**MINI PROJECT REPORT - COMPILER DESIGN**

**INFIX TO POSTFIX CONVERSION**

**B1000109117**

**RA1911003010109**

**RA1911003010117**

**INDEX**

**S.NO TITLE PAGE NO.**

1. ABSTRACT 3
2. INTRODUCTION 4
3. OBJECTIVE 5
4. IMPLEMENTATION 7-9

1. CONCLUSION 9

1. REFERENCES 10

**ABSTRACT**

* Infix expressions are readable and solvable by humans.
* We can easily distinguish the order of operators, and also can use the parenthesis to solve that part first during solving mathematical expressions.
* The computer cannot differentiate the operators and parenthesis easily, that’s why postfix conversion is needed.
* Postfix notation is the useful form of intermediate code if the given language is expressions.
* Postfix notation is also called 'suffix notation' and 'reverse polish'.
* Postfix notation is a linear representation of a syntax tree.
* In the postfix notation, any expression can be written unambiguously without parentheses.
* The ordinary (infix) way of writing the sum of x and y is with operator in the middle: x \* y. But in the postfix notation, we place the operator at the right end as xy \*.

**INTRODUCTION**

* **Infix expression:** The expression of the form a op b. When an operator is in-between every pair of operands.
* **Postfix expression:** The expression of the form a b op. When an operator is followed for every pair of operands.

**Why postfix representation of the expression?**

* The compiler scans the expression either from left to right or from right to left.

Consider the below expression: a op1 b op2 c op3 d

If op1 = +, op2 = \*, op3 = +

* The compiler first scans the expression to evaluate the expression b \* c, then again scans the expression to add a to it. The result is then added to d after another scan.
* The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.
* The corresponding expression in postfix form is abc\*+d+. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.

**OBJECTIVE**

* As the operators are binary (we are not talking about unary operators here), so two operands are needed for each operator.
* The nature of these operators is not affected in the postfix form i.e. the plus operator (+) will apply on two operands.
* Each time we read an operand, we will push it on the stack.
* We are going to evaluate the postfix expression with the help of stack.
* After reaching an operator, we pop the two operands from the top of the stack, apply the operator and push the result back on the stack.

**IMPLEMENTATION**

**Algorithm**

1. Scan the infix expression from left to right.

2. If the scanned character is an operand, output it.

3. Else,

1 If the precedence and associativity of the scanned operator is greater than the precedence and associativity of the operator in the stack(or the stack is empty or the stack contains a ‘(‘ ), push it.

2 ‘^’ operator is right associative and other operators like ‘+’,’-‘,’\*’ and ‘/’ are left associative. Check especially for a condition when both top of the operator stack and scanned operator are ‘^’. In this condition the precedence of scanned operator is higher due to it’s right associativity. So it will be pushed in the operator stack. In all the other cases when the top of operator stack is same as scanned operator we will pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.

3 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

4. If the scanned character is an ‘(‘, push it to the stack.

5. If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.

6. Repeat steps 2-6 until infix expression is scanned.

7. Print the output

8. Pop and output from the stack until it is not empty.

**CODE**

Operators = set(['+', '-', '\*', '/', '(', ')', '^']) # collection of Operators

Priority = {'+':1, '-':1, '\*':2, '/':2, '^':3} # dictionary having priorities of Operators

def infixToPostfix(expression):

stack = [] # initialization of empty stack

output = ''

for character in expression:

if character not in Operators: # if an operand append in postfix expression

output+= character

elif character=='(': # else Operators push onto stack

stack.append('(')

elif character==')':

while stack and stack[-1]!= '(':

output+=stack.pop()

stack.pop()

else:

while stack and stack[-1]!='(' and Priority[character]<=Priority[stack[-1]]:

output+=stack.pop()

stack.append(character)

while stack:

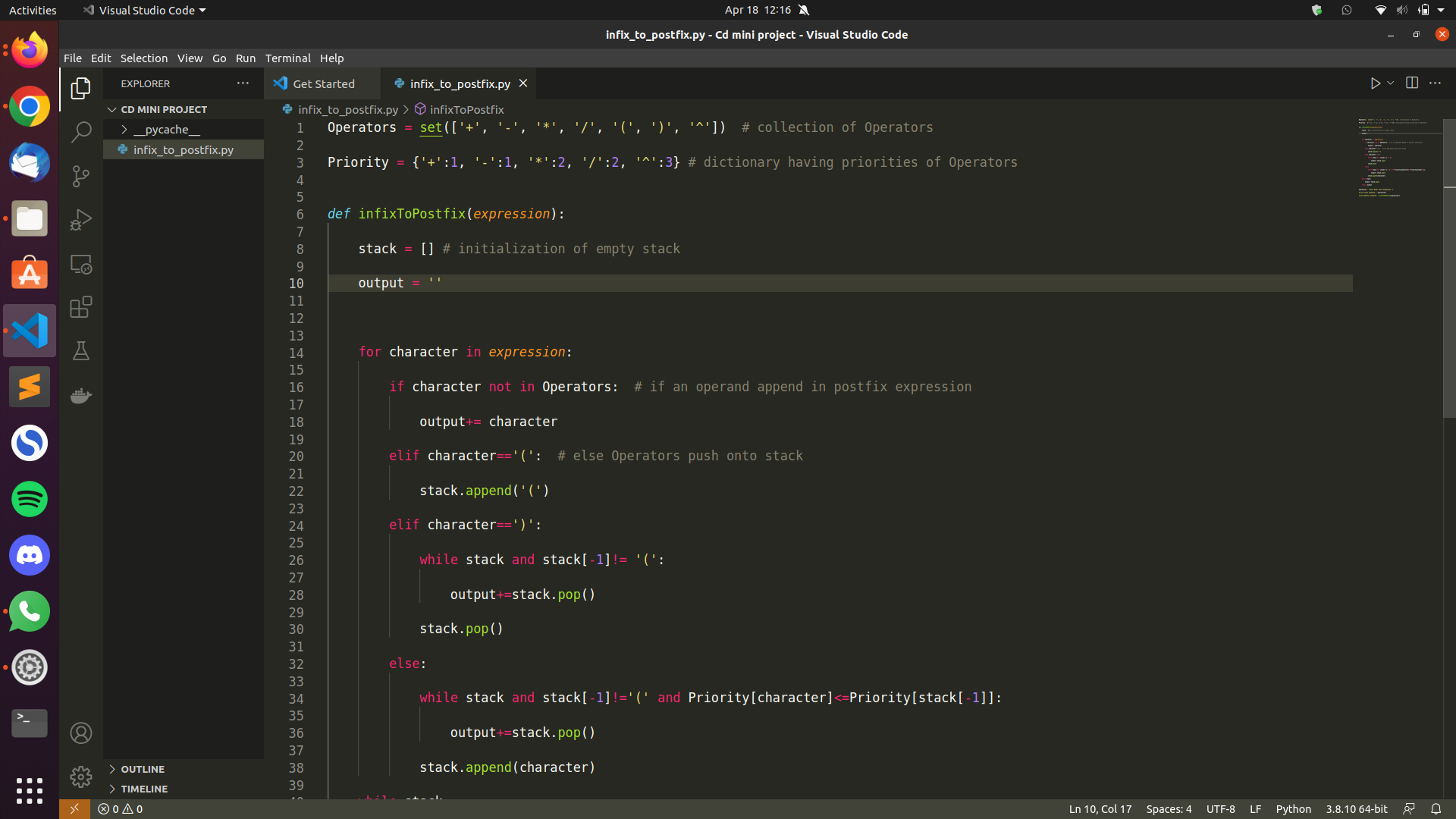
output+=stack.pop()

return output

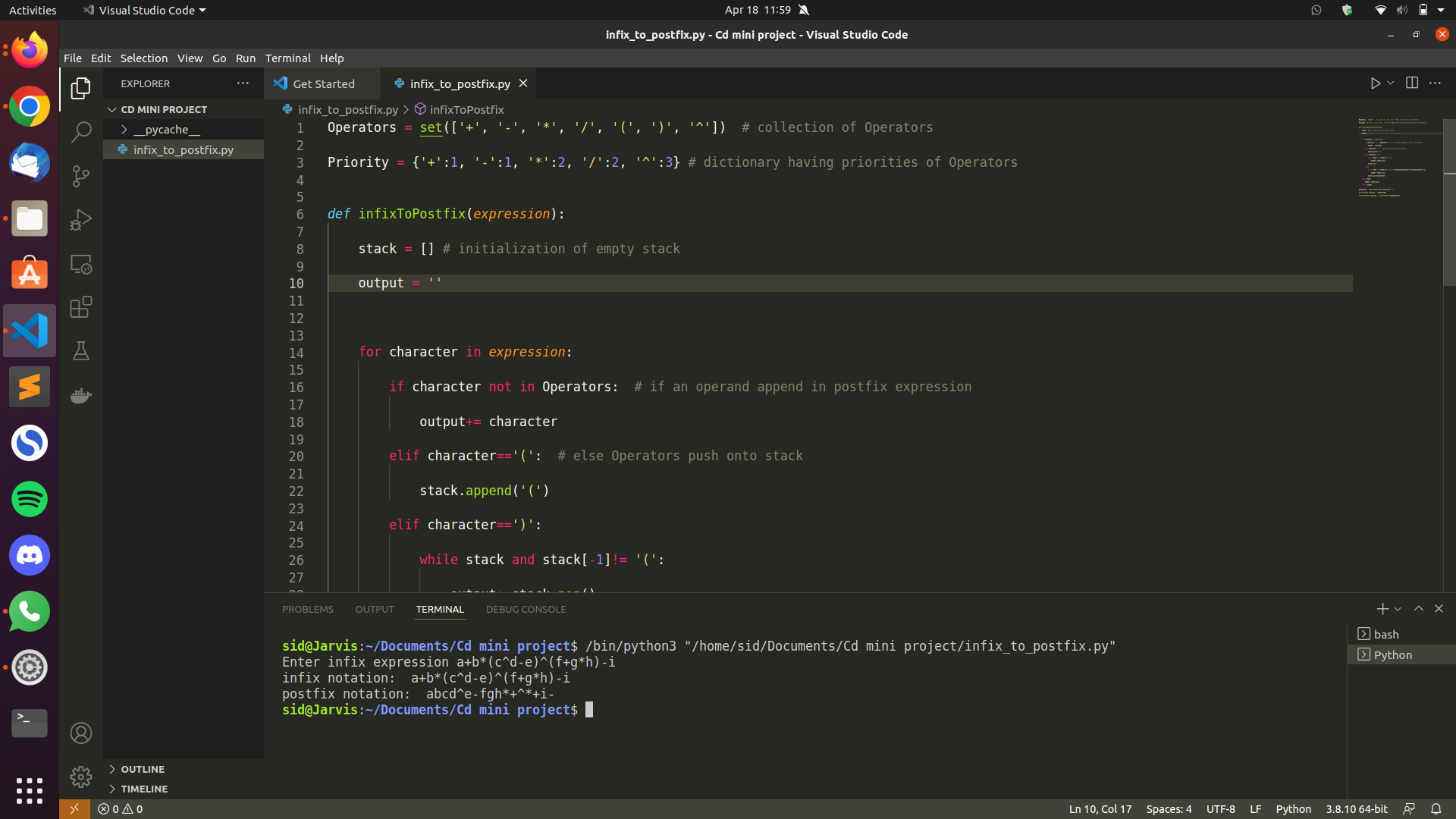
expression = input('Enter infix expression ')

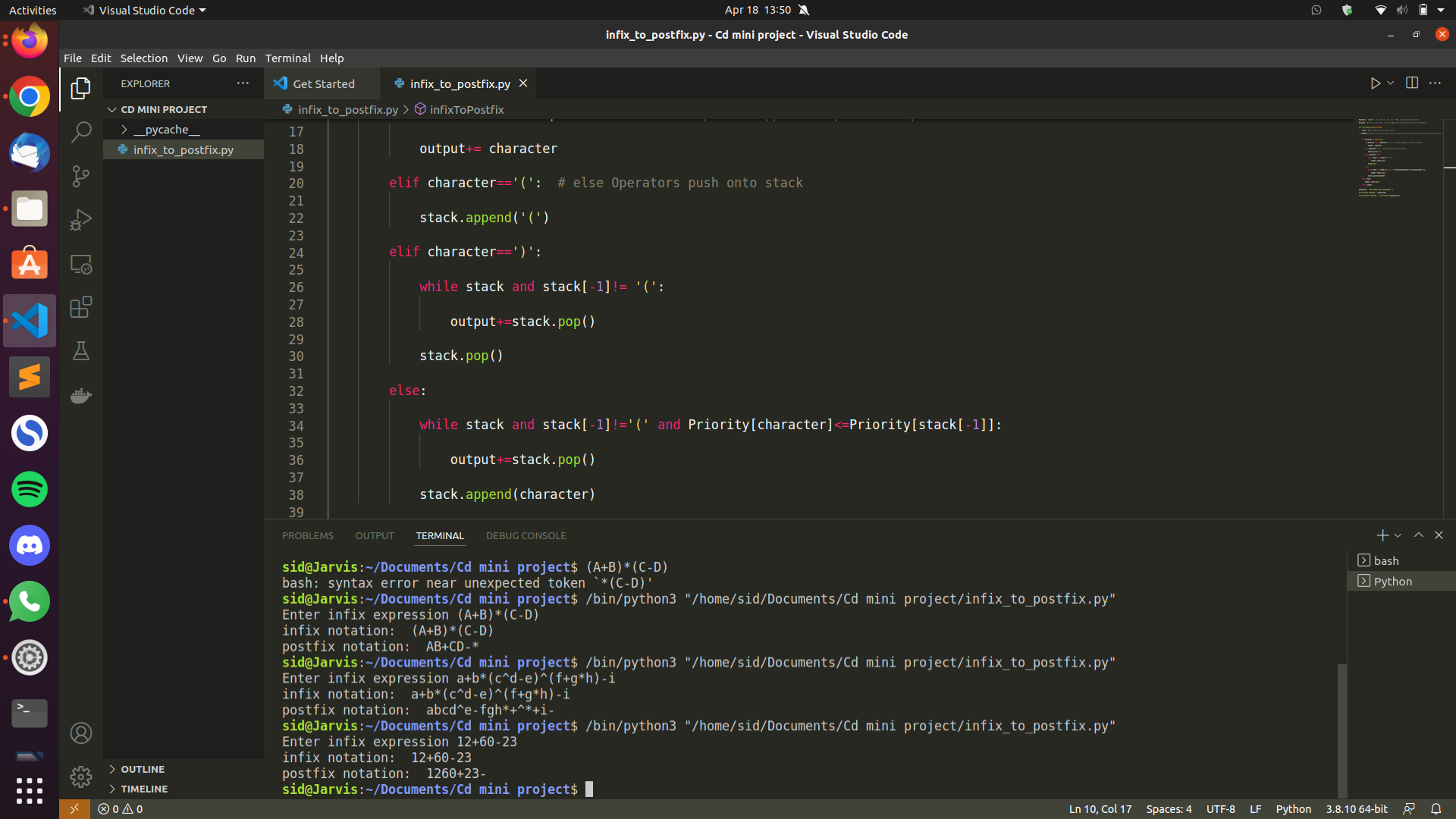
print('infix notation: ',expression)

print('postfix notation: ',infixToPostfix(expression))

****

**OUTPUT**

****

****

**CONCLUSION**

Infix expressions are what we humans use to solve problems normally. However, computers require a stack to solve expressions. Without taking care of the operator’s precedence, it is easy for the systems to solve the expressions using prefix and postfix notation. In this project, I studied a detailed view of infix and postfix notation along with the simplest technique to convert infix to postfix notation using the stack data structure.

**References**

1. <https://www.geeksforgeeks.org/stack-set-2-infix-to-postfix/>
2. <https://www.javatpoint.com/postfix-notation>
3. <http://web.cecs.pdx.edu/~harry/compilers/slides/IntroPart2.pdf>