Python Project Report

<SciCal: Scientific Calculator Development>

01286121 Computer Programming

Software Engineering Program

By

66011606 Thura Aung

Python Project

<SciCal: Scientific Calculator Development>

Project Introduction

The Scientific Calculator and the Unit Converter project is about the development of an integrated software solution aimed at providing users with a versatile tool for mathematical calculations and unit conversions. The primary goal of this project was to create an intuitive and user-friendly interface that offers advanced scientific calculator functionalities while seamlessly incorporating a comprehensive unit conversion feature.

Motivation

Creating a scientific calculator is exciting because it's like having a super-smart math assistant on your computer. With this project, I'll get to explore how to build functions that solve complex math problems. I'll be able to use it for my own math homework or whenever I need quick calculations too. Building this calculator as my first Python project is not only useful but also a great way for me to learn and improve my programming skills.

Functionalities

1. Scientific Calculator:
   - Performs basic arithmetic operations, trigonometric functions, logarithmic functions, exponentiation, square root, and factorial.
   - Handles parentheses and maintains the proper order of operations.
   - Includes memory functions for storing and recalling values.

2. Unit Converter:
   - Converts various units in categories like length, temperature, mass, volume, time, etc.
   - Enables bidirectional conversions for seamless unit switching.
   - Provides an intuitive interface for easy unit selection and input.

3. Graphical User Interface (GUI):
   - Presents an intuitive design for effortless navigation between the calculator and unit converter.
   - Offers clear visual feedback for user inputs, calculations, and converted results.

4. Error Handling and Validation:
   - Validates inputs to prevent calculation or conversion errors.
   - Displays clear error messages for incorrect inputs or unsupported operations.

Screen Captures

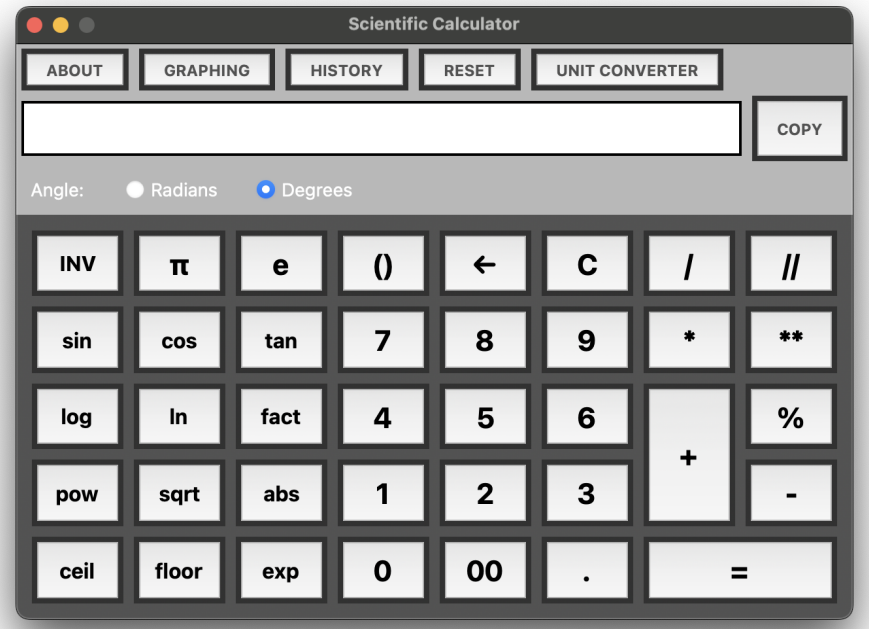


Figure 1: First page of **SciCal**

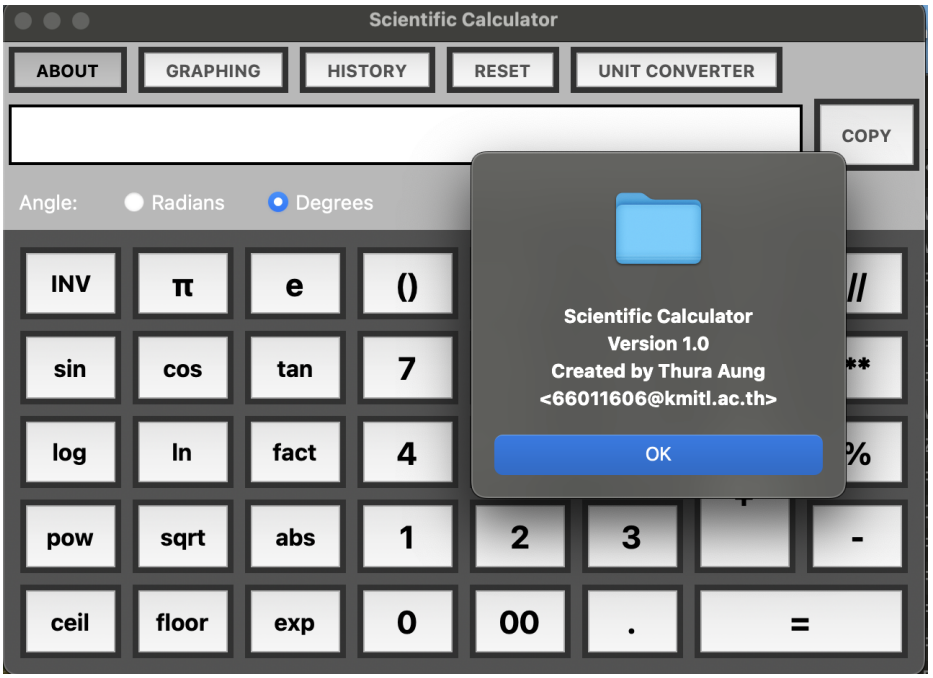


Figure 2: About page when **ABOUT** button is clicked

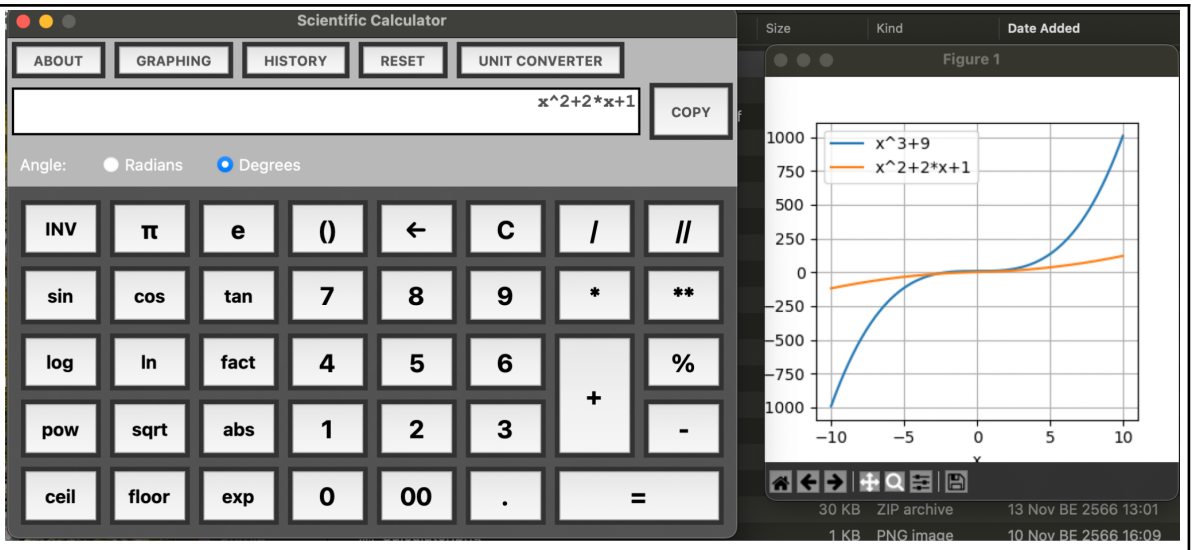
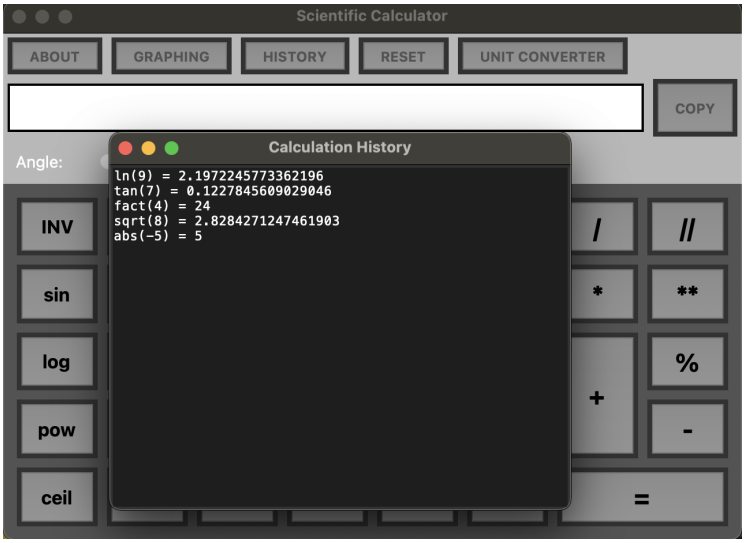


Figure 3: Example usage of **GRAPHING** button
(It can plot more than one equation)



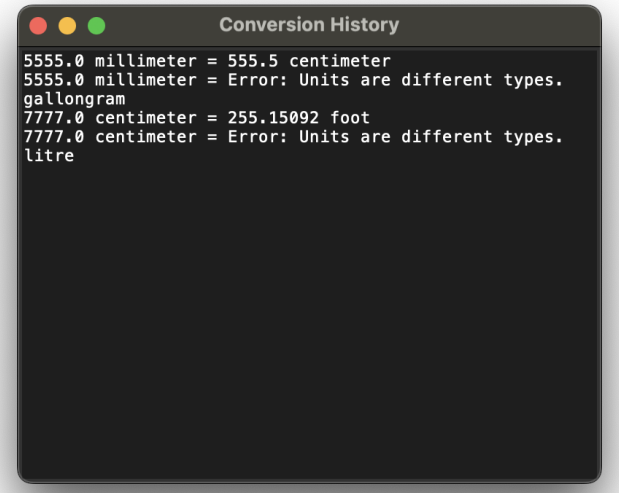


Figure 4: **HISTORY** button showing calculation history for calculator and conversion history for unit converter

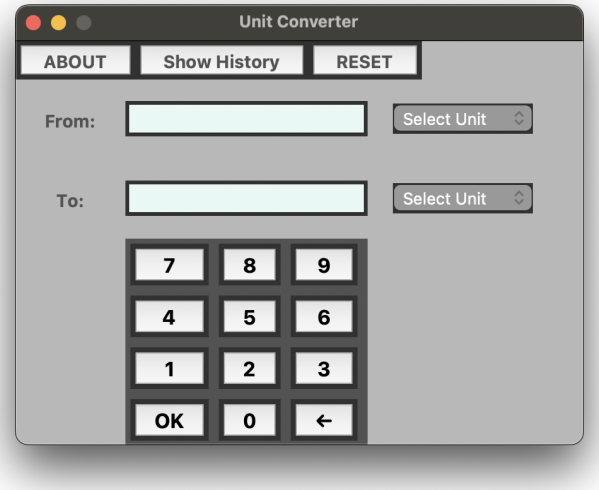


Figure 5: **UNIT CONVERTER** button to open Unit converter window

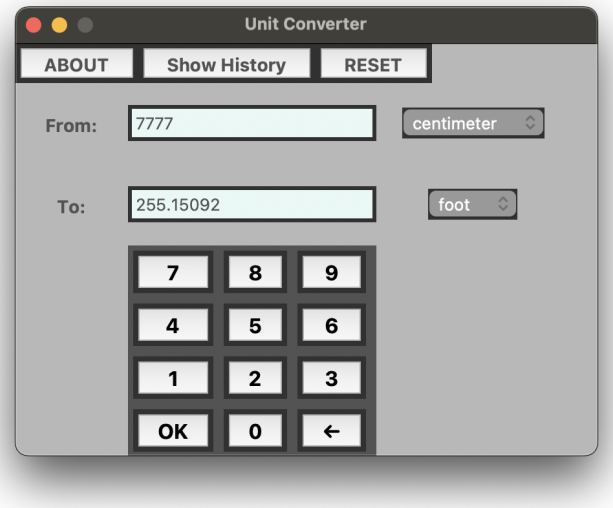


Figure 6: Unit conversion example (When OK button is clicked)

Errors

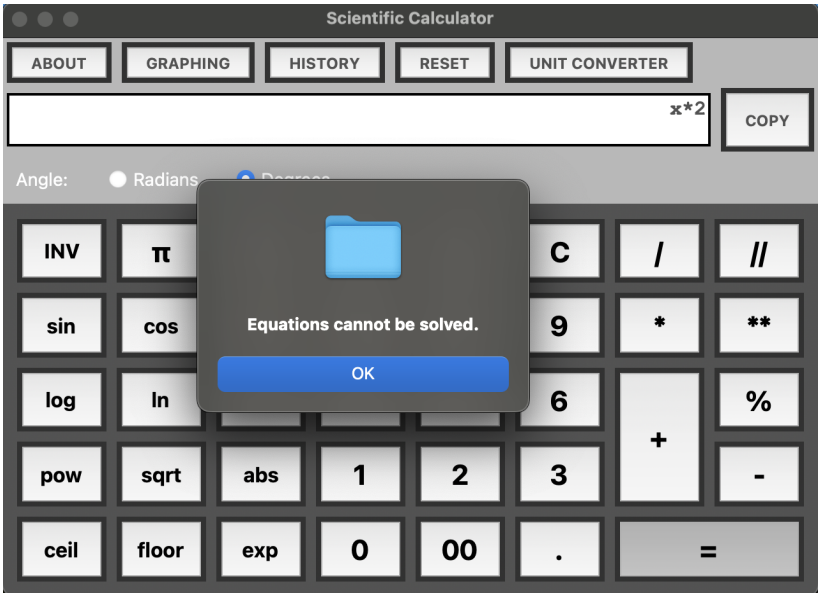


Figure 7: Graphing example (Error when the equations are tried to solve with "=")

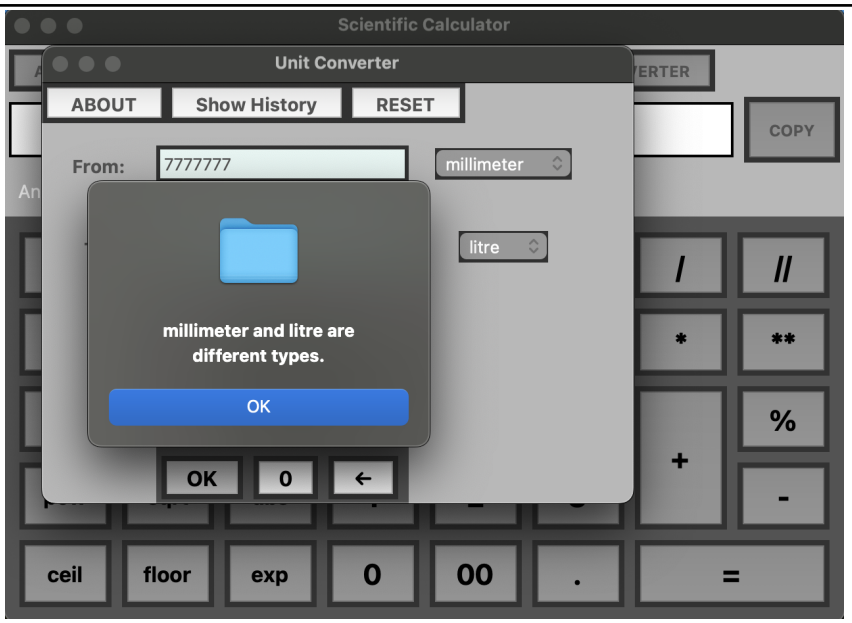


Figure 8: Unit conversion example (Error when units are different.)

Figure 9: Unit conversion example (Error when one of input/output unit is missing.)

Python Source codes

```
'''
01286121 Computer Programming, Software Engineering Program
For Year 1, Semester 1
Project: SciCal:Scientific Calculator Development
Author: Thura Aung (66011606@kmitl.ac.th)
Repository: https://github.com/ThuraAung1601/mySciCal/blob/main/scientific_calculator.py
'''
import sys
import subprocess
from abc import ABC, abstractmethod
from tkinter import *
from tkinter import messagebox
import math

# Check for numpy module
try:
    import numpy as np
except ImportError:
    print(">> 'numpy' module is missing!\n" +
          "Trying to install required module: numpy")
    subprocess.check_call([sys.executable, "-m", "pip", "install", "numpy"])
    print()
finally:
    import numpy as np

# Check for matplotlib module
try:
    import matplotlib.pyplot as plt
except ImportError:
    print(">> 'matplotlib' module is missing!\n" +
```

```
        "Trying to install required module: matplotlib")
    subprocess.check_call([sys.executable, "-m", "pip", "install", "matplotlib"])
    print()
finally:
    import matplotlib.pyplot as plt

class CalculatorInterface(ABC):
    """
    Abstract base class defining the interface for a calculator.

    Attributes:
    - colors (dict): Dictionary containing color codes for GUI elements.

    Methods:
    - __init__(self): Constructor method initializing color codes for GUI elements.
    - display_about(self): Abstract method to display information about the calculator.
    - reset(self): Abstract method to reset the calculator.
    - show_history(self): Abstract method to display calculation history.
    """
    def __init__(self):
        self.colors = {
            "light_grey": "#B8B8B8",
            "dark_grey": "#535353",
            "light_green": "#A3CDA8",
            "dark_green": "#729177",
            "orange": "#FFA662",
            "dark_orange": "#C48542"
        }

    @abstractmethod
    def display_about(self):
```

```
        """
        Abstract method to display information about the calculator.
        """
        pass

    @abstractmethod
    def reset(self):
        """
        Abstract method to reset the calculator.
        """
        pass

    @abstractmethod
    def show_history(self):
        """
        Abstract method to display calculation history.
        """
        pass

def is_number(value):
    """
    Checks if the given value is a number (integer or float).

    Args:
        value (str): The input value to be checked.

    Returns:
        bool: True if the input value is a number, False otherwise.
    """
    if value.isdecimal() or (value.count('.') == 1 and value.replace('.', '').isdecimal()):
        return True
```

```
    return False


def decimal2Scientific(value):
    """
    Converts a decimal value into scientific notation.

    Args:
        value (float/int): The decimal value to be converted.

    Returns:
        str/float/int: The value in scientific notation or the original value if conversion is not
needed.
    """
    if len(str(int(value))) > 1:
        prefix = value / (10 ** (len(str(int(value))) - 1))
        return f"{prefix}E{len(str(int(value)))-1}"
    return value


class Operations(object):
    """
    Class Operations performs various mathematical operations and equation solving.

    Attributes:
        result (int/float): Result of the equation solving operation.
        deg_or_rad (str): String indicating whether calculations are in degrees or radians.

    Methods:
        equation_solver: Solves mathematical equations and expressions.
        log_value: Computes the logarithm of a given value.
        ln_value: Computes the natural logarithm of a given value.
        factorial_value: Computes the factorial of a given value.
```

```
    squareRoot_value: Computes the square root of a given value.
    absolute_value: Computes the absolute value of a given value.
    ceil_value: Computes the ceiling value of a given value.
    floor_value: Computes the floor value of a given value.
    radian_converter: Converts an angle from degrees to radians.
    degree_converter: Converts an angle from radians to degrees.
    sin_value: Computes the sine of a given angle.
    cos_value: Computes the cosine of a given angle.
    tan_value: Computes the tangent of a given angle.
    asin_value: Computes the arcsine of a given angle.
    acos_value: Computes the arccosine of a given angle.
    atan_value: Computes the arctangent of a given angle.
"""
def __init__(self):
    self.result = 0
    self.deg_or_rad = 'radian'

def equation_solver(self, equation):
    log = self.log_value
    ln = self.ln_value
    fact = self.factorial_value
    sqrt = self.squareRoot_value
    abs = self.absolute_value
    ceil = self.ceil_value
    floor = self.floor_value

    sin = self.sin_value
    cos = self.cos_value
    tan = self.tan_value
    asin = self.asin_value
    acos = self.acos_value
```

```
    atan = self.atan_value

    e = math.e
    pi = math.pi
    x = "x"
    equation = equation.replace('π', 'pi')
    equation = equation.replace('^', '**')

    try:
        if "x" in equation:
            messagebox.showerror("Error", "Equations cannot be solved.")
            return ""
        else:
            return eval(equation)
    except:
        return "Error: Cannot be calculated."

@staticmethod
def log_value(value):
    if value == 0:
        return "Undefined"
    base = 10
    result = math.log(value, base)
    return result

@staticmethod
def ln_value(value):
    if value == 0:
        return "Undefined"
    base = math.e
    result = math.log(value, base)
```

```
    return result

@staticmethod
def factorial_value(value):
    if int(value) < 0:
        return "Invalid Input."
    result = 1
    for i in range(2, value + 1):
        result *= i
    return result

@staticmethod
def squareRoot_value(value):
    result = value ** (1/2)
    return result

@staticmethod
def absolute_value(value):
    result = abs(value)
    return result

@staticmethod
def ceil_value(value):
    result = math.ceil(value)
    return result

@staticmethod
def floor_value(value):
    result = int(value)
    return result
```

```
@staticmethod
def radian_converter(angle):
    return math.radians(angle)

@staticmethod
def degree_converter(angle):
    return math.degrees(angle)

def sin_value(self, angle):
    if self.deg_or_rad == 'degree':
        return math.sin(self.radian_converter(angle))
    return math.sin(angle)

def cos_value(self, angle):
    if self.deg_or_rad == 'degree':
        return math.cos(self.radian_converter(angle))
    return math.cos(angle)

def tan_value(self, angle):
    if self.deg_or_rad == 'degree':
        return math.tan(self.radian_converter(angle))
    return math.tan(angle)

def asin_value(self, angle):
    if self.deg_or_rad == 'degree':
        return math.asin(self.radian_converter(angle))
    return math.asin(angle)

def acos_value(self, angle):
    if self.deg_or_rad == 'degree':
        return math.acos(self.radian_converter(angle))
```

```
        return math.acos(angle)

    def atan_value(self, angle):
        if self.deg_or_rad == 'degree':
            return math.atan(self.radian_converter(angle))
        return math.atan(angle)

class BackEnd(Operations):
    """
    Class BackEnd handles the back-end operations of the scientific calculator.

    Attributes:
        ops (list): List of mathematical operators.
        nums (str): String containing numeric characters.
        funcs (list): List of mathematical functions.
        allow_operator (bool): Control the input of mathematical operators.
        allow_constants (bool): Manage the input of specific mathematical constants.
        allow_any (bool): Control the ability to input any character.
        angle_unit (StringVar): String variable to store the angle unit.
        cursor (int): Cursor position for display text.
        display_text (list): List containing the text displayed in the calculator.
        textvar (StringVar): String variable used for the calculator's display.

    Methods:
        operator_counter: Counts the number of operators in the expression.
        append_char: Appends characters to the display in the calculator.
        clear_text: Clears the displayed text and resets parameters.
        backspace_text: Handles the backspace action for deleting characters.
        send_press: Handles the button presses and performs respective actions.
    """
    def __init__(self):
```

```
        Operations.__init__(self)
        self.ops = ['+', '-', '*', '**', '^', '/', '//', '%']
        self.nums = '12345678900'
        self.funcs = ['asin', 'sin', 'acos', 'cos', 'atan', 'tan', 'fact', 'log', 'ln', 'sqrt', 'sqr', 'abs', 'ceil',
'floor']
        # to control the input of mathematical operators (+, -, *, /, etc.)
        self.allow_operator = False
        # to manage the input of specific mathematical constants - 'pi' and 'e'.
        self.allow_constants = True
        # to to control the ability to input any character
        self.allow_any = True

        self.angle_unit = StringVar()
        self.cursor = 0
        self.display_text = []
        self.textvar = StringVar()

    def operator_counter(self, expr):
        count = 0
        for i in expr:
            if i in self.ops + self.funcs:
                count += 1
        return count

    def append_char(self, *char):
        # append each character to display in textbox
        for i in char:
            a = len(self.display_text)
            self.display_text.insert(a + self.cursor, i)

    def clear_text(self):
```

```
        # delete the display text
        self.display_text.clear()
        # reset the parameters for display
        self.allow_operator, self.allow_constants, self.allow_any = False, True, True
        self.cursor = 0

    def backspace_text(self):
        # to check where the backspace action should be performed
        index = len(self.display_text) + self.cursor - 1
        if self.display_text[index] == '(':
            # remove '('
            self.display_text.pop(index)
            # increase cursor to check next character
            self.cursor += 1
            # redo
            self.backspace_text()
            # check the ( at the function eg. sqrt() and delete (
            if self.display_text and self.display_text[len(self.display_text) + self.cursor - 1] in
self.funcs:
                self.backspace_text()
        else:
            index = len(self.display_text) + self.cursor - 1
            if is_number(self.display_text[index]) and len(self.display_text[index]) > 1:
                self.display_text[index] = self.display_text[index][:-1]
            else:
                self.display_text.pop(len(self.display_text) + self.cursor - 1)

    def send_press(self, button):
        # limit - prevent user from entering many operators
        if button in self.ops + self.funcs and (self.operator_counter(self.display_text) > 14):
            return 0
```

```
        # to put () for the functions
        if button in self.funcs:
            if not self.allow_operator and self.allow_any:
                self.append_char(button, '(', ')')
                self.cursor -= 1
        # for showing the equation to graph
        elif button == 'x':
            self.append_char(button)
            self.allow_operator, self.allow_any, self.allow_constants = True, True, True
        # for power function
        elif button == 'pow':
            if self.allow_operator:
                self.append_char('^')
                self.allow_operator, self.allow_any, self.allow_constants = False, True, True
        # for exponential function
        elif button == 'exp':
            if self.allow_operator and self.display_text[len(self.display_text) + self.cursor - 1] !=
'e':
                self.append_char('E')
                self.allow_operator, self.allow_any, self.allow_constants = False, True, False
        # for arthmetic operations
        elif button in self.ops:
            if self.allow_operator:
                if button == "**":
                    self.append_char('^')
                else:
                    self.append_char(button)
                self.allow_operator, self.allow_any, self.allow_constants = False, True, True
            else:
                if button == '-' and ((self.display_text and self.display_text[len(self.display_text) +
self.cursor - 1] != '-') or (not self.display_text)):
```

```
            self.append_char('-')
            self.allow_operator, self.allow_any, self.allow_constants = False, True, True
        # for constants
        elif button in ('e', 'pi'):
            if self.allow_constants:
                if button == 'pi':
                    self.append_char('π')
                else:
                    self.append_char(button)
                self.allow_operator, self.allow_any, self.allow_constants = True, False, False
        # for numbers
        elif button in self.nums:
            if self.allow_any:
                if not self.display_text:
                    self.append_char(button)
                else:
                    if not is_number(self.display_text[len(self.display_text) + self.cursor - 1]):
                        self.append_char(button)
                    else:
                        self.display_text[len(self.display_text) + self.cursor - 1] =
self.display_text[len(self.display_text) + self.cursor - 1] + str(button)
                self.allow_constants = False
                self.allow_operator = True
        # for clear expression
        elif button == 'C':
            self.clear_text()
        # for getting the answer
        elif button == '=':
            if self.display_text:
                print(self.display_text)
                self.deg_or_rad = self.angle_unit.get()
```

```
            b = self.equation_solver(self.textvar.get())
            self.display_text = [str(b)]
            if type(b) is str:
                self.allow_operator, self.allow_any, self.allow_constants = False, False, False
                self.cursor = 0
            else:
                if b >= 10 ** 10000:
                    self.display_text = ['Overflow']
                    self.allow_any, self.allow_constants, self.allow_constants = False, False, False
                else:
                    if 10 ** 45 <= b < 10 ** 10000:
                        self.display_text = [str(self.dec_to_e(b))]
                    self.allow_operator, self.allow_any, self.allow_constants = True, False, False
                    self.cursor = 0

                    # append the result to history.txt
                    with open('history.txt', 'a') as history_file:
                        history_file.write(f"{self.textvar.get()} = {b}\n")
        # for float point
        elif button == '.':
            if self.display_text and self.display_text[len(self.display_text) + self.cursor -
1].isdecimal():
                self.display_text[len(self.display_text) + self.cursor - 1] += '.'
        # for natural number power
        elif button in ['10^x', 'e^x']:
            if not self.allow_operator and self.allow_any:
                self.append_char(button.split('^')[0], '^')
                self.allow_operator, self.allow_any, self.allow_constants = False, True, True
        # for power of 2
        elif button == 'x^2':
            if self.allow_operator:
```

```
            self.append_char('^', '2')
            self.allow_operator, self.allow_any, self.allow_constants = True, False, False
        # for ()
        elif button == '()':
            if not self.allow_operator and self.allow_any:
                self.append_char('(', ')')
                self.cursor -= 1
                self.allow_operator, self.allow_any, self.allow_constants = False, True, True
        # backspace - for delete one last character
        elif button == 'backspace':
            if self.display_text:
                self.backspace_text()
            curr = len(self.display_text) + self.cursor - 1
            if not self.display_text:
                self.clear_text()
            elif is_number(self.display_text[curr]):
                self.allow_operator, self.allow_any, self.allow_constants = True, True, False
            elif self.display_text[curr] in self.ops + ['(']:
                self.allow_operator, self.allow_any, self.allow_constants = False, True, True
            elif self.display_text[curr] in 'πe':
                self.allow_operator, self.allow_any, self.allow_constants = True, False, False
        # for root
        elif button == 'root':
            if self.allow_operator:
                self.append_char('^', '(', '1', '/', ')')
                self.cursor -= 1
                self.allow_operator, self.allow_any, self.allow_constants = False, True, True
        # for any character
        else:
            self.append_char(button)
            self.allow_operator = True
```

```
        self.textvar.set("".join(self.display_text))

class UnitConverter(CalculatorInterface, BackEnd):
    """
    Class implementing unit conversion via a graphical user interface (GUI).

    Inherits from BackEnd class to utilize conversion methods.

    Attributes:
    - root (Tk): The root Tkinter instance for the GUI.
    - unitDict (dict): Dictionary of conversion factors for different units.
    - length_units (list): List of length-related unit types.
    - temperature_units (list): List of temperature-related unit types.
    - area_units (list): List of area-related unit types.
    - volume_units (list): List of volume-related unit types.
    - weight_units (list): List of weight-related unit types.
    - SELECTIONS (list): List of unit selections for the GUI dropdown menu.
    - buttons (list): List containing buttons layout for the GUI.

    Methods:
    - __init__(self, root): Constructor method initializing the GUI and variables.
    - create_gui(self): Creates the graphical user interface.
    - create_buttons(self): Generates buttons for the GUI layout for user input.
    - display_about(self): Displays information about the Unit Converter.
    - reset(self): Resets the input and output fields and clears the history.
    - convert(self): Performs unit conversion based on user input and selected units.
    - add_digit(self, digit): Appends a digit to the input field.
    - delete_character(self): Deletes the last character from the input field.
    - update_history(self): Updates the conversion history displayed in the GUI.
    - load_conversion_history(self): Loads conversion history from a file.
    - save_conversion_history(self): Saves conversion history to a file.
```

```
- show_history(self): Displays conversion history in a new window.
"""
def __init__(self, root):
    CalculatorInterface.__init__(self)
    BackEnd.__init__(self)
    self.root = root
    # dictionary of conversion factors
    self.unitDict = {
        "millimeter" : 0.001,
        "centimeter" : 0.01,
        "meter" : 1.0,
        "kilometer" : 1000.0,
        "foot" : 0.3048,
        "mile" : 1609.344,
        "yard" : 0.9144,
        "inch" : 0.0254,
        "square meter" : 1.0,
        "square kilometer" : 1000000.0,
        "square centimeter" : 0.0001,
        "square millimeter" : 0.000001,
        "are" : 100.0,
        "hectare" : 10000.0,
        "acre" : 4046.856,
        "square mile" : 2590000.0,
        "square foot" : 0.0929,
        "cubic meter" : 1000.0,
        "cubic centimeter" : 0.001,
        "litre" :  1.0,
        "millilitre" : 0.001,
        "gallon" : 3.785,
        "gram" : 1.0,
```

```
    "kilogram" : 1000.0,
    "milligram" : 0.001,
    "quintal" : 100000.0,
    "ton" : 1000000.0,
    "pound" : 453.592,
    "ounce" : 28.3495
}

# charts for units conversion
self.length_units = [
    "millimeter", "centimeter", "meter", "kilometer", "foot", "mile", "yard", "inch"
    ]
self.temperature_units = [
    "celsius", "fahrenheit"
]
self.area_units = [
    "square meter", "square kilometer", "square centimeter", "square millimeter",
    "are", "hectare", "acre", "square mile", "square foot"
    ]
self.volume_units = [
    "cubic meter", "cubic centimeter", "litre", "millilitre", "gallon"
]
self.weight_units = [
    "gram", "kilogram", "milligram", "quintal", "ton", "pound", "ounce"
]

# creating the list of options for selection menu
self.SELECTIONS = [
    "Select Unit",
    "millimeter",
    "centimeter",
```

```
    "meter",
    "kilometer",
    "foot",
    "mile",
    "yard",
    "inch",
    "celsius",
    "fahrenheit",
    "square meter",
    "square kilometer",
    "square centimeter",
    "square millimeter",
    "are",
    "hectare",
    "acre",
    "square mile",
    "square foot",
    "cubic meter",
    "cubic centimeter",
    "litre",
    "millilitre",
    "gallon"
    "gram",
    "kilogram",
    "milligram",
    "quintal",
    "ton",
    "pound",
    "ounce"
]
```

```
        self.buttons = [
            ['7', '8', '9'],
            ['4', '5', '6'],
            ['1', '2', '3'],
            ['OK', '0', '←']
        ]

        self.create_gui()

        self.conversion_history = []
        self.load_conversion_history()

        self.history_text = Text(self.root, wrap=WORD)
        self.history_text.grid(row=2, column=0, padx=10, pady=10, columnspan=4)
        self.history_text.config(state=DISABLED)

    def create_gui(self):
        self.root.title("Unit Converter")
        self.root.geometry("450x320")
        self.root.resizable(False, False)
        self.root.configure(bg=self.colors["light_grey"])

        self.top_frame = Frame(self.root, bg=self.colors["light_grey"])
        self.top_frame.grid(row=0, column=0, sticky=NSEW)

        self.about = Button(self.top_frame, text="ABOUT", bg=self.colors["light_grey"],
fg=self.colors["dark_grey"], font=("Segoe UI", 14, 'bold'), borderwidth=3, padx=4,
command=self.display_about)
        self.reset_button = Button(self.top_frame, text="RESET", bg=self.colors["light_grey"],
fg=self.colors["dark_grey"], font=("Segoe UI", 14, 'bold'), borderwidth=3, padx=4,
command=self.reset)
        self.about.grid(row=1, column=1)
```

```
self.reset_button.grid(row=1, column=3)

self.history_button = Button(
    self.top_frame,
    text="Show History",
    bg=self.colors["light_grey"],
    fg=self.colors["dark_grey"],
    font=("Segoe UI", 14, 'bold'),
    borderwidth=3,
    padx=4,
    command=self.show_history
)
self.history_button.grid(row=1, column=2)

self.bottom_frame = Frame(self.root, bg=self.colors["light_grey"])
self.bottom_frame.grid(row=1, column=0, sticky=NSEW)

self.input_value = StringVar()
self.output_value = StringVar()
self.input_value.set(self.SELECTIONS[0])
self.output_value.set(self.SELECTIONS[0])

# creating the labels for the body of the main window
self.input_label = Label(
    self.bottom_frame,
    text = "From:",
    bg = self.colors["light_grey"],
    fg = self.colors["dark_grey"],
    font=("Segoe UI", 14, 'bold')
)
self.output_label = Label(
```

```
            self.bottom_frame,
            text = "To:",
            bg = self.colors["light_grey"],
            fg = self.colors["dark_grey"],
            font=("Segoe UI", 14, 'bold')
        )

        # using the grid() method to set the position of the above labels
        self.input_label.grid(row = 0, column = 0, padx = 20, pady = 20, sticky = NSEW)
        self.output_label.grid(row = 1, column = 0, padx = 20, pady = 20, sticky = NSEW)

        self.input_field = Entry(self.bottom_frame, bg="#e8f8f5", fg=self.colors['dark_grey'])
        self.output_field = Entry(self.bottom_frame, bg="#e8f8f5", fg=self.colors['dark_grey'])
        self.input_field.grid(row=0, column=1)
        self.output_field.grid(row=1, column=1)

        self.input_menu = OptionMenu(self.bottom_frame, self.input_value, *self.SELECTIONS)
        self.output_menu = OptionMenu(self.bottom_frame, self.output_value,
*self.SELECTIONS)
        self.input_menu.grid(row=0, column=2, padx=20)
        self.output_menu.grid(row=1, column=2, padx=20)

        self.create_buttons()

    def create_buttons(self):
        self.buttons_frame = Frame(self.bottom_frame, bg=self.colors["dark_grey"])
        for i, row_buttons in enumerate(self.buttons):
            for j, button_text in enumerate(row_buttons):
                button = Button(
                    self.buttons_frame,
                    text=button_text,
```

```
                bg=self.colors["light_grey"],
                borderwidth=3,
                font=('Segoe UI', 16, 'bold')
            )
            if button_text.isdigit():
                button.config(command=lambda x=button_text: self.add_digit(x))
            elif button_text == "OK":
                button.config(command=lambda x=button_text: self.convert())
            elif button_text == "←":
                button.config(command=lambda x=button_text: self.delete_character())

            button.grid(row=i, column=j, sticky=NSEW, padx=4, pady=4)


        self.buttons_frame.grid(row=2, column=1, sticky=NSEW)

    def display_about(self):
        about_text = "Unit Converter\nVersion 1.0\nCreated by Thura Aung
<66011606@kmitl.ac.th>"
        messagebox.showinfo("ABOUT", about_text)

    def reset(self):
        self.input_field.delete(0, END)
        self.output_field.delete(0, END)
        self.input_value.set(self.SELECTIONS[0])
        self.output_value.set(self.SELECTIONS[0])
        self.input_field.focus_set()

        # Clear conversion history when resetting
        self.conversion_history = []
        self.update_history()
```

```
        self.save_conversion_history()


    def convert(self):
        # getting the string from entry field and converting it into float
        inputVal = float(self.input_field.get())
        # getting the values from selection menus
        input_unit = self.input_value.get()
        output_unit = self.output_value.get()

        # list of the required combination of the conversion factors
        conversion_factors = [input_unit in self.length_units and output_unit in
self.length_units,
        input_unit in self.weight_units and output_unit in self.weight_units,
        input_unit in self.temperature_units and output_unit in self.temperature_units,
        input_unit in self.area_units and output_unit in self.area_units,
        input_unit in self.volume_units and output_unit in self.volume_units]

        if any(conversion_factors): # If both the units are of same type, perform the conversion
            if input_unit == "celsius" and output_unit == "fahrenheit":
                self.output_field.delete(0, END)
                self.output_field.insert(0, (inputVal * 1.8) + 32)
            elif input_unit == "fahrenheit" and output_unit == "celsius":
                self.output_field.delete(0, END)
                self.output_field.insert(0, (inputVal - 32) * (5/9))
            else:
                self.output_field.delete(0, END)
                self.output_field.insert(0, round(inputVal * self.unitDict[input_unit] /
self.unitDict[output_unit], 5))

        elif input_unit == self.SELECTIONS[0] or output_unit == self.SELECTIONS[0]:
            if input_unit == self.SELECTIONS[0]:
```

```
            # displaying error if units are of different types
            self.output_field.delete(0, END)
            messagebox.showwarning("Warning", "Give the input unit type.")
        else:
            # displaying error if units are of different types
            self.output_field.delete(0, END)
            messagebox.showwarning("Warning", "Give the output unit type.")
    else:
        # displaying error if units are of different types
        self.output_field.delete(0, END)
        messagebox.showerror("Error", f"{input_unit} and {output_unit} are different types.")

    # Append conversion details to the history list
    history_entry = f"{inputVal} {input_unit} = {self.output_field.get()} {output_unit}\n"
    self.conversion_history.append(history_entry)

    # Update the conversion history displayed in the Text widget
    self.update_history()
    self.save_conversion_history()

def add_digit(self, digit):
    current_value = self.input_field.get()
    self.input_field.insert(END, digit)

def delete_character(self):
    current_value = self.input_field.get()
    self.input_field.delete(len(current_value) - 1, END)

def update_history(self):
    # Clear the history text widget and update it with the latest conversion history
    self.history_text.delete(1.0, END)
```

```
    for entry in self.conversion_history:
        self.history_text.insert(END, entry)

def load_conversion_history(self):
    try:
        with open("conversion_history.txt", "r") as file:
            self.conversion_history = file.readlines()
    except FileNotFoundError:
        self.conversion_history = []

def save_conversion_history(self):
    with open("conversion_history.txt", "w") as file:
        file.writelines(self.conversion_history)

def show_history(self):
        # Create a new window for history
        history_window = Toplevel(self.root)
        history_window.title("Conversion History")
        history_window.geometry("400x300")

        # Read history from the file
        try:
            with open('conversion_history.txt', 'r') as history_file:
                history_data = history_file.readlines()
        except FileNotFoundError:
            history_data = []
        except Exception as e:
            # Handle other exceptions with a generic error message
            messagebox.showerror("Error", f"An error occurred: {e}")

        # Display history in a Text widget
```

```
        history_text = Text(history_window, wrap=WORD)
        history_text.pack(expand=YES, fill=BOTH)

        for entry in history_data:
            history_text.insert(END, entry)

        # Disable text editing in the history window
        history_text.config(state=DISABLED)

class App(CalculatorInterface, BackEnd):
    """
    Class representing a Scientific Calculator application using Tkinter GUI.

    Inherits from BackEnd class for backend calculation functionalities.

    Attributes:
    - colors (dict): Dictionary containing color codes for GUI elements.
    - buttons (list): Nested list containing buttons layout for the calculator.
    - inv_buttons (dict): Dictionary containing inverse function mappings for trigonometric and other operations.
    - inv_rows (tuple): Tuple specifying rows where inverse functions are active.
    - inv_cols (tuple): Tuple specifying columns where inverse functions are active.
    - w (Tk): The main Tkinter instance for the GUI.

    Methods:
    - __init__(self, w): Constructor method initializing the calculator interface.
    - meta_window(self): Configures and sets metadata for the calculator window.
    - place_mainframes(self): Places and configures main frames for top and bottom sections of the calculator.
    - place_tf_widgets(self): Places widgets in the top frame of the calculator.
    - copy_result(self): Copies the calculation result to the clipboard.
```

```
    - toggle_angle(self): Toggles between radian and degree angles.
    - display_about(self): Displays information about the Scientific Calculator.
    - graphing(self): Generates a graph based on the entered expression.
    - show_history(self): Displays calculation history in a separate window.
    - to_unit_converter(self): Opens a Unit Converter window.
    - config_tf_widgets(self): Configures widgets in the top frame of the calculator.
    - inverse(self): Handles the toggling of inverse functions in the calculator.
    - place_bf_widgets(self): Places and configures buttons in the bottom frame of the
calculator.
    - start(self): Starts the calculator application.
    """
    def __init__(self, w):
        CalculatorInterface.__init__(self)
        BackEnd.__init__(self)

        self.buttons = [
            ['INV', 'π', 'e', '()', '←', 'C', '/', '//'],
            ['sin', 'cos', 'tan', '7', '8', '9', '*', '**'],
            ['log', 'ln', 'fact', '4', '5', '6', '+', '%'],
            ['pow', 'sqrt', 'abs', '1', '2', '3', False, '-'],
            ['ceil', 'floor', 'exp', '0', '00', '.', '=', False]
        ]

        self.inv_buttons = {
            'sin': 'asin', 'cos': 'acos', 'tan': 'atan',
            'log': '10^x', 'ln': 'e^x', 'fact': 'x',
            'pow': 'root', 'sqrt': 'x^2', 'abs': 'abs'
        }

        # inverse key will be active from row 1 (2nd row) to row 4
        self.inv_rows = (1, 4)
```

```
# inverse key will be active from column 0 (1st column) to column 3
self.inv_cols = (0, 3)

self.w = w
self.meta_window()

self.top_frame = Frame(self.w, bg=self.colors["light_grey"])
self.bottom_frame = Frame(self.w, bg=self.colors["dark_grey"])
self.place_mainframes()

# top frame widgets
self.tf_buttons = Frame(self.top_frame)
self.tf_copy_button = Button(self.top_frame)
self.tf_about_button = Button(self.tf_buttons)
self.tf_graph_button = Button(self.tf_buttons)
self.tf_history_button = Button(self.tf_buttons)
self.tf_reset_button = Button(self.tf_buttons)
self.tf_menu_button = Button(self.tf_buttons)
self.tf_confirm_text = Label(self.tf_buttons)
self.tf_textbox = Label(self.top_frame)
self.tf_angle_selection_frame = Frame(self.top_frame)
self.tf_asf_text = Label(self.tf_angle_selection_frame)
self.tf_asf_rad_choice = Radiobutton(self.tf_angle_selection_frame)
self.tf_asf_deg_choice = Radiobutton(self.tf_angle_selection_frame)

self.place_tf_widgets()
self.config_tf_widgets()

# bottom frame widgets
self.bf_buttons = []
self.inverse_State = False
```

```
        self.place_bf_widgets()

    def meta_window(self):
        self.w.geometry("640x440")
        self.w.title("Scientific Calculator")
        self.w.config(bg=self.colors["dark_grey"])
        self.w.resizable(False, False)
        # column 0 and row 1 will resize with window
        self.w.columnconfigure(0, weight=1)
        self.w.rowconfigure(1, weight=1)

    def place_mainframes(self):
        self.top_frame.grid(row=0, column=0, sticky=NSEW)
        self.top_frame.columnconfigure(0, weight=1)

        self.bottom_frame.grid(row=1, column=0, sticky=NSEW, padx=8, pady=8)
        self.bottom_frame.grid_propagate(False)
        for i in range(5):
            self.bottom_frame.rowconfigure(i, weight=1, uniform="xyz")
        for i in range(8):
            self.bottom_frame.columnconfigure(i, weight=1, uniform="xyz")

    def place_tf_widgets(self):
        self.tf_buttons.grid(row=0, column=0, pady=4, sticky=NSEW)
        self.tf_buttons.rowconfigure(0, weight=1)
        self.tf_buttons.grid_propagate(False)
        self.tf_copy_button.grid(row=1, column=1, padx=4, sticky=NSEW)
        self.tf_about_button.grid(row=0, column=1, padx=4, sticky=NSEW)
        self.tf_graph_button.grid(row=0, column=2, padx=4, sticky=NSEW)
        self.tf_history_button.grid(row=0, column=3, padx=4, sticky=NSEW)
        self.tf_reset_button.grid(row=0, column=4, padx=4, sticky=NSEW)
```

```
        self.tf_menu_button.grid(row=0, column=5, padx=4, sticky=NSEW)
        self.tf_textbox.grid(row=1, column=0, sticky=NSEW, padx=4, pady=4)

        self.tf_angle_selection_frame.grid(row=2, column=0, sticky=W, pady=8, padx=4)
        self.tf_asf_text.grid(row=0, column=0, padx=4)
        self.tf_asf_rad_choice.grid(row=0, column=1, padx=16)
        self.tf_asf_deg_choice.grid(row=0, column=2, padx=4)

    def copy_result(self):
        text = self.textvar.get()
        self.w.clipboard_clear()
        self.w.clipboard_append(text)
        self.w.update()

        self.tf_confirm_text.grid(row=0, column=4, padx=4, sticky=NSEW)
        self.tf_confirm_text.after(2000, lambda: self.tf_confirm_text.grid_remove())

    def toggle_angle(self):
        curr_ang = self.angle_unit.get()
        print(f"You selected {curr_ang}")
        if curr_ang == "degree":
            self.tf_asf_deg_choice.config(selectcolor="lime")
            self.tf_asf_rad_choice.config(selectcolor=self.colors["light_grey"])
        else:
            self.tf_asf_deg_choice.config(selectcolor=self.colors["light_grey"])
            self.tf_asf_rad_choice.config(selectcolor="lime")

    def display_about(self):
        about_text = "Scientific Calculator\nVersion 1.0\nCreated by Thura Aung
<66011606@kmitl.ac.th>"
        messagebox.showinfo("About", about_text)
```

```
def graphing(self):
    if self.display_text:
        try:
            # Extract the expression from the display text
            expression = self.textvar.get()
            # Generate x values
            x_values = np.linspace(-10, 10, 400)
            # Evaluate the expression for each x value
            y_values = [self.equation_solver(expression.replace('x', str(x))) for x in x_values]
            # Plot the graph
            plt.plot(x_values, y_values, label=expression)
            plt.xlabel("x")
            plt.ylabel("y")
            plt.grid(True)
            plt.legend()  # Set legend
            plt.show()
        except Exception as e:
            messagebox.showerror("Error", f"Error during graphing: {e}")

def show_history(self):
        # Create a new window for history
        history_window = Toplevel(self.w)
        history_window.title("Calculation History")
        history_window.geometry("400x300")

        # Read history from the file
        try:
            with open('history.txt', 'r') as history_file:
                history_data = history_file.readlines()
        except FileNotFoundError:
```

```
            history_data = []

        # Display history in a Text widget
        history_text = Text(history_window, wrap=WORD)
        history_text.pack(expand=YES, fill=BOTH)

        for entry in history_data:
            history_text.insert(END, entry)

        # Disable text editing in the history window
        history_text.config(state=DISABLED)

def reset(self):
    self.textvar.set('')
    try:
        with open('history.txt', 'w') as history_file:
            history_file.close()
    except Exception as e:
        messagebox.showerror("Error", f"An error occurred: {e}")

def to_unit_converter(self):
    self.custom_root = Toplevel(self.w)
    unit_converter = UnitConverter(self.custom_root)

def config_tf_widgets(self):
    self.tf_buttons.config(bg=self.colors["light_grey"], height=32)
    self.tf_copy_button.config(
        text="COPY",
        bg=self.colors["dark_grey"],
        activebackground=self.colors["light_grey"],
        fg=self.colors["dark_grey"],
```

```
    font=("Segoe UI", 12, 'bold'),
    borderwidth=3,
    command=lambda: self.copy_result())
self.tf_about_button.config(
    text="ABOUT",
    bg=self.colors["dark_grey"],
    activebackground=self.colors["light_grey"],
    fg=self.colors["dark_grey"],
    font=("Segoe UI", 12, 'bold'),
    borderwidth=3,
    command=lambda: self.display_about())
self.tf_graph_button.config(
    text="GRAPHING",
    bg=self.colors["dark_grey"],
    activebackground=self.colors["light_grey"],
    fg=self.colors["dark_grey"],
    font=("Segoe UI", 12, 'bold'),
    borderwidth=3,
    command=lambda: self.graphing())
self.tf_history_button.config(
    text="HISTORY",
    bg=self.colors["dark_grey"],
    activebackground=self.colors["light_grey"],
    fg=self.colors["dark_grey"],
    font=("Segoe UI", 12, 'bold'),
    borderwidth=3,
    command=lambda: self.show_history())
self.tf_reset_button.config(
    text="RESET",
    bg=self.colors["dark_grey"],
    activebackground=self.colors["light_grey"],
```

```
        fg=self.colors["dark_grey"],
        font=("Segoe UI", 12, 'bold'),
        borderwidth=3,
        command=self.reset)
    self.tf_menu_button.config(
        text="UNIT CONVERTER",
        bg=self.colors["dark_grey"],
        activebackground=self.colors["light_grey"],
        fg=self.colors["dark_grey"],
        font=("Segoe UI", 12, 'bold'),
        borderwidth=3,
        command=lambda: self.to_unit_converter())
    self.tf_confirm_text.config(
        text="Copied to Clipboard",
        bg="white",
        fg=self.colors["dark_grey"],
        font=('Segoe UI', 10))
    self.tf_textbox.update()
    self.tf_textbox.config(
        height=2,
        font=("Courier New", 16, "bold"),
        textvariable=self.textvar,
        fg=self.colors["dark_grey"],
        bg="white",
        borderwidth=2,
        relief=SOLID,
        wraplength=self.tf_textbox.winfo_width() - 6,
        justify=RIGHT,
        anchor=NE
    )
```

```
    self.tf_angle_selection_frame.config(bg=self.colors["light_grey"])
    self.tf_asf_text.config(
        text="Angle: ",
        bg=self.colors["light_grey"],
        font=('Segoe UI', 14, 'normal'))
    self.tf_asf_rad_choice.config(
        text="Radians",
        selectcolor=self.colors["light_grey"],
        activebackground=self.colors["light_grey"],
        font=('Segoe UI', 14, 'normal'),
        variable=self.angle_unit,
        value="radian",
        command=self.toggle_angle,
        bg=self.colors["light_grey"])
    self.tf_asf_deg_choice.config(
        text="Degrees",
        selectcolor="lime",
        activebackground=self.colors["light_grey"],
        font=('Segoe UI', 14, 'normal'),
        variable=self.angle_unit,
        value="degree",
        command=self.toggle_angle,
        bg=self.colors["light_grey"])
    self.angle_unit.set("degree")

def inverse(self):
    if not self.inverse_State:
        # for i in range(1,4):
        for i in range(*self.inv_rows):
            if i == 1:
                self.bf_buttons[i][2].config(
```

```
                text=self.inv_buttons[self.buttons[i][2]],
                bg=self.colors["light_green"],
                command=lambda x=self.inv_buttons[self.buttons[i][2]]: [self.send_press(x),
self.inverse()])
            for j in range(*self.inv_cols):
                self.bf_buttons[i][j].config(
                text=self.inv_buttons[self.buttons[i][j]],
                bg=self.colors["light_green"],
                command=lambda x=self.inv_buttons[self.buttons[i][j]]: [self.send_press(x),
self.inverse()])
        self.bf_buttons[0][0].config(bg=self.colors["dark_green"], relief=SUNKEN)
        self.inverse_State = True
    else:
        for i in range(*self.inv_rows):
            if i == 1:
                self.bf_buttons[i][2].config(
                text=self.buttons[i][2],
                bg=self.colors["light_grey"],
                command=lambda x=self.buttons[i][2]: self.send_press(x))
            for j in range(*self.inv_cols):
                self.bf_buttons[i][j].config(
                text=self.buttons[i][j],
                bg=self.colors["light_grey"],
                command=lambda x=self.buttons[i][j]: self.send_press(x))
        self.bf_buttons[0][0].config(bg=self.colors["light_green"], relief=RAISED)
        self.inverse_State = False

def place_bf_widgets(self):
    for i in range(len(self.buttons)):
        row = []
        for j in range(len(self.buttons[i])):
```

```
            if self.buttons[i][j]:
                b = Button(
                    self.bottom_frame,
                    text=self.buttons[i][j],
                    bg=self.colors["light_grey"],
                    borderwidth=3,
                    font=('Segoe UI', 16, 'bold'),
                    command=lambda x=self.buttons[i][j]: self.send_press(x))
                if (self.buttons[i][j].isdecimal() and self.buttons[i][j] != '00') or self.buttons[i][j] in
('+', '-', '=', '*', '/', '.', '()'):
                    self.w.bind(f"{self.buttons[i][j]}", lambda event, a=self.buttons[i][j]:
self.send_press(a))
                if j > 2 or (i == 0 and 0 < j < 3):
                    b.config(font=('Segoe UI', 22, 'bold'))
                row.append(b)
                b.grid(row=i, column=j, sticky=NSEW, padx=4, pady=4)

        self.bf_buttons.append(row)

    self.w.bind('<Return>', lambda event, a='=': self.send_press(a))
    self.w.bind('<BackSpace>', lambda event, a='backspace': self.send_press(a))
    self.w.bind('<Delete>', lambda event, a='C': self.send_press(a))

    self.bf_buttons[2][6].grid(row=2, column=6, rowspan=2)
    self.bf_buttons[4][6].grid(row=4, column=6, columnspan=2)
    self.bf_buttons[0][0].config(bg=self.colors["light_green"],
activebackground=self.colors["dark_green"])
    self.bf_buttons[0][5].config(bg=self.colors["orange"],
activebackground=self.colors["dark_orange"])
    self.bf_buttons[0][4].config(bg=self.colors["orange"],
activebackground=self.colors["dark_orange"])
    self.bf_buttons[0][1].config(command=lambda f='pi': self.send_press(f))
```

```
        self.bf_buttons[0][4].config(command=lambda f='backspace': self.send_press(f))

        self.bf_buttons[0][0].config(
            command=lambda x=self.buttons[0][0]: [print(f"You pressed {x}"), self.inverse()])

    def start(self):
        self.w.mainloop()


if __name__ == "__main__":
    root = Tk()
    calculator = App(root)
    calculator.start()
```