Arithmetic Expression Evaluation - Copy

The stack organization is very effective in evaluating arithmetic expressions. Expressions are usually represented in what is known as **Infix notation**, in which each operator is written between two operands (i.e., A + B). With this notation, we must distinguish between (A + B)*C and A + (B * C) by using either parentheses or some operator-precedence convention. Thus, the order of operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

1. Polish notation (prefix notation) –
It refers to the notation in which the operator is placed before its two operands. Here no parentheses are required, i.e.,

+AB

2. Reverse Polish notation(postfix notation) –
It refers to the analogous notation in which the operator is placed after its two operands. Again, no parentheses is required in Reverse Polish notation, i.e.,

AB+

Stack-organized computers are better suited for post-fix notation than the traditional infix notation. Thus, the infix notation must be converted to the postfix notation. The conversion from infix notation to postfix notation must take into consideration the operational hierarchy.

There are 3 levels of precedence for 5 binary operators as given below:

Highest: Exponentiation (^)
Next highest: Multiplication (*) and division (/)
Lowest: Addition (+) and Subtraction (-)

The procedure for getting the result is:

1. Convert the expression in Reverse Polish notation(post-fix notation).
2. Push the operands into the stack in the order they appear.
3. When any operator encounters then pop two topmost operands for executing the operation.
4. After execution push the result obtained into the stack.
5. After the complete execution of expression, the final result remains on the top of the stack.

But My code directly converts the infix to its value

```
def infixEvaluation(string):
    nums = []
    signs = []
    for i in range(len(string)):
        ch = string[i]
        if ch=="(":
            signs.append(ch)
        elif ch in {'1','2','3','4','5','6','7','8','9'}:
            nums.append(int(ch))
        elif ch ==')':
            while signs[-1]!='(':
                v2 = nums.pop()
                v1 = nums.pop()
                op = signs.pop()
                nums.append(eval(v1,v2,op))
            signs.pop()
        elif ch in {'+','-','*','/'}:

        #keep popping unless:
        #1. the stack is not empty
        #2. the peek of stack is not equal to '('
        #3. the precedence of top of stack is greater than or equal to the
current operator because higher precedence is solved first. For equal
precedence the one
        #which comes first is solved first.
            while len(signs)>0 and signs[-1]!='(' and
precedence(signs[-1])>=precedence(ch):
                v2 = nums.pop()
                v1 = nums.pop()
                op = signs.pop()
                nums.append(eval(v1, v2, op))
            signs.append(ch)

    while len(signs)>0:
        v2 = nums.pop()
        v1 = nums.pop()
        op = signs.pop()
        nums.append(eval(v1, v2, op))
    return nums[-1]


#Checks Precedence of operator.     "/"="*" > "+" = "-"
def precedence(operator):
```

```
    if operator == "+" or operator == "-":
        return 1
    else:
        return 2



#For evaluating the value of 2 operand and 1 operator
def eval(v1,v2,operator):
    if operator=='+':
        return v1+v2
    elif operator=="-":
        return v1-v2
    elif operator == "*":
        return v1*v2
    else:
        return v1//v2


string = "5-4*2+(4)"
print(infixEvaluation(string))
```