📘 Lab Manual — OOP Data Structures Lab (2 hours)

References:

- **Lecture 4:** Algorithms, Linear Search, Balanced Parentheses, OOP Encapsulation
- **Lecture 6:** Stacks, Operator Precedence, Infix→Postfix Conversion, Postfix Evaluation

---

🔧 Python OOP Concepts Used in This Lab (Quick Primer)

- **Class & Object:** A *class* is a blueprint; an *object* is an instance created from that class.
- `__init__` **(Constructor):** Runs when an object is created; initialize internal state here.
- **Attributes (State):** Variables stored on the object (e.g., `self._data`, `self._stack`).
- **Methods (Behavior):** Functions defined inside a class that operate on the object's state.
- **Encapsulation:** Keep internals "private" by convention (single leading underscore, e.g., `_data`) and expose a clean API. *(Lecture 4)*
- **Composition:** Build bigger abstractions by using objects inside other objects (e.g., `BracketChecker` uses a `Stack`). *(Lecture 4)*
- **Abstraction:** Focus on what the object does; hide how it's implemented.
- **Exceptions:** Raise clear errors when a contract is violated (e.g., popping from an empty stack → `IndexError`).
- **Unit-style checks:** Use `assert` in cells to quickly verify behavior—mini tests embedded in the notebook.

🗓️ Agenda (120 minutes)

1. Warm-up & scaffold (10 min)
2. **Task-1:** `Stack` class (array-based) (20 min) — *Lecture 6*
3. **Task-2:** `BracketChecker` (balanced parentheses using Stack) (25 min) — *Lecture 4*
4. **Task-3:** `InfixToPostfix` (Shunting-yard lite) (25 min) — *Lecture 6*
5. **Task-4:** `PostfixEvaluator` (evaluate RPN) (20 min) — *Lecture 6*
6. **Task-5:** `LinearSearch` (class + method + tests) (15 min) — *Lecture 4*
7. Wrap-up & quick quiz (5 min)

> Work in this single notebook. Keep I/O minimal; focus on classes, docstrings, correctness.

🔥 Warm-up (10 min)

- Fill your **Name, Roll, Section** below.
- Then proceed task-by-task. Run each cell after writing/reading it.

Student Info:

- Name: *type here*
- Roll: *type here*
- Section: *type here*

✅ Task-1: Stack Class (array-backed) — *Lecture 6*

What & Why:
A **stack** is a Last-In-First-Out (LIFO) data structure. It's fundamental for expression conversion and evaluation.
You will implement a generic `Stack` with methods `push`, `pop`, `peek`, `is_empty`, and `size`.

Concepts Used: Encapsulation, methods, exceptions, unit-style asserts.
From Lecture 6: What a stack is; how it powers infix/postfix topics.

In [2]:
```
class Stack:
    def __init__(self):
        """Initialize internal storage for the stack using a Python list.
        Top of the stack will be at the *end* of the list for O(1) amortized pus
        """
        self._data = []  # internal list to hold items

    def push(self, item):
        """Place a new item on the top of the stack."""
        self._data.append(item)  # append adds at end (the top)

    def pop(self):
        """Remove and return the top item from the stack.
        Raises:
            IndexError: if the stack is empty (underflow).
        """
        if self.is_empty():  # guard against removing from empty stack
            raise IndexError("Stack underflow")
        return self._data.pop()  # remove and return last element

    def peek(self):
        """Return the top item without removing it.
        Raises:
            IndexError: if the stack is empty.
        """
        if self.is_empty():
```

```
            raise IndexError("Empty stack")
        return self._data[-1]  # last element is the top

    def is_empty(self):
        """Return True if the stack has no elements; otherwise False."""
        return len(self._data) == 0

    def size(self):
        """Return the current number of elements in the stack."""
        return len(self._data)
```

In [3]:

```
# --- Quick tests for your Stack class ---

s = Stack()   # create a stack

# Check empty
print("Is stack empty?", s.is_empty())   # True
print("Current Stack:", s._data)

# Push elements
s.push(10)
print("Pushed 10 → Stack:", s._data)

s.push(20)
print("Pushed 20 → Stack:", s._data)

# Peek element
print("Peek →", s.peek())
print("Stack after peek:", s._data)

# Pop elements
print("Pop →", s.pop())
print("Stack after pop:", s._data)

print("Pop →", s.pop())
print("Stack after pop:", s._data)

# Check empty again
print("Is stack empty?", s.is_empty())
print("Current Stack:", s._data)

# Try pop on empty (will raise error)
try:
    s.pop()
except IndexError as e:
    print("Error:", e)
```

```
Is stack empty? True
Current Stack: []
Pushed 10 → Stack: [10]
Pushed 20 → Stack: [10, 20]
Peek → 20
Stack after peek: [10, 20]
Pop → 20
Stack after pop: [10]
Pop → 10
Stack after pop: []
Is stack empty? True
Current Stack: []
Error: Stack underflow
```

✅ Task-2: BracketChecker (Balanced Parentheses) — *Lecture 4*

What & Why:
Use a **stack** to check if an expression has balanced brackets: `()`, `[]`, `{}`.
Algorithm (Lecture 4): scan left→right; push openings; on closing, pop and match; at end, stack must be empty.

Concepts Used: Composition (uses `Stack`), control flow, early returns, encapsulation.

In [5]:
```python
class BracketChecker:
    def __init__(self):
        """Create a new bracket checker that uses an internal Stack instance."""
        self._stack = Stack()  # composition: use a Stack inside

    def _is_open(self, ch):
        """Return True if character is an opening bracket."""
        return ch in "([{"

    def _matches(self, open_br, close_br):
        """Return True if open_br correctly matches close_br."""
        pairs = {')': '(', ']': '[', '}': '{'}  # mapping closing -> opening
        return pairs.get(close_br) == open_br

    def is_balanced(self, expr: str) -> bool:
        """Return True if expr has balanced (), [], {}; False otherwise.
        Ignores non-bracket characters.
        """
        self._stack = Stack()  # reset stack per call
        for ch in expr:
            if self._is_open(ch):
                self._stack.push(ch)  # push openings
            elif ch in ")]}":
                if self._stack.is_empty():  # unmatched closing bracket
                    return False
                top = self._stack.pop()     # get last opening
                if not self._matches(top, ch):  # mismatch pair
                    return False
            # ignore other characters
        return self._stack.is_empty()  # balanced only if nothing left
```

In [6]:
```python
# --- Quick tests for Task-2 ---

bc = BracketChecker()   # create object

# Test cases with prints
expr1 = "{[()]}"
print(expr1, "→ Balanced?", bc.is_balanced(expr1))

expr2 = "([)]"
print(expr2, "→ Balanced?", bc.is_balanced(expr2))

expr3 = "(((())))"
```

```
print(expr3, "→ Balanced?", bc.is_balanced(expr3))

expr4 = ")("
print(expr4, "→ Balanced?", bc.is_balanced(expr4))

print("✅ Task-2: BracketChecker tests done")
```

```
{[()]} → Balanced? True
([)] → Balanced? False
(((()))) → Balanced? True
)( → Balanced? False
✅ Task-2: BracketChecker tests done
```

✅ Task-3: Infix→Postfix Converter — *Lecture 6*

What & Why:

Computers prefer **postfix (RPN)** because it avoids parentheses and is easy to evaluate. We'll convert infix (e.g., `A + B * C`) to postfix (e.g., `ABC*+`) using **operator precedence** and a **stack**.

Rules (short):

- Operands → output immediately.
- Operators → pop higher/equal precedence from stack first (handle `^` as right-associative).
- `(` pushes; `)` pops until matching `(`.
- End: pop remaining operators.

In [7]:
```
class InfixToPostfix:
    def __init__(self):
        """Initialize precedence and the internal operator stack."""
        self._prec = {'^': 3, '*': 2, '/': 2, '+': 1, '-': 1}  # precedence map
        self._right_assoc = {'^'}                               # right-associat
        self._stack = Stack()                                   # operator stack

    def _is_operand(self, ch):
        """Return True if ch is an operand (letter or digit)."""
        return ch.isalnum()

    def convert(self, infix: str) -> str:
        """Convert an infix expression (single-char tokens) to postfix (RPN)."""
        out = []                # output token list
        self._stack = Stack()   # reset operator stack

        # remove spaces so we can scan char by char
        for ch in infix.replace(" ", ""):
            if self._is_operand(ch):
                out.append(ch)  # operands go straight to output
            elif ch == '(':
                self._stack.push(ch)  # push opening parenthesis
            elif ch == ')':
                # pop operators until '(' is found
```

```
                while not self._stack.is_empty() and self._stack.peek() != '(':
                    out.append(self._stack.pop())
                if self._stack.is_empty():
                    raise ValueError("Mismatched parentheses")  # no matching '('
                self._stack.pop()  # discard '('
            else:
                # operator case: pop while stack top has higher precedence
                # or equal precedence for left-associative operators
                while (not self._stack.is_empty()
                       and self._stack.peek() != '('
                       and (self._prec[self._stack.peek()] > self._prec[ch]
                            or (self._prec[self._stack.peek()] == self._prec[ch]
                                and ch not in self._right_assoc))):
                    out.append(self._stack.pop())
                self._stack.push(ch)  # finally push current operator

        # flush remaining operators
        while not self._stack.is_empty():
            top = self._stack.pop()
            if top == '(':
                raise ValueError("Mismatched parentheses")  # stray '('
            out.append(top)

        return "".join(out)  # join tokens to make postfix string
```

In [8]:
```
# --- Quick tests for Task-3 with printing ---
conv = InfixToPostfix()

expr1 = "A+B*C"
result1 = conv.convert(expr1)
print(f"{expr1} → {result1}")

expr2 = "(A+B)*C"
result2 = conv.convert(expr2)
print(f"{expr2} → {result2}")

expr3 = "A^B^C"
result3 = conv.convert(expr3)
print(f"{expr3} → {result3}")

expr4 = "A*(B+C*D)"
result4 = conv.convert(expr4)
print(f"{expr4} → {result4}")

print("✅ Task-3: InfixToPostfix tests done")
```

```
A+B*C → ABC*+
(A+B)*C → AB+C*
A^B^C → ABC^^
A*(B+C*D) → ABCD*+*
✅ Task-3: InfixToPostfix tests done
```

✅ Task-4: PostfixEvaluator — *Lecture 6*

What & Why:

Evaluate **postfix (RPN)** with a stack:

- If token is an operand → push it.
- If token is an operator → pop two, apply operator, push result.
- End state must have exactly one value = result.

Note: This simple version handles **single-digit** operands for clarity.

In [10]:
```
class PostfixEvaluator:
    def __init__(self):
        """Initialize with an internal stack of numbers."""
        self._stack = Stack()

    def _apply(self, op, b, a):
        """Apply binary operator 'op' to operands a (left) and b (right)."""
        if op == '+': return a + b
        if op == '-': return a - b
        if op == '*': return a * b
        if op == '/': return a / b
        if op == '^': return a ** b
        raise ValueError(f"Unknown operator {op}")

    def evaluate(self, postfix: str) -> float:
        """Evaluate a postfix string containing single-digit operands."""
        self._stack = Stack()  # reset per call
        for ch in postfix.replace(" ", ""):
            if ch.isdigit():                # operand
                self._stack.push(float(ch))  # push numeric value
            elif ch in "+-*/^":            # operator
                if self._stack.size() < 2:
                    raise ValueError("Malformed expression: insufficient operand
                b = self._stack.pop()        # right operand
                a = self._stack.pop()        # left operand
                self._stack.push(self._apply(ch, b, a))
            else:
                raise ValueError(f"Bad token {ch}")  # reject unknown tokens
        if self._stack.size() != 1:
            raise ValueError("Malformed expression: leftover values")
        return self._stack.pop()             # final result
```

In [11]:
```
# --- Quick tests for Task-4 with printing ---
ev = PostfixEvaluator()

expr1 = "432+*"
res1 = ev.evaluate(expr1)
print(f"{expr1} → {res1}")   # (4 * (3+2)) = 20

expr2 = "23+5*"
res2 = ev.evaluate(expr2)
print(f"{expr2} → {res2}")   # (2+3)*5 = 25

expr3 = "82/3-"
res3 = ev.evaluate(expr3)
print(f"{expr3} → {res3}")   # (8/2) - 3 = 1

print("✅ Task-4: PostfixEvaluator tests done")
```

```
432+* → 20.0
23+5* → 25.0
82/3- → 1.0
✅ Task-4: PostfixEvaluator tests done
```

In [ ]:

✅ Task-5: LinearSearch — *Lecture 4*

What & Why:
Linear search is the simplest search algorithm: scan from the start, compare each element, return index if found, else `-1`.
Time: O(n), Space: O(1). Great for understanding algorithm steps (Lecture 4).

Concepts Used: Simple class wrapper, iteration, conditionals, unit-style asserts.

In [13]:
```python
class LinearSearch:
    def __init__(self, data):
        """Store a copy of the data to avoid external mutation side-effects."""
        self.data = list(data)

    def find(self, target):
        """Return index of first occurrence of target; -1 if not found."""
        for i, x in enumerate(self.data):  # scan left to right
            if x == target:                # match?
                return i                   # return index of first match
        return -1                          # not found
```

In [14]:
```python
# --- Quick tests for Task-5 with printing ---
ls = LinearSearch([10, 30, 20, 50])

# Search for an existing element
target1 = 20
res1 = ls.find(target1)
print(f"Searching for {target1} → Index: {res1}")   # expected 2

# Search for a missing element
target2 = 99
res2 = ls.find(target2)
print(f"Searching for {target2} → Index: {res2}")   # expected -1

print("✅ Task-5: LinearSearch tests done")
```

```
Searching for 20 → Index: 2
Searching for 99 → Index: -1
✅ Task-5: LinearSearch tests done
```

🎯 Wrap-up & Quick Quiz

What you built:

- Reusable `Stack` class (Lecture 6)
- `BracketChecker` using `Stack` (Lecture 4)
- `InfixToPostfix` converter (Lecture 6)
- `PostfixEvaluator` (Lecture 6)
- `LinearSearch` (Lecture 4)

Quick Quiz (write your answers below):

1. Why does postfix evaluation not need parentheses? *(Hint: operator order is encoded by position)*
2. What condition makes parentheses **not balanced** during scanning? *(Hint: early closing, mismatched pair, leftover openings)*
3. In infix→postfix, when do we pop an operator of **equal** precedence from the stack? *(Hint: for left-associative ops)*