stack data structure in python
---

introduction:
~~~~~~~~~~~~
=> a stack is linear data structure.
=> memory allocation for objects are continueos memory locations.
=> last - in - first - out LIFO

Eg:
---
    stack of plates
    disks in the rack
    function calls are stored inside stack
    web page navigations
    parenthesis balancing
    reversing item
    infix to prefix/postfix
    evaluation of prefix/postfix
    undo/redo or forward/backward
    etc

operations that can be performed on stack

The following are some of operations that can be performed on stack

1) push ---> inserting an object into stack
2) pop ----> deleteing an object from stack
3) peek ---> returning top most object in the stack
4) size ---> returns num of objects in the stack
5) isempty-> returns True if the stack is empty else False
6) display-> display/traverse elements in the stack

implementation of stack
---
container and top

when an obj is inserted top will be incremented by one unit
when an obj is deleted top will be decremented by one unit

1) using list
2) collections. deque
3) using our own implementation

stack implementation by using list

```
L = []
print(L) #[]
L.append(111)
L.append(222)
L.append(333)
L.append(444)
L.append(555)
print(L) #[111, 222, 333, 444, 555]
print(L[-1]) #555
print(L) #[111, 222, 333, 444, 555]
L.pop()
print(L) #[111, 222, 333, 444]
print(len(L)==0)
print(len(L))
```

```
C:\test>py test.py
[]
[111, 222, 333, 444, 555]
```
~~~~~~~~~~~~

```
555
[111, 222, 333, 444, 555]
[111, 222, 333, 444]
False
4


stack implementation by using collections.deque
-----------------------------------------------
from collections import deque

stack = deque()

stack.append(10)
stack.append(20)
stack.append(30)
stack.append(40)

print(stack) #[10,20,30,40]
print(stack[-1]) #40
stack.pop()
print(stack)

C:\test>py test.py
deque([10, 20, 30, 40])
40
deque([10, 20, 30])

stack implementation by our own list
------------------------------------
class stack:

      #constructor for initializations
      def __init__(self):
            self.stk = []

      #isempty operation
      def isempty(self):
            return len(self.stk)==0

      #size of stack
      def size(self):
            return len(self.stk)

      #print or display or traversing
      def display(self):
            print(self.stk)

      #push operation
      def push(self,data):
            self.stk.append(data)

      #pop operation
      def pop(self):
            if self.isempty():
                  print("stack is underflow")
                  return
            return self.stk.pop()


s = stack()
s.push(111)
s.push(222)
s.display()
```

```
print(s.pop())
s.display()

C:\test>py test.py
[111, 222]
222
[111]

stack implementation by using linked list
-----------------------------------------
class stack:

      class node:

            def __init__(self,data,next=None):
                  self.data = data
                  self.next = next

      #constructor for initializations
      def __init__(self):
            self.head = None
            self.count = 0

      #isempty operation
      def isempty(self):
            return self.count==0

      #peek operation
      def peek(self):
            if self.isempty():
                  print("stack is empty")
                  return
            return self.head.data

      #size of stack
      def size(self):
            return self.count

      #print or display or traversing: O(n)
      def display(self):
            if self.isempty():
                  print("list is empty")
                  return
            temp = self.head
            while temp!=None:
                  print(temp.data,end=" ")
                  temp = temp.next
            print()

      #push operation
      def push(self,data):
            self.head = self.node(data,self.head)
            self.count = self.count + 1

      #pop operation
      def pop(self):
            if self.isempty():
                  print("stack is underflow")
                  return
            val = self.head.data
            self.count = self.count -1
            self.head = self.head.next
            return val
```

```
s = stack()
s.push(100)
s.push(200)
s.push(300)
s.display()
print(s.pop())
s.display()
print(s.peek())
print(s.size())

C:\test>py test.py
300 200 100
300
200 100
200
2

system stack and function calls
------------------------------
def fun2():
      print("function2 line num 1")

def fun1():
      print("function1 line num 1")
      fun2()
      print("function1 line num 2")

def main():
      print("main line num 1")
      fun1()
      print("main line num 2")

main()
'''
main line num1
function1 line num 1
function2 line num 1
function1 line num 2
main line num 2
'''

stack.py
--------
class stack:

      class node:

            def __init__(self,data,next=None):
                  self.data = data
                  self.next = next

      #constructor for initializations
      def __init__(self):
            self.head = None
            self.count = 0

      #isempty operation
      def isempty(self):
            return self.count==0

      #peek operation
      def peek(self):
            if self.isempty():
```

```
                print("stack is empty")
                return
          return self.head.data

    #size of stack
    def size(self):
          return self.count

    #print or display or traversing: O(n)
    def display(self):
          if self.isempty():
                print("list is empty")
                return
          temp = self.head
          while temp!=None:
                print(temp.data,end=" ")
                temp = temp.next
          print()

    #push operation
    def push(self,data):
          self.head = self.node(data,self.head)
          self.count = self.count + 1

    #pop operation
    def pop(self):
          if self.isempty():
                print("stack is underflow")
                return
          val = self.head.data
          self.count = self.count -1
          self.head = self.head.next
          return val


stack to maintain student objects
---------------------------------
from stack import *

class student:
    def __init__(self,sid,sname):
          self.sid = sid
          self.sname = sname
    def __str__(self):
          return f"({self.sid}, {self.sname})"

s1 = student(111,"AAA")
s2 = student(222,"BBB")
s3 = student(333,"CCC")
s4 = student(444,"DDD")
s = stack()
s.push(s1)
s.push(s2)
s.push(s3)
s.push(s4)
s.display()
print(s.pop())
s.display()

C:\test>py test.py
(444, DDD) (333, CCC) (222, BBB) (111, AAA)
(444, DDD)
(333, CCC) (222, BBB) (111, AAA)
```

sorted insertion into stack

```
--------------------------
from stack import *

def sortedinsert(s,data):
      if s.isempty() or data>s.peek():
            s.push(data)
      else:
            temp = s.pop()
            sortedinsert(s,data)
            s.push(temp)

s = stack()
sortedinsert(s,4)
sortedinsert(s,5)
sortedinsert(s,1)
sortedinsert(s,2)
sortedinsert(s,3)
sortedinsert(s,6)
s.display()
```

sorting stack elements

```
----------------------
from stack import *

def sortedinsert(s,data):
      if s.isempty() or data>s.peek():
            s.push(data)
      else:
            temp = s.pop()
            sortedinsert(s,data)
            s.push(temp)

def sortstack(s):
      if not s.isempty():
            temp = s.pop()
            sortstack(s)
            sortedinsert(s,temp)


s = stack()
s.push(444)
s.push(111)
s.push(222)
s.push(333)
s.push(666)
s.push(555)
s.display()
sortstack(s)
s.display()
```

bottom insert in the stack

```
--------------------------
from stack import *

def bottominsert(s,data):
      if s.isempty():
            s.push(data)
      else:
            temp = s.pop()
            bottominsert(s,data)
            s.push(temp)

s = stack()
```

```
s.push(444)
s.push(111)
s.push(222)
s.push(333)
s.display()
bottominsert(s,999)
s.display()


C:\test>py test.py
333
222
111
444

333
222
111
444
999
```

reverse stack
~~~~~~~~~~~~~

```
from stack import *

def reversestack(s):
      if not s.isempty():
            temp = s.pop()
            reversestack(s)
            bottominsert(s,temp)

def bottominsert(s,data):
      if s.isempty():
            s.push(data)
      else:
            temp = s.pop()
            bottominsert(s,data)
            s.push(temp)

s = stack()
s.push(444)
s.push(111)
s.push(222)
s.push(333)
s.display()
reversestack(s)
s.display()

C:\test>py test.py
333 222 111 444
444 111 222 333
```

reverse a stack by using queue

```
stack ----> LIFO
queue ----> FIFO

from queue import *
from stack import *

def reversestack(s):
      q = Queue(maxsize=5)
      while not s.isempty():
            q.put(s.pop())
```
~~~~~~~~~~~~~

```
      while not q.empty():
            s.push(q.get())

s = stack()
s.push(111)
s.push(222)
s.push(333)
s.push(444)
s.push(555)
s.display()
reversestack(s)
s.display()


C:\test>py test.py
555 444 333 222 111
111 222 333 444 555
```

reverse k elements in stack

```
---------------------------
from queue import *
from stack import *

def reversestack(s,k):
      q = Queue(maxsize=5)
      i=1
      while not s.isempty() and i<=k:
            q.put(s.pop())
            i=i+1
      while not q.empty():
            s.push(q.get())

s = stack()
s.push(111)
s.push(222)
s.push(333)
s.push(444)
s.push(555)
s.push(666)
s.push(777)
s.display()
reversestack(s,3)
s.display()

C:\test>py test.py
777 666 555 444 333 222 111
555 666 777 444 333 222 111
```

balanced parenthesis

```
~~~~~~~~~~~~~~~~~~~~
()    True
{}    True
[]    True
([])  True
()[]  True
([{]})      False

def balanced_parenthesis(expr):
      s = []
      for ch in expr:
            if ch=='(' or ch=='[' or ch=='{':
                  s.append(ch)
            elif ch==')':
                  if s.pop() != '(':
```

```
                        return False
            elif ch==']':
                  if s.pop()!='[':
                        return False
            elif ch=='}':
                  if s.pop()!='{':
                        return False
      return len(s)==0

print(balanced_parenthesis("()")) #True
print(balanced_parenthesis("([])")) #True
print(balanced_parenthesis("([{}])")) #True
print(balanced_parenthesis("([{]})")) #False

C:\test>py test.py
True
True
True
False

Representation of expressions in programming
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
infix ----> operand operator operand
prefix ---> operator operand operand
postfix --> operand operand operator

infix       prefix            postfix
---------------------------------------
A+B         +AB         AB+
A+(B*C)           +A*BC       ABC*+
(A+B)*C           *+ABC       AB+C*
A*B+C*D           +*AB*CD           AB*CD*+

steps to convert infix expr into postfix form
---------------------------------------------
1) print operands in the same order they arrive.
2) if stack is empty or contains ( on top, then push incoming operator
3) if incoming symbol is ( then push into stack.
4) if incoming symbol is ) pop all items into output until ( came.
5) if the precedence of incoming symbol is >= precedence of symbol existed in
the top of stack, push that symbol into stack.
6) if the precedence of incoming symbol is < precedence of symbol existed in the
top of stack,pop the symbol from stack put it into output, then compare next
symbol else push that symbol into stack.
7) pop all symbols from stack put into output.

infix to postfix conversion implementation
------------------------------------------
def precedence(x):
      if x=='(':
            return 0
      if x=='+' or x=='-':
            return 1
      if x=='*' or x=='/':
            return 2
      return 4

def infix_to_postfix_conversion(expn):
      s = []
      tokens=list(expn)
      result = ""
      for item in tokens:
            if item in "+-*/":
                  while len(s)!=0 and precedence(item) <= precedence(s[-1]):
```

```
                        result=result+s.pop()
                  s.append(item)
            elif item=='(':
                  s.append(item)
            elif item==')':
                  temp=None
                  while len(s)!=0 and temp!='(':
                        temp = s.pop()
                        if temp!='(':
                              result = result + temp
            else:
                  result=result+item
      while len(s)!=0:
            result = result + s.pop()
      return result

print(infix_to_postfix_conversion("A+B")) #A
print(infix_to_postfix_conversion("A+(B*C)"))
print(infix_to_postfix_conversion("(A+B)*C)"))
print(infix_to_postfix_conversion("A*B+C*D"))

C:\test>py test.py
AB+
ABC*+
AB+C*
AB*CD*+

steps to convert infix expr into prefix form
--------------------------------------------
1) reverse the given expression
2) replace '(' with ')' and '(' with ')'
3) apply infix to postfix conversion
4) reverse generated output

infix to prefix conversion implementation
-----------------------------------------
def precedence(x):
      if x=='(' or x==')':
            return 0
      if x=='+' or x=='-':
            return 1
      if x=='*' or x=='/':
            return 2
      return 4

def infix_to_postfix_conversion(expn):
      s = []
      tokens=list(expn)
      result = ""
      for item in tokens:
            if item in "+-*/":
                  while len(s)!=0 and precedence(item) <= precedence(s[-1]):
                        result=result+s.pop()
                  s.append(item)
            elif item=='(':
                  s.append(item)
            elif item==')':
                  temp=None
                  while len(s)!=0 and temp!='(':
                        temp = s.pop()
                        if temp!='(':
                              result = result + temp
            else:
                  result=result+item
```

```
      while len(s)!=0:
            result = result + s.pop()
      return result

def replace(expn):
      s = ""
      for i in expn:
            if i=='(':
                  s=s+')'
            elif i==')':
                  s=s+'('
            else:
                  s=s+i
      #print(s)
      return s

def infix_to_prefix_conversion(expn):
      expn = expn[::-1]
      expn = replace(expn)
      expn = infix_to_postfix_conversion(expn)
      expn = expn[::-1]
      return expn

print(infix_to_postfix_conversion("A+(B*C)")) #ABC*+
print(infix_to_prefix_conversion("A+(B*C)")) #+A*BC
print(infix_to_postfix_conversion("A+B")) #AB+
print(infix_to_prefix_conversion("A+B")) #+AB
print(infix_to_postfix_conversion("(A+B)*C)")) #AB+C*
print(infix_to_prefix_conversion("(A+B)*C"))#*+ABC
print(infix_to_postfix_conversion("A*B+C*D"))
print(infix_to_prefix_conversion("A*B+C*D"))

C:\test>py test.py
ABC*+
+A*BC
AB+
+AB
AB+C*
*+ABC
AB*CD*+
+*AB*CD

evalutation of postfix
----------------------
def precedence(x):
      if x=='(' or x==')':
            return 0
      if x=='+' or x=='-':
            return 1
      if x=='*' or x=='/':
            return 2
      return 4

def infix_to_postfix_conversion(expn):
      s = []
      tokens=list(expn)
      result = ""
      for item in tokens:
            if item in "+-*/":
                  while len(s)!=0 and precedence(item) <= precedence(s[-1]):
                        result=result+s.pop()
                  s.append(item)
            elif item=='(':
                  s.append(item)
```

```
            elif item==')':
                  temp=None
                  while len(s)!=0 and temp!='(':
                        temp = s.pop()
                        if temp!='(':
                              result = result + temp
            else:
                  result=result+item
      while len(s)!=0:
            result = result + s.pop()
      return result

def replace(expn):
      s = ""
      for i in expn:
            if i=='(':
                  s=s+')'
            elif i==')':
                  s=s+'('
            else:
                  s=s+i
      #print(s)
      return s

def infix_to_prefix_conversion(expn):
      expn = expn[::-1]
      expn = replace(expn)
      expn = infix_to_postfix_conversion(expn)
      expn = expn[::-1]
      return expn

def postfix_eval(expn):
      s = []
      tokens = list(expn)
      for i in tokens:
            if i=='+':
                  n1 = s.pop()
                  n2 = s.pop()
                  s.append(n1+n2)
            elif i=='-':
                  n1 = s.pop()
                  n2 = s.pop()
                  s.append(n1-n2)
            elif i=='*':
                  n1 = s.pop()
                  n2 = s.pop()
                  s.append(n1*n2)
            elif i=='/':
                  n1 = s.pop()
                  n2 = s.pop()
                  s.append(n1/n2)
            else:
                  s.append(int(i))
      return s.pop()

print(postfix_eval(infix_to_postfix_conversion("1+2"))) #3
print(postfix_eval(infix_to_postfix_conversion("1+2*3"))) #7

C:\test>py test.py
3
7
```