Problem 1: Galactic Explorer's Map

```
# CelestialBody: base class for any space object, storing name and coordinates
class CelestialBody:
    def __init__(self, name, position):
        self.name = name
        self.position = position

    def describe(self):  # returns a human-readable description of the body
        return f"{self.name} at position {self.position}"

# Star: a CelestialBody with a temperature, shows in description
class Star(CelestialBody):
    def __init__(self, name, position, temperature):
        super().__init__(name, position)
        self.temperature = temperature

    def describe(self):  # overrides to include temperature in the text
        return f"{super().describe()}, a star with temperature
{self.temperature}K"

# Planet: adds gravity and list of moons, handles no-moons case gracefully
class Planet(CelestialBody):
    def __init__(self, name, position, gravity, moons):
        super().__init__(name, position)
        self.gravity = gravity
        self.moons = moons

    def describe(self):  # builds a comma-separated moons string or 'no moons'
        moons_str = ", ".join(self.moons) if self.moons else "no moons"
        return f"{super().describe()}, a planet with gravity {self.gravity}m/s²
and moons: {moons_str}"

# Galaxy: container for bodies, iterable, recursive lookup by name
class Galaxy:
    def __init__(self, name, bodies):
        self.name = name
        self.bodies = bodies

    def find_planet(self, name, current=None):  # recursively finds a Planet by
name
        if current is None:
            current = self.bodies
        for body in current:
            if isinstance(body, Planet) and body.name == name:
                return body
            if hasattr(body, 'moons'):
                result = self.find_planet(name, body.moons)
                if result:
                    return result
        return None
```

```
    def __iter__(self):  # makes Galaxy directly iterable over its bodies
        return iter(self.bodies)

# NavigationStack: simple stack with error checks for empty operations
class NavigationStack:
    def __init__(self):
        self.stack = []

    def push(self, location):  # add item on top
        self.stack.append(location)

    def pop(self):  # remove and return top, error if empty
        if self.is_empty():
            raise IndexError("Stack is empty")
        return self.stack.pop()

    def top(self):  # peek top without removing, error if empty
        if self.is_empty():
            raise IndexError("Stack is empty")
        return self.stack[-1]

    def is_empty(self):  # check if no elements
        return len(self.stack) == 0

    def __len__(self):  # support len(stack)
        return len(self.stack)
```

Problem 2: Election Tally

```
# You may assume that the CSV file is in the same directory as this script.
import micropip # Don't delete this
await micropip.install('pandas') # Don't delete this
import pandas as pd # Delete this if you found a better way...? I guess...?

class InvalidDataError(Exception):
    pass

# analyze_election_data: loads CSV, validates, aggregates votes, finds winners
def analyze_election_data(filename):
    try:
        df = pd.read_csv(filename)  # may raise FileNotFoundError
    except FileNotFoundError:
        raise FileNotFoundError(f"File {filename} not found")
    except Exception:
        raise InvalidDataError("CSV is empty.")  # no rows or unreadable

    # ensure required columns exist
    if not all(col in df.columns for col in ['program', 'candidate', 'votes']):
        raise InvalidDataError("CSV must contain 'program', 'candidate', and
'votes' columns")
```

```
    # convert votes to int, error if invalid values
    numeric_votes = pd.to_numeric(df['votes'], errors='coerce')
    if numeric_votes.isnull().any():
        raise InvalidDataError("CSV contains missing or invalid vote entries.")
    df['votes'] = numeric_votes.astype(int)


    if df.empty:
        return set(), "", {}

    # group by program and candidate, sum votes
    total_votes_by_candidate = df.groupby('candidate')['votes'].sum()
    candidate_votes_set = set((str(c), v) for c, v in
total_votes_by_candidate.items())

    overall_winner = total_votes_by_candidate.idxmax() if not
total_votes_by_candidate.empty else ""

    # find winner per program
    program_winners = {}
    if 'program' in df.columns:
        for program, group in df.groupby('program'):
            if not group.empty and not group['votes'].empty:
                program_winner_candidate = group.loc[group['votes'].idxmax()]
['candidate']
                program_winners[str(program)] = str(program_winner_candidate)

    return [candidate_votes_set, overall_winner, program_winners]
```

Problem 3: Broken Weather Stations

```
# max_temperature: flatten grid, ignore None, return max or None if no data
def max_temperature(grid):
    if not grid or not grid[0]:
        raise ValueError("Grid is empty")
    flat = [temp for row in grid for temp in row if temp is not None]
    return max(flat) if flat else None

# average_temperature: compute mean of non-None values, error on empty grid
def average_temperature(grid):
    if not grid or not grid[0]:
        raise ValueError("Grid is empty")
    flat = [temp for row in grid for temp in row if temp is not None]
    return sum(flat) / len(flat) if flat else None

# increase_temperatures: list comprehension to add delta to each non-None entry
def increase_temperatures(grid, delta=2):
    return [[temp + delta if temp is not None else None for temp in row] for row
in grid]
```

```
# local_maxima: check each cell against four neighbors, collect positions
def local_maxima(grid):
    maxima = []
    rows, cols = len(grid), len(grid[0])
    for i in range(1, rows-1):
        for j in range(1, cols-1):
            current = grid[i][j]
            if current is None:
                continue
            neighbors = [grid[i-1][j], grid[i+1][j], grid[i][j-1], grid[i][j+1]]
            # only consider if all neighbors exist and current is strictly greater
            if all(n is not None for n in neighbors) and all(current > n for n in
neighbors):
                maxima.append((i, j))
    return maxima

# above_average: filter flat temperatures by those above overall mean
def above_average(grid):
    avg = average_temperature(grid)
    flat = [temp for row in grid for temp in row if temp is not None]
    return list(filter(lambda x: x > avg, flat))

# unique_temperatures: set of all non-None readings for deduplication
def unique_temperatures(grid):
    return set(temp for row in grid for temp in row if temp is not None)
```

Problem 4: Python Is Just a Fancy Calculator

```
# Expression: abstract base with evaluate() stub
class Expression:
    def evaluate(self):
        pass

# Number: leaf node, returns its numeric value
class Number(Expression):
    def __init__(self, value):
        self.value = value

    def evaluate(self):
        return self.value

# BinaryOperation: holds left/right Expressions and operator, performs calculation
class BinaryOperation(Expression):
    def __init__(self, left, operator, right):
        self.left = left
        self.operator = operator
        self.right = right

    def evaluate(self):
        left_val = self.left.evaluate()
        right_val = self.right.evaluate()
```

```
        if self.operator == '+':
            return left_val + right_val
        elif self.operator == '-':
            return left_val - right_val
        elif self.operator == '*':
            return left_val * right_val
        elif self.operator == '/':
            if right_val == 0:
                raise ValueError("Division by zero")
            return left_val / right_val
        else:
            raise ValueError("Invalid operator")

# parse_expression: recursively strip parentheses, split by operators in
precedence order
def parse_expression(s):
    s = s.strip()
    if not s:
        raise ValueError("Empty expression")

    # remove outer parentheses if they wrap entire expression
    while s.startswith('(') and s.endswith(')'):
        count = 0
        balanced = True
        for i in range(len(s)):
            if s[i] == '(':
                count += 1
            elif s[i] == ')':
                count -= 1
            if count == 0 and i != len(s) - 1:
                balanced = False
                break
        if balanced:
            s = s[1:-1].strip()
        else:
            break

    # simple number case
    try:
        return Number(float(s))
    except ValueError:
        pass

    # parse addition/subtraction at base depth
    depth = 0
    for i in range(len(s)-1, -1, -1):
        if s[i] == ')': depth += 1
        elif s[i] == '(': depth -= 1
        elif depth == 0 and s[i] in '+-':
            left = parse_expression(s[:i])
            right = parse_expression(s[i+1:])
            return BinaryOperation(left, s[i], right)

    # parse multiplication/division next
```

```
    depth = 0
    for i in range(len(s)-1, -1, -1):
        if s[i] == ')': depth += 1
        elif s[i] == '(': depth -= 1
        elif depth == 0 and s[i] in '*/':
            left = parse_expression(s[:i])
            right = parse_expression(s[i + 1:])
            return BinaryOperation(left, s[i], right)

    raise ValueError("Invalid expression")
```

This took too long 😅