PRACTICAL 1

What is the basic functionality of turtle in python in computer graphics?

Turtle in Python is a simple module for drawing graphics. It allows you to move a virtual "turtle" around the screen to create shapes and patterns. You can control its movement (forward, backward, left, right), draw lines, change colours, and even fill shapes. It's great for learning programming basics and creating simple graphics.

Functions of turtle in python

1. Movement Functions

- forward(distance): Moves the turtle forward by the specified distance.
- backward(distance): Moves the turtle backward by the specified distance.
- left(angle): Turns the turtle left by the specified angle.
- right(angle): Turns the turtle right by the specified angle.
- goto(x, y): Moves the turtle to the specified coordinates.

2. Drawing Control

- penup(): Lifts the pen, so the turtle moves without drawing.
- pendown(): Lowers the pen, so the turtle draws as it moves.
- pensize(width): Sets the width of the pen.
- pencolor(color): Sets the color of the pen.
- begin_fill(): Starts filling the shape the turtle will draw.
- end_fill(): Fills the shape drawn after begin_fill().

3. Shape and Appearance

- shape(name): Sets the shape of the turtle (e.g., "turtle", "arrow").
- stamp(): Leaves an imprint of the turtle's shape at its current position.
- speed(speed): Sets the speed of the turtle's movement (range from 1 to 10, or "fastest").

4. Screen and Display Control

- bgcolor(color): Changes the background color of the drawing screen.
- clear(): Clears the turtle's drawings from the screen.

- reset(): Resets the turtle's position and clears the screen.
- hideturtle(): Makes the turtle invisible.
- showturtle(): Makes the turtle visible again.

5. Event Handling

- onclick(function): Calls a function when the turtle screen is clicked.
- onkey(function, key): Calls a function when a specific key is pressed.

PRACTICAL 2

Aim: Drawing of Shapes like Pixel, Line, Rectangle, Circle, Arc, Sector, Concentric Circles and Hut

I. Code (for drawing Pixel, Line, Rectangle, Circle, Arc and Sector):

```
import turtle
import random
screen = turtle.Screen()
screen.title("Pixel, Line, Rectangle, Circle, Arc, and Sector Plotting with Turtle")
screen.bgcolor("white")
pen = turtle.Turtle()
pen.hideturtle()
pen.speed(0)
def plot_pixel(x, y, color="black"):
    pen.penup()
    pen.goto(x, y)
    pen.dot(5, color)
    pen.penup()
def draw_line(x1, y1, x2, y2, color="black"):
    pen.penup()
    pen.goto(x1, y1)
    pen.pendown()
    pen.pencolor(color)
    pen.pensize(3)
    pen.goto(x2, y2)
    pen.penup()
def draw_rectangle(x, y, width, height, color="black"):
    pen.penup()
    pen.goto(x, y)
    pen.pendown()
```

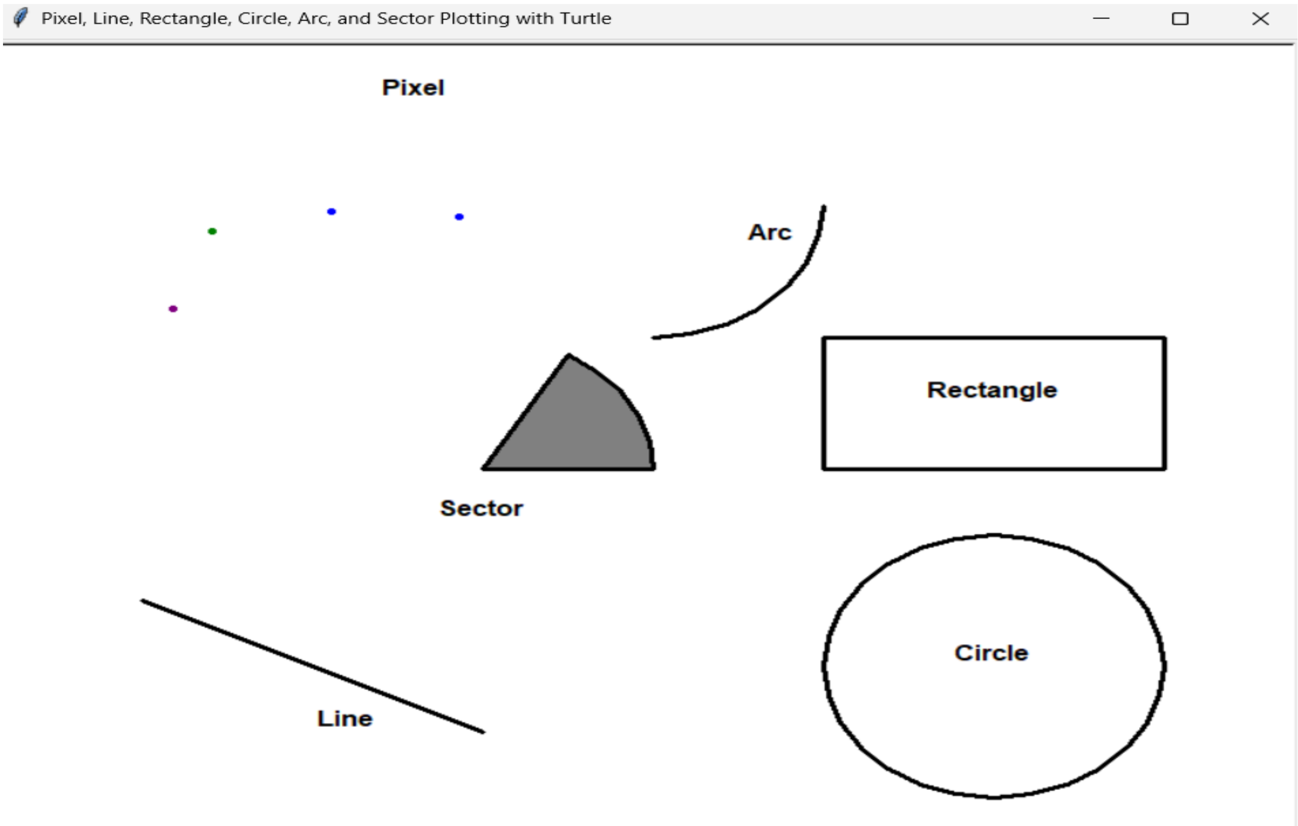
```
    pen.pencolor(color)
    pen.pensize(3)
    for _ in range(2):
        pen.forward(width)
        pen.right(90)
        pen.forward(height)
        pen.right(90)
    pen.penup()
def draw_circle(x, y, radius, color="black"):
    pen.penup()
    pen.goto(x, y - radius)
    pen.pendown()
    pen.pencolor(color)
    pen.pensize(3)
    pen.circle(radius)
    pen.penup()
def draw_arc(x, y, radius, extent, color="black"):
    pen.penup()
    pen.goto(x, y - radius)
    pen.setheading(0)
    pen.pendown()
    pen.pencolor(color)
    pen.pensize(3)
    pen.circle(radius, extent)
    pen.penup()
def draw_sector(x, y, radius, angle, color="black"):
    pen.penup()
    pen.goto(x, y)
    pen.setheading(0)
```

```
    pen.pendown()
    pen.fillcolor(color)
    pen.begin_fill()
    pen.forward(radius)
    pen.left(90)
    pen.circle(radius, angle)
    pen.left(90)
    pen.forward(radius)
    pen.end_fill()
    pen.penup()
def label_shape(x, y, text, offset_x=0, offset_y=0):
    pen.penup()
    pen.goto(x + offset_x, y + offset_y)
    pen.write(text, align="center", font=("Arial", 12, "bold"))
    pen.penup()
for _ in range(4):
    x = random.randint(-300, -100)
    y = random.randint(100, 300)
    plot_pixel(x, y, random.choice(["blue", "red", "green", "purple", "orange"]))
label_shape(-150, 300, "Pixel", offset_x=10, offset_y=-20)
draw_line(-300, -100, -100, -200, "black")
label_shape(-200, -180, "Line", offset_x=20, offset_y=-20)
draw_rectangle(100, 100, 200, 100, "black")
label_shape(200, 50, "Rectangle")
draw_circle(200, -150, 100, "black")
label_shape(200, -150, "Circle")
draw_arc(0, 200, 100, 90, "black")
label_shape(70, 170, "Arc")
draw_sector(-100, 0, 100, 60, "grey")
```

label_shape(-100, -40, "Sector")

screen.exitonclick()

Output:



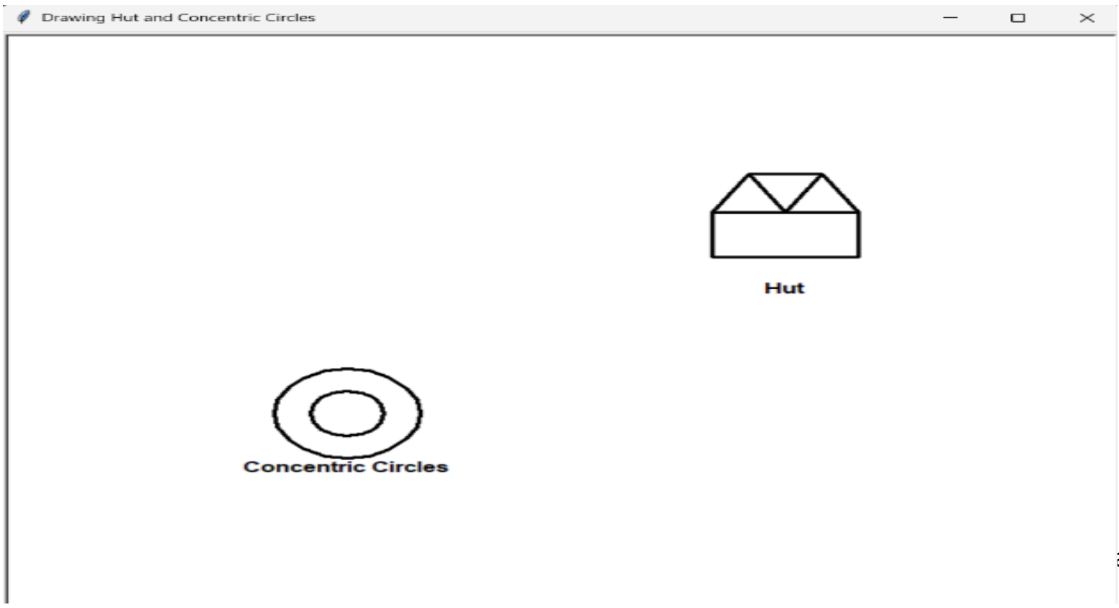
II. Code (for drawing Concentric Circles and Hut):

```
import turtle
screen = turtle.Screen()
screen.title("Drawing Hut and Concentric Circles")
screen.bgcolor("white")
pen = turtle.Turtle()
pen.speed(5)
pen.hideturtle()
```

```
def draw_concentric_circles(x, y, radii, color="black"):
    for radius in radii:
        pen.penup()
        pen.goto(x, y - radius)
        pen.pendown()
        pen.pencolor(color)
        pen.pensize(3)
        pen.circle(radius)
        pen.penup()
def draw_hut(x, y, length, width):
    pen.penup()
    pen.goto(x - length / 2, y + width / 2)
    pen.pendown()
    pen.pensize(3)
    pen.pencolor("black")
    for _ in range(2):
        pen.forward(length)
        pen.right(90)
        pen.forward(width)
        pen.right(90)
    pen.goto(x - length / 2, y + width / 2)
    pen.forward(length / 2)
    pen.right(240)
    pen.forward(width)
    pen.right(240)
    pen.forward(length / 2)
    pen.penup()
    pen.goto(x + length / 2, y + width / 2)
    pen.pendown()
```

```
    pen.left(240)
    pen.forward(length / 2)
    pen.left(120)
    pen.forward(width)
    pen.right(120)
    pen.forward(width)
    pen.right(120)
    pen.forward(length / 2)
    pen.penup()
def label_shape(x, y, text, offset_x=0, offset_y=0):
    pen.penup()
    pen.goto(x + offset_x, y + offset_y)
    pen.write(text, align="center", font=("Arial", 12, "bold"))
    pen.penup()
draw_concentric_circles(-150, -100, [50, 25], "black")
label_shape(-150, -160, "Concentric Circles", offset_x=0, offset_y=-10)
draw_hut(150, 100, 100, 50)
label_shape(150, 50, "Hut", offset_x=0, offset_y=-20)
screen.exitonclick()
```

Output:



PRACTICAL 3

Aim: Digital Differential Analyzer (DDA) Line Drawing Algorithm

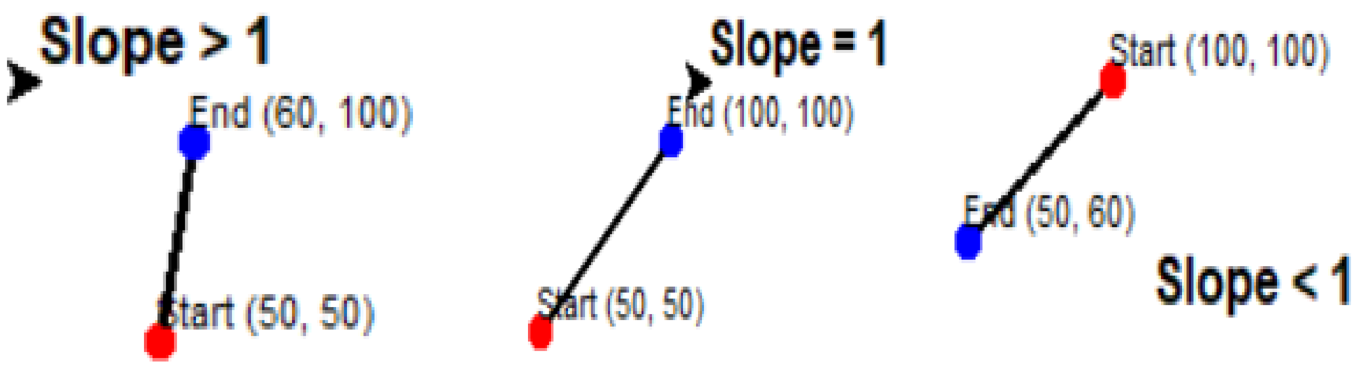
Code:

```
import turtle
import time
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
def drawLineDDA(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    if dx == 0:
        slope = float('inf')
    elif dy == 0:
        slope = 0
    else:
        slope = dy / dx
    steps = abs(dx) if abs(dx) > abs(dy) else abs(dy)
    Xinc = dx / steps
    Yinc = dy / steps
    x = x1
    y = y1
    points = []
    for _ in range(steps + 1):
        turtle.goto(round(x), round(y))
        turtle.dot(3, "black")
        points.append(Point(round(x), round(y)))
```

```python
        x += Xinc
        y += Yinc
    turtle.penup()
    turtle.goto(x1, y1)
    turtle.dot(10, "red")
    turtle.write(f"Start ({x1}, {y1})", font=("Arial", 8, "normal"))
    turtle.goto(x2, y2)
    turtle.dot(10, "blue")
    turtle.write(f"End ({x2}, {y2})", font=("Arial", 8, "normal"))
    mid_x = (x1 + x2) / 2
    mid_y = (y1 + y2) / 2
    offset_x = 40
    offset_y = 40
    if slope > 1:
        turtle.goto(mid_x - offset_x, mid_y + offset_y)
        turtle.write("Slope > 1", font=("Arial", 12, "bold"))
    elif slope == 1:
        turtle.goto(mid_x + offset_x, mid_y + offset_y)
        turtle.write("Slope = 1", font=("Arial", 12, "bold"))
    elif slope == 0:
        turtle.goto(mid_x + offset_x, mid_y + offset_y)
        turtle.write("Slope = 0", font=("Arial", 12, "bold"))
    elif slope < 0:
        turtle.goto(mid_x + offset_x, mid_y - offset_y)
        turtle.write("Slope < 0", font=("Arial", 12, "bold"))
    else:
        turtle.goto(mid_x + offset_x, mid_y - offset_y)
        turtle.write("Slope < 1", font=("Arial", 12, "bold"))
def main():
```

```
    turtle.speed(0)
    turtle.penup()
    x1, y1 = 50, 50
    x2, y2 = 60, 100
    drawLineDDA(x1, y1, x2, y2)
    time.sleep(2)
    turtle.clear()
    x1, y1 = 50, 50
    x2, y2 = 100, 100
    drawLineDDA(x1, y1, x2, y2)
    time.sleep(2)
    turtle.clear()
    x1, y1 = 100, 100
    x2, y2 = 50, 60
    drawLineDDA(x1, y1, x2, y2)
    turtle.hideturtle()
    turtle.done()
if __name__ == "__main__":
    main()
```

Output:



PRACTICAL 4

Aim: Bresenham's Line Drawing Algorithm

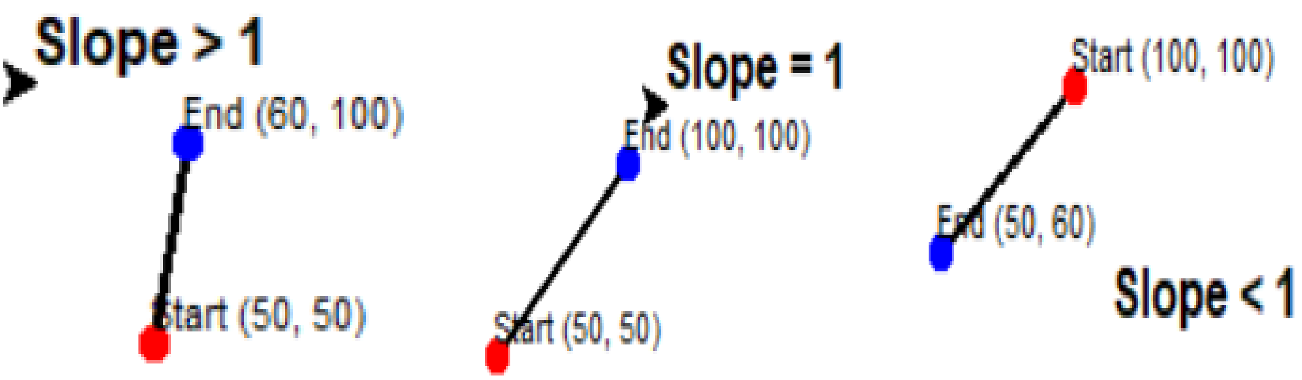
Code:

```
import turtle
import time
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
def drawLineBresenham(x1, y1, x2, y2):
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)
    sx = 1 if x2 > x1 else -1
    sy = 1 if y2 > y1 else -1
    err = dx - dy
    slope = dy / dx if dx != 0 else float('inf')
    points = []
    x, y = x1, y1
    while True:
        turtle.goto(x, y)
        turtle.dot(3, "black")
        points.append(Point(x, y))
        if x == x2 and y == y2:
            break
        e2 = 2 * err
        if e2 > -dy:
            err -= dy
            x += sx
```

```
    if e2 < dx:
        err += dx
        y += sy
turtle.penup()
turtle.goto(x1, y1)
turtle.dot(10, "red")
turtle.write(f"Start ({x1}, {y1})", font=("Arial", 8, "normal"))
turtle.goto(x2, y2)
turtle.dot(10, "blue")
turtle.write(f"End ({x2}, {y2})", font=("Arial", 8, "normal"))
mid_x = (x1 + x2) / 2
mid_y = (y1 + y2) / 2
offset_x = 40
offset_y = 40
if slope > 1:
    turtle.goto(mid_x - offset_x, mid_y + offset_y)
    turtle.write("Slope > 1", font=("Arial", 12, "bold"))
elif slope == 1:
    turtle.goto(mid_x + offset_x, mid_y + offset_y)
    turtle.write("Slope = 1", font=("Arial", 12, "bold"))
elif slope == 0:
    turtle.goto(mid_x + offset_x, mid_y + offset_y)
    turtle.write("Slope = 0", font=("Arial", 12, "bold"))
elif slope < 0:
    turtle.goto(mid_x + offset_x, mid_y - offset_y)
    turtle.write("Slope < 0", font=("Arial", 12, "bold"))
else:
    turtle.goto(mid_x + offset_x, mid_y - offset_y)
    turtle.write("Slope < 1", font=("Arial", 12, "bold"))
```

```python
def main():
    turtle.speed(0)
    turtle.penup()
    x1, y1 = 50, 50
    x2, y2 = 60, 100
    drawLineBresenham(x1, y1, x2, y2)
    time.sleep(2)
    turtle.clear()
    x1, y1 = 50, 50
    x2, y2 = 100, 100
    drawLineBresenham(x1, y1, x2, y2)
    time.sleep(2)
    turtle.clear()
    x1, y1 = 100, 100
    x2, y2 = 50, 60
    drawLineBresenham(x1, y1, x2, y2)
    turtle.hideturtle()
    turtle.done()
if __name__ == "__main__":
    main()
```

Output:



PRACTICAL 5

Aim: Mid-Point Circle Line Drawing Algorithm

Code:

```
import turtle
def draw_point(x, y):
    turtle.penup()
    turtle.goto(x, y)
    turtle.pendown()
    turtle.dot(3, "blue")
def draw_circle(xc, yc, r):
    x = 0
    y = r
    d = 1 - r
    turtle.speed(0)
    turtle.bgcolor("white")
    turtle.color("blue")
    turtle.hideturtle()
    turtle.tracer(0, 0)
    draw_point(xc + x, yc + y)
    draw_point(xc - x, yc + y)
    draw_point(xc + x, yc - y)
    draw_point(xc - x, yc - y)
    draw_point(xc + y, yc + x)
    draw_point(xc - y, yc + x)
    draw_point(xc + y, yc - x)
    draw_point(xc - y, yc - x)
    print(f"Point: ({xc + x}, {yc + y})")
    print(f"Point: ({xc - x}, {yc + y})")
```

```
print(f"Point: ({xc + x}, {yc - y})")
print(f"Point: ({xc - x}, {yc - y})")
print(f"Point: ({xc + y}, {yc + x})")
print(f"Point: ({xc - y}, {yc + x})")
print(f"Point: ({xc + y}, {yc - x})")
print(f"Point: ({xc - y}, {yc - x})")
while x < y:
    if d < 0:
        d = d + 2 * x + 3
    else:
        d = d + 2 * (x - y) + 5
        y -= 1
    x += 1
    draw_point(xc + x, yc + y)
    draw_point(xc - x, yc + y)
    draw_point(xc + x, yc - y)
    draw_point(xc - x, yc - y)
    draw_point(xc + y, yc + x)
    draw_point(xc - y, yc + x)
    draw_point(xc + y, yc - x)
    draw_point(xc - y, yc - x)
    print(f"Point: ({xc + x}, {yc + y})")
    print(f"Point: ({xc - x}, {yc + y})")
    print(f"Point: ({xc + x}, {yc - y})")
    print(f"Point: ({xc - x}, {yc - y})")
    print(f"Point: ({xc + y}, {yc + x})")
    print(f"Point: ({xc - y}, {yc + x})")
    print(f"Point: ({xc + y}, {yc - x})")
    print(f"Point: ({xc - y}, {yc - x})")
```

```
    turtle.update()
def main():
    xc, yc = 0, 0
    r = 10
    draw_circle(xc, yc, r)
    turtle.done()
if __name__ == "__main__":
    main()
```

Output:

Point: (0, 10)

Point: (0, 10)

Point: (0, -10)

Point: (0, -10)

Point: (10, 0)

Point: (-10, 0)

Point: (10, 0)

Point: (-10, 0)

Point: (1, 10)

Point: (-1, 10)

Point: (1, -10)

Point: (-1, -10)

Point: (10, 1)

Point: (-10, 1)

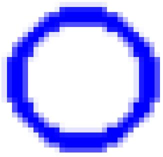
Point: (10, -1)

Point: (-10, -1)

Point: (2, 10)

Point: (-2, 10)

Point: (2, -10)



Point: (-2, -10)

Point: (10, 2)

Point: (-10, 2)

Point: (10, -2)

Point: (-10, -2)

Point: (3, 10)

Point: (-3, 10)

Point: (3, -10)

Point: (-3, -10)

Point: (10, 3)

Point: (-10, 3)

Point: (10, -3)

Point: (-10, -3)

Point: (4, 9)

Point: (-4, 9)

Point: (4, -9)

Point: (-4, -9)

Point: (9, 4)

Point: (-9, 4)

Point: (9, -4)

Point: (-9, -4)

Point: (5, 9)

Point: (-5, 9)

Point: (5, -9)

Point: (-5, -9)

Point: (9, 5)

Point: (-9, 5)

Point: (9, -5)

Point: (-9, -5)

Point: (6, 8)

Point: (-6, 8)

Point: (6, -8)

Point: (-6, -8)

Point: (8, 6)

Point: (-8, 6)

Point: (8, -6)

Point: (-8, -6)

Point: (7, 7)

Point: (-7, 7)

Point: (7, -7)

Point: (-7, -7)

Point: (7, 7)

Point: (-7, 7)

Point: (7, -7)

Point: (-7, -7)

PRACTICAL 6

Aim: Transformation (Rotation and Translation)

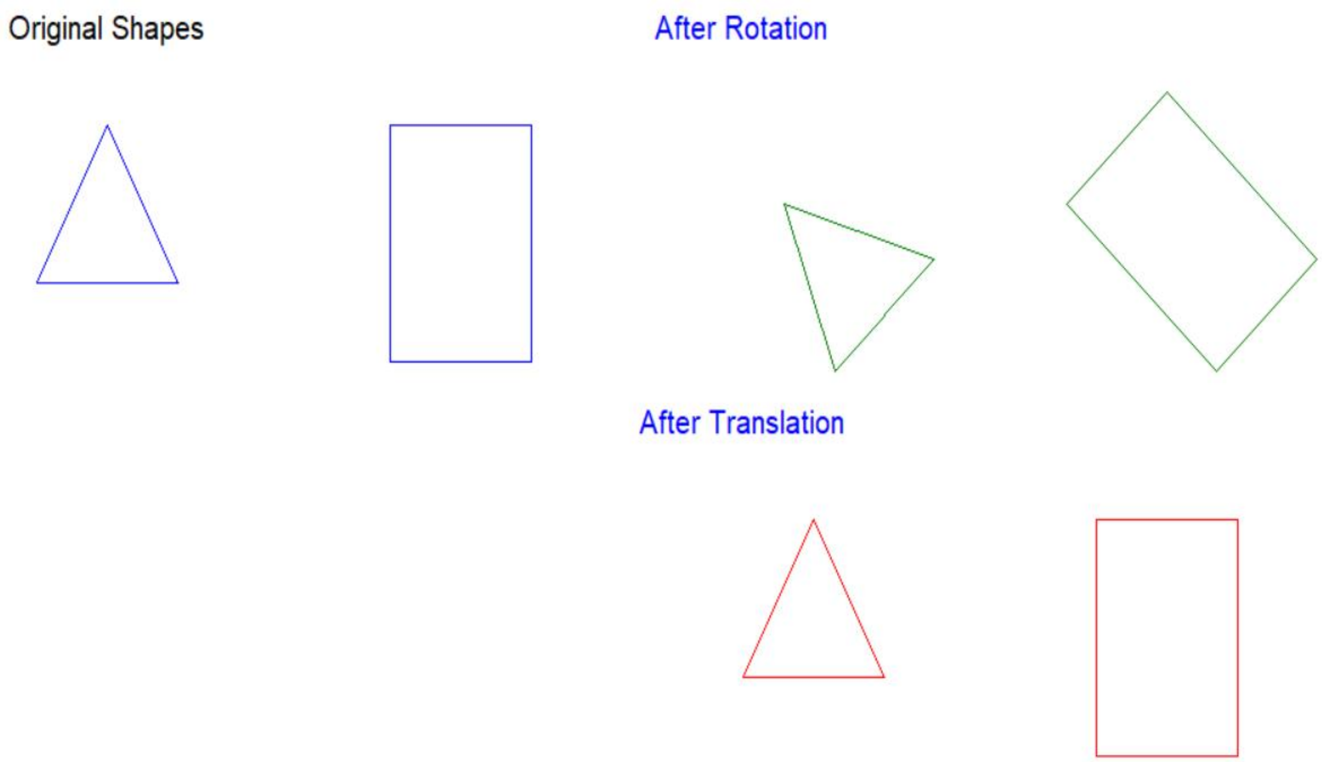
Code:

```
import turtle
import math
WIDTH, HEIGHT = 1000, 700
screen = turtle.Screen()
screen.setup(WIDTH, HEIGHT)
screen.bgcolor("white")
screen.title("Shapes Drawing with Turtle")
def draw_polygon(points, color):
    turtle.penup()
    turtle.goto(points[0][0], points[0][1])
    turtle.pendown()
    turtle.pencolor(color)
    for point in points[1:]:
        turtle.goto(point[0], point[1])
    turtle.goto(points[0][0], points[0][1])
def translate_point(x, y, tx, ty):
    return x + tx, y + ty
def rotate_point(x, y, cx, cy, angle):
    rad = math.radians(angle)
    s = math.sin(rad)
    c = math.cos(rad)
    x -= cx
    y -= cy
    xnew = x * c - y * s
    ynew = x * s + y * c
```

```
    x = xnew + cx
    y = ynew + cy
    return x, y
def print_text(text, position, font_size=15, color="black"):
    turtle.penup()
    turtle.goto(position)
    turtle.pendown()
    turtle.pencolor(color)
    turtle.write(text, align="center", font=("Arial", font_size, "normal"))
def draw_shapes():
    turtle.speed(0)
    turtle.hideturtle()
    start_x = -400
    y_positions = [250, -50]
    print_text("Original Shapes", (start_x, y_positions[0]))
    triangle = [(-400, 200), (-350, 100), (-450, 100)]
    rectangle = [(-200, 200), (-100, 200), (-100, 50), (-200, 50)]
    draw_polygon(triangle, "blue")
    draw_polygon(rectangle, "blue")
    print_text("After Rotation", (start_x + 450, y_positions[0]), color="blue")
    rotated_triangle = [rotate_point(x, y, -350, 150, 45) for x, y in triangle]
    rotated_rectangle = [rotate_point(x, y, -150, 150, 45) for x, y in rectangle]
    rotated_triangle = [translate_point(x, y, 500, 0) for x, y in rotated_triangle]
    rotated_rectangle = [translate_point(x, y, 500, 0) for x, y in rotated_rectangle]
    draw_polygon(rotated_triangle, "green")
    draw_polygon(rotated_rectangle, "green")
    print_text("After Translation", (start_x + 450, y_positions[1] + 50), color="blue")  #
Moved further up
```

```
    translated_triangle = [translate_point(x, y, 500, -250) for x, y in triangle]  # Adjusted translation
    translated_rectangle = [translate_point(x, y, 500, -250) for x, y in rectangle]  # Adjusted translation
    draw_polygon(translated_triangle, "red")
    draw_polygon(translated_rectangle, "red")
    turtle.hideturtle()
draw_shapes()
turtle.done()
```

Output:



PRACTICAL 7

Aim: Transformation (Reflection)

Code:

```
import turtle
def write_label(label, position):
    turtle.penup()
    turtle.goto(position[0], position[1] + 50)
    turtle.pendown()
    turtle.write(label, align="center", font=("Arial", 14, "bold"))
    turtle.penup()
def draw_triangle(vertices, position_offset=(0, 0)):
    turtle.penup()
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
    turtle.pendown()
    for vertex in vertices[1:]:
        turtle.goto(vertex[0] + position_offset[0], vertex[1] + position_offset[1])
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
def draw_rectangle(vertices, position_offset=(0, 0)):
    turtle.penup()
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
    turtle.pendown()
    for vertex in vertices[1:]:
        turtle.goto(vertex[0] + position_offset[0], vertex[1] + position_offset[1])
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
def reflect_shape(vertices, axis):
    if axis == 'x':
        return [(x, -y) for x, y in vertices]
```

```
    elif axis == 'y':
        return [(-x, y) for x, y in vertices]
    else:
        raise ValueError("Axis must be 'x' or 'y'")
turtle.speed(2)
turtle.bgcolor("white")
triangle_vertices = [(-100, 0), (0, 100), (100, 0)]
rectangle_vertices = [(-50, -50), (-50, 50), (50, 50), (50, -50)]
gap_x = 300
gap_y = 200
reduced_gap_x = 150
turtle.color("black")
write_label("Original Triangle", (-400, 100))
draw_triangle(triangle_vertices, position_offset=(-400, 50))
write_label("Original Rectangle", (-400, -50))
draw_rectangle(rectangle_vertices, position_offset=(-400, -100))
reflected_triangle_x = reflect_shape(triangle_vertices, 'x')
reflected_rectangle_x = reflect_shape(rectangle_vertices, 'x')
turtle.color("magenta")
write_label("Reflection About X-Axis (Triangle)", (-400 + gap_x, 100))
draw_triangle(reflected_triangle_x, position_offset=(-400 + gap_x, 130))
turtle.color("cyan")
write_label("Reflection About X-Axis (Rectangle)", (-400 + gap_x, -50))
draw_rectangle(reflected_rectangle_x, position_offset=(-400 + gap_x, -100))
reflected_triangle_y = reflect_shape(triangle_vertices, 'y')
reflected_rectangle_y = reflect_shape(rectangle_vertices, 'y')
turtle.color("orange")
write_label("Reflection About Y-Axis (Triangle)", (-200 + gap_x + reduced_gap_x,
100))
```

draw_triangle(reflected_triangle_y, position_offset=(-200 + gap_x + reduced_gap_x, 50))

turtle.color("brown")

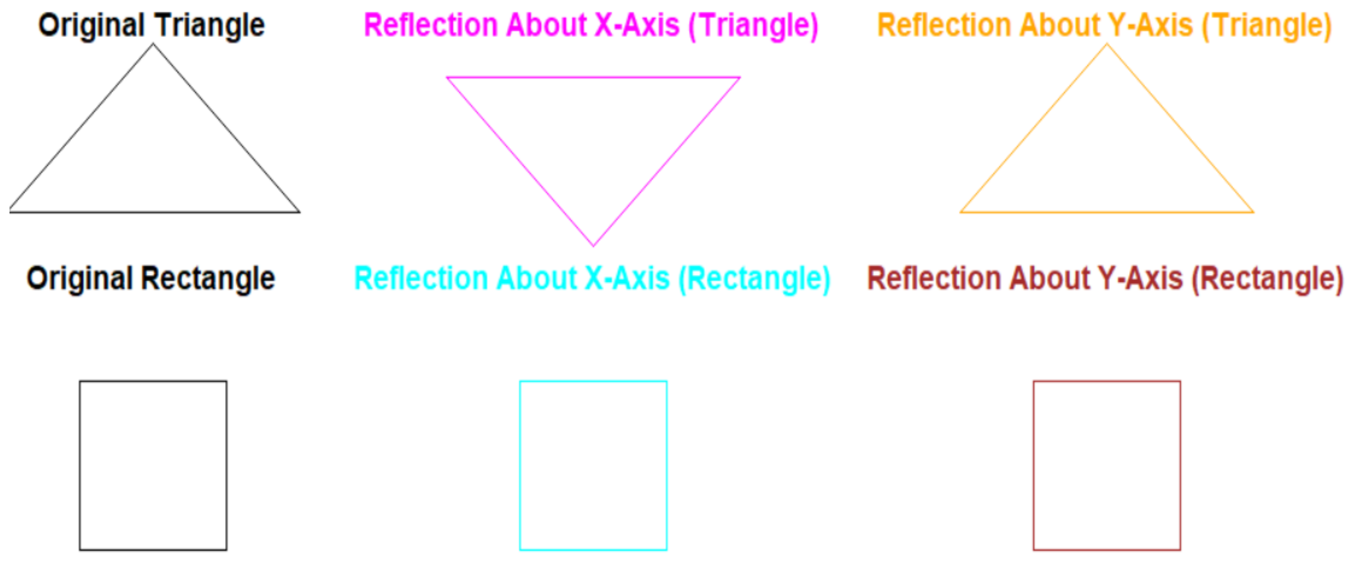
write_label("Reflection About Y-Axis (Rectangle)", (-200 + gap_x + reduced_gap_x, -50))

draw_rectangle(reflected_rectangle_y, position_offset=(-200 + gap_x + reduced_gap_x, -100))

turtle.hideturtle()

turtle.done()

Output:



PRACTICAL 8

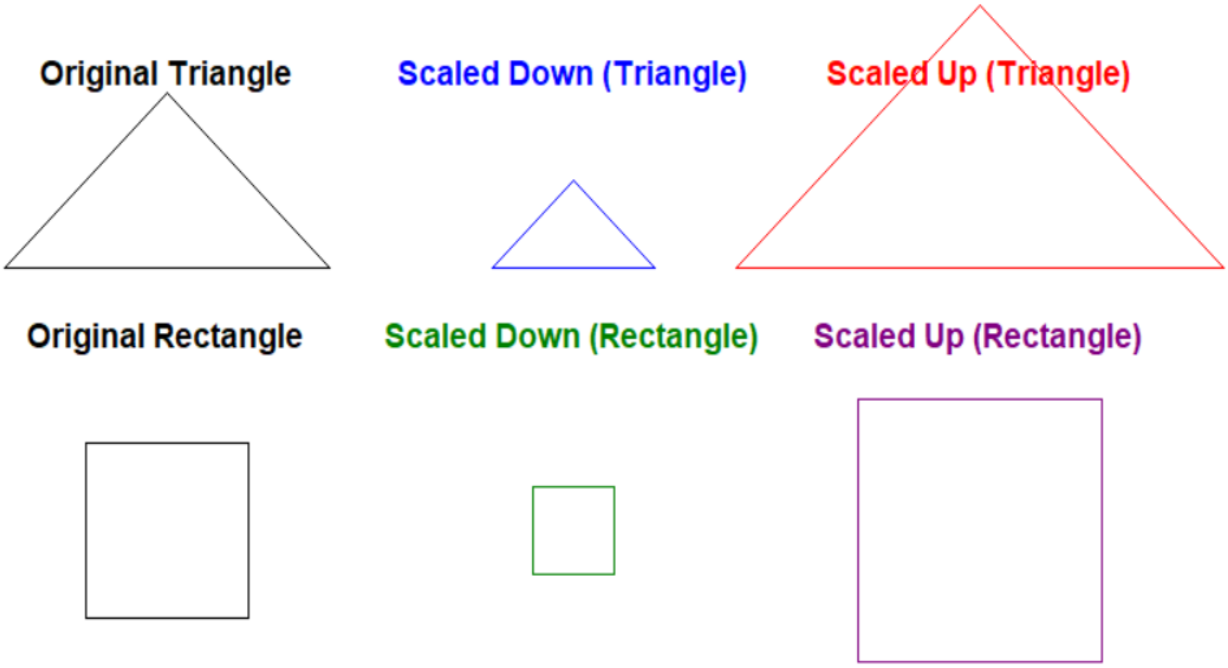
Aim: Transformation (Scaling)

Code:

```
import turtle
def write_label(label, position):
    turtle.penup()
    turtle.goto(position[0], position[1] + 50)
    turtle.pendown()
    turtle.write(label, align="center", font=("Arial", 14, "bold"))
    turtle.penup()
def draw_triangle(vertices, position_offset=(0, 0)):
    turtle.penup()
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
    turtle.pendown()
    for vertex in vertices[1:]:
        turtle.goto(vertex[0] + position_offset[0], vertex[1] + position_offset[1])
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
def draw_rectangle(vertices, position_offset=(0, 0)):
    turtle.penup()
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
    turtle.pendown()
    for vertex in vertices[1:]:
        turtle.goto(vertex[0] + position_offset[0], vertex[1] + position_offset[1])
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
def scale_shape(vertices, Sx, Sy):
    return [(x * Sx, y * Sy) for x, y in vertices]
turtle.speed(2)
```

```
turtle.bgcolor("white")
triangle_vertices = [(-100, 0), (0, 100), (100, 0)]
rectangle_vertices = [(-50, -50), (-50, 50), (50, 50), (50, -50)]
gap_x = 250
gap_y = 200
turtle.color("black")
write_label("Original Triangle", (-400, 100))
draw_triangle(triangle_vertices, position_offset=(-400, 50))
write_label("Original Rectangle", (-400, -50))
draw_rectangle(rectangle_vertices, position_offset=(-400, -100))
scaled_triangle = scale_shape(triangle_vertices, 0.5, 0.5)
scaled_rectangle = scale_shape(rectangle_vertices, 0.5, 0.5)
turtle.color("blue")
write_label("Scaled Down (Triangle)", (-400 + gap_x, 100))
draw_triangle(scaled_triangle, position_offset=(-400 + gap_x, 50))
turtle.color("green")
write_label("Scaled Down (Rectangle)", (-400 + gap_x, -50))
draw_rectangle(scaled_rectangle, position_offset=(-400 + gap_x, -100))
scaled_triangle = scale_shape(triangle_vertices, 1.5, 1.5)
scaled_rectangle = scale_shape(rectangle_vertices, 1.5, 1.5)
turtle.color("red")
write_label("Scaled Up (Triangle)", (-400 + 2 * gap_x, 100))
draw_triangle(scaled_triangle, position_offset=(-400 + 2 * gap_x, 50))
turtle.color("purple")
write_label("Scaled Up (Rectangle)", (-400 + 2 * gap_x, -50))
draw_rectangle(scaled_rectangle, position_offset=(-400 + 2 * gap_x, -100))
turtle.hideturtle()
turtle.done()
```

Output:


Original Triangle
Scaled Down (Triangle)
Scaled Up (Triangle)
Original Rectangle
Scaled Down (Rectangle)
Scaled Up (Rectangle)

PRACTICAL 9

Aim: Transformation (Shearing)

Code:

```
import turtle
def write_label(label, position):
    turtle.penup()
    turtle.goto(position[0], position[1] + 50)
    turtle.pendown()
    turtle.write(label, align="center", font=("Arial", 14, "bold"))
    turtle.penup()
def draw_triangle(vertices, position_offset=(0, 0)):
    turtle.penup()
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
    turtle.pendown()
    for vertex in vertices[1:]:
        turtle.goto(vertex[0] + position_offset[0], vertex[1] + position_offset[1])
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
def draw_rectangle(vertices, position_offset=(0, 0)):
    turtle.penup()
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
    turtle.pendown()
    for vertex in vertices[1:]:
        turtle.goto(vertex[0] + position_offset[0], vertex[1] + position_offset[1])
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
def shear_shape(vertices, shear_factor_x, shear_factor_y):
    return [(x + shear_factor_x * y, y + shear_factor_y * x) for x, y in vertices]
turtle.speed(2)
```

```
turtle.bgcolor("white")
triangle_vertices = [(-100, 0), (0, 100), (100, 0)]
rectangle_vertices = [(-50, -50), (-50, 50), (50, 50), (50, -50)]
shear_factor_x = 0.5
shear_factor_y = 0.5
sheared_triangle = shear_shape(triangle_vertices, shear_factor_x, shear_factor_y)
sheared_rectangle = shear_shape(rectangle_vertices, shear_factor_x, shear_factor_y)
turtle.color("blue")
write_label("Original Triangle", (-400, 100))
draw_triangle(triangle_vertices, position_offset=(-400, 50))
turtle.color("green")
write_label("Original Rectangle", (-400, -50))
draw_rectangle(rectangle_vertices, position_offset=(-400, -100))
turtle.color("orange")
write_label("Sheared Triangle", (200, 100))
draw_triangle(sheared_triangle, position_offset=(200, 50))
turtle.color("brown")
write_label("Sheared Rectangle", (200, -70))
draw_rectangle(sheared_rectangle, position_offset=(200, -120))
turtle.hideturtle()
turtle.done()
```

Output:

PRACTICAL 10

Aim: Composite Transformation (Rotation and Translation)

Code:

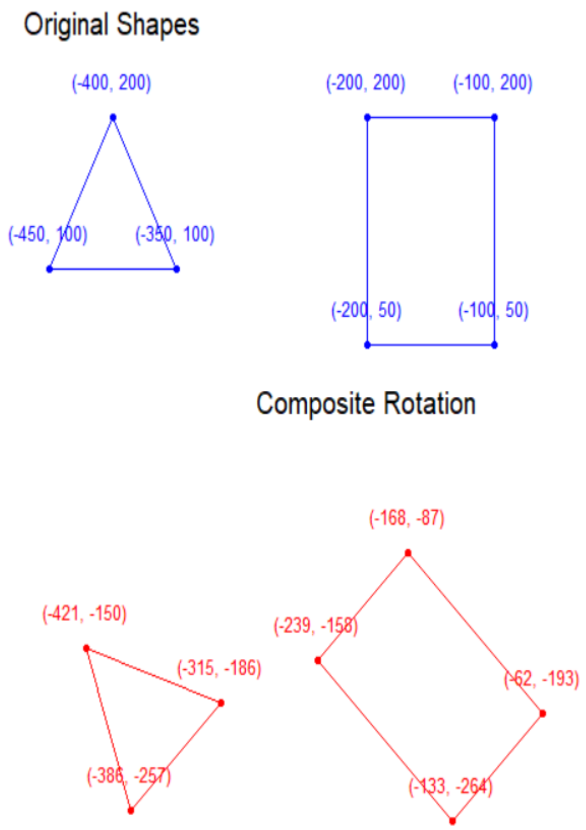
```
import turtle
import math
WIDTH, HEIGHT = 1000, 700
screen = turtle.Screen()
screen.setup(WIDTH, HEIGHT)
screen.bgcolor("white")
screen.title("Composite Translation and Rotation with Turtle")
def draw_polygon(points, color):
    turtle.penup()
    turtle.goto(points[0][0], points[0][1])
    turtle.pendown()
    turtle.pencolor(color)
    for point in points[1:]:
        turtle.goto(point[0], point[1])
    turtle.goto(points[0][0], points[0][1])
def print_text(text, position, font_size=15, color="black"):
    turtle.penup()
    turtle.goto(position)
    turtle.pendown()
    turtle.pencolor(color)
    turtle.write(text, align="center", font=("Arial", font_size, "normal"))
def draw_corner_points(points, color="black"):
    turtle.penup()
    for x, y in points:
```

```
        turtle.goto(x, y)
        turtle.dot(5, color)
        turtle.goto(x, y + 15)
        turtle.pendown()
        turtle.write(f"({x}, {y})", align="center", font=("Arial", 10, "normal"))
        turtle.penup()
def translate_point(x, y, tx, ty):
    return x + tx, y + ty
def rotate_point(x, y, cx, cy, angle):
    rad = math.radians(angle)
    s = math.sin(rad)
    c = math.cos(rad)
    x -= cx
    y -= cy
    xnew = x * c - y * s
    ynew = x * s + y * c
    return math.floor(xnew + cx), math.floor(ynew + cy)
def composite_translation(points, translations):
    return [translate_point(x, y, translations[0], translations[1]) for x, y in points]
def composite_rotation(points, cx, cy, angle):
    return [rotate_point(x, y, cx, cy, angle) for x, y in points]
def draw_shapes():
    turtle.speed(0)
    turtle.hideturtle()
    start_x = -400
    y_positions = [250, -50, -250]
    triangle = [(-400, 200), (-350, 100), (-450, 100)]
    rectangle = [(-200, 200), (-100, 200), (-100, 50), (-200, 50)]
```

```
    print_text("Original Shapes", (start_x, y_positions[0]))
    draw_polygon(triangle, "blue")
    draw_polygon(rectangle, "blue")
    draw_corner_points(triangle, color="blue")
    draw_corner_points(rectangle, color="blue")
    translations = (500, 0)
    print_text("Composite Translation", (start_x + 500, y_positions[0]))
    translated_triangle = composite_translation(triangle, translations)
    translated_rectangle = composite_translation(rectangle, translations)
    draw_polygon(translated_triangle, "green")
    draw_polygon(translated_rectangle, "green")
    draw_corner_points(translated_triangle, color="green")
    draw_corner_points(translated_rectangle, color="green")
    rotation_center_triangle = (-350, 150)
    rotation_center_rectangle = (-150, 125)
    angles = 45
    print_text("Composite Rotation", (start_x + 200, y_positions[1] + 50))
    rotated_triangle = composite_rotation(triangle, rotation_center_triangle[0],
rotation_center_triangle[1], angles)
    rotated_rectangle = composite_rotation(rectangle, rotation_center_rectangle[0],
rotation_center_rectangle[1], angles)
    rotated_triangle_shifted = composite_translation(rotated_triangle, (0, -300))
    rotated_rectangle_shifted = composite_translation(rotated_rectangle, (0, -300))
    draw_polygon(rotated_triangle_shifted, "red")
    draw_polygon(rotated_rectangle_shifted, "red")
    draw_corner_points(rotated_triangle_shifted, color="red")
    draw_corner_points(rotated_rectangle_shifted, color="red")
    turtle.hideturtle()
draw_shapes()
```

turtle.done()

Output:



PRACTICAL 11

Aim: Composite Transformation (Shearing)

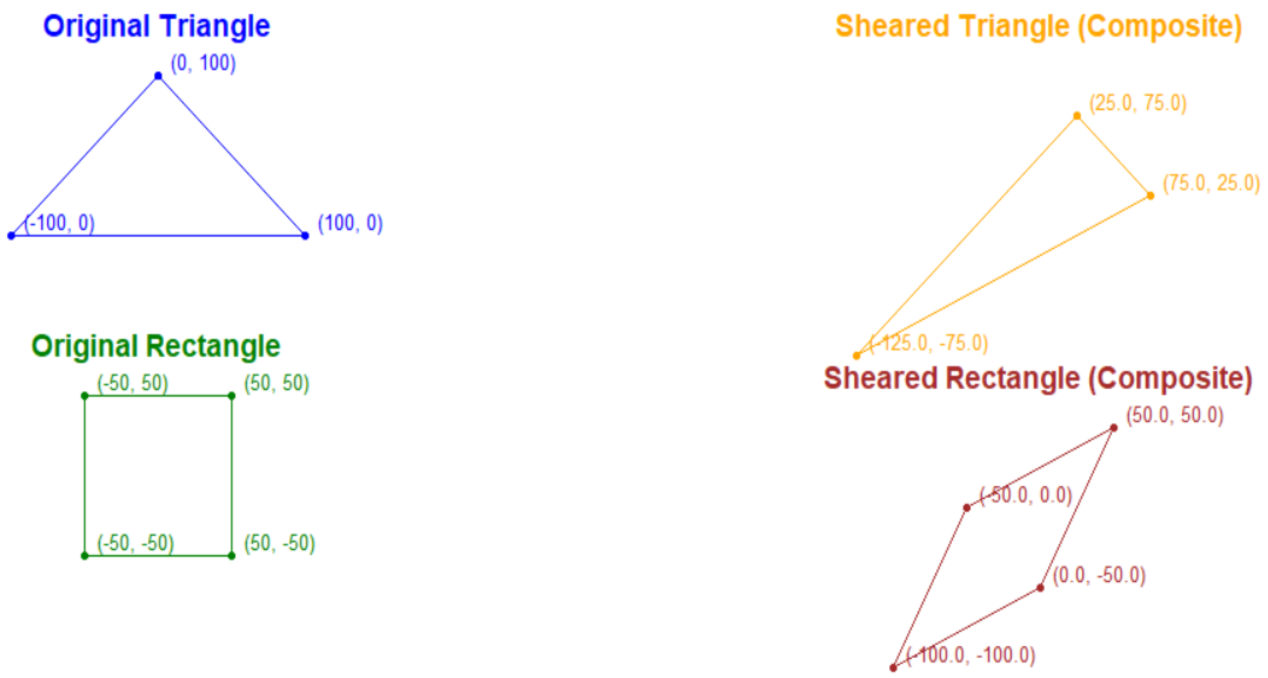
Code:

```
import turtle
def write_label(label, position):
    turtle.penup()
    turtle.goto(position[0], position[1] + 20)
    turtle.pendown()
    turtle.write(label, align="center", font=("Arial", 14, "bold"))
    turtle.penup()
def draw_shape(vertices, position_offset=(0, 0)):
    turtle.penup()
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
    turtle.pendown()
    for vertex in vertices[1:]:
        turtle.goto(vertex[0] + position_offset[0], vertex[1] + position_offset[1])
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
def draw_points(vertices, position_offset=(0, 0), color="black"):
    turtle.color(color)
    for x, y in vertices:
        turtle.penup()
        turtle.goto(x + position_offset[0], y + position_offset[1])
        turtle.pendown()
        turtle.dot(5)
        turtle.penup()
        turtle.goto(x + position_offset[0] + 10, y + position_offset[1])
        turtle.pendown()
```

```
        turtle.write(f"({x}, {y})", align="left", font=("Arial", 10, "normal"))
def shear_shape_composite(vertices, shear_factor_x, shear_factor_y, shear_origin):
    translated_vertices = [(x - shear_origin[0], y - shear_origin[1]) for x, y in vertices]
    sheared_vertices = [(x + shear_factor_x * y, y + shear_factor_y * x) for x, y in translated_vertices]
    final_vertices = [(x + shear_origin[0], y + shear_origin[1]) for x, y in sheared_vertices]
    return final_vertices
turtle.speed(2)
turtle.bgcolor("white")
triangle_vertices = [(-100, 0), (0, 100), (100, 0)]
rectangle_vertices = [(-50, -50), (-50, 50), (50, 50), (50, -50)]
shear_factor_x = 0.5
shear_factor_y = 0.5
shear_origin_triangle = (50, 50)
shear_origin_rectangle = (50, 50)
sheared_triangle = shear_shape_composite(triangle_vertices, shear_factor_x, shear_factor_y, shear_origin_triangle)
sheared_rectangle = shear_shape_composite(rectangle_vertices, shear_factor_x, shear_factor_y, shear_origin_rectangle)
turtle.color("blue")
write_label("Original Triangle", (-400, 150))
draw_shape(triangle_vertices, position_offset=(-400, 50))
draw_points(triangle_vertices, position_offset=(-400, 50), color="blue")
turtle.color("green")
write_label("Original Rectangle", (-400, -50))
draw_shape(rectangle_vertices, position_offset=(-400, -100))
draw_points(rectangle_vertices, position_offset=(-400, -100), color="green")
turtle.color("orange")
write_label("Sheared Triangle (Composite)", (200, 150))
```

```
draw_shape(sheared_triangle, position_offset=(200, 50))
draw_points(sheared_triangle, position_offset=(200, 50), color="orange")
turtle.color("brown")
write_label("Sheared Rectangle (Composite)", (200, -70))
draw_shape(sheared_rectangle, position_offset=(200, -120))
draw_points(sheared_rectangle, position_offset=(200, -120), color="brown")
turtle.hideturtle()
turtle.done()
```

Output:



PRACTICAL 12

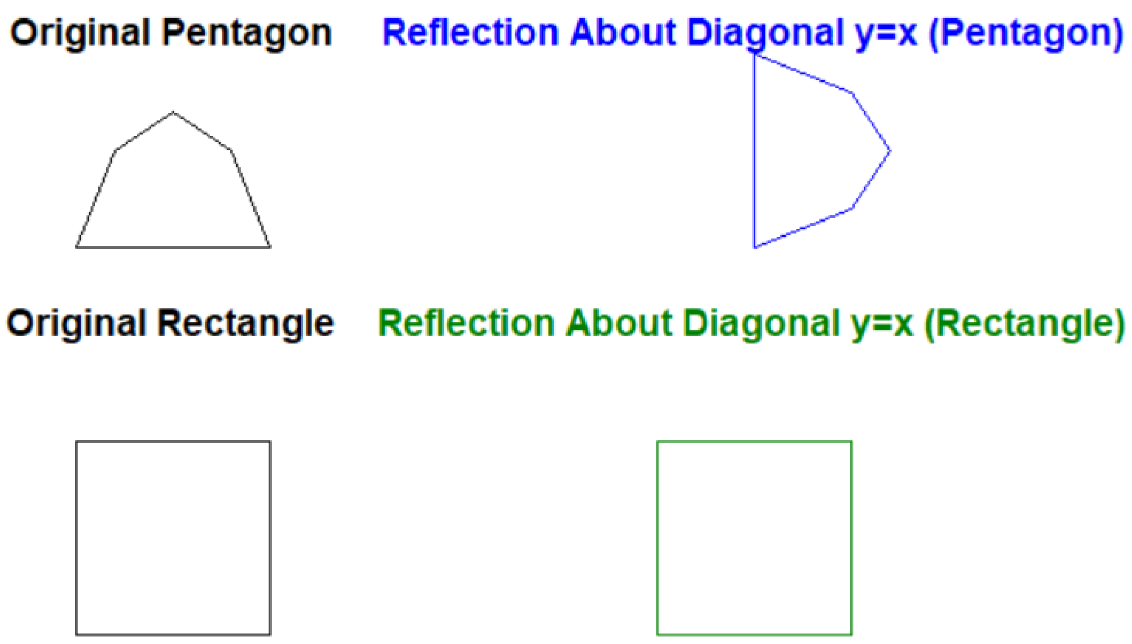
Aim: Reflection about Diagonal (Line Y = X)

Code:

```
import turtle
def write_label(label, position):
    turtle.penup()
    turtle.goto(position[0], position[1] + 50)
    turtle.pendown()
    turtle.write(label, align="center", font=("Arial", 14, "bold"))
    turtle.penup()
def draw_shape(vertices, position_offset=(0, 0)):
    turtle.penup()
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
    turtle.pendown()
    for vertex in vertices[1:]:
        turtle.goto(vertex[0] + position_offset[0], vertex[1] + position_offset[1])
    turtle.goto(vertices[0][0] + position_offset[0], vertices[0][1] + position_offset[1])
def reflect_diagonal(vertices):
    return [(y, x) for x, y in vertices]
turtle.speed(2)
turtle.bgcolor("white")
pentagon_vertices = [(-50, 0), (-30, 50), (0, 70), (30, 50), (50, 0)]
rectangle_vertices = [(-50, -50), (-50, 50), (50, 50), (50, -50)]
gap_x = 300
gap_y = 200
turtle.color("black")
write_label("Original Pentagon", (-400, 100))
```

```
draw_shape(pentagon_vertices, position_offset=(-400, 50))
write_label("Original Rectangle", (-400, -50))
draw_shape(rectangle_vertices, position_offset=(-400, -100))
reflected_pentagon_diagonal = reflect_diagonal(pentagon_vertices)
reflected_rectangle_diagonal = reflect_diagonal(rectangle_vertices)
turtle.color("blue")
write_label("Reflection About Diagonal y=x (Pentagon)", (-400 + gap_x, 100))
draw_shape(reflected_pentagon_diagonal, position_offset=(-400 + gap_x, 100))
turtle.color("green")
write_label("Reflection About Diagonal y=x (Rectangle)", (-400 + gap_x, -50))
draw_shape(reflected_rectangle_diagonal, position_offset=(-400 + gap_x, -100))
turtle.hideturtle()
turtle.done()
```

Output:



PRACTICAL 13

Aim: Point Clipping

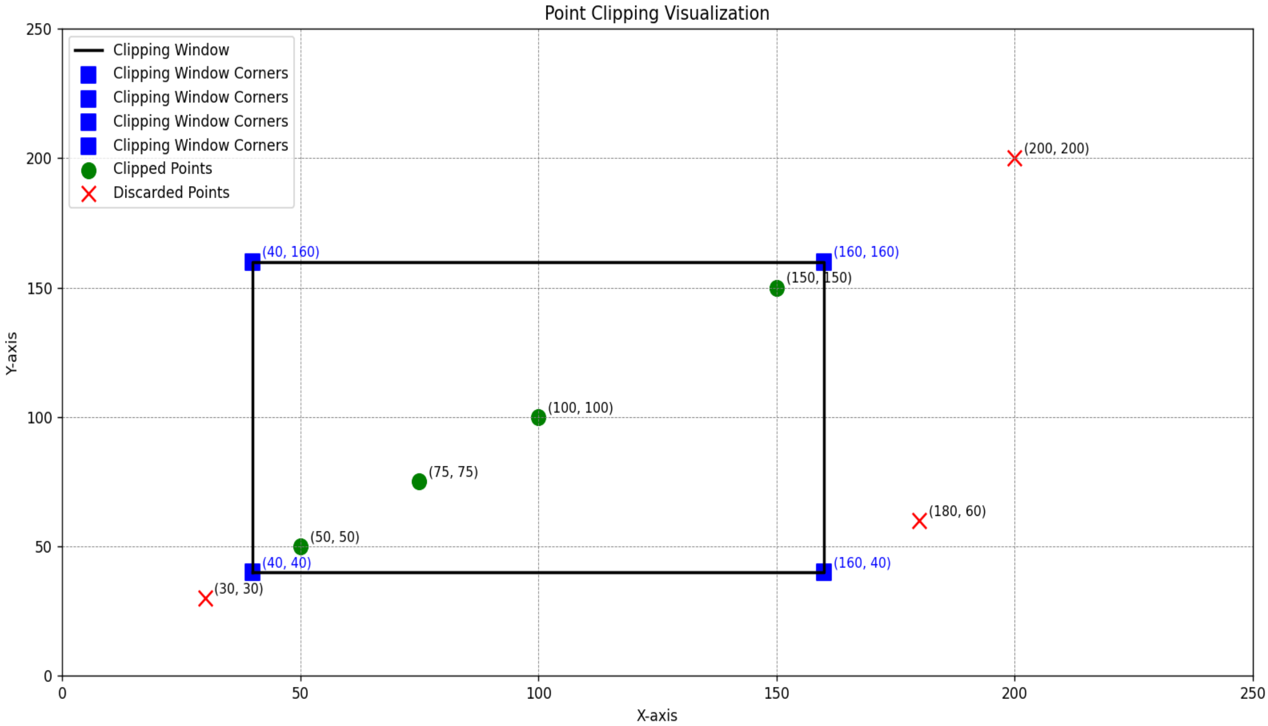
Code:

```
import matplotlib.pyplot as plt
def point_clipping(points, window):
    x_min, y_min, x_max, y_max = window
    clipped_points = []
    discarded_points = []
    for point in points:
        x, y = point
        if (x_min <= x <= x_max) and (y_min <= y <= y_max):
            clipped_points.append(point)
        else:
            discarded_points.append(point)
    return clipped_points, discarded_points
if __name__ == "__main__":
    points = [
        (50, 50),
        (150, 150),
        (200, 200),
        (30, 30),
        (100, 100),
        (75, 75),
        (180, 60)
    ]
    window = (40, 40, 160, 160)
    clipped_points, discarded_points = point_clipping(points, window)
```

```
    print("Clipped Points:", clipped_points)
    print("Discarded Points:", discarded_points)
    plt.figure(figsize=(8, 8))
    plt.plot([window[0], window[0], window[2], window[2], window[0]],
             [window[1], window[3], window[3], window[1], window[1]], color='black', linewidth=2, label='Clipping Window')
    window_corners = [(window[0], window[1]), (window[0], window[3]),
                      (window[2], window[3]), (window[2], window[1])]
    for corner in window_corners:
        plt.scatter(*corner, color='blue', s=100, marker='s', label='Clipping Window Corners')
        plt.text(corner[0] + 2, corner[1] + 2, f'{corner}', fontsize=9, color='blue')
    if clipped_points:
        x_clipped, y_clipped = zip(*clipped_points)
        plt.scatter(x_clipped, y_clipped, color='green', label='Clipped Points', s=100, marker='o')
        for point in clipped_points:
            plt.text(point[0] + 2, point[1] + 2, f'{point}', fontsize=9, color='black')
    if discarded_points:
        x_discarded, y_discarded = zip(*discarded_points)
        plt.scatter(x_discarded, y_discarded, color='red', label='Discarded Points', s=100, marker='x')
        for point in discarded_points:
            plt.text(point[0] + 2, point[1] + 2, f'{point}', fontsize=9, color='black')
    plt.xlim(0, 250)
    plt.ylim(0, 250)
    plt.axhline(0, color='black', linewidth=0.5, ls='--')
    plt.axvline(0, color='black', linewidth=0.5, ls='--')
    plt.title('Point Clipping Visualization')
    plt.xlabel('X-axis')
```

```
plt.ylabel('Y-axis')
plt.grid(color='gray', linestyle='--', linewidth=0.5)
plt.legend()
plt.show()
```

Output:



PRACTICAL 14

Aim: Line Clipping (Cohen Sutherland)

Code:

```
import matplotlib.pyplot as plt
INSIDE = 0  # 0000
LEFT = 1    # 0001
RIGHT = 2   # 0010
BOTTOM = 4  # 0100
TOP = 8     # 1000
def compute_outcode(x, y, window):
    x_min, y_min, x_max, y_max = window
    code = INSIDE
    if x < x_min:
        code |= LEFT
    elif x > x_max:
        code |= RIGHT
    if y < y_min:
        code |= BOTTOM
    elif y > y_max:
        code |= TOP
    return code
def cohen_sutherland_line_clip(line, window):
    x_min, y_min, x_max, y_max = window
    (x0, y0), (x1, y1) = line
    outcode0 = compute_outcode(x0, y0, window)
    outcode1 = compute_outcode(x1, y1, window)
    accept = False
    clipped = False
```

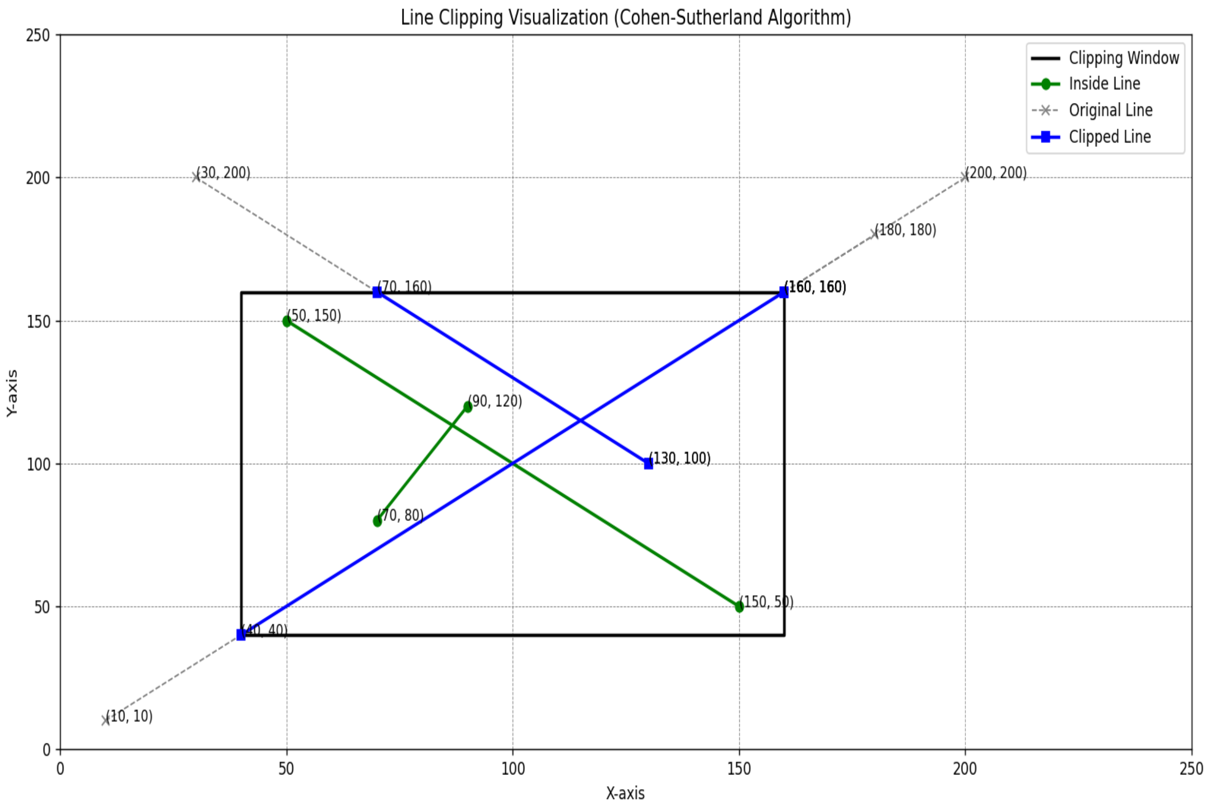
```
while True:
    if outcode0 == 0 and outcode1 == 0:
        accept = True
        break
    elif (outcode0 & outcode1) != 0:
        break
    else:
        x, y = 0, 0
        outcode_out = outcode0 if outcode0 != 0 else outcode1
        clipped = True
        if outcode_out & TOP:
            x = x0 + (x1 - x0) * (y_max - y0) / (y1 - y0)
            y = y_max
        elif outcode_out & BOTTOM:
            x = x0 + (x1 - x0) * (y_min - y0) / (y1 - y0)
            y = y_min
        elif outcode_out & RIGHT:
            y = y0 + (y1 - y0) * (x_max - x0) / (x1 - x0)
            x = x_max
        elif outcode_out & LEFT:
            y = y0 + (y1 - y0) * (x_min - x0) / (x1 - x0)
            x = x_min
        if outcode_out == outcode0:
            x0, y0 = x, y
            outcode0 = compute_outcode(x0, y0, window)
        else:
            x1, y1 = x, y
            outcode1 = compute_outcode(x1, y1, window)
```

```
    if accept:
        if clipped:
            return "partially", [(x0, y0), (x1, y1)], line
        else:
            return "inside", [(x0, y0), (x1, y1)], line
    else:
        return "outside", line, line
if __name__ == "__main__":
    lines = [
        [(10, 10), (200, 200)],
        [(50, 150), (150, 50)],
        [(30, 200), (130, 100)],
        [(160, 160), (180, 180)],
        [(70, 80), (90, 120)]
    ]
    window = (40, 40, 160, 160)
    inside_lines = []
    partially_clipped_lines = []
    outside_lines = []
    for line in lines:
        category, clipped_line, original_line = cohen_sutherland_line_clip(line, window)
        if category == "inside":
            inside_lines.append(line)
        elif category == "partially":
            partially_clipped_lines.append((clipped_line, original_line))
        elif category == "outside":
            outside_lines.append(line)
    print("Inside Lines:", inside_lines)
    print("Partially Clipped Lines:", partially_clipped_lines)
```

```
    print("Outside Lines:", outside_lines)
    plt.figure(figsize=(8, 8))
    plt.plot([window[0], window[0], window[2], window[2], window[0]],
             [window[1], window[3], window[3], window[1], window[1]], color='black', linewidth=2, label='Clipping Window')
    for line in inside_lines:
        (x0, y0), (x1, y1) = line
        plt.plot([x0, x1], [y0, y1], color='green', linewidth=2, marker='o', label='Inside Line' if line == inside_lines[0] else "")
        plt.text(x0, y0, f'({x0}, {y0})', fontsize=9, color='black')
        plt.text(x1, y1, f'({x1}, {y1})', fontsize=9, color='black')
    for clipped_line, original_line in partially_clipped_lines:
        (ox0, oy0), (ox1, oy1) = original_line
        plt.plot([ox0, ox1], [oy0, oy1], color='gray', linewidth=1, linestyle='--', marker='x', label='Original Line' if original_line == partially_clipped_lines[0][1] else "")
        plt.text(ox0, oy0, f'({ox0}, {oy0})', fontsize=9, color='black')
        plt.text(ox1, oy1, f'({ox1}, {oy1})', fontsize=9, color='black')
        (x0, y0), (x1, y1) = clipped_line
        plt.plot([x0, x1], [y0, y1], color='blue', linewidth=2, marker='s', label='Clipped Line' if clipped_line == partially_clipped_lines[0][0] else "")
        plt.text(x0, y0, f'({x0:.0f}, {y0:.0f})', fontsize=9, color='black')
        plt.text(x1, y1, f'({x1:.0f}, {y1:.0f})', fontsize=9, color='black')
    for line in outside_lines:
        (x0, y0), (x1, y1) = line
        plt.plot([x0, x1], [y0, y1], color='red', linewidth=2, linestyle=':', marker='x', label='Outside Line' if line == outside_lines[0] else "")
        plt.text(x0, y0, f'({x0}, {y0})', fontsize=9, color='black')
        plt.text(x1, y1, f'({x1}, {y1})', fontsize=9, color='black')
    plt.xlim(0, 250)
    plt.ylim(0, 250)
```

```
plt.axhline(0, color='black', linewidth=0.5, ls='--')
plt.axvline(0, color='black', linewidth=0.5, ls='--')
plt.title('Line Clipping Visualization (Cohen-Sutherland Algorithm)')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.grid(color='gray', linestyle='--', linewidth=0.5)
plt.legend()
plt.show()
```

Output:



PRACTICAL 15

Aim: Mid-Point Ellipse Line Drawing Algorithm

Code:

```
import turtle
def draw_point(x, y):
    turtle.penup()
    turtle.goto(x, y)
    turtle.pendown()
    turtle.dot(3, "blue")
def draw_ellipse(xc, yc, a, b):
    x = 0
    y = b
    d1 = (b**2) - (a**2 * b) + (0.25 * a**2)
    draw_point(xc + x, yc + y)
    draw_point(xc - x, yc + y)
    draw_point(xc + x, yc - y)
    draw_point(xc - x, yc - y)
    print(f"Point: ({xc + x}, {yc + y})")
    print(f"Point: ({xc - x}, {yc + y})")
    print(f"Point: ({xc + x}, {yc - y})")
    print(f"Point: ({xc - x}, {yc - y})")
    while (a**2 * (y - 0.5)) > (b**2 * x):
        if d1 < 0:
            d1 = d1 + (b**2 * (2 * x + 3))
        else:
            d1 = d1 + (b**2 * (2 * x + 3)) + (a**2 * (-2 * y + 2))
            y -= 1
        x += 1
```

```
        draw_point(xc + x, yc + y)
        draw_point(xc - x, yc + y)
        draw_point(xc + x, yc - y)
        draw_point(xc - x, yc - y)
        if x % 5 == 0:
            print(f"Point: ({xc + x}, {yc + y})")
            print(f"Point: ({xc - x}, {yc + y})")
            print(f"Point: ({xc + x}, {yc - y})")
            print(f"Point: ({xc - x}, {yc - y})")
    d2 = (b**2) * (x + 0.5)**2 + (a**2) * (y - 1)**2 - (a**2) * (b**2)
    while y >= 0:
        if d2 > 0:
            d2 = d2 + (a**2 * (-2 * y + 3))
        else:
            d2 = d2 + (a**2 * (-2 * y + 3)) + (b**2 * (2 * x + 2))
            x += 1
        y -= 1
        draw_point(xc + x, yc + y)
        draw_point(xc - x, yc + y)
        draw_point(xc + x, yc - y)
        draw_point(xc - x, yc - y)
        if y % 5 == 0:
            print(f"Point: ({xc + x}, {yc + y})")
            print(f"Point: ({xc - x}, {yc + y})")
            print(f"Point: ({xc + x}, {yc - y})")
            print(f"Point: ({xc - x}, {yc - y})")
    turtle.update()
def main():
    xc, yc = 0, 0
```

```python
    a, b = 50, 30
    turtle.speed(0)
    turtle.bgcolor("white")
    turtle.color("blue")
    turtle.hideturtle()
    turtle.tracer(0, 0)
    draw_ellipse(xc, yc, a, b)
    turtle.done()
if __name__ == "__main__":
    main()
```

Output: