

```
import numpy as np
import math

class Calculator:
    def __init__(self, stack_size=100):
        self.stack = []
        self.stack_size = stack_size
        self.operations = {}
        self.register_default_operations()

    def push(self, value):
        if len(self.stack) >= self.stack_size:
            raise OverflowError("Stack overflow")
        self.stack.append(value)

    def pop(self):
        if not self.stack:
            raise IndexError("Stack underflow")
        return self.stack.pop()

    def register_operation(self, name, func):
        self.operations[name] = func

    def execute(self, op_name, *args):
        if op_name not in self.operations:
            raise ValueError(f"Operation '{op_name}' not registered.")
        return self.operations[op_name](self, *args)

    def register_default_operations(self):
        # Basic arithmetic
        self.register_operation('add', lambda self:
self.push(self.pop() + self.pop()))
        self.register_operation('sub', lambda self:
self._binary_op(lambda a,b: b - a))
        self.register_operation('mul', lambda self:
self.push(self.pop() * self.pop()))
        self.register_operation('div', lambda self:
self._binary_op(lambda a,b: b / a))

        # Engineering functions (scalar)
        self.register_operation('sin', lambda self:
self.push(math.sin(self.pop())))
        self.register_operation('cos', lambda self:
self.push(math.cos(self.pop())))
        self.register_operation('tan', lambda self:
self.push(math.tan(self.pop())))
        self.register_operation('log', lambda self:
self.push(math.log10(self.pop())))
        self.register_operation('ln', lambda self:
```

```

self.push(math.log(self.pop()))
    self.register_operation('pow', lambda self:
self._binary_op(lambda a,b: b ** a))

    # Matrix / Vector operations
    self.register_operation('mat_add', lambda self:
self._binary_op(np.add))
    self.register_operation('mat_sub', lambda self:
self._binary_op(np.subtract))
    self.register_operation('mat_mul', lambda self:
self._binary_op(np.matmul))
    self.register_operation('mat_transpose', lambda
self: self.push(np.transpose(self.pop())))
    self.register_operation('mat_det', lambda self:
self.push(np.linalg.det(self.pop())))
    self.register_operation('mat_inv', lambda self:
self.push(np.linalg.inv(self.pop())))
    self.register_operation('vec_dot', lambda self:
self._binary_op(np.dot))
    self.register_operation('vec_cross', lambda self:
self._binary_op(np.cross))

def _binary_op(self, func):
    a = self.pop()
    b = self.pop()
    result = func(a, b)
    self.push(result)

def __str__(self):
    return f"Stack: {self.stack}"

def main():
    calc = Calculator()

    print("Welcome to the scalable Casio-like calculator!")
    print("Type 'push <value>', 'pop', 'exec <operation>',
'stack', or 'quit'")
    print("Example: push 5, push 3, exec add")
    print("Supports scalars and numpy arrays
(vectors/matrices).")
    print("Operations: add, sub, mul, div, sin, cos, tan, log,
ln, pow, mat_add, mat_sub, mat_mul, mat_transpose,
mat_det, mat_inv, vec_dot, vec_cross")

    while True:
        inp = input(">>> ").strip()
        if inp.lower() in ('quit', 'exit'):
            break
        if inp.startswith('push '):
            val_str = inp[5:]
            try:

```

```

# Try to parse numpy array
val = eval(val_str, {"np": np, "array": np.array,
"matrix": np.matrix})
except:
    # fallback: try float
    try:
        val = float(val_str)
    except:
        print("Invalid value. Use float or numpy array
notation, e.g., np.array([1,2,3])")
        continue
    calc.push(val)
    print(calc)
elif inp == 'pop':
    try:
        val = calc.pop()
        print(f"Popped: {val}")
    except Exception as e:
        print(e)
elif inp.startswith('exec '):
    op = inp[5:]
    try:
        calc.execute(op)
        print(calc)
    except Exception as e:
        print(e)
elif inp == 'stack':
    print(calc)
else:
    print("Unknown command.")

if __name__ == "__main__":
    main() write for arduino lcd shield with crete the
    arduino by key pad calculatior

```

You want to create an **Arduino calculator** that uses a **keypad** for input and an **LCD shield** to display results — basically a handheld calculator similar to your Python example but on Arduino hardware.

Here's a simple but complete example showing how to build a calculator on an Arduino using:

- A **4x4 matrix keypad** (like the common keypad shield)
- A **16x2 LCD display** (the common LCD shield or LCD module)
- Basic arithmetic operations (+, -, \*, /)

## What you need:

- Arduino Uno (or compatible)
- 4x4 Keypad
- 16x2 LCD (with I2C or direct pins)
- Jumper wires or shield setup

## Arduino Calculator Sketch

This sketch:

- Reads digits and operators from the keypad
- Displays input on the LCD
- Performs calculation on pressing '='
- Supports basic arithmetic only (no matrix ops)
- Clears on pressing 'C'

cpp



Copy



Edit

```
#include <Keypad.h> #include <LiquidCrystal.h> // LCD pins (change
according to your wiring) LiquidCrystal lcd(8, 9, 4, 5, 6, 7); // RS, E,
D4, D5, D6, D7 // Keypad setup const byte ROWS = 4; const byte COLS = 4;
// Define the keymap (common calculator layout) char keys[ROWS][COLS] =
{ {'1', '2', '3', '+'}, {'4', '5', '6', '-'}, {'7', '8', '9', '*'},
{'C', '0', '=', '/'} }; byte rowPins[ROWS] = {A0, A1, A2, A3}; //
Connect to row pins of the keypad byte colPins[COLS] = {A4, A5, 2, 3};
// Connect to column pins of the keypad Keypad keypad =
Keypad(makeKeymap(keys), rowPins, colPins, ROWS, COLS); String
currentInput = ""; // To store input digits/operators double operand1 =
0, operand2 = 0; char operation = 0; bool waitingForSecondOperand =
false; void setup() { lcd.begin(16, 2); lcd.print("Arduino Calc");
delay(1500); lcd.clear(); lcd.print("Enter number:"); } void loop() {
char key = keypad.getKey(); if (key != NO_KEY) { if (key >= '0' && key
<= '9') { // If we already have an operation and waiting second operand,
append to currentInput currentInput += key; lcd.clear();
lcd.print(currentInput); } else if (key == '+' || key == '-' || key ==
'*' || key == '/') { if (!waitingForSecondOperand &&
currentInput.length() > 0) { operand1 = currentInput.toDouble();
operation = key; waitingForSecondOperand = true; currentInput = ""; //
Clear input for second operand lcd.clear(); lcd.print(operand1);
lcd.print(" "); lcd.print(operation); } } else if (key == '=') { if
(waitingForSecondOperand && currentInput.length() > 0) { operand2 =
currentInput.toDouble(); double result = calculate(operand1, operand2,
operation); lcd.clear(); lcd.print("Result:"); lcd.setCursor(0, 1);
lcd.print(result, 6); // Display result with 6 decimal places // Reset
for next calculation currentInput = ""; waitingForSecondOperand = false;
```

```
operation = 0; operand1 = result; // Optionally keep result as new
operand1 for chaining } } else if (key == 'C') { clearCalculator(); } }
} double calculate(double a, double b, char op) { switch (op) { case
'+': return a + b; case '-': return a - b; case '*': return a * b; case
'/': return (b != 0) ? a / b : 0; // simple divide by zero handling
default: return 0; } } void clearCalculator() { currentInput = "";
operand1 = 0; operand2 = 0; operation = 0; waitingForSecondOperand =
false; lcd.clear(); lcd.print("Cleared"); delay(1000); lcd.clear();
lcd.print("Enter number:"); }
```

## How this works:

- You press digits on the keypad — they show on the LCD.
- Press an operator ( + , - , \* , / ).
- Enter second number.
- Press = to calculate and display the result.
- Press c to clear and reset.

## Wiring hints:

- Connect your keypad rows and cols to the Arduino analog/digital pins as declared ( A0 - A5 , 2 , 3 ) or change pins accordingly.
- Connect the LCD according to your module. The example assumes a 6-pin LCD (RS, E, D4-D7) connected to Arduino pins 8,9,4,5,6,7. Adjust if you use I2C or other wiring.

If you want me to generate code for **I2C LCD** or add **more advanced features** (like decimal points, negative numbers, or a bigger expression parser), just ask!



in over riding stack the python code for all 100 numbers  
in basic scientific calculatpor

Got it! You want to **override the stack in your Python calculator** so it can handle **up to 100 numbers** robustly and fully support basic scientific calculator operations over that stack.

I'll help you:

- Expand your existing calculator class
- Enforce max stack size = 100
- Add full support for basic scientific operations (sin, cos, tan, log, ln, pow)
- Ensure proper handling of stack overflow/underflow
- Make sure operations work on the last two or one stack elements correctly
- Provide a clean overridden pop/push and operation handling that can scale to 100+ numbers

Here's a full improved version of your Python calculator **with overridden stack behavior and proper scientific operations:**

python



Copy



Edit

```
import numpy as np import math class Calculator: def __init__(self,
stack_size=100): self.stack = [] self.stack_size = stack_size
self.operations = {} self.register_default_operations() # Override push
to enforce max stack size def push(self, value): if len(self.stack) >=
self.stack_size: raise OverflowError(f"Stack overflow: max size
{self.stack_size} reached.") self.stack.append(value) # Override pop
with underflow check def pop(self): if not self.stack: raise
IndexError("Stack underflow: stack is empty.") return self.stack.pop()
def register_operation(self, name, func): self.operations[name] = func
def execute(self, op_name, *args): if op_name not in self.operations:
raise ValueError(f"Operation '{op_name}' not registered.") return
self.operations[op_name](self, *args) def
register_default_operations(self): # Binary ops use _binary_op helper,
unary use _unary_op helper # Basic arithmetic
self.register_operation('add', lambda self: self._binary_op(lambda a,b:
b + a)) self.register_operation('sub', lambda self:
self._binary_op(lambda a,b: b - a)) self.register_operation('mul',
lambda self: self._binary_op(lambda a,b: b * a))
self.register_operation('div', lambda self: self._binary_op(lambda a,b:
b / a if a != 0 else float('inf')))) # Scientific unary functions
self.register_operation('sin', lambda self: self._unary_op(math.sin))
self.register_operation('cos', lambda self: self._unary_op(math.cos))
self.register_operation('tan', lambda self: self._unary_op(math.tan))
self.register_operation('log', lambda self: self._unary_op(lambda x:
math.log10(x) if x > 0 else float('nan')))) self.register_operation('ln',
lambda self: self._unary_op(lambda x: math.log(x) if x > 0 else
float('nan')))) self.register_operation('pow', lambda self:
self._binary_op(lambda a,b: b ** a)) self.register_operation('sqrt',
lambda self: self._unary_op(lambda x: math.sqrt(x) if x >= 0 else
float('nan')))) self.register_operation('neg', lambda self:
self._unary_op(lambda x: -x)) # Matrix / Vector operations
self.register_operation('mat_add', lambda self: self._binary_op(np.add))
self.register_operation('mat_sub', lambda self:
self._binary_op(np.subtract)) self.register_operation('mat_mul', lambda
self: self._binary_op(np.matmul))
self.register_operation('mat_transpose', lambda self:
```

```

self._unary_op(np.transpose)) self.register_operation('mat_det', lambda
self: self._unary_op(np.linalg.det)) self.register_operation('mat_inv',
lambda self: self._unary_op(np.linalg.inv))
self.register_operation('vec_dot', lambda self: self._binary_op(np.dot))
self.register_operation('vec_cross', lambda self:
self._binary_op(np.cross)) # Binary operation helper: pops two values,
applies func, pushes result def _binary_op(self, func): a = self.pop() b
= self.pop() result = func(a, b) self.push(result) # Unary operation
helper: pops one value, applies func, pushes result def _unary_op(self,
func): a = self.pop() result = func(a) self.push(result) def
__str__(self): # Show stack nicely, limit display for large stacks if
len(self.stack) > 20: displayed = ', '.join(map(str, self.stack[-10:]))
return f"Stack (last 10 of {len(self.stack)}): [{displayed}]" else:
return f"Stack: {self.stack}" # Example usage: if __name__ ==
"__main__": calc = Calculator() # Push 100 numbers to test overflow
protection try: for i in range(100): calc.push(i + 1) print(calc) #
Pushing one more should raise OverflowError calc.push(101) except
OverflowError as e: print(e) # Test basic arithmetic calc.push(5)
calc.push(3) calc.execute('add') print(calc) # Should show 8 on top #
Test scientific function calc.push(math.pi / 2) calc.execute('sin')
print(calc) # Should show 1 # Test matrix operations
calc.push(np.array([[1, 2], [3, 4]])) calc.execute('mat_det')
print(calc) # Should show determinant -2.0

```

## What's new/different:

- push() now **raises** OverflowError if you exceed 100 items.
- pop() raises IndexError if stack empty.
- Two helpers: \_binary\_op and \_unary\_op for ease of adding more ops.
- Added some common scientific functions including sqrt , neg (negate).
- For math domain errors (like log of negative), returns nan .
- Stack \_\_str\_\_ limits display when large.
- Included a simple test in \_\_main\_\_ .

**You can extend this for more operations easily by calling register\_operation with appropriate lambda or function.**

If you want me to write an **interactive console** like your first example that supports pushing 100+ numbers and scientific functions, or to add **history or undo** features, just ask!