Lab19_SDA

March 18, 2025

[2]:
```python
#lab19_1
# Initialize previous number
previous_num = 0

print("Printing sum of current and previous number in a range(10):")

# Iterate through the first 10 numbers
for i in range(10):
    # Calculate sum of current and previous number
    sum_nums = i + previous_num
    print("Current Number:", i, "Previous Number:", previous_num, "Sum:",␣
 ↪sum_nums)

    # Update previous number to current number
    previous_num = i
```

```
Printing sum of current and previous number in a range(10):
Current Number: 0 Previous Number: 0 Sum: 0
Current Number: 1 Previous Number: 0 Sum: 1
Current Number: 2 Previous Number: 1 Sum: 3
Current Number: 3 Previous Number: 2 Sum: 5
Current Number: 4 Previous Number: 3 Sum: 7
Current Number: 5 Previous Number: 4 Sum: 9
Current Number: 6 Previous Number: 5 Sum: 11
Current Number: 7 Previous Number: 6 Sum: 13
Current Number: 8 Previous Number: 7 Sum: 15
Current Number: 9 Previous Number: 8 Sum: 17
```

[3]:
```python
#lab19_2
# Given string
str_x = """Human health can be monitored using several types of data. The data␣
 ↪collected
from scans, labs, and procedures can be used for research purposes. The key to␣
 ↪using data
effectively is .... """

# Count occurrences of substring "data"
count_data = str_x.count("data")
```

```python
# Print the result
print(f'The substring "data" appears {count_data} times in the given string.')
```

```
The substring "data" appears 3 times in the given string.
```

[4]:
```python
#lab19_3
# Given number
num = 4858643

# Print message
print("Digits in reverse order:")

# Loop to extract digits
while num > 0:
    digit = num % 10  # Get the last digit
    print(digit, end=" ")  # Print the digit with space
    num = num // 10  # Remove the last digit
```

```
Digits in reverse order:
3 4 6 8 5 8 4
```

[5]:
```python
#lab19_4
# Calculator Program

# Taking user input for two numbers
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

# Performing calculations
addition = num1 + num2
subtraction = num1 - num2
multiplication = num1 * num2

# Handling division to avoid division by zero error
if num2 != 0:
    division = num1 / num2
else:
    division = "Undefined (division by zero is not allowed)"

# Displaying results
print("\nResults:")
print(f"Addition: {num1} + {num2} = {addition}")
print(f"Subtraction: {num1} - {num2} = {subtraction}")
print(f"Multiplication: {num1} × {num2} = {multiplication}")
print(f"Division: {num1} ÷ {num2} = {division}")
```

```
Enter the first number:  18
Enter the second number:  20
```

```
Results:
Addition: 18.0 + 20.0 = 38.0
Subtraction: 18.0 - 20.0 = -2.0
Multiplication: 18.0 × 20.0 = 360.0
Division: 18.0 ÷ 20.0 = 0.9
```

[9]:
```python
#lab19_5
# Initialize the first two numbers
num1, num2 = 0, 1

print("Fibonacci Series up to 10 terms:")

# Loop to generate 10 terms
for _ in range(10):
    print(num1, end=" ")  # Print the current term
    res = num1 + num2  # Calculate the next term
    num1, num2 = num2, res  # Update values for the next iteration
```

```
Fibonacci Series up to 10 terms:
0 1 1 2 3 5 8 13 21 34
```

[ ]:
```python
import math
import turtle
import random
import tkinter as tk
from tkinter import ttk, colorchooser

class FractalTreeApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Fractal Tree Generator")
        self.root.geometry("800x600")

        # Setup main frames
        self.control_frame = ttk.Frame(root, padding="10")
        self.control_frame.pack(side=tk.LEFT, fill=tk.Y)

        self.canvas_frame = ttk.Frame(root)
        self.canvas_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

        # Create turtle canvas
        self.canvas = tk.Canvas(self.canvas_frame, bg="white", width=600,␣
↪height=500)
        self.canvas.pack(fill=tk.BOTH, expand=True)

        # Create turtle screen
        self.screen = turtle.TurtleScreen(self.canvas)
```

```
        self.screen.tracer(0)  # Turn off animation for speed
        self.turtle = turtle.RawTurtle(self.screen)
        self.turtle.hideturtle()
        self.turtle.speed(0)

        # Control variables
        self.iterations = tk.IntVar(value=8)
        self.branch_length = tk.DoubleVar(value=100.0)
        self.angle = tk.DoubleVar(value=30.0)
        self.reduction = tk.DoubleVar(value=0.75)
        self.branch_color = "#008000"  # Default green
        self.randomize_var = tk.BooleanVar(value=False)

        # Create controls
        self.create_controls()

    def create_controls(self):
        ttk.Label(self.control_frame, text="Fractal Tree Controls",␣
↪font=("Arial", 14, "bold")).pack(pady=10)

        # Iterations slider
        ttk.Label(self.control_frame, text="Iterations:").pack(anchor=tk.W,␣
↪pady=(10, 0))
        iterations_slider = ttk.Scale(self.control_frame, from_=1, to=12,
                                     variable=self.iterations, orient=tk.
↪HORIZONTAL, length=200)
        iterations_slider.pack(fill=tk.X, pady=5)

        # Branch length slider
        ttk.Label(self.control_frame, text="Initial Branch Length:").
↪pack(anchor=tk.W, pady=(10, 0))
        length_slider = ttk.Scale(self.control_frame, from_=20, to=200,
                                 variable=self.branch_length, orient=tk.
↪HORIZONTAL, length=200)
        length_slider.pack(fill=tk.X, pady=5)

        # Angle slider
        ttk.Label(self.control_frame, text="Branch Angle (degrees):").
↪pack(anchor=tk.W, pady=(10, 0))
        angle_slider = ttk.Scale(self.control_frame, from_=5, to=60,
                                variable=self.angle, orient=tk.HORIZONTAL,␣
↪length=200)
        angle_slider.pack(fill=tk.X, pady=5)

        # Reduction slider
```

```
        ttk.Label(self.control_frame, text="Length Reduction:").pack(anchor=tk.
↪W, pady=(10, 0))
        reduction_slider = ttk.Scale(self.control_frame, from_=0.5, to=0.9,
                                     variable=self.reduction, orient=tk.
↪HORIZONTAL, length=200)
        reduction_slider.pack(fill=tk.X, pady=5)

        # Color picker
        ttk.Label(self.control_frame, text="Branch Color:").pack(anchor=tk.W,␣
↪pady=(10, 0))
        color_button = ttk.Button(self.control_frame, text="Choose Color",
                                  command=self.choose_color)
        color_button.pack(fill=tk.X, pady=5)

        # Color indicator
        self.color_indicator = tk.Canvas(self.control_frame, width=200,␣
↪height=20, bg=self.branch_color)
        self.color_indicator.pack(pady=5)

        # Randomize option
        randomize_check = ttk.Checkbutton(self.control_frame, text="Randomize␣
↪Parameters",
                                          variable=self.randomize_var)
        randomize_check.pack(anchor=tk.W, pady=10)

        # Generate button
        generate_button = ttk.Button(self.control_frame, text="Generate Tree",
                                     command=self.generate_tree)
        generate_button.pack(fill=tk.X, pady=10)

        # Reset button
        reset_button = ttk.Button(self.control_frame, text="Reset Canvas",
                                 command=self.reset_canvas)
        reset_button.pack(fill=tk.X, pady=5)

    def choose_color(self):
        color = colorchooser.askcolor(initialcolor=self.branch_color)
        if color[1]:  # If color is selected (not canceled)
            self.branch_color = color[1]
            self.color_indicator.config(bg=self.branch_color)

    def reset_canvas(self):
        self.screen.clear()
        self.screen.update()

    def generate_tree(self):
        # Reset canvas
```

```
        self.reset_canvas()

        # Randomize if needed
        if self.randomize_var.get():
            self.iterations.set(random.randint(5, 10))
            self.branch_length.set(random.uniform(80, 150))
            self.angle.set(random.uniform(15, 45))
            self.reduction.set(random.uniform(0.6, 0.85))
            r, g, b = random.randint(0, 255), random.randint(0, 255), random.
↪randint(0, 255)
            self.branch_color = f'#{r:02x}{g:02x}{b:02x}'
            self.color_indicator.config(bg=self.branch_color)

        # Get canvas dimensions
        canvas_width = self.canvas.winfo_width()
        canvas_height = self.canvas.winfo_height()

        # If dimensions are 1 (not yet realized), use defaults
        if canvas_width <= 1:
            canvas_width = 600
        if canvas_height <= 1:
            canvas_height = 500

        # Position turtle in the center bottom of the canvas
        self.turtle.penup()
        self.turtle.setposition(0, -canvas_height/4)  # Bottom quarter of the␣
↪canvas
        self.turtle.setheading(90)  # Point upward
        self.turtle.pendown()

        # Draw tree
        self.turtle.pencolor(self.branch_color)
        self.draw_tree(self.branch_length.get(), self.iterations.get())

        # Update display
        self.screen.update()

    def draw_tree(self, branch_length, iterations):
        if iterations <= 0:
            return

        # Calculate pen width based on iteration level
        self.turtle.pensize(iterations)

        # Darken color slightly for trunk/larger branches
        if iterations > 3:
            # Darken the color
```

```
            color = self.branch_color
            r = int(color[1:3], 16) * 0.8
            g = int(color[3:5], 16) * 0.8
            b = int(color[5:7], 16) * 0.8
            self.turtle.pencolor(f'#{int(r):02x}{int(g):02x}{int(b):02x}')
        else:
            self.turtle.pencolor(self.branch_color)

        # Draw branch
        self.turtle.forward(branch_length)

        # Save position and heading
        pos = self.turtle.position()
        heading = self.turtle.heading()

        # Right branch
        self.turtle.right(self.angle.get())
        self.draw_tree(branch_length * self.reduction.get(), iterations - 1)

        # Left branch
        self.turtle.penup()
        self.turtle.setposition(pos)
        self.turtle.setheading(heading)
        self.turtle.pendown()
        self.turtle.left(self.angle.get())
        self.draw_tree(branch_length * self.reduction.get(), iterations - 1)

# Special function for Jupyter notebooks
def run_in_jupyter():
    # Create the main window
    root = tk.Tk()

    # Create and set up the app
    app = FractalTreeApp(root)

    # Generate a tree immediately
    root.update()  # This forces the window to update and realize its widgets
    app.generate_tree()

    # Return the application and root for Jupyter to keep reference
    return root, app

# Use this line in Jupyter to run the app
root, app = run_in_jupyter()
root.mainloop()  # In some Jupyter environments, you might need to comment this out
```

```
[ ]:
```