链滴

栈

作者：xuqil
原文链接：https://ld246.com/article/1563891860060
来源网站：链滴

栈

- 栈遵循LIFO(后进先出)
- 栈是结构化的，是一个有序的项的集
- 添加和删除的一端称为“顶”

栈的实现

```
class Stack:
    """
    栈的实现
    """
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)


if __name__ == '__main__':
    s = Stack()
    print(s.is_empty())
    s.push(4)
    s.push('dog')
    print(s.peek())
    s.push(True)
    print(s.size())
    print(s.is_empty())
    s.push(8.4)
    print(s.pop())
    print(s.pop())
    print(s.size())
```

```
True
dog
3
False
8.4
True
2
```

j简单括号匹配

```
from ProblemSolving_python.three.Stack.stack import Stack


def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol == '(':
            s.push(symbol)
        else:
            if s.is_empty():
                balanced = False
            else:
                s.pop()
        index += 1

    if balanced and s.is_empty():
        return True
    else:
        return False


print(parChecker('((()))'))
print(parChecker('(((()mm'))
print(parChecker('mm'))
```

存在巨大漏洞

符号匹配

```
from ProblemSolving_python.three.Stack.stack import Stack


def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol in '([{':
            s.push(symbol)
        else:
            if s.is_empty():
                balanced = False
            else:
                top = s.pop()
                if not matches(top, symbol):
```

```
                balanced = False
        index += 1

    if balanced and s.is_empty():
        return True
    else:
        return False


def matches(open, close):
    opens = '([{'
    closers = ')]}'
    return opens.index(open) == closers.index(close)


print(parChecker('((()))'))
print(parChecker('{{([][])}()}'))
print(parChecker('(()'))
```

同样存在巨大漏洞

正确的符号匹配

```
from ProblemSolving_python.three.Stack.stack import Stack


def check_parens(text):
    """
    :param text: 包含括号的字符串，被检测字符串
    :return: 检查成功与否
    """
    parens = "()[]{}"
    open_parens = "([{"
    opposite = {")": "(", "]": "[", "}": "{"}  # 表示配对关系的字典

    def parentheses(text):
        """
        括号生成器，每次调用返回text里的下一括号及其位置
        每次调用生成括号的及其位置如test[1]=(
        :param text: 包含括号的字符串，被检测字符串
        :return:
        """
        i, text_len = 0, len(text)
        while True:
            while i < text_len and text[i] not in parens:
                i += 1
            if i >= text_len:
                return
            yield text[i], i
            i += 1

    st = Stack()
    for pr, i in parentheses(text):
        if pr in open_parens:
```

```
            # 将所有的开括号入栈
            st.push(pr)
        elif st.is_empty() and pr in parens:
            # 闭括号多于开括号的情况
            print("Unmatching is found at", i, "for", pr)
        elif st.pop() != opposite[pr]:
            # 将开括号出栈，与闭口号对于字典的值（闭括号对应的开括号）相比较
            print("Unmatching is found at", i, "for", pr)
    print("all parentheses are correctly matched")
    return True


if __name__ == '__main__':
    print(check_parens("(ddddd{ddds}))"))
    print(check_parens("(((ddddd{ddds}))"))
```

十进制转换成二进制

```
from ProblemSolving_python.three.Stack.stack import Stack


def divideBy2(decNumber):
    """
    十进制转换成二进制
    :param decNumber:
    :return:
    """
    remstack = Stack()
    while decNumber > 0:
        rem = decNumber % 2
        remstack.push(rem)
        decNumber = decNumber // 2

    binString = ''
    while not remstack.is_empty():
        binString = binString + str(remstack.pop())

    return binString


print(divideBy2(10))
```

进制转换

```
from ProblemSolving_python.three.Stack.stack import Stack


def baseConverter(decNumber, base):
    """
    十进制转换成二进制
    :param decNumber:
    :param base:
    :return:
```

```
    """
    digits = "0123456789ABCDEF"
    remstack = Stack()
    while decNumber > 0:
        rem = decNumber % base
        remstack.push(rem)
        decNumber = decNumber // base

    binString = ''
    while not remstack.is_empty():
        binString = binString + digits[remstack.pop()]

    return binString


print(baseConverter(10, 2))
print(baseConverter(10, 16))
print(baseConverter(10, 8))
```

输出

```
1010
A
12
```

中缀转换为后缀

```
from ProblemSolving_python.three.Stack.stack import Stack


def infixToPostfix(infixexpr):
    prec = dict()
    prec['*'] = 3
    prec['/'] = 3
    prec['+'] = 2
    prec['-'] = 2
    prec['('] = 1
    opStack = Stack()
    postfixList = []
    tokenList = infixexpr.split()

    for token in tokenList:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789":
            postfixList.append(token)
        elif token == '(':
            opStack.push(token)
        elif token == ')':
            topToken = opStack.pop()
            while topToken != '(':
                postfixList.append(topToken)
                topToken = opStack.pop()
        else:
            while (not opStack.is_empty()) and (prec[opStack.peek()] >= prec[token]):
```

```
            postfixList.append(opStack.pop())
        opStack.push(token)
    while not opStack.is_empty():
        postfixList.append(opStack.pop())
    return " ".join(postfixList)


print(infixToPostfix("A * B + C * D"))
print(infixToPostfix("( A + B ) * C - ( D - E ) * ( F + G )"))
```

输出结果：

```
A B * C D * +
A B + C * D E - F G + * -
```

后缀表达式求值

```
from ProblemSolving_python.three.Stack.stack import Stack


def postfixEval(postfixExpr):
    operandStack = Stack()
    tokenList = postfixExpr.split()
    for token in tokenList:
        if token in "0123456789":
            operandStack.push(int(token))
        else:
            operand2 = operandStack.pop()
            operand1 = operandStack.pop()
            result = doMath(token, operand1, operand2)
            operandStack.push(result)
    return operandStack.pop()


def doMath(op, op1, op2):
    if op == "*":
        return op1 * op2

    elif op == "/":
        return op1 / op2

    elif op == "+":
        return op1 + op2

    elif op == "-":
        return op1 - op2


print(postfixEval('1 2 * 3 2 * +'))
```

存在漏洞

```
from ProblemSolving_python.three.Stack.stack import Stack
```

```
def postfixEval(postfixExpr):
    operandStack = Stack()
    tokenList = postfixExpr.split()
    for token in tokenList:
        if token.isdigit():
            operandStack.push(int(token))
        else:
            operand2 = operandStack.pop()
            operand1 = operandStack.pop()
            result = doMath(token, operand1, operand2)
            operandStack.push(result)
    return operandStack.pop()


def doMath(op, op1, op2):
    if op == "*":
        return op1 * op2

    elif op == "/":
        return op1 / op2

    elif op == "+":
        return op1 + op2

    elif op == "-":
        return op1 - op2


print(postfixEval('1 2 * 32 2 * +'))
```

修复部分漏洞