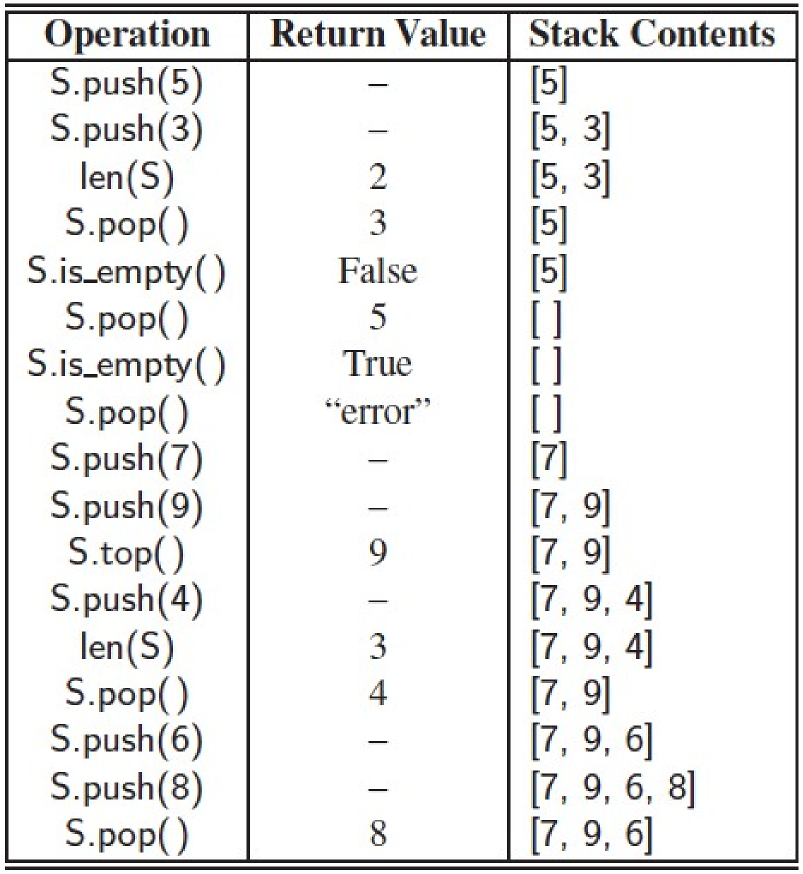
Stacks

Example



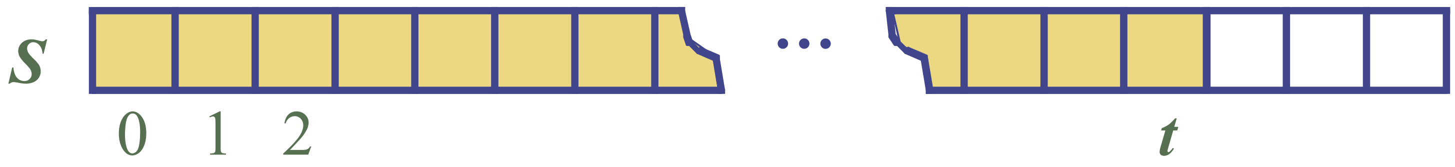
Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop()	3	[5]
S.is_empty()	False	[5]
S.pop()	5	[ ]
S.is_empty()	True	[ ]
S.pop()	“error”	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top()	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop()	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop()	8	[7, 9, 6]

Applications of Stacks

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Expression Evaluations
- Function Calls
- Backtracking Algorithms
- Auxiliary data structure for algorithms
- Component of other data structures

Array-based Stack

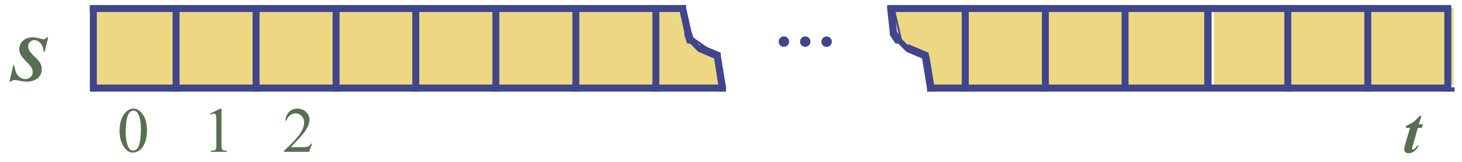
- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element



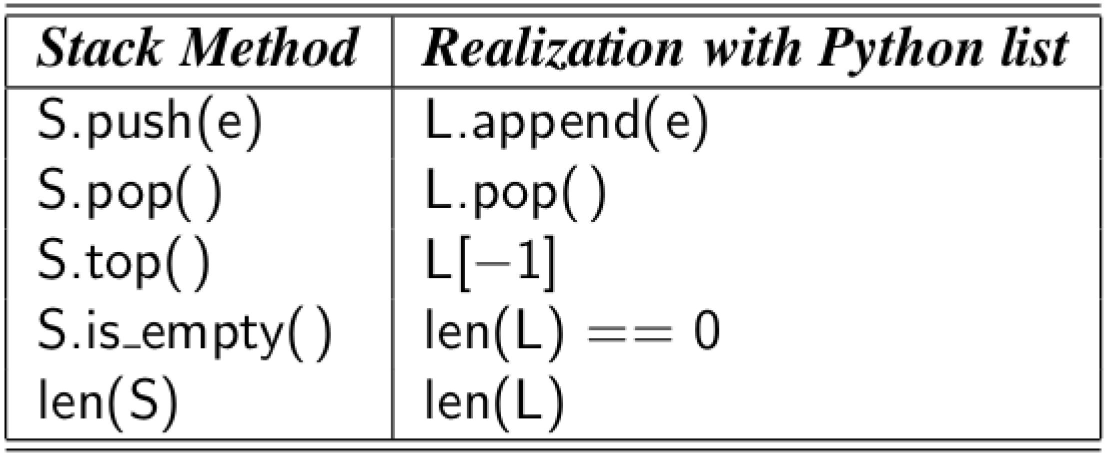
Array-based Stack (cont.)

The array storing the stack elements may become full

A push operation will then need to grow the array and copy all the elements over.



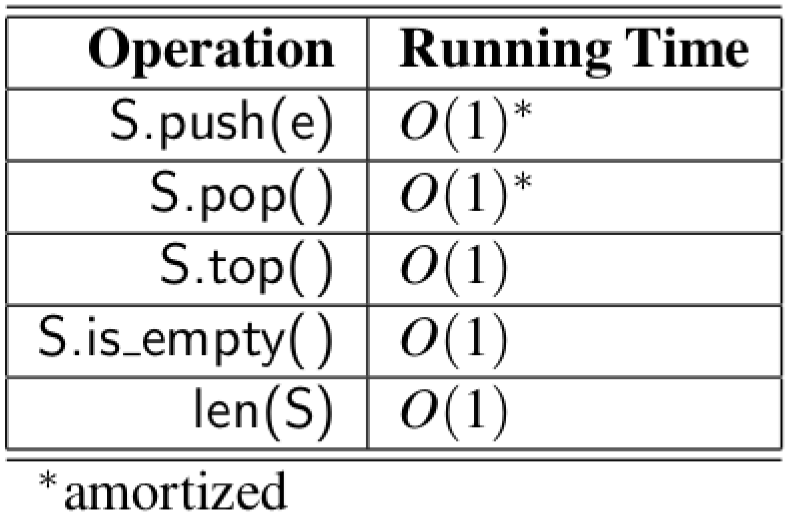
Applying adapter design pattern



Stack Method	*Realization with Python list*
S.push(e)	L.append(e)
S.pop( )	L.pop( )
S.top( )	L[−1]
S.is_empty( )	len(L) == 0
len(S)	len(L)

Realization of a stack S as an adaptation of a Python list L

Performance of our array-based stack implementation



Operation	Running Time
S.push(e)	$O(1)^*$
S.pop()	$O(1)^*$
S.top()	$O(1)$
S.is_empty()	$O(1)$
len(S)	$O(1)$

*amortized

Performance and Limitations

Performance

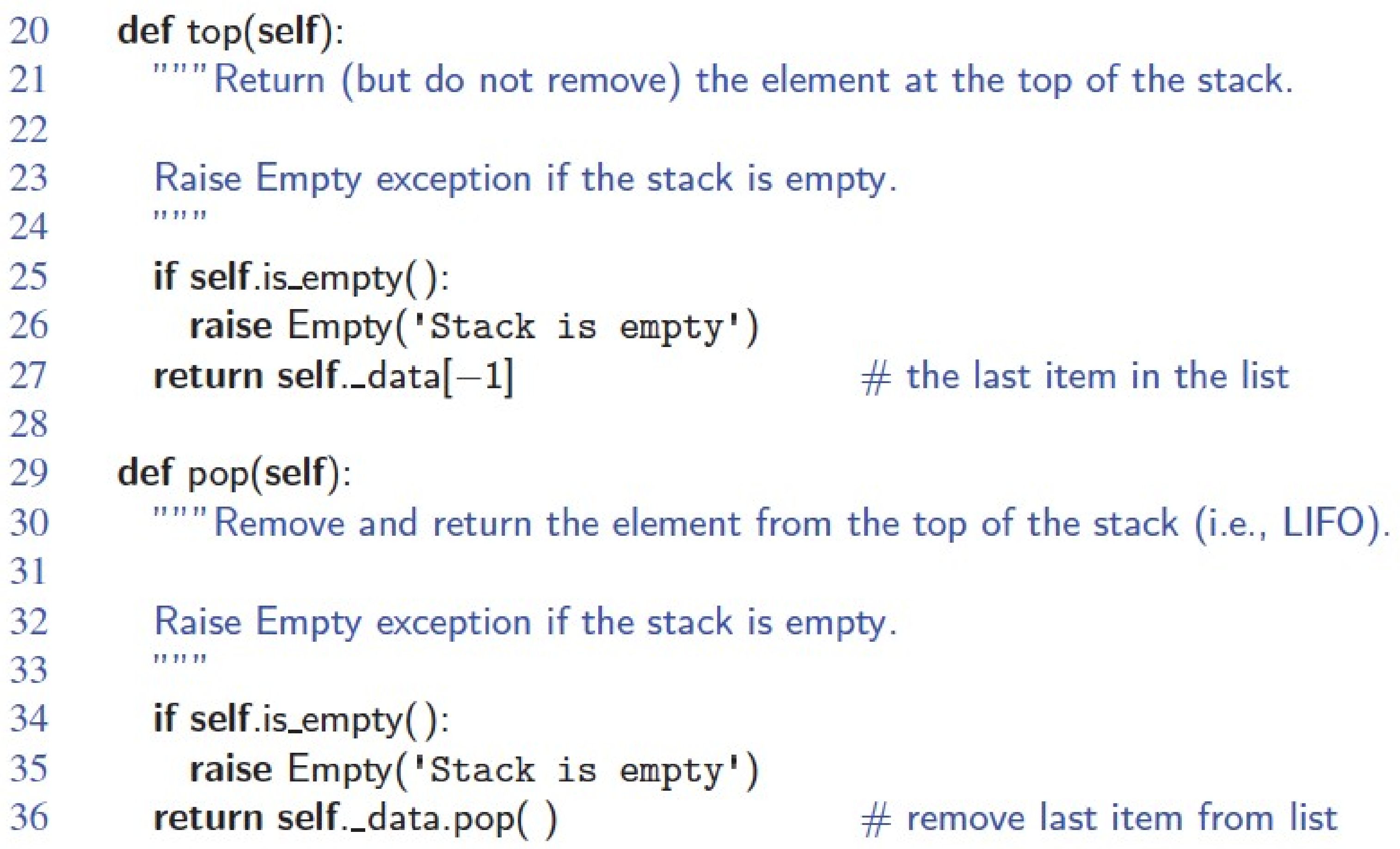
Let n be the number of elements in the stack: The space used is $O(n)$

Each operation runs in time $O(1)$ (amortized in the case of a push)

Array-based Stack in Python

```
class ArrayStack:
  """LIFO Stack implementation using a Python list as underlying storage."""

  def __init__(self):
    """Create an empty stack."""
    self._data = [ ]                        # nonpublic list instance

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

  def is_empty(self):
    """Return True if the stack is empty."""
    return len(self._data) == 0

  def push(self, e):
    """Add element e to the top of the stack."""
    self._data.append(e)                    # new item stored at end of list

```

Array-based Stack in Python



```
  def top(self):
    """Return (but do not remove) the element at the top of the stack.

    Raise Empty exception if the stack is empty.
    """
    if self.is_empty():
      raise Empty('Stack is empty')
    return self._data[−1]                    # the last item in the list

  def pop(self):
    """Remove and return the element from the top of the stack (i.e., LIFO).

    Raise Empty exception if the stack is empty.
    """
    if self.is_empty():
      raise Empty('Stack is empty')
    return self._data.pop( )                 # remove last item from list
```

Parentheses Matching

Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”

correct: ( )(( )){([( )])}

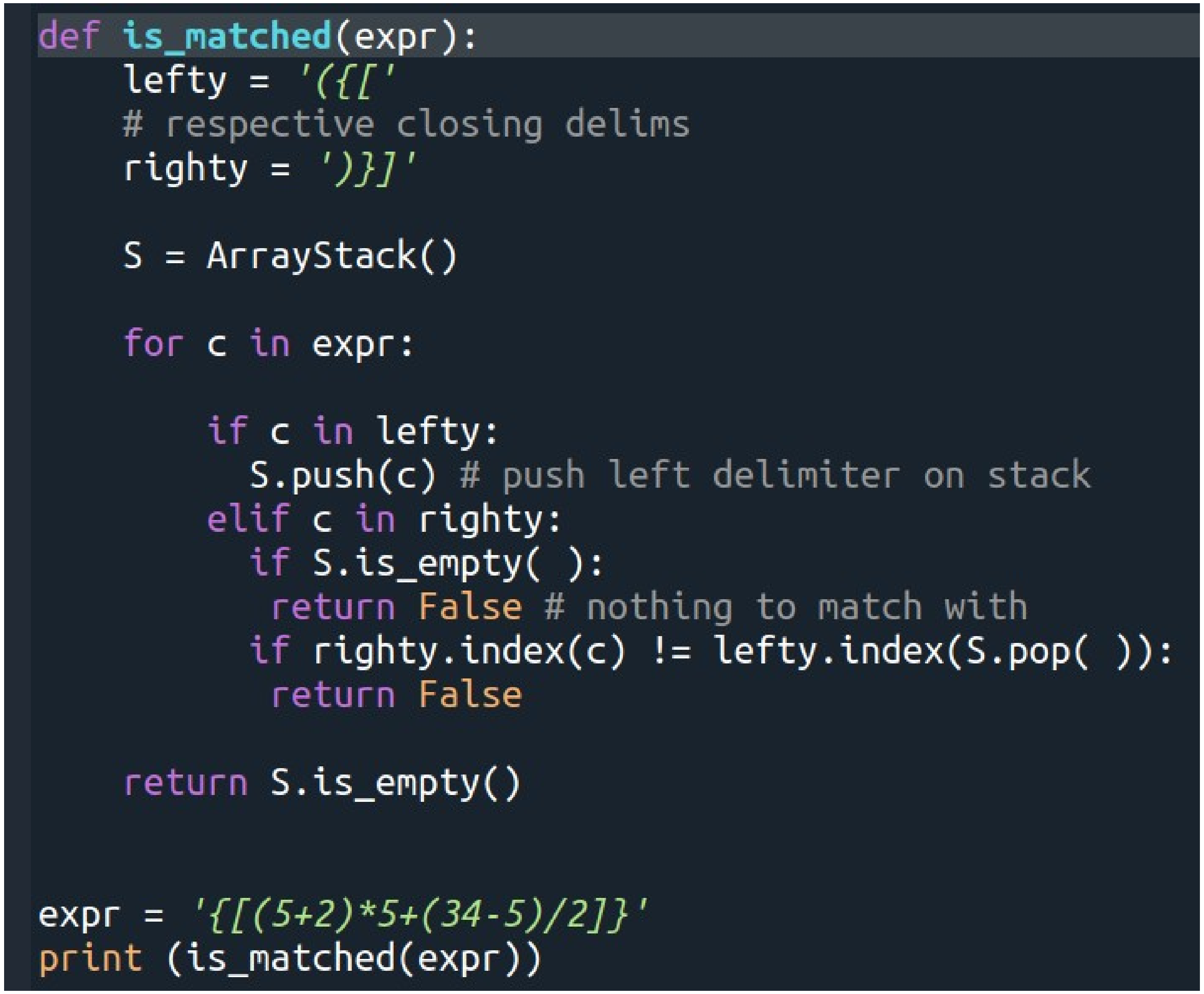
correct: ((( )(( )){([( )])}

incorrect:)(( )){([( )])}

incorrect: ({[ ])}

incorrect: (

Parentheses Matching in Python



```python
def is_matched(expr):
    lefty = '({['
    # respective closing delims
    righty = ')}]'

    S = ArrayStack()

    for c in expr:

        if c in lefty:
          S.push(c) # push left delimiter on stack
        elif c in righty:
          if S.is_empty( ):
           return False # nothing to match with
          if righty.index(c) != lefty.index(S.pop( )):
           return False

    return S.is_empty()


expr = '{[(5+2)*5+(34-5)/2]}'
print (is_matched(expr))
```

Questions

What values are returned during the following series of stack operations, if executed upon an initially empty stack? push(5), push(3), pop(), push(2), push(8), pop(), pop(), push(9), push(1), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop().

Suppose an initially empty stack S has executed a total of 25 push operations, 12 top operations, and 10 pop operations, 3 of which raised Empty errors that were caught and ignored. What is the current size of S?

Dijkstra’s Two-Stack Algorithm for Infix notation expressions

For each item of the infix expression:

1. A number: push it onto the value stack
2. An operator: push it onto the operator stack
3. Left parenthesis: ignore
4. Right parenthesis: Pop the operator from the operator stack. Pop the value stack twice, getting two operands. Apply the operator to those operands, in the correct order and Push the result onto the value stack again.

Caveat: every binary operation should be enclosed in parenthesis

eg. (5 + ((12 - 2) * (24 * 45)))

Generic Two-Stack Algorithm for Infix notation expressions

I. While there are still items to read, get the next item

For each item of the infix expression:

1. A number: push it onto the value stack
2. A left parenthesis: Push onto operator stack
3. A right parenthesis: do the following

   While item != left parenthesis:

   - Pop the operator from the operator stack.
   - Pop the value stack twice, getting two operands.
   - Apply the operator to the operands, in the correct order.
   - Push the result onto the value stack.

   Pop out the left parenthesis

Generic Two-Stack Algorithm for Infix notation expressions

4. An operator op:

   a) While operator stack != empty, and the top of the operator stack has the same or greater precedence than op:

      - Pop the operator from the operator stack.
      - Pop the value stack twice, getting two operands.
      - Apply the operator to the operands, in the correct order.
      - Push the result onto the value stack.

   b) Push op onto the operator stack.

Generic Two-Stack Algorithm for Infix notation expressions

II. Repeat till operator stack is empty,

- Pop the operator from the operator stack.
- Pop the value stack twice, getting two operands.
- Apply the operator to the operands, in the correct order.
- Push the result onto the value stack.

At this juncture the operator stack should be empty, and the value stack should have only one value in it, which is the final result.

Evaluation of a postfix expression using a single stack

```
For each item in postfix expression
        If an operand is found
           push it onto the stack
        End-If
        If an operator op is found
           Pop the stack and assign stack element as  B
           Pop the stack and assign stack element as  A
           Evaluate A op B using the operator just found.
           Push the resulting value onto the stack
        End-If
End-For
Pop the stack (this is the computed result)
```

Questions

How to you modify the previous algorithm for Postfix expressions if the inputs are of prefix expressions?

References

Data Structures and Algorithms in Python
Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser

Introduction to Algorithms
Leiserson,Stein,Rivest,Cormen

Algorithms, 4th Edition
Robert Sedgewick and Kevin Wayne

Few Images from the internet