



PRACTICE

COMPETE

JOBS

LEADERBOARD

Search



swatantragoswam1

[All Contests](#) > [GCFL\\_3\\_year\\_6\\_sem](#) > [Infix Conversion \(STO2\)](#)

# Infix Conversion (STO2)

locked

Problem

Submissions

Leaderboard

Discussions

Editorial

Convert the given prefix expression to infix expression.

## Input Format

A string  $s$  representing the prefix expression.

## Constraints

 $1 \leq |s| \leq 1000$ 

## Output Format

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

## Sample Input 0

```
*+AB-CD
```

Submissions: 0

Max Score: 1

Difficulty: Medium

Rate This Challenge:

[More](#)

## Sample Output 0

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

Current Buffer (saved locally, editable) 🔗 ↺

Python 3



```
1
2
3 #!/usr/bin/env python
4 '''
5 Q: Infix, postfix, and prefix expression conversion and evaluation
6 For example:
7 Infix: 1 * (2 + 3) / 4
8 Postfix: 1 2 3 + * 4 /
9 Prefix: / * 1 + 2 3 4
10 Build up a binary tree with numbers as leaves and operators as internal nodes.
11 In-order traversal → infix
12 Post-order traversal → postfix
13 Pre-order traversal → prefix
14 '''
15 from __future__ import division
16 import random
17
18
19 OPERATORS = set(['+', '-', '*', '/', '(', ')'])
20 PRIORITY = {'+':1, '-':1, '*':2, '/':2}
21
22
23 ### INFIX ==> POSTFIX ###
24 '''
25 1)Fix a priority level for each operator. For example, from high to low:
26     3.     - (unary negation)
```

```

27     2.    * /
28     1.    + - (subtraction)
29 2) If the token is an operand, do not stack it. Pass it to the output.
30 3) If token is an operator or parenthesis:
31     3.1) if it is '(', push
32     3.2) if it is ')', pop until '('
33     3.3) push the incoming operator if its priority > top operator; otherwise pop.
34     *The popped stack elements will be written to output.
35 4) Pop the remainder of the stack and write to the output (except left parenthesis)
36 '''
37 # def infix_to_postfix(formula):
38 #     stack = [] # only pop when the coming op has priority
39 #     output = ''
40 #     for ch in formula:
41 #         if ch not in OPERATORS:
42 #             output += ch
43 #         elif ch == '(':
44 #             stack.append('(')
45 #         elif ch == ')':
46 #             while stack and stack[-1] != '(':
47 #                 output += stack.pop()
48 #             stack.pop() # pop '('
49 #         else:
50 #             while stack and stack[-1] != '(' and PRIORITY[ch] <= PRIORITY[stack[-1]]:
51 #                 output += stack.pop()
52 #             stack.append(ch)
53 #     # leftover
54 #     while stack: output += stack.pop()
55 #     print(output)
56 #     return output
57
58
59 ### POSTFIX ==> INFIX ###
60 '''
61 1) When see an operand, push

```

```
62 2) When see an operator, pop out two numbers, connect them into a substring and push back to the
    stack
63 3) the top of the stack is the final infix expression.
64 '''
65 # def postfix_to_infix(formula):
66 #     stack = []
67 #     prev_op = None
68 #     for ch in formula:
69 #         if not ch in OPERATORS:
70 #             stack.append(ch)
71 #         else:
72 #             b = stack.pop()
73 #             a = stack.pop()
74 #             if prev_op and len(a) > 1 and PRIORITY[ch] > PRIORITY[prev_op]:
75 #                 # if previous operator has lower priority
76 #                 # add '()' to the previous a
77 #                 expr = '('+a+')' + ch + b
78 #             else:
79 #                 expr = a + ch + b
80 #             stack.append(expr)
81 #             prev_op = ch
82 #     print(stack[-1])
83 #     return stack[-1]
84
85
86 ### INFIX ==> PREFIX ###
87 # def infix_to_prefix(formula):
88 #     op_stack = []
89 #     exp_stack = []
90 #     for ch in formula:
91 #         if not ch in OPERATORS:
92 #             exp_stack.append(ch)
93 #         elif ch == '(':
94 #             op_stack.append(ch)
95 #         elif ch == ')':
```

```

96 #         while op_stack[-1] != '(':
97 #             op = op_stack.pop()
98 #             a = exp_stack.pop()
99 #             b = exp_stack.pop()
100 #             exp_stack.append( op+b+a )
101 #             op_stack.pop() # pop '('
102 #         else:
103 #             while op_stack and op_stack[-1] != '(' and PRIORITY[ch] <= PRIORITY[op_stack[-1]]:
104 #                 op = op_stack.pop()
105 #                 a = exp_stack.pop()
106 #                 b = exp_stack.pop()
107 #                 exp_stack.append( op+b+a )
108 #                 op_stack.append(ch)
109
110 #     # leftover
111 #     while op_stack:
112 #         op = op_stack.pop()
113 #         a = exp_stack.pop()
114 #         b = exp_stack.pop()
115 #         exp_stack.append( op+b+a )
116 #     print(exp_stack[-1])
117 #     return exp_stack[-1]
118
119
120 ### PREFIX ==> INFIX ###
121 '''
122 Scan the formula reversely
123 1) When the token is an operand, push into stack
124 2) When the token is an operator, pop out 2 numbers from stack, merge them and push back to the
    stack
125 '''
126 def prefix_to_infix(formula):
127     stack = []
128     prev_op = None
129     for ch in reversed(formula):

```

```
130     if not ch in OPERATORS:
131         stack.append(ch)
132     else:
133         a = stack.pop()
134         b = stack.pop()
135         if prev_op and PRIORITY[prev_op] < PRIORITY[ch]:
136             exp = '('+a+ch+b+')'
137         else:
138             exp = '('+a+ch+b+')'
139         stack.append(exp)
140         prev_op = ch
141     # print(stack[-1])
142     return stack[-1]
143
144
145 '''
146 Scan the formula:
147 1) When the token is an operand, push into stack;
148 2) When an operator is encountered:
149     2.1) If the operator is binary, then pop the stack twice
150     2.2) If the operator is unary (e.g. unary minus), pop once
151 3) Perform the indicated operation on two popped numbers, and push the result back
152 4) The final result is the stack top.
153 '''
154 # def evaluate_postfix(formula):
155 #     stack = []
156 #     for ch in formula:
157 #         if ch not in OPERATORS:
158 #             stack.append(float(ch))
159 #         else:
160 #             b = stack.pop()
161 #             a = stack.pop()
162 #             c = {'+':a+b, '-':a-b, '*':a*b, '/':a/b}[ch]
163 #             stack.append(c)
164 #     print(stack[-1])
```

```
165 #     return stack[-1]
166
167
168 # def evaluate_infix(formula):
169 #     return evaluate_postfix(infix_to_postfix(formula))
170
171
172 ''' Whenever we see an operator following by two numbers,
173 we can compute the result. '''
174 # def evaluate_prefix(formula):
175 #     exps = list(formula)
176 #     while len(exps) > 1:
177 #         for i in range(len(exps)-2):
178 #             if exps[i] in OPERATORS:
179 #                 if not exps[i+1] in OPERATORS and not exps[i+2] in OPERATORS:
180 #                     op, a, b = exps[i:i+3]
181 #                     a,b = map(float, [a,b])
182 #                     c = {'+':a+b, '-':a-b, '*':a*b, '/':a/b}[op]
183 #                     exps = exps[:i] + [c] + exps[i+3:]
184 #                     break
185 #         prin(exps)
186 #     return exps[-1]
187
188
189
190 #infix_to_postfix('1+(3+4*6+6*1)*2/3')
191 #infix_to_prefix('1+(3+4*6+6*1)*2/3')
192 # print
193 #evaluate_infix('1+(3+4*6+6*1)*2/3')
194 #evaluate_postfix('1346**61**2*3/+')
195 #evaluate_prefix('+1/*++3*46*6123')
196 # print
197 #postfix_to_infix('1346**61**2*3/+')
198 s=input()
199 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** $((A+B) * (C-D))$