Infix Conversion (STO2)

locked

Convert the given prefix expression to infix expression.

Input Format

A string **s** representing the prefix expression.

Constraints

1 <= |s| <= 1000

Output Format

Print the corresponding infix expression of **s** after conversion.

Sample Input 0

```
*+AB-CD
```

Sample Output 0

```
((A+B)*(C-D))
```

Current Buffer (saved locally, editable)

Python 3

```
#!/usr/bin/env python
'''
Q: Infix,  postfix, and prefix expression conversion and evaluation
For example:
Infix: 1 * (2 + 3) / 4
Postfix: 1 2 3 + * 4 /
Prefix: / * 1 + 2 3 4
Build up a binary tree with numbers as leaves and operators as internal nodes.
In-order traversal → infix
Post-order traversal → postfix
Pre-order traversal → prefix
'''
from __future__ import division
import random


OPERATORS = set(['+', '-', '*', '/', '(', ')'])
PRIORITY = {'+':1, '-':1, '*':2, '/':2}


### INFIX ===> POSTFIX ###
'''
1)Fix a priority level for each operator. For example, from high to low:
    3.    - (unary negation)
```

```
    2.    * /
    1.    + - (subtraction)
2) If the token is an operand, do not stack it. Pass it to the output.
3) If token is an operator or parenthesis:
    3.1) if it is '(', push
    3.2) if it is ')', pop until '('
    3.3) push the incoming operator if its priority > top operator; otherwise pop.
    *The popped stack elements will be written to output.
4) Pop the remainder of the stack and write to the output (except left parenthesis)
'''
# def infix_to_postfix(formula):
#     stack = [] # only pop when the coming op has priority
#     output = ''
#     for ch in formula:
#         if ch not in OPERATORS:
#             output += ch
#         elif ch == '(':
#             stack.append('(')
#         elif ch == ')':
#             while stack and stack[-1] != '(':
#                 output += stack.pop()
#             stack.pop() # pop '('
#         else:
#             while stack and stack[-1] != '(' and PRIORITY[ch] <= PRIORITY[stack[-1]]:
#                 output += stack.pop()
#             stack.append(ch)
#     # leftover
#     while stack: output += stack.pop()
#     print(output)
#     return output


### POSTFIX ===> INFIX ###
'''
1) When see an operand, push
```

```
2) When see an operator, pop out two numbers, connect them into a substring and push back to the stack
3) the top of the stack is the final infix expression.
'''
# def postfix_to_infix(formula):
#     stack = []
#     prev_op = None
#     for ch in formula:
#         if not ch in OPERATORS:
#             stack.append(ch)
#         else:
#             b = stack.pop()
#             a = stack.pop()
#             if prev_op and len(a) > 1 and PRIORITY[ch] > PRIORITY[prev_op]:
#                 # if previous operator has lower priority
#                 # add '()' to the previous a
#                 expr = '('+a+')' + ch + b
#             else:
#                 expr = a + ch + b
#             stack.append(expr)
#             prev_op = ch
#     print(stack[-1])
#     return stack[-1]


### INFIX ===> PREFIX ###
# def infix_to_prefix(formula):
#     op_stack = []
#     exp_stack = []
#     for ch in formula:
#         if not ch in OPERATORS:
#             exp_stack.append(ch)
#         elif ch == '(':
#             op_stack.append(ch)
#         elif ch == ')':
```

```
#             while op_stack[-1] != '(':
#                 op = op_stack.pop()
#                 a = exp_stack.pop()
#                 b = exp_stack.pop()
#                 exp_stack.append( op+b+a )
#             op_stack.pop() # pop '('
#         else:
#             while op_stack and op_stack[-1] != '(' and PRIORITY[ch] <= PRIORITY[op_stack[-1]]:
#                 op = op_stack.pop()
#                 a = exp_stack.pop()
#                 b = exp_stack.pop()
#                 exp_stack.append( op+b+a )
#             op_stack.append(ch)

#     # leftover
#     while op_stack:
#         op = op_stack.pop()
#         a = exp_stack.pop()
#         b = exp_stack.pop()
#         exp_stack.append( op+b+a )
#     print(exp_stack[-1])
#     return exp_stack[-1]


### PREFIX ===> INFIX ###
'''
Scan the formula reversely
1) When the token is an operand, push into stack
2) When the token is an operator, pop out 2 numbers from stack, merge them and push back to the stack
'''
def prefix_to_infix(formula):
    stack = []
    prev_op = None
    for ch in reversed(formula):
```

```
        if not ch in OPERATORS:
            stack.append(ch)
        else:
            a = stack.pop()
            b = stack.pop()
            if prev_op and PRIORITY[prev_op] < PRIORITY[ch]:
                exp = '('+a+ch+b+')'
            else:
                exp = '('+a+ch+b+')'
            stack.append(exp)
            prev_op = ch
    # print(stack[-1])
    return stack[-1]


'''
Scan the formula:
1) When the token is an operand, push into stack;
2) When an operator is encountered:
    2.1) If the operator is binary, then pop the stack twice
    2.2) If the operator is unary (e.g. unary minus), pop once
3) Perform the indicated operation on two poped numbers, and push the result back
4) The final result is the stack top.
'''
# def evaluate_postfix(formula):
#     stack = []
#     for ch in formula:
#         if ch not in OPERATORS:
#             stack.append(float(ch))
#         else:
#             b = stack.pop()
#             a = stack.pop()
#             c = {'+':a+b, '-':a-b, '*':a*b, '/':a/b}[ch]
#             stack.append(c)
#     print(stack[-1])
```

```
#     return stack[-1]


# def evaluate_infix(formula):
#     return evaluate_postflix(inflix_to_postfix(formula))


''' Whenever we see an operator following by two numbers,
we can compute the result.'''
# def evaluate_prefix(formula):
#     exps = list(formula)
#     while len(exps) > 1:
#         for i in range(len(exps)-2):
#             if exps[i] in OPERATORS:
#                 if not exps[i+1] in OPERATORS and not exps[i+2] in OPERATORS:
#                     op, a, b = exps[i:i+3]
#                     a,b = map(float, [a,b])
#                     c = {'+':a+b, '-':a-b, '*':a*b, '/':a/b}[op]
#                     exps = exps[:i] + [c] + exps[i+3:]
#                     break
#         prin(exps)
#     return exps[-1]



#infix_to_postfix('1+(3+4*6+6*1)*2/3')
#infix_to_prefix('1+(3+4*6+6*1)*2/3')
# print
#evaluate_inflix('1+(3+4*6+6*1)*2/3')
#evaluate_postfix('1346*+61*+2*3/+')
#evaluate_prefix('+1/*++3*46*6123')
# print
#postfix_to_infix('1346*+61*+2*3/+')
s=input()
print(prefix_to_infix(s))
```

200

Line: 27 Col: 14

Upload Code as File Test against custom input Run Code Submit Code

Testcase 0 ✔

Congratulations, you passed the sample test case.

Click the **Submit Code** button to run your code against all the test cases.

Input (stdin)

```
*+AB-CD
```

Your Output (stdout)

```
((A+B)*(C-D))
```

Expected Output

```
((A+B)*(C-D))
```