Chapter 2 - Scientific Calculator Project

This chapter will discuss the creation of a Scientific Calculator using Python and CustomTkinter. A good practice in software design is to define the initial features and specifications of the calculator application before starting development.

Features and Specifications:

- [x] Numeric keys - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- [x] Decimal point
- [x] Sign (+/-)
- [x] Basic math - addition, subtraction, multiplication, division
- [x] ln
- [x] log
- [x] 10^x
- [x] square root
- [x] Parentheses - ( )
- [x] n!
- [x] x^2
- [x] 1/x
- [x] absolute value
- [x] exp
- [x] mod(%)
- [x] pi
- [x] e
- [x] clear
- [x] backspace
- [x] Trigonometry
    - [x] sin, cos, tan
    - [ ] asin, acos, atan
    - [x] hyp
    - [x] sec, csc, cot
    - [x] radians, degrees
- [x] Memory
    - [x] MC

- MR
- M+
- M-
- MS
- Wide screen design
- Assume 10 rows x 10 columns

Optional Features

- ~~[ ] Graphing~~
- ~~[ ] Programming Language - Python?~~
- ~~[ ] Large equation library~~
- ~~[ ] RPN or Algebraic data entry~~

Project Setup

Open PyCharm and create a new project called ScientificCalculator using Python 3.11. From a terminal in PyCharm, install CustomTkinter

pip install customtkinter

The installation may already be satisfied by the installation during the Python3Tutorial.

Project Directory: D:/EETools/ScientificCalculator

GUI Design

CustomTkinter has 3 layout options: place, pack, and grid. For a calculator, the grid layout is optimal for a calculator interface. As defined in the specification, the initial grid design is 10 rows x 10 columns. The top two rows will be the display area. Rows 3 to 10 are buttons. Some of the buttons may be two columns in width or two rows in height.

Create a new file called scientific_calculator.py. We could create each button individually and assign it to a grid position. A more efficient method is to create and array of buttons and an array of button text which makes it easy for the programmer to visualize the buttons and assign test.

```
# Initialize a button array (list of lists)
self.buttons = [[None]*10 for _ in range(5)]

# Create buttons
self.button_text = [
    ["sin", "cos", "tan", "hyp", "MC", "MR", "M+", "M-", "MS", "bs"],  # Row 5
    ["sec", "csc", "cot", "eng", "sci", "clr", "7", "8", "9", "/"],  # Row 6
    ["exp", "mod", "pi", "e", "x<>y", "4", "4", "5", "6", "*"],  # Row 7
    ["x^2", "1/x", "|x|", "n!", "(", ")", "1", "2", "3", "-"],  # Row 8
    ["ln", "log", "10^x", "x^y", "sqrt", "+/-", "0", "=", ".", "+"],  # Row 9
]
```

We can display the 5x10 button array using two for loops.

```
# Display button in a 5x10 array
for x in range(5):
    for y in range(10):
        self.buttons[x][y] = ctk.CTkButton(self, text=self.button_text[x][y],
height=60,
                                           font=('arial',16),
                                           command=lambda x1=x, y1=y:
self.press(x1, y1))
        if 0 < x < 5 < y < 9:
            self.buttons[x][y].configure(fg_color="#3b3b3b")
        else:
            self.buttons[x][y].configure(fg_color="#7b7b7b")
        self.buttons[x][y].grid(row=x+4, column=y, padx=5, pady=5)
```

Finally, we need a single button callback function that can select the button that was pressed and print the button text, for now.

```
def press(self, x, y):
    print(self.button_text[x][y])
```

Nice! Here is the complete program
scientific_calculator.py

```
import customtkinter as ctk


class App(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("1525x460x10x10")
        self.title("Scientific Calculator")

        calculation = ""

        # Create a label to display results of the calculation
        label = ctk.CTkLabel(self, text="0.000000", padx=10, anchor=ctk.SE,
font=('arial',22), fg_color="transparent",
                             width=1500, height=60)
        label.grid(row=0, column=0, columnspan=10, rowspan=4, padx=10, pady=5)
        label.configure(bg_color="#555500")

        # Initialize a button array (list of lists)
        self.buttons = [[None]*10 for _ in range(5)]

        # Create buttons
        self.button_text = [
            ["sin", "cos", "tan", "hyp", "MC", "MR", "M+", "M-", "MS", "bs"],
# Row 5
            ["sec", "csc", "cot", "eng", "sci", "clr", "7", "8", "9", "/"],  #
Row 6
            ["exp", "mod", "pi", "e", "x<>y", "4", "4", "5", "6", "*"],  # Row
7
            ["x^2", "1/x", "|x|", "n!", "(", ")", "1", "2", "3", "-"],  # Row 8
            ["ln", "log", "10^x", "x^y", "sqrt", "+/-", "0", "=", ".", "+"],  #
Row 9
        ]

        # Display button in a 5x10 array
        for x in range(5):
            for y in range(10):
                self.buttons[x][y] = ctk.CTkButton(self,
text=self.button_text[x][y], height=60,
                                                   font=('arial',16),
                                                   command=lambda x1=x, y1=y:
self.press(x1, y1))
                if 0 < x < 5 < y < 9:
                    self.buttons[x][y].configure(fg_color="#3b3b3b")
                else:
                    self.buttons[x][y].configure(fg_color="#7b7b7b")
```

```
                self.buttons[x][y].grid(row=x+4, column=y, padx=5, pady=5)

    def press(self, x, y):
        print(self.button_text[x][y])


if __name__ == "__main__":
    app = App()
    app.mainloop()
```

Here are some additional tweaks:

- Adjust the window height
- Find unicode text for divide symbol
- Find unicode text for backspace key
- Find unicode text for square root key
- Find text for PI symbol

After a few tweaks to the button text array

```
        # Create buttons
        self.button_text = [
            ["sin", "cos", "tan", "hyp", "MC", "MR", "M+", "M-", "MS", "\u232B"],  # Row 5
            ["sec", "csc", "cot", "eng", "sci", "clr", "7", "8", "9", "\u00F7"],  # Row 6
            ["exp", "mod(%)", "\u03C0", "e", "x<>y", "4", "4", "5", "6", "x"],  # Row 7
            ["x^2", "1/x", "|x|", "n!", "(", ")", "1", "2", "3", "-"],  # Row 8
            ["ln", "log", "10^x", "x^y", "\u221A", "+/-", "0", "=", ".", "+"],
```

```
# Row 9
        ]
```

With the initial user-interface defined, we can proceed to implement the calculator logic.

Calculator Logic

The key to calculating the result of the expression in the label is to process eval(expression) if possible. eval will evaluate the expression using the Python math library but the expression must use the syntax defined for the math library. Many of the keys have the correct syntax however some have special cases to improve the text on the button or expression that cannot be directly calculated by eval(). Memory buttons are also a special case.

The label class has an argument called "textvariable" which can be set to a special string variable called "ctk.StringVar". The result of the eval() calculation will be stored in this variable called "self.equation". eval() will be called when the user presses the "=" key. A string variable called "self.expression" will contain the expression in the label to be evaluated, i.e. eval(self.expression). Memory value will be stored in "self.memory".

```
self.expression = ""
self.equation = ctk.StringVar()
self.sel = ""
self.memory = 0
```

Here is the update label definition with the textvariable set to self.equation.

```
label = ctk.CTkLabel(self, text="0.00", padx=10, anchor=ctk.SE, font=
('arial',22), textvariable=self.equation,
```

```
                        fg_color="transparent", width=1500, height=60)
```

Updated button text array

```
        self.button_text = [
            ["sin", "cos", "tan", "\u03C0", "MC", "MR", "M+", "M-", "MS",
"\u232B"],  # Row 5
            ["asin", "acos", "atan", "radians", "degrees", "CLR", "7", "8",
"9", "\u00F7"],  # Row 6
            ["sec", "csc", "cot", "e", "exp", "%", "4", "5", "6", "*"],  # Row
7
            ["x^2", "1/x", "|x|", "n!", "(", ")", "1", "2", "3", "-"],  # Row 8
            ["log", "log10", "10^x", "2\u03C0", "\u221A", "+/-", "0", "=", ".",
"+"],  # Row 9
        ]
```

The press() function will check for any special cases, memory operations, or simply update the label expression.

```
def press(self, x, y):
    # Check for special cases where eval can't evaluate the button text
    self.sel = self.button_text[x][y]  # self.sel is a short alias for
self.button_text[x][y]
    if self.sel == "CLR" or self.sel == "=" or self.sel == "\u00F7" or self.sel
== "1/x" or \
            self.sel == "\u232B" or self.sel == "\u03C0" or self.sel == "x^2"
or self.sel == "|x|" or \
            self.sel == "+/-" or self.sel == "n!" or self.sel == "10^x" or
self.sel == "\u221A" or \
            self.sel == "2\u03C0":
        self.special_case(x, y)
    elif self.sel == "MC" or self.sel == "MR" or self.sel == "M+" or self.sel
== "M-" or self.sel == "MS":
        self.memory_operation(x,y)
    else:
        self.expression += self.button_text[x][y]
        self.equation.set(self.expression)
```

The special_case() function detects the key pressed and processes the expression as needed. For example, pi is defined as a unicode character "\u03C0" on the key which displays the Greek letter for pi. However, math.pi is substituted for the Greek character in the expression.

```
    def special_case(self, x, y):
        if self.sel == "CLR":  # Clear button selected
            self.expression = ""
            self.equation.set("")
        elif self.sel == "=":  # Equal button pressed, evaluate the current
expression
            total = str(eval(self.expression))
            self.equation.set(total)
            self.expression = ""
        elif self.sel == "\u00F7":  # Divide button pressed
            self.expression += "/"
            self.equation.set(self.expression)
        elif self.sel == "1/x":  # 1/x button pressed
            self.expression = "1 / " + self.expression
            self.equation.set(self.expression)
        elif self.sel == "\u232B":  # backspace button pressed
            self.expression = self.expression.rstrip(self.expression[-1])
            self.equation.set(self.expression)
        elif self.sel == "\u03C0":   # pi
            self.expression += "pi"
            self.equation.set(self.expression)
        elif self.sel == "2\u03C0":  # 2pi
            self.expression += "2*pi"
            self.equation.set(self.expression)
        elif self.sel == "x^2":  # x raised to the power of 2
            self.expression = pow(float(self.expression), 2)
            self.equation.set(str(self.expression))
        elif self.sel == "|x|":  # abs(x)
            self.expression = fabs(float(self.expression))
            self.equation.set(str(self.expression))
        elif self.sel == "+/-":  # invert sign
            self.expression = -float(self.expression)
            self.equation.set(str(self.expression))
        elif self.sel == "n!":  # factorial
            self.expression = factorial(int(self.expression))
            self.equation.set(str(self.expression))
        elif self.sel == "10^x":  # 10 to the power of x
            self.expression = pow(10, float(self.expression))
            self.equation.set(str(self.expression))
        elif self.sel == "\u221A":  # square root
```

```
                self.expression = sqrt(float(self.expression))
                self.equation.set(str(self.expression))
```

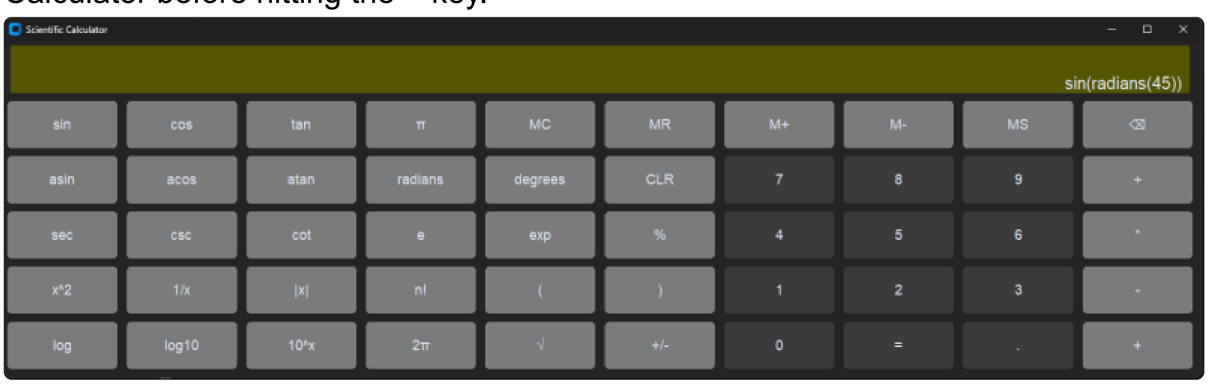
memory_operation() function detects which memory key is pressed and processes the operation for that key.

```
def memory_operation(self, x, y):
    if self.sel == "MC":
        self.memory = 0
    elif self.sel == "MR":
        self.expression += str(self.memory)
        self.equation.set(str(self.expression))
    elif self.sel == "M+":
        self.memory += int(self.expression)
    elif self.sel == "M-":
        self.memory -= int(self.expression)
    elif self.sel == "MS":
        self.memory = int(self.expression)
```

Note that trig functions such as sine and cosine must be entered as follows:

- Press the sin key
- Press the left parenthesis key
- Press the desired value key (value must be in radians)
- Press the right parenthesis key
- Press the = key or add more to the expression

Calculator before hitting the = key.



Calculator after hitting the = key.

Congratulations on completing the Scientific Calculator project. The complete source code is 119 software lines of code (SLOC) and is shown below.

scientific_calculator.py

```
import customtkinter as ctk
from math import *


class App(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("1525x425x10x10")
        self.title("Scientific Calculator")

        self.expression = ""
        self.equation = ctk.StringVar()
        self.sel = ""
        self.memory = 0

        # Create a label to display results of the calculation
        label = ctk.CTkLabel(self, text="0.00", padx=10, anchor=ctk.SE, font=('arial',22), textvariable=self.equation,
                             fg_color="transparent", width=1500, height=60)
        label.grid(row=0, column=0, columnspan=10, rowspan=4, padx=10, pady=5)
        label.configure(bg_color="#555500")

        # Initialize a button array (list of lists)
        self.buttons = [[None]*10 for _ in range(5)]

        # Create buttons
        self.button_text = [
            ["sin", "cos", "tan", "\u03C0", "MC", "MR", "M+", "M-", "MS", "\u232B"],  # Row 5
            ["asin", "acos", "atan", "radians", "degrees", "CLR", "7", "8", "9", "\u00F7"],  # Row 6
            ["sec", "csc", "cot", "e", "exp", "%", "4", "5", "6", "*"],  # Row
```

```
7
            ["x^2", "1/x", "|x|", "n!", "(", ")", "1", "2", "3", "-"],  # Row 8
            ["log", "log10", "10^x", "2\u03C0", "\u221A", "+/-", "0", "=", ".",
"+"],  # Row 9
        ]

        # Display button in a 5x10 array
        for x in range(5):
            for y in range(10):
                self.buttons[x][y] = ctk.CTkButton(self,
text=self.button_text[x][y], height=60,
                                                   font=('arial',18),
                                                   command=lambda x1=x, y1=y:
self.press(x1, y1))
                if 0 < x < 5 < y < 9:
                    self.buttons[x][y].configure(fg_color="#3b3b3b")
                else:
                    self.buttons[x][y].configure(fg_color="#7b7b7b")
                self.buttons[x][y].grid(row=x+4, column=y, padx=5, pady=5)

    def press(self, x, y):
        # Check for special cases where eval can't evaluate the button text
        self.sel = self.button_text[x][y]  # self.sel is a short alias for
self.button_text[x][y]
        if self.sel == "CLR" or self.sel == "=" or self.sel == "\u00F7" or
self.sel == "1/x" or \
                self.sel == "\u232B" or self.sel == "\u03C0" or self.sel ==
"x^2" or self.sel == "|x|" or \
                self.sel == "+/-" or self.sel == "n!" or self.sel == "10^x" or
self.sel == "\u221A" or \
                self.sel == "2\u03C0":
            self.special_case(x, y)
        elif self.sel == "MC" or self.sel == "MR" or self.sel == "M+" or
self.sel == "M-" or self.sel == "MS":
            self.memory_operation(x,y)
        else:
            self.expression += self.button_text[x][y]
            self.equation.set(self.expression)

    def special_case(self, x, y):
        if self.sel == "CLR":  # Clear button selected
            self.expression = ""
            self.equation.set("")
        elif self.sel == "=":  # Equal button pressed, evaluate the current
expression
            total = str(eval(self.expression))
```

```
            self.equation.set(total)
            self.expression = ""
        elif self.sel == "\u00F7":  # Divide button pressed
            self.expression += "/"
            self.equation.set(self.expression)
        elif self.sel == "1/x":  # 1/x button pressed
            self.expression = "1 / " + self.expression
            self.equation.set(self.expression)
        elif self.sel == "\u232B":  # backspace button pressed
            self.expression = self.expression.rstrip(self.expression[-1])
            self.equation.set(self.expression)
        elif self.sel == "\u03C0":   # pi
            self.expression += "pi"
            self.equation.set(self.expression)
        elif self.sel == "2\u03C0":  # 2pi
            self.expression += "2*pi"
            self.equation.set(self.expression)
        elif self.sel == "x^2":  # x raised to the power of 2
            self.expression = pow(float(self.expression), 2)
            self.equation.set(str(self.expression))
        elif self.sel == "|x|":  # abs(x)
            self.expression = fabs(float(self.expression))
            self.equation.set(str(self.expression))
        elif self.sel == "+/-":  # invert sign
            self.expression = -float(self.expression)
            self.equation.set(str(self.expression))
        elif self.sel == "n!":  # factorial
            self.expression = factorial(int(self.expression))
            self.equation.set(str(self.expression))
        elif self.sel == "10^x":  # 10 to the power of x
            self.expression = pow(10, float(self.expression))
            self.equation.set(str(self.expression))
        elif self.sel == "\u221A":  # square root
            self.expression = sqrt(float(self.expression))
            self.equation.set(str(self.expression))

    def memory_operation(self, x, y):
        if self.sel == "MC":
            self.memory = 0
        elif self.sel == "MR":
            self.expression += str(self.memory)
            self.equation.set(str(self.expression))
        elif self.sel == "M+":
            self.memory += int(self.expression)
        elif self.sel == "M-":
            self.memory -= int(self.expression)
```

```
        elif self.sel == "MS":
            self.memory = int(self.expression)


if __name__ == "__main__":
    app = App()
    app.mainloop()
```

Summary

This tutorial covered the design and development of a scientific calculator program. This is a beginner project because the total lines of code are fairly small. In the next chapters, we will move on to intermediate projects to create a shape editor and a line editor which will form the basis for a diagram editor similar to Microsoft Visio. Electrical Engineering simulators are essentially schematics (diagrams) with analysis capability.