:

```
--- Case 1: Set a = 4 in matrices A and B ...

Matrix: A
    1.00     2.00    -3.00
    3.00    -1.00     5.00
    4.00     1.00     2.00

Matrix: B
    4.00
    2.00
    6.00

--- determinant(A)  --> 0.000000 ...
--- rank [A]        --> 2.000000 ...

--- Create augmented matrix [ A | B ] ...

Matrix: Matrix [A|B]
    1.00     2.00    -3.00     4.00
    3.00    -1.00     5.00     2.00
    4.00     1.00     2.00     6.00

--- rank [A|B]      --> 2.000000 ...

--- Case 2: Set a = -4 in matrices A and B ...

Matrix: A
    1.00     2.00    -3.00
    3.00    -1.00     5.00
    4.00     1.00     2.00
Matrix: B
    4.00
    2.00
   -2.00

--- determinant(A)  --> 0.000000 ...
--- rank [A]        --> 2.000000 ...

--- Create augmented matrix [ A | B ] ...
```

```
Matrix: Matrix [A|B]
    1.00     2.00    -3.00     4.00
    3.00    -1.00     5.00     2.00
    4.00     1.00     2.00    -2.00

--- rank [A|B]      --> 3.000000 ...
```

Note: When a = 4 (Case 1), rank $[A]$ = rank $[A|B]$ = 2 – the equations overlap and there are an infinite number of solutions. Conversely, a = -4 (Case 2), rank $[A]$ = 2 and rank $[A|B]$ = 3 – the equations are inconsistent and there are zero solutions.

Program 2: Python Source Code: Implementation with SymPy.

```
# ============================================================
# TestMatrixOperationsSympy01.py: Compute symbolic solutions to
# linear matrix equations ...
#
# ============================================================

import sympy as sp
from sympy import Integral, Matrix, pi, pprint

# main method ...

def main():
    # Define symbolic representation of matrices ...

    print("--- Create matrices A and B ...");

    a = sp.symbols('a')
    A = sp.Matrix(( [ 1,  2, -3],
                    [ 3, -1,  5],
                    [ 4,  1, a*a - 14 ] ))
    B = sp.Matrix(( [4], [2], [a+2] ))

    pprint(A)
    pprint(B)

    print("--- Compute and print matrix determinant ...\n");

    print("--- A.det() --> {:s} ...".format( str(A.det() ) ))

    print("--- General symbolic solution to matrix system ...\n");

    solution = A.solve(B)
    print(solution)

    print("--- Expressions for individual solution elements ...\n");
    print("--- x1 = {:s} ...".format( str( solution[0] ) ))
    print("--- x2 = {:s} ...".format( str( solution[1] ) ))
    print("--- x2 = {:s} ...".format( str( solution[2] ) ))
```

```
    print("");
    print("--- Create augmented matrix [ A | B ] ...\n");

    C = A.row_join(B)
    pprint(C)

    print("--- Case 1: Set a = 4 in matrix [A|B] ...");

    A1 = A.subs( {a:4} )
    C1 = C.subs( {a:4} )
    pprint(C1)

    print("");
    print("--- A1.det()  --> {:s} ...".format( str( A1.det() ) ))
    print("--- A1.rank() --> {:s} ...".format( str( A1.rank() ) ))
    print("--- C1.rank() --> {:s} ...".format( str( C1.rank() ) ))

    print("--- Case 2: Set a = -4 in matrix [A|B] ...");

    A2 = A.subs( {a:-4} )
    C2 = C.subs( {a:-4} )
    pprint(C2)

    print("--- A2.det()  --> {:s} ...".format( str( A2.det() ) ))
    print("--- A2.rank() --> {:s} ...".format( str( A2.rank() ) ))
    print("--- C2.rank() --> {:s} ...".format( str( C2.rank() ) ))

# call the main method ...

main()
```

Program 2 Output:

```
--- Create matrices A and B ...

| 1  2     -3    |
|                |
| 3  -1     5    |
|                |
|        2       |
| 4  1   a  - 14 |

|   4   |
|       |
|   2   |
|       |
| a + 2 |

--- Compute and print matrix determinant ...

--- A.det() --> 112 - 7*a**2 ...

--- General symbolic solution to matrix system ...
```

```
Matrix([[(8*a + 25)/(7*a + 28)], [(10*a + 54)/(7*a + 28)], [1/(a + 4)]])

--- Expressions for individual solution elements ...

--- x1 = (8*a + 25)/(7*a + 28) ...
--- x2 = (10*a + 54)/(7*a + 28) ...
--- x2 = 1/(a + 4) ...

--- Create augmented matrix [ A | B ] ...

| 1  2     -3       4   |
|                       |
| 3  -1     5       2   |
|                       |
|        2              |
| 4  1   a  - 14  a + 2 |

--- Case 1: Set a = 4 in matrix [A|B] ...

| 1  2   -3  4 |
|              |
| 3  -1  5   2 |
|              |
| 4  1   2   6 |

--- A1.det()  --> 0 ...
--- A1.rank() --> 2 ...
--- C1.rank() --> 2 ...

--- Case 2: Set a = -4 in matrix [A|B] ...

| 1  2   -3  4 |
|              |
| 3  -1  5   2 |
|              |
| 4  1   2  -2 |

--- A2.det()  --> 0 ...
--- A2.rank() --> 2 ...
--- C2.rank() --> 3 ...
```

Note: Using SymPy leads to results consistent with NumPy.