Python Project 01: Calculator

Complete Frontend + Backend Project

1. Project Overview

In this project, we will build a **modern desktop Calculator application using Python**.

This is not just a simple calculator where buttons perform arithmetic. The project is structured like a small real-world application with:

- A graphical frontend
- A separate calculation backend
- Safe expression evaluation
- Input validation
- Error handling
- Calculation history
- Keyboard support
- Animated UI feedback
- Modular project structure

The frontend will be built using **Tkinter**, Python's standard GUI toolkit. Tkinter provides Python interfaces for creating windows, widgets, event bindings, and event-driven applications. (Python documentation)

The backend will contain the actual calculator logic so that the UI does not directly handle the mathematical rules.

---

2. What We Are Building

The final application will look and behave like a modern desktop calculator.

The user will be able to:

- Enter numbers
- Add numbers
- Subtract numbers
- Multiply numbers
- Divide numbers
- Calculate percentages
- Use powers
- Use decimal values
- Use parentheses
- Use positive and negative values
- Clear the calculator
- Delete the last character
- Press `Enter` to calculate
- Use keyboard numbers and operators
- View calculation history
- See animated feedback
- Receive friendly error messages

Example:

Input:

```
25 * 4 + 10
```

Output:

```
110
```

Another example:

```
(20 + 5) * 3
```

Output:

```
75
```

---

3. Project Features

The project includes the following features.

Calculator Features

+ Addition
- Subtraction
* Multiplication
/ Division
% Modulo
** Power
() Parentheses
Decimal numbers
Positive / negative values

UI Features

Modern dark interface
Large calculator display
Organized keypad
History panel
Hover effects
Button press effects
Result animation
Error animation
Display flash

Keyboard controls

Backend Features

Separate CalculatorEngine class
Safe expression parsing
No direct eval()
Custom CalculatorError
Input validation
Division by zero protection
Exponent protection
Invalid expression handling
Finite result validation

---

4. What You Will Learn

By completing this project, you will practice several Python concepts together.

Python Concepts

Variables
Functions
Classes
Objects
Methods
Conditionals
Exceptions
Custom Exceptions
Strings
Lists
Dictionaries
Modules
Imports

OOP
Static methods
Type hints

Application Development Concepts

Frontend
Backend
Separation of concerns
Event-driven programming
UI callbacks
Input validation
Error handling
Application architecture
User interaction
Testing

---

5. Technology Stack

We will use:

Python
Tkinter
Standard Python Library
AST module

No external Python package is required.

Tkinter is commonly included with Python and provides the GUI layer for desktop applications. (Python documentation)

---

6. Requirements

You need:

```
Python 3.10+
VS Code or another code editor
Terminal / Command Prompt
```

Check Python:

```
python --version
```

You should get something similar to:

```
Python 3.x.x
```

---

7. Project Folder Structure

Create this structure:

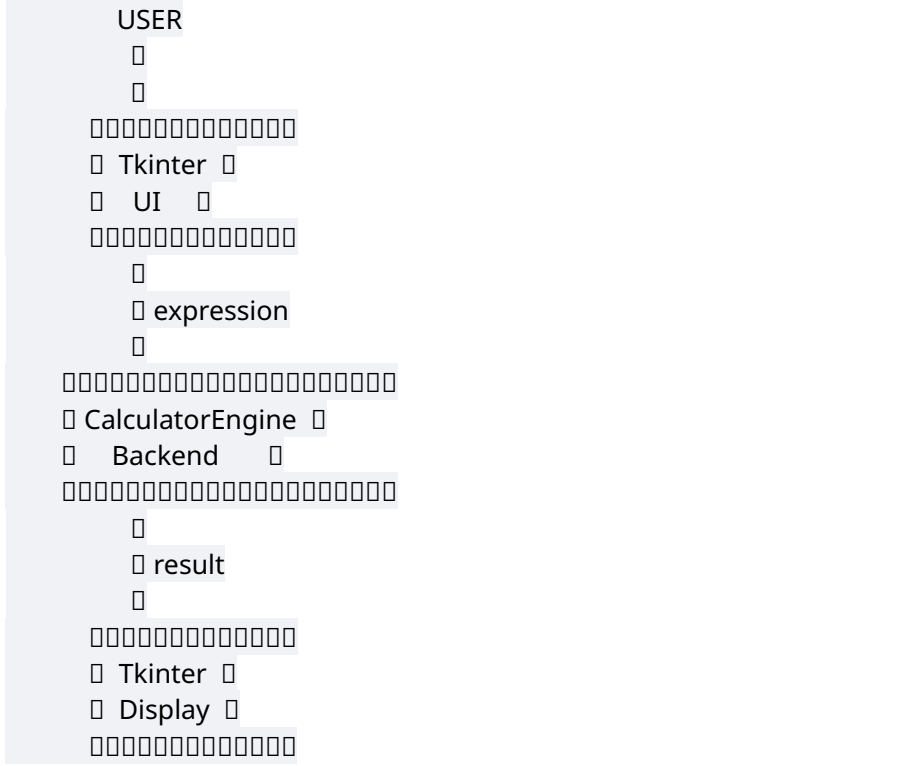
```
calculator_project/
□
□□□ main.py
□□□ ui.py
□□□ calculator.py
□□□ README.md
```

Each file has a specific responsibility.

---

8. Project Architecture

Our application follows a simple frontend/backend architecture.



The important idea is:

`UI ≠ Calculation Logic`

The UI collects user input.

The backend performs the calculation.

This separation makes the project easier to maintain and extend.

---

9. Application Data Flow

When a user clicks:

`7`

the UI receives the click.

When the user clicks:

`+`

the UI adds the operator.

Then:

`3`

The expression becomes:

`7 + 3`

When the user clicks:

`=`

the frontend sends:

```
7 + 3
```

to the backend.

The backend calculates:

```
10
```

The result is then returned to the frontend.

```
User
 
Button Click
 
UI
 
Expression
 
CalculatorEngine
 
Result
 
UI Display
```

---

10. Why We Separate Frontend and Backend

A beginner might write everything inside one file:

```
if button == "+":
    ...
```

That works for a tiny calculator, but it becomes difficult to maintain.

Instead, we separate responsibilities.

Frontend

Responsible for:

```
Buttons
Display
History
Animations
Keyboard
User interaction
```

Backend

Responsible for:

```
Calculation
Validation
Errors
Expression parsing
Safety
```

This is called **separation of concerns**.

---

11. Backend: calculator.py

Create:

```
calculator.py
```

This file contains the calculation engine.

The complete code is:

```
import ast
import math
import operator


class CalculatorError(Exception):
    """Custom exception for calculator errors."""


class CalculatorEngine:
    """Safely evaluates calculator expressions."""

    OPERATORS = {
        ast.Add: operator.add,
        ast.Sub: operator.sub,
        ast.Mult: operator.mul,
        ast.Div: operator.truediv,
        ast.Mod: operator.mod,
        ast.Pow: operator.pow,
        ast.UAdd: operator.pos,
        ast.USub: operator.neg,
    }

    MAX_EXPRESSION_LENGTH = 200
    MAX_POWER = 100

    def calculate(self, expression: str) -> float:
        expression = expression.strip()

        if not expression:
```

```
            raise CalculatorError("Enter an expression.")

        if len(expression) > self.MAX_EXPRESSION_LENGTH:
            raise CalculatorError("Expression is too long.")

        try:
            tree = ast.parse(expression, mode="eval")
            result = self._evaluate(tree.body)

            if not math.isfinite(result):
                raise CalculatorError("Result is not finite.")

            return result

        except CalculatorError:
            raise

        except ZeroDivisionError:
            raise CalculatorError("Cannot divide by zero.")

        except (SyntaxError, ValueError, TypeError):
            raise CalculatorError("Invalid expression.")

        except OverflowError:
            raise CalculatorError("Result is too large.")

    def _evaluate(self, node):
        if isinstance(node, ast.Constant):
            if isinstance(node.value, bool):
                raise CalculatorError("Invalid value.")

            if isinstance(node.value, (int, float)):
                return node.value
```

```
            raise CalculatorError("Only numbers are allowed.")

        if isinstance(node, ast.BinOp):
            left = self._evaluate(node.left)
            right = self._evaluate(node.right)

            operation = self.OPERATORS.get(type(node.op))

            if operation is None:
                raise CalculatorError("Operator not supported.")

            if isinstance(node.op, ast.Pow):
                if abs(right) > self.MAX_POWER:
                    raise CalculatorError("Exponent is too large.")

            return operation(left, right)

        if isinstance(node, ast.UnaryOp):
            value = self._evaluate(node.operand)

            operation = self.OPERATORS.get(type(node.op))

            if operation is None:
                raise CalculatorError("Operator not supported.")

            return operation(value)

        raise CalculatorError("Expression contains unsupported syntax.")


def format_result(value: float) -> str:
    """Convert a numeric result into a user-friendly string."""

    if value == int(value):
```

```
        return str(int(value))

    return f"{value:.10g}"
```

---

12. What This Code Does

The backend creates a class:

```
class CalculatorEngine:
```

This class represents the calculator's calculation system.

The UI does not need to know how the calculation works.

It only needs to do:

```
engine.calculate("10 + 5")
```

and receive:

```
15
```

---

13. Why We Use AST Instead of eval()

A simple calculator could use:

```
eval(expression)
```

For example:

`eval("10 + 5")`

But directly evaluating arbitrary user input with `eval()` is unsafe.

Our project instead parses the expression using Python's `ast` module and explicitly allows only the operations we support.

This gives us control over what the calculator can execute.

Allowed operations include:

`+`
`-`
`*`
`/`
`%`
`**`
`unary +`
`unary -`

---

14. CalculatorError

We create our own exception:

```
class CalculatorError(Exception):
    """Custom exception for calculator errors."""
```

This allows the UI to distinguish calculator errors from unrelated Python errors.

For example:

```
raise CalculatorError("Cannot divide by zero.")
```

The UI can then display:

```
Cannot divide by zero.
```

---

15. Supported Operators

The backend contains:

```
OPERATORS = {
    ast.Add: operator.add,
    ast.Sub: operator.sub,
    ast.Mult: operator.mul,
    ast.Div: operator.truediv,
    ast.Mod: operator.mod,
    ast.Pow: operator.pow,
    ast.UAdd: operator.pos,
    ast.USub: operator.neg,
}
```

This maps Python AST operations to actual Python functions.

For example:

```
+
```

maps to:

```
operator.add
```

and:

`*`

maps to:

`operator.mul`

---

16. Expression Length Protection

The backend contains:

`MAX_EXPRESSION_LENGTH = 200`

Then:

```
if len(expression) > self.MAX_EXPRESSION_LENGTH:
    raise CalculatorError("Expression is too long.")
```

This prevents unnecessarily large input.

---

17. Power Protection

The calculator supports:

`**`

For example:

`2 ** 5`

returns:

`32`

But extremely large powers can create enormous numbers.

Therefore:

`MAX_POWER = 100`

is used.

If someone enters an excessively large exponent, the application rejects it.

---

18. Result Formatting

The function:

`def format_result(value: float) -> str:`

makes results easier to read.

For example:

`10.0`

becomes:

`10`

while:

`3.1415926535`

can be displayed in a compact format.

---

19. Frontend: ui.py

Now create:

`ui.py`

This file creates the graphical interface.

Complete code:

```
import tkinter as tk
from tkinter import font

from calculator import CalculatorEngine, CalculatorError, format_result


class CalculatorUI:
    """Modern Tkinter frontend for the calculator."""

    BG = "#0b1020"
    PANEL = "#11182b"
    DISPLAY = "#080d1a"
    TEXT = "#f5f7ff"
    MUTED = "#8993aa"
    BUTTON = "#18223a"
```

```
    BUTTON_HOVER = "#243252"
    OPERATOR = "#243b63"
    OPERATOR_HOVER = "#31517f"
    EQUAL = "#3d72e6"
    EQUAL_HOVER = "#4e82f0"
    DANGER = "#5a2630"
    DANGER_HOVER = "#71303b"
    BORDER = "#202c46"

    def __init__(self, root: tk.Tk):
        self.root = root
        self.engine = CalculatorEngine()

        self.expression = ""
        self.history = []

        self.display_var = tk.StringVar(value="0")
        self.status_var = tk.StringVar(value="Ready")

        self.normal_font = font.Font(
            family="Segoe UI",
            size=18,
            weight="bold",
        )

        self.display_font = font.Font(
            family="Segoe UI",
            size=32,
            weight="bold",
        )

        self.setup_window()
        self.build_ui()
        self.bind_keyboard()
```

```
    def setup_window(self):
        self.root.title("HAAS Calculator")
        self.root.geometry("980x650")
        self.root.minsize(850, 580)
        self.root.configure(bg=self.BG)

    def build_ui(self):
        self.main = tk.Frame(
            self.root,
            bg=self.BG,
        )

        self.main.pack(
            fill="both",
            expand=True,
            padx=24,
            pady=24,
        )

        self.main.grid_columnconfigure(0, weight=1)
        self.main.grid_columnconfigure(1, weight=0)
        self.main.grid_rowconfigure(1, weight=1)

        self.build_header()
        self.build_calculator_panel()
        self.build_history_panel()

    def build_header(self):
        header = tk.Frame(
            self.main,
            bg=self.BG,
        )
```

```
        header.grid(
            row=0,
            column=0,
            columnspan=2,
            sticky="ew",
            pady=(0, 18),
        )

        title = tk.Label(
            header,
            text="HAAS CALCULATOR",
            bg=self.BG,
            fg=self.TEXT,
            font=("Segoe UI", 20, "bold"),
        )

        title.pack(side="left")

        subtitle = tk.Label(
            header,
            text="Python Desktop Project",
            bg=self.BG,
            fg=self.MUTED,
            font=("Segoe UI", 10),
        )

        subtitle.pack(
            side="left",
            padx=(14, 0),
            pady=(5, 0),
        )

    def build_calculator_panel(self):
        panel = tk.Frame(
```

```
            self.main,
            bg=self.PANEL,
            highlightbackground=self.BORDER,
            highlightthickness=1,
        )

        panel.grid(
            row=1,
            column=0,
            sticky="nsew",
            padx=(0, 14),
        )

        panel.grid_columnconfigure(
            0,
            weight=1,
        )

        for row in range(6):
            panel.grid_rowconfigure(
                row,
                weight=1,
            )

        display_frame = tk.Frame(
            panel,
            bg=self.DISPLAY,
        )

        display_frame.grid(
            row=0,
            column=0,
            sticky="nsew",
            padx=18,
```

```
            pady=18,
        )

        self.display_label = tk.Label(
            display_frame,
            textvariable=self.display_var,
            bg=self.DISPLAY,
            fg=self.TEXT,
            font=self.display_font,
            anchor="e",
            padx=18,
            pady=18,
        )

        self.display_label.pack(
            fill="both",
            expand=True,
        )

        status = tk.Label(
            panel,
            textvariable=self.status_var,
            bg=self.PANEL,
            fg=self.MUTED,
            font=("Segoe UI", 9),
            anchor="e",
        )

        status.grid(
            row=1,
            column=0,
            sticky="ew",
            padx=20,
            pady=(0, 8),
```

```
        )

        keypad = tk.Frame(
            panel,
            bg=self.PANEL,
        )

        keypad.grid(
            row=2,
            column=0,
            rowspan=4,
            sticky="nsew",
            padx=18,
            pady=(0, 18),
        )

        for column in range(4):
            keypad.grid_columnconfigure(
                column,
                weight=1,
            )

        for row in range(5):
            keypad.grid_rowconfigure(
                row,
                weight=1,
            )

        buttons = [
            ["C", "⌫", "(", ")"],
            ["7", "8", "9", "/"],
            ["4", "5", "6", "*"],
            ["1", "2", "3", "-"],
            ["0", ".", "=", "+"],
```

```
        ]

        for row_index, row in enumerate(buttons):
            for column_index, value in enumerate(row):
                button = self.create_button(
                    keypad,
                    value,
                )

                button.grid(
                    row=row_index,
                    column=column_index,
                    sticky="nsew",
                    padx=5,
                    pady=5,
                )

    def build_history_panel(self):
        history_panel = tk.Frame(
            self.main,
            bg=self.PANEL,
            width=260,
            highlightbackground=self.BORDER,
            highlightthickness=1,
        )

        history_panel.grid(
            row=1,
            column=1,
            sticky="ns",
        )

        history_panel.grid_propagate(False)
```

```
        title = tk.Label(
            history_panel,
            text="HISTORY",
            bg=self.PANEL,
            fg=self.TEXT,
            font=("Segoe UI", 12, "bold"),
        )

        title.pack(
            anchor="w",
            padx=18,
            pady=(18, 10),
        )

        self.history_text = tk.Text(
            history_panel,
            bg=self.PANEL,
            fg=self.MUTED,
            insertbackground=self.TEXT,
            relief="flat",
            borderwidth=0,
            font=("Consolas", 10),
            state="disabled",
            wrap="word",
        )

        self.history_text.pack(
            fill="both",
            expand=True,
            padx=18,
            pady=(0, 12),
        )

        clear_button = tk.Button(
```

```
            history_panel,
            text="Clear History",
            command=self.clear_history,
            bg=self.BUTTON,
            fg=self.TEXT,
            activebackground=self.BUTTON_HOVER,
            activeforeground=self.TEXT,
            relief="flat",
            bd=0,
            cursor="hand2",
            font=("Segoe UI", 10, "bold"),
            pady=10,
        )

        clear_button.pack(
            fill="x",
            padx=18,
            pady=(0, 18),
        )

    def create_button(self, parent, text):
        if text == "=":
            background = self.EQUAL
            hover = self.EQUAL_HOVER
        elif text in {"+", "-", "*", "/"}:
            background = self.OPERATOR
            hover = self.OPERATOR_HOVER
        elif text in {"C", "⌫"}:
            background = self.DANGER
            hover = self.DANGER_HOVER
        else:
            background = self.BUTTON
            hover = self.BUTTON_HOVER
```

```
        button = tk.Button(
            parent,
            text=text,
            bg=background,
            fg=self.TEXT,
            activebackground=hover,
            activeforeground=self.TEXT,
            relief="flat",
            bd=0,
            cursor="hand2",
            font=self.normal_font,
            command=lambda value=text: self.handle_input(value),
        )

        button.bind(
            "<Enter>",
            lambda event: button.configure(
                bg=hover
            ),
        )

        button.bind(
            "<Leave>",
            lambda event: button.configure(
                bg=background
            ),
        )

        button.bind(
            "<ButtonPress-1>",
            lambda event: self.button_press_animation(button),
        )

        return button
```

```
    def button_press_animation(self, button):
        original_font = button.cget("font")

        button.configure(
            font=("Segoe UI", 16, "bold")
        )

        self.root.after(
            80,
            lambda: button.configure(
                font=original_font
            ),
        )

    def handle_input(self, value):
        if value == "=":
            self.calculate()
            return

        if value == "C":
            self.clear()
            return

        if value == "⌫":
            self.backspace()
            return

        self.expression += value
        self.display_var.set(self.expression)
        self.status_var.set("Typing...")

    def calculate(self):
        if not self.expression:
```

```
            return

        try:
            result = self.engine.calculate(
                self.expression
            )

            formatted = format_result(result)

            self.history.append(
                f"{self.expression} = {formatted}"
            )

            self.update_history()

            self.expression = formatted
            self.display_var.set(formatted)
            self.status_var.set("Calculated")

            self.animate_result()

        except CalculatorError as error:
            self.status_var.set(str(error))
            self.animate_error()

    def clear(self):
        self.expression = ""
        self.display_var.set("0")
        self.status_var.set("Ready")

    def backspace(self):
        self.expression = self.expression[:-1]

        if self.expression:
```

```
            self.display_var.set(
                self.expression
            )
        else:
            self.display_var.set("0")

        self.status_var.set("Editing...")

    def update_history(self):
        self.history_text.configure(
            state="normal"
        )

        self.history_text.delete(
            "1.0",
            "end",
        )

        for item in reversed(self.history):
            self.history_text.insert(
                "end",
                item + "\n\n",
            )

        self.history_text.configure(
            state="disabled"
        )

    def clear_history(self):
        self.history.clear()

        self.history_text.configure(
            state="normal"
        )
```

```
        self.history_text.delete(
            "1.0",
            "end",
        )

        self.history_text.configure(
            state="disabled"
        )

        self.status_var.set(
            "History cleared"
        )

    def animate_result(self):
        original = self.display_label.cget(
            "fg"
        )

        self.display_label.configure(
            fg=self.EQUAL
        )

        self.root.after(
            160,
            lambda: self.display_label.configure(
                fg=original
            ),
        )

    def animate_error(self):
        original = self.display_label.cget(
            "fg"
        )
```

```
        self.display_label.configure(
            fg="#ff6b7a"
        )

        self.root.after(
            160,
            lambda: self.display_label.configure(
                fg=original
            ),
        )

    def bind_keyboard(self):
        self.root.bind(
            "<Key>",
            self.handle_keyboard,
        )

    def handle_keyboard(self, event):
        key = event.keysym
        char = event.char

        if char in "0123456789.+-*/%()":
            self.handle_input(char)

        elif key == "Return":
            self.calculate()

        elif key == "BackSpace":
            self.backspace()

        elif key == "Escape":
            self.clear()
```

```
def run_app():
    root = tk.Tk()
    CalculatorUI(root)
    root.mainloop()
```

---

20. What This Code Does

The frontend creates a class:

```
class CalculatorUI:
```

This class controls the entire graphical interface.

The constructor:

```
def __init__(self, root):
```

initializes:

```
Calculator backend
Expression
History
Display
Fonts
Window
UI
Keyboard controls
```

---

21. Creating the Window

The application begins with:

`root = tk.Tk()`

This creates the main Tkinter window.

Then:

`CalculatorUI(root)`

creates our calculator interface.

Finally:

`root.mainloop()`

starts the Tkinter event loop.

Tkinter applications are event-driven, and `mainloop()` processes events until the windows are destroyed. (Python documentation)

---

22. Why We Use mainloop()

Without:

`root.mainloop()`

the window would appear and immediately terminate.

The main loop continuously waits for events such as:

Mouse clicks
Keyboard input
Window events
Scheduled callbacks

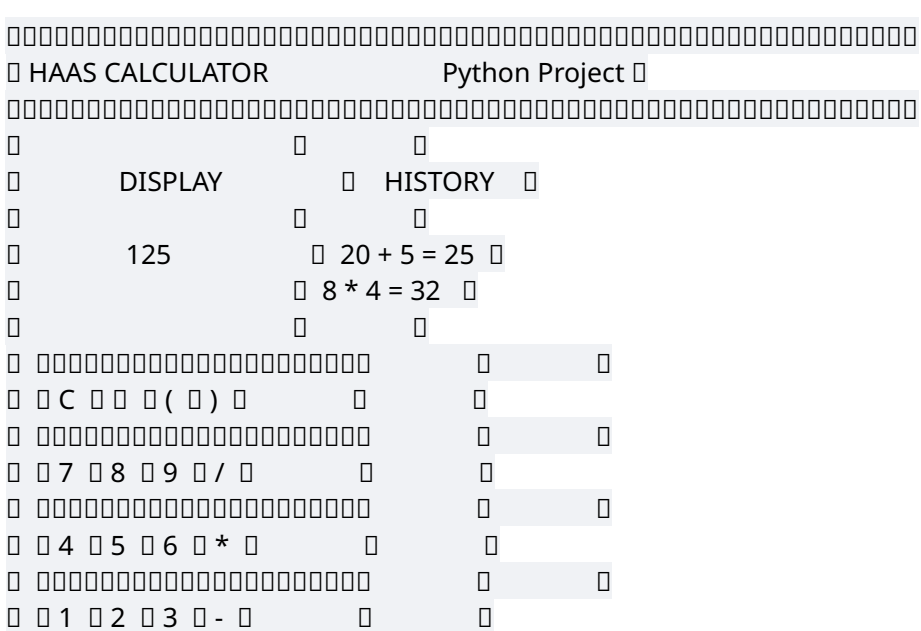
---

23. Building the UI

The UI contains three major areas:

Header
Calculator
History

Visually:



```
□ □□□□□□□□□□□□□□□□□□□□□□□□□     □     □
□ □0 □. □= □+ □     □     □
□ □□□□□□□□□□□□□□□□□□□□□□□□□     □     □
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

---

24. Modern UI Design

The application uses a dark interface.

The major colors are:

```
BG = "#0b1020"
PANEL = "#11182b"
DISPLAY = "#080d1a"
TEXT = "#f5f7ff"
```

The design uses different visual levels:

```
Background
   □
Panels
   □
Display
   □
Buttons
   □
Primary action
```

This creates visual hierarchy.

---

25. Button Categories

Buttons are divided into different categories.

Normal Buttons

0
1
2
3
4
5
6
7
8
9
.
(
)

Operator Buttons

+
-
*
/

Danger / Control Buttons

C
⌫

Primary Action

=

This gives the interface a clear visual hierarchy.

---

26. Hover Animation

Each button receives:

```
button.bind(
    "<Enter>",
    ...
)
```

and:

```
button.bind(
    "<Leave>",
    ...
)
```

When the mouse enters the button, its background changes.

When the mouse leaves, the original color returns.

Tkinter's `bind()` associates an event pattern with a callback function. (Python documentation)

---

27. Button Press Animation

When the user clicks a button:

```
<ButtonPress-1>
```

the button font temporarily becomes smaller.

Then:

```
self.root.after(
    80,
    ...
)
```

restores the original font.

Tkinter's `after()` schedules a callback to run after a specified number of milliseconds, which makes it useful for lightweight UI animations. (Python documentation)

---

28. Result Animation

After a successful calculation:

```
self.animate_result()
```

temporarily changes the display color.

The result appears highlighted for a short time and then returns to its normal color.

This provides visual feedback without interrupting the user.

---

29. Error Animation

If an invalid calculation occurs:

`self.animate_error()`

temporarily changes the display color to indicate an error.

For example:

`10 / 0`

produces:

`Cannot divide by zero.`

The display also receives visual feedback.

---

30. Frontend and Backend Connection

The most important connection happens here:

```
result = self.engine.calculate(
    self.expression
)
```

The UI sends the expression to:

`CalculatorEngine`

The backend calculates it.

Then the frontend receives the result.

For example:

`User enters:`

`25 * 4`

Frontend:

`self.expression`

contains:

`25 * 4`

Backend:

`self.engine.calculate("25 * 4")`

returns:

`100`

Frontend:

`self.display_var.set("100")`

shows:

`100`

---

31. History System

The calculator stores previous calculations:

```
self.history = []
```

After a successful calculation:

```
self.history.append(
    f"{self.expression} = {formatted}"
)
```

For example:

```
20 + 5 = 25
8 * 4 = 32
100 / 5 = 20
```

The history is then displayed in the sidebar.

---

32. Clearing History

The user can click:

```
Clear History
```

which executes:

```
self.clear_history()
```

The list is cleared:

```
self.history.clear()
```

and the history display is refreshed.

---

33. Keyboard Support

The application also supports keyboard input.

We use:

```
self.root.bind(
    "<Key>",
    self.handle_keyboard,
)
```

The application can detect:

```
Numbers
+
-
*
/
%
(
)
.
Enter
Backspace
Escape
```

For example:

Keyboard:

2
+
5
Enter

Result:

7

---

34. Keyboard Mapping

The keyboard handler checks:

```
if char in "0123456789.+-*/%()":
```

and sends the character to:

```
self.handle_input(char)
```

Enter:

```
self.calculate()
```

Backspace:

```
self.backspace()
```

Escape:

```
self.clear()
```

---

35. main.py

Now create:

`main.py`

This file is intentionally small.

Complete code:

```
from ui import run_app


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

---

36. Why main.py Is So Small

`main.py` should be the application entry point.

It should not contain:

```
UI design
Calculator logic
Validation
History logic
Animation code
```

Instead, it simply starts the application.

This keeps responsibilities separated.

---

37. README.md

Create:

README.md

Use:

```
# HAAS Calculator

A modern desktop calculator built with Python and Tkinter.

## Features

- Arithmetic operations
- Parentheses
- Decimal numbers
- Modulo
- Powers
- Calculation history
- Keyboard support
- Error handling
- Animated UI feedback
- Safe expression parsing

## Requirements

Python 3.10+

## Run
```

```bash
python main.py
```

Project Structure

```
calculator_project/
□
□□□ main.py
□□□ ui.py
□□□ calculator.py
□□□ README.md
```

Architecture

The application separates the frontend and backend.

Frontend:

- Tkinter UI
- Buttons
- Display
- History
- Animations
- Keyboard controls

Backend:

- Expression parsing
- Calculation
- Validation
- Error handling

Learning Goals

This project demonstrates:

- Python OOP
- Modules
- Exceptions
- Tkinter
- Event handling
- Input validation
- Application architecture

---

38. Complete Final Source Code

At this point the project contains:

```text
calculator_project/
□
□□□ main.py
□□□ ui.py
□□□ calculator.py
□□□ README.md
```

main.py

```
from ui import run_app


if __name__ == "__main__":
```

```
    run_app()
```

calculator.py

```
import ast
import math
import operator


class CalculatorError(Exception):
    """Custom exception for calculator errors."""


class CalculatorEngine:
    """Safely evaluates calculator expressions."""

    OPERATORS = {
        ast.Add: operator.add,
        ast.Sub: operator.sub,
        ast.Mult: operator.mul,
        ast.Div: operator.truediv,
        ast.Mod: operator.mod,
        ast.Pow: operator.pow,
        ast.UAdd: operator.pos,
        ast.USub: operator.neg,
    }

    MAX_EXPRESSION_LENGTH = 200
    MAX_POWER = 100

    def calculate(self, expression: str) -> float:
        expression = expression.strip()

        if not expression:
            raise CalculatorError("Enter an expression.")
```

```
        if len(expression) > self.MAX_EXPRESSION_LENGTH:
            raise CalculatorError("Expression is too long.")

        try:
            tree = ast.parse(expression, mode="eval")
            result = self._evaluate(tree.body)

            if not math.isfinite(result):
                raise CalculatorError("Result is not finite.")

            return result

        except CalculatorError:
            raise

        except ZeroDivisionError:
            raise CalculatorError("Cannot divide by zero.")

        except (SyntaxError, ValueError, TypeError):
            raise CalculatorError("Invalid expression.")

        except OverflowError:
            raise CalculatorError("Result is too large.")

    def _evaluate(self, node):
        if isinstance(node, ast.Constant):
            if isinstance(node.value, bool):
                raise CalculatorError("Invalid value.")

            if isinstance(node.value, (int, float)):
                return node.value

            raise CalculatorError("Only numbers are allowed.")
```

```
        if isinstance(node, ast.BinOp):
            left = self._evaluate(node.left)
            right = self._evaluate(node.right)

            operation = self.OPERATORS.get(type(node.op))

            if operation is None:
                raise CalculatorError("Operator not supported.")

            if isinstance(node.op, ast.Pow):
                if abs(right) > self.MAX_POWER:
                    raise CalculatorError("Exponent is too large.")

            return operation(left, right)

        if isinstance(node, ast.UnaryOp):
            value = self._evaluate(node.operand)

            operation = self.OPERATORS.get(type(node.op))

            if operation is None:
                raise CalculatorError("Operator not supported.")

            return operation(value)

        raise CalculatorError("Expression contains unsupported syntax.")


def format_result(value: float) -> str:
    """Convert a numeric result into a user-friendly string."""

    if value == int(value):
        return str(int(value))
```

```
    return f"{value:.10g}"
```

ui.py

```
import tkinter as tk
from tkinter import font

from calculator import CalculatorEngine, CalculatorError, format_result


class CalculatorUI:
    """Modern Tkinter frontend for the calculator."""

    BG = "#0b1020"
    PANEL = "#11182b"
    DISPLAY = "#080d1a"
    TEXT = "#f5f7ff"
    MUTED = "#8993aa"
    BUTTON = "#18223a"
    BUTTON_HOVER = "#243252"
    OPERATOR = "#243b63"
    OPERATOR_HOVER = "#31517f"
    EQUAL = "#3d72e6"
    EQUAL_HOVER = "#4e82f0"
    DANGER = "#5a2630"
    DANGER_HOVER = "#71303b"
    BORDER = "#202c46"

    def __init__(self, root: tk.Tk):
        self.root = root
        self.engine = CalculatorEngine()

        self.expression = ""
        self.history = []
```

```
        self.display_var = tk.StringVar(value="0")
        self.status_var = tk.StringVar(value="Ready")

        self.normal_font = font.Font(
            family="Segoe UI",
            size=18,
            weight="bold",
        )

        self.display_font = font.Font(
            family="Segoe UI",
            size=32,
            weight="bold",
        )

        self.setup_window()
        self.build_ui()
        self.bind_keyboard()

    def setup_window(self):
        self.root.title("HAAS Calculator")
        self.root.geometry("980x650")
        self.root.minsize(850, 580)
        self.root.configure(bg=self.BG)

    def build_ui(self):
        self.main = tk.Frame(
            self.root,
            bg=self.BG,
        )

        self.main.pack(
            fill="both",
```

```
            expand=True,
            padx=24,
            pady=24,
        )

        self.main.grid_columnconfigure(0, weight=1)
        self.main.grid_columnconfigure(1, weight=0)
        self.main.grid_rowconfigure(1, weight=1)

        self.build_header()
        self.build_calculator_panel()
        self.build_history_panel()

    def build_header(self):
        header = tk.Frame(
            self.main,
            bg=self.BG,
        )

        header.grid(
            row=0,
            column=0,
            columnspan=2,
            sticky="ew",
            pady=(0, 18),
        )

        title = tk.Label(
            header,
            text="HAAS CALCULATOR",
            bg=self.BG,
            fg=self.TEXT,
            font=("Segoe UI", 20, "bold"),
        )
```

```
        title.pack(side="left")

        subtitle = tk.Label(
            header,
            text="Python Desktop Project",
            bg=self.BG,
            fg=self.MUTED,
            font=("Segoe UI", 10),
        )

        subtitle.pack(
            side="left",
            padx=(14, 0),
            pady=(5, 0),
        )

    def build_calculator_panel(self):
        panel = tk.Frame(
            self.main,
            bg=self.PANEL,
            highlightbackground=self.BORDER,
            highlightthickness=1,
        )

        panel.grid(
            row=1,
            column=0,
            sticky="nsew",
            padx=(0, 14),
        )

        panel.grid_columnconfigure(
            0,
```

```
            weight=1,
        )

        for row in range(6):
            panel.grid_rowconfigure(
                row,
                weight=1,
            )

        display_frame = tk.Frame(
            panel,
            bg=self.DISPLAY,
        )

        display_frame.grid(
            row=0,
            column=0,
            sticky="nsew",
            padx=18,
            pady=18,
        )

        self.display_label = tk.Label(
            display_frame,
            textvariable=self.display_var,
            bg=self.DISPLAY,
            fg=self.TEXT,
            font=self.display_font,
            anchor="e",
            padx=18,
            pady=18,
        )

        self.display_label.pack(
```

```
            fill="both",
            expand=True,
        )

        status = tk.Label(
            panel,
            textvariable=self.status_var,
            bg=self.PANEL,
            fg=self.MUTED,
            font=("Segoe UI", 9),
            anchor="e",
        )

        status.grid(
            row=1,
            column=0,
            sticky="ew",
            padx=20,
            pady=(0, 8),
        )

        keypad = tk.Frame(
            panel,
            bg=self.PANEL,
        )

        keypad.grid(
            row=2,
            column=0,
            rowspan=4,
            sticky="nsew",
            padx=18,
            pady=(0, 18),
        )
```

```
        for column in range(4):
            keypad.grid_columnconfigure(
                column,
                weight=1,
            )

        for row in range(5):
            keypad.grid_rowconfigure(
                row,
                weight=1,
            )

        buttons = [
            ["C", "⌫", "(", ")"],
            ["7", "8", "9", "/"],
            ["4", "5", "6", "*"],
            ["1", "2", "3", "-"],
            ["0", ".", "=", "+"],
        ]

        for row_index, row in enumerate(buttons):
            for column_index, value in enumerate(row):
                button = self.create_button(
                    keypad,
                    value,
                )

                button.grid(
                    row=row_index,
                    column=column_index,
                    sticky="nsew",
                    padx=5,
                    pady=5,
```

```
            )

    def build_history_panel(self):
        history_panel = tk.Frame(
            self.main,
            bg=self.PANEL,
            width=260,
            highlightbackground=self.BORDER,
            highlightthickness=1,
        )

        history_panel.grid(
            row=1,
            column=1,
            sticky="ns",
        )

        history_panel.grid_propagate(False)

        title = tk.Label(
            history_panel,
            text="HISTORY",
            bg=self.PANEL,
            fg=self.TEXT,
            font=("Segoe UI", 12, "bold"),
        )

        title.pack(
            anchor="w",
            padx=18,
            pady=(18, 10),
        )

        self.history_text = tk.Text(
```

```
            history_panel,
            bg=self.PANEL,
            fg=self.MUTED,
            insertbackground=self.TEXT,
            relief="flat",
            borderwidth=0,
            font=("Consolas", 10),
            state="disabled",
            wrap="word",
        )

        self.history_text.pack(
            fill="both",
            expand=True,
            padx=18,
            pady=(0, 12),
        )

        clear_button = tk.Button(
            history_panel,
            text="Clear History",
            command=self.clear_history,
            bg=self.BUTTON,
            fg=self.TEXT,
            activebackground=self.BUTTON_HOVER,
            activeforeground=self.TEXT,
            relief="flat",
            bd=0,
            cursor="hand2",
            font=("Segoe UI", 10, "bold"),
            pady=10,
        )

        clear_button.pack(
```

```
            fill="x",
            padx=18,
            pady=(0, 18),
        )

    def create_button(self, parent, text):
        if text == "=":
            background = self.EQUAL
            hover = self.EQUAL_HOVER

        elif text in {"+", "-", "*", "/"}:
            background = self.OPERATOR
            hover = self.OPERATOR_HOVER

        elif text in {"C", "⌫"}:
            background = self.DANGER
            hover = self.DANGER_HOVER

        else:
            background = self.BUTTON
            hover = self.BUTTON_HOVER

        button = tk.Button(
            parent,
            text=text,
            bg=background,
            fg=self.TEXT,
            activebackground=hover,
            activeforeground=self.TEXT,
            relief="flat",
            bd=0,
            cursor="hand2",
            font=self.normal_font,
            command=lambda value=text: self.handle_input(value),
```

```
        )

        button.bind(
            "<Enter>",
            lambda event: button.configure(
                bg=hover
            ),
        )

        button.bind(
            "<Leave>",
            lambda event: button.configure(
                bg=background
            ),
        )

        button.bind(
            "<ButtonPress-1>",
            lambda event: self.button_press_animation(button),
        )

        return button

    def button_press_animation(self, button):
        original_font = button.cget("font")

        button.configure(
            font=("Segoe UI", 16, "bold")
        )

        self.root.after(
            80,
            lambda: button.configure(
                font=original_font
```

```
            ),
        )

    def handle_input(self, value):
        if value == "=":
            self.calculate()
            return

        if value == "C":
            self.clear()
            return

        if value == "⌫":
            self.backspace()
            return

        self.expression += value
        self.display_var.set(self.expression)
        self.status_var.set("Typing...")

    def calculate(self):
        if not self.expression:
            return

        try:
            result = self.engine.calculate(
                self.expression
            )

            formatted = format_result(result)

            self.history.append(
                f"{self.expression} = {formatted}"
            )
```

```
            self.update_history()

            self.expression = formatted
            self.display_var.set(formatted)
            self.status_var.set("Calculated")

            self.animate_result()

        except CalculatorError as error:
            self.status_var.set(str(error))
            self.animate_error()

    def clear(self):
        self.expression = ""
        self.display_var.set("0")
        self.status_var.set("Ready")

    def backspace(self):
        self.expression = self.expression[:-1]

        if self.expression:
            self.display_var.set(
                self.expression
            )
        else:
            self.display_var.set("0")

        self.status_var.set("Editing...")

    def update_history(self):
        self.history_text.configure(
            state="normal"
        )
```

```
        self.history_text.delete(
            "1.0",
            "end",
        )

        for item in reversed(self.history):
            self.history_text.insert(
                "end",
                item + "\n\n",
            )

        self.history_text.configure(
            state="disabled"
        )

    def clear_history(self):
        self.history.clear()

        self.history_text.configure(
            state="normal"
        )

        self.history_text.delete(
            "1.0",
            "end",
        )

        self.history_text.configure(
            state="disabled"
        )

        self.status_var.set(
            "History cleared"
```

```
        )

    def animate_result(self):
        original = self.display_label.cget(
            "fg"
        )

        self.display_label.configure(
            fg=self.EQUAL
        )

        self.root.after(
            160,
            lambda: self.display_label.configure(
                fg=original
            ),
        )

    def animate_error(self):
        original = self.display_label.cget(
            "fg"
        )

        self.display_label.configure(
            fg="#ff6b7a"
        )

        self.root.after(
            160,
            lambda: self.display_label.configure(
                fg=original
            ),
        )
```

```
    def bind_keyboard(self):
        self.root.bind(
            "<Key>",
            self.handle_keyboard,
        )

    def handle_keyboard(self, event):
        key = event.keysym
        char = event.char

        if char in "0123456789.+-*/%()":
            self.handle_input(char)

        elif key == "Return":
            self.calculate()

        elif key == "BackSpace":
            self.backspace()

        elif key == "Escape":
            self.clear()


def run_app():
    root = tk.Tk()
    CalculatorUI(root)
    root.mainloop()
```

---

39. How to Run the Project

Open the terminal inside the project folder.

Run:

```
python main.py
```

The calculator window should open.

---

40. First Test

Enter:

```
10 + 5
```

Press:

```
=
```

Expected result:

```
15
```

---

41. Test Multiplication

Enter:

```
8 * 7
```

Expected:

```
56
```

---

42. Test Division

Enter:

```
100 / 4
```

Expected:

```
25
```

---

43. Test Parentheses

Enter:

```
(10 + 5) * 2
```

Expected:

```
30
```

---

44. Test Power

Enter:

```
2 ** 5
```

Expected:

```
32
```

---

45. Test Modulo

Enter:

```
10 % 3
```

Expected:

```
1
```

---

46. Test Decimal Numbers

Enter:

```
10.5 + 2.5
```

Expected:

```
13
```

---

47. Test Negative Numbers

Enter:

```
-10 + 5
```

Expected:

`-5`

---

48. Test Division by Zero

Enter:

`10 / 0`

Expected:

`Cannot divide by zero.`

The application should not crash.

---

49. Test Invalid Expression

Try:

`10 +`

Expected:

`Invalid expression.`

Again, the application should continue running.

---

50. Test Backspace

Enter:

`12345`

Press:

`⌫`

Expected:

`1234`

Press again:

`123`

---

51. Test Clear

Enter:

`12345`

Press:

`C`

Expected:

```
0
```

---

52. Test Keyboard

Type:

```
25+15
```

Press:

```
Enter
```

Expected:

```
40
```

Press:

```
Escape
```

Expected:

```
0
```

---

53. Common Error: ModuleNotFoundError

If you see:

```
ModuleNotFoundError
```

check that the files are in the same folder:

```
calculator_project/
□□□ main.py
□□□ ui.py
□□□ calculator.py
□□□ README.md
```

Run:

```
python main.py
```

from inside the project folder.

---

54. Common Error: Tkinter Not Available

If Python cannot import Tkinter, your Python installation may not include the Tk components.

On some operating systems, Tkinter needs to be installed separately.

Test it with:

```
python -m tkinter
```

If Tkinter is available, a small test window should open.

---

55. Common Error: Calculator Does Not Respond

Make sure the application has:

```
root.mainloop()
```

The main event loop is what allows the Tkinter application to continuously process user events. (Python documentation)

---

56. Common Error: Buttons Do Not Work

Check the button command:

```
command=lambda value=text: self.handle_input(value)
```

This sends the button value to:

```
handle_input()
```

For example:

```
7
```

becomes:

```
self.handle_input("7")
```

---

57. Common Error: Division by Zero

The backend catches:

```
ZeroDivisionError
```

and converts it into:

```
CalculatorError(
    "Cannot divide by zero."
)
```

The UI then displays the error instead of crashing.

---

58. Common Error: Invalid Syntax

For:

```
10 +
```

Python's AST parser raises an error.

Our backend catches it:

```
except (SyntaxError, ValueError, TypeError):
    raise CalculatorError("Invalid expression.")
```

---

59. Why Error Handling Matters

A real application should not crash just because a user enters invalid input.

Bad:

```
User input
↓
Error
↓
```

```
Application crashes
```

Better:

```
User input
↓
Validate
↓
Error?
↓
Show message
↓
Keep application running
```

This is an important application-development principle.

---

60. How the Project Works Internally

Suppose the user enters:

```
20 + 5 * 2
```

The frontend stores:

```
self.expression
```

as:

```
20 + 5 * 2
```

Then:

```
self.engine.calculate(
    self.expression
)
```

sends it to the backend.

The backend creates an AST:

```
Expression
    □
    □□□ 20
    □
    □□□ +
        □
        □□□ 5 * 2
```

The evaluator processes the tree.

The result becomes:

```
30
```

The frontend displays:

```
30
```

---

61. Why AST Is Useful

AST means:

```
Abstract Syntax Tree
```

Python can represent an expression as a tree of operations.

For:

```
2 + 3
```

the tree conceptually becomes:

```
  +
 / \
2  3
```

For:

```
2 + 3 * 4
```

the tree represents operator precedence:

```
   +
  / \
 2   *
    / \
   3   4
```

Therefore:

```
3 * 4
```

is calculated before:

```
2 +
```

---

62. Project Design Principles

This project demonstrates several good development practices.

Separation of Concerns

UI ui.py
Logic calculator.py
Entry Point main.py
Documentation README.md

Reusability

The backend can theoretically be reused by another interface.

For example:

Tkinter UI

CalculatorEngine

could later become:

Web UI

CalculatorEngine

or:

Mobile UI

CalculatorEngine

---

63. Version 1

The current project is Version 1.

It includes:

```
Calculator
+
Modern UI
+
History
+
Keyboard
+
Animations
+
Error handling
+
Safe parsing
```

---

64. Version 2 Improvements

Students can extend the project.

Possible features:

```
Scientific calculator
Square root
Sin
Cos
Tan
```

Log
Memory buttons
M+
M-
MR
MC
Calculation history persistence
Themes
Light mode
Dark mode
Settings
Keyboard shortcuts
Resizable UI

---

65. Version 3 Improvements

The project can become a more complete application.

Add:

SQLite database

Then history can survive application restarts.

Architecture:

Frontend

Calculator Backend

History Service

□
SQLite Database

---

66. Version 4 Improvements

The calculator can eventually become a web application.

Possible architecture:

HTML
CSS
JavaScript
□
FastAPI
□
Python Calculator Engine

This would allow students to move from a desktop Python project toward backend development.

---

67. Student Challenge

Now modify the calculator by yourself.

Challenge: Add a ± Button

Add a button:

±

The button should allow the user to change the current value from:

`25`

to:

`-25`

and:

`-25`

back to:

`25`

---

68. Challenge Requirements

The new button should:

```
Appear in the UI
Respond to mouse clicks
Work with positive numbers
Work with negative numbers
Update the display
Not crash the application
```

---

69. Challenge Hint

You can add:

```
if value == "±":
    self.toggle_sign()
    return
```

Then create:

```
def toggle_sign(self):
    ...
```

Think about how a string can be changed from:

```
25
```

to:

```
-25
```

---

70. Challenge Solution

Add:

```
def toggle_sign(self):
    if not self.expression:
        return

    try:
        value = float(self.expression)

        value *= -1
```

```
            self.expression = format_result(value)

            self.display_var.set(
                self.expression
            )

        except ValueError:
            self.status_var.set(
                "Enter a valid number."
            )
```

Then add:

```
±
```

to the button layout.

---

71. Mini Exercises

Exercise 1

Add a square operation.

Example:

```
5²
```

should return:

```
25
```

Exercise 2

Add a percentage button.

Example:

`50%`

should return:

`0.5`

Exercise 3

Add a light theme.

Create alternative colors for:

`Background`
`Panel`
`Display`
`Buttons`
`Text`
`Operators`

Exercise 4

Add a keyboard shortcut for clear:

C

Exercise 5

Save calculation history to a JSON file.

Example:

history.json

Then reload it when the application starts.

---

72. Interview Questions

1. What is Tkinter?

Tkinter is Python's interface for the Tk GUI toolkit.

---

2. What is an event-driven application?

An event-driven application waits for events such as:

Click
Key press
Mouse movement
Window event

and executes callback functions in response.

---

3. What does mainloop() do?

It starts Tkinter's event-processing loop and keeps the application responsive. (Python documentation)

---

4. What does bind() do?

It associates an event pattern with a callback function. (Python documentation)

---

5. What does after() do?

It schedules a callback to execute after a specified delay. (Python documentation)

---

6. Why did we separate UI and backend?

To separate responsibilities and make the application easier to maintain and extend.

---

7. Why should we avoid directly using eval() for user input?

Because arbitrary expressions passed to `eval()` can execute Python code. The project instead parses the expression and explicitly permits supported operations.

---

8. What is AST?

AST stands for:

`Abstract Syntax Tree`

It represents source expressions as a structured tree.

---

9. What is a custom exception?

A custom exception is an exception class created specifically for an application's error conditions.

Example:

```
class CalculatorError(Exception):
    pass
```

---

10. Why do we use classes?

Classes allow us to group related data and behavior into reusable objects.

---

73. Final Project Checklist

Before considering the project complete, verify:

```
[ ] Python installed
[ ] Project folder created
[ ] main.py created
[ ] ui.py created
[ ] calculator.py created
[ ] README.md created
[ ] Application launches
```

[ ] Buttons work
[ ] Addition works
[ ] Subtraction works
[ ] Multiplication works
[ ] Division works
[ ] Modulo works
[ ] Power works
[ ] Parentheses work
[ ] Decimal numbers work
[ ] Negative numbers work
[ ] Backspace works
[ ] Clear works
[ ] History works
[ ] Clear history works
[ ] Keyboard works
[ ] Hover animation works
[ ] Button animation works
[ ] Result animation works
[ ] Error animation works
[ ] Division by zero is handled
[ ] Invalid expressions are handled
[ ] Application does not crash

---

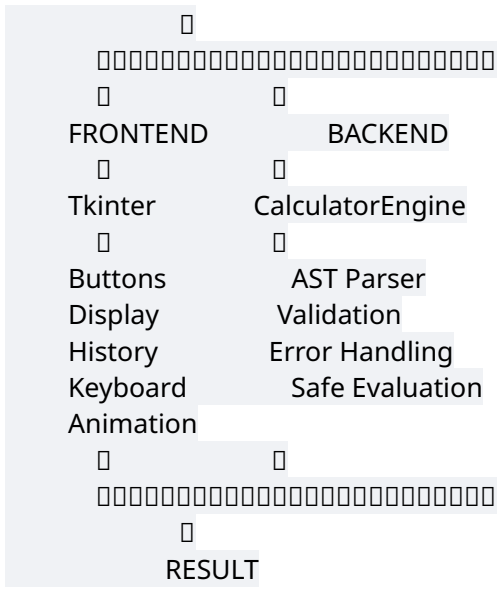
74. What You Built

You started with a simple idea:

```
Calculator
```

and turned it into a structured Python application:

```
                CALCULATOR
```



The important lesson is not only how to build a calculator.

The important lesson is how to take Python concepts and combine them into a complete application.

You have now practiced:

```
Python
+
OOP
+
Modules
+
Exceptions
+
Validation
+
GUI Development
```

+
Event Handling
+
Application Architecture
+
Testing
+
User Experience

This is the transition from:

Learning Python syntax

to:

Building with Python.

Visit haas.dev for more resources and guides.