**Homework : 10733101**

**10733169**

**Stack in Python**

A **stack** is a linear data structure that stores items in a  or First-In/Last-Out (FILO) manner. In stack, a new element is added at one end and an element is removed from that end only. The insert and delete operations are often called push and pop.

**The functions associated with stack are:**

* **empty()** – Returns whether the stack is empty – Time Complexity: O(1)
* **size()** – Returns the size of the stack – Time Complexity: O(1)
* **top() / peek()**– Returns a reference to the topmost element of the stack – Time Complexity: O(1)
* **push(a)** – Inserts the element ‘a’ at the top of the stack – Time Complexity: O(1)
* **pop()** – Deletes the topmost element of the stack – Time Complexity: O(1)

### Implementation:

There are various ways from which a stack can be implemented in Python. This article covers the implementation of a stack using data structures and modules from the Python library.   
Stack in Python can be implemented using the following ways:

* list
* Collections.deque
* queue.LifoQueue

### Implementation using list:

Python’s built-in data structure list can be used as a stack. Instead of push(), append() is used to add elements to the top of the stack while pop() removes the element in LIFO order.   
Unfortunately, the list has a few shortcomings. The biggest issue is that it can run into speed issues as it grows. The items in the list are stored next to each other in memory, if the stack grows bigger than the block of memory that currently holds it, then Python needs to do some memory allocations. This can lead to some append() calls taking much longer than other ones.

# Python program to

# demonstrate stack implementation

# using list

stack = []

# append() function to push

# element in the stack

stack.append('a')

stack.append('b')

stack.append('c')

print('Initial stack')

print(stack)

# pop() function to pop

# element from stack in

# LIFO order

print('\nElements popped from stack:')

print(stack.pop())

print(stack.pop())

print(stack.pop())

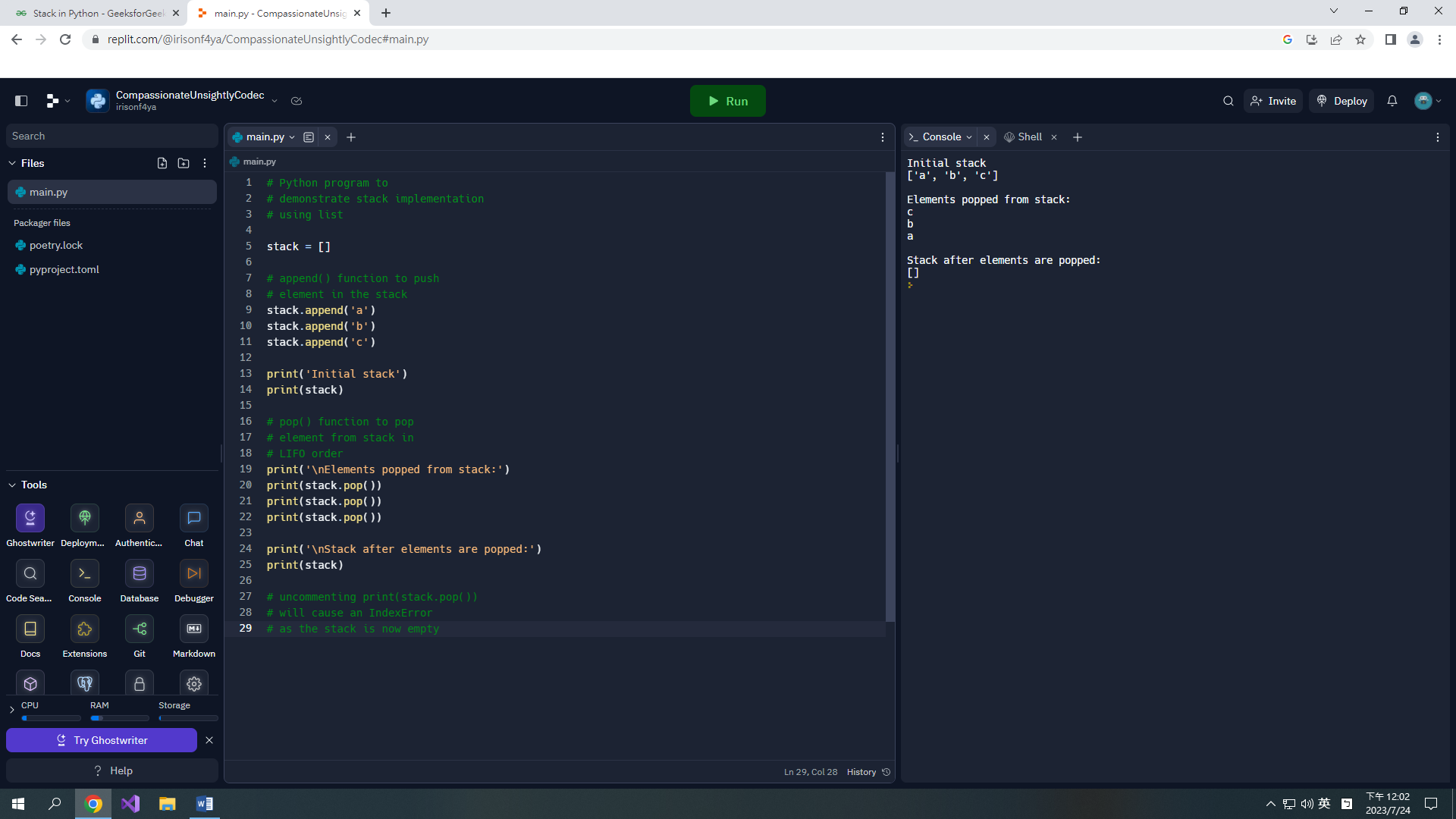
print('\nStack after elements are popped:')

print(stack)

# uncommenting print(stack.pop())

# will cause an IndexError

# as the stack is now empty



**Output**

Initial stack

['a', 'b', 'c']

Elements popped from stack:

c

b

a

Stack after elements are popped:

[]

### Implementation using a singly linked list:

The linked list has two methods addHead(item) and removeHead() that run in constant time. These two methods are suitable to implement a stack.

* **getSize()**– Get the number of items in the stack.
* **isEmpty()** – Return True if the stack is empty, False otherwise.
* **peek()** – Return the top item in the stack. If the stack is empty, raise an exception.
* **push(value)** – Push a value into the head of the stack.
* **pop()** – Remove and return a value in the head of the stack. If the stack is empty, raise an exception.

**Below is the implementation of the above approach:**

# Python program to demonstrate

# stack implementation using a linked list.

# node class

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.next = None

class Stack:

# Initializing a stack.

# Use a dummy node, which is

# easier for handling edge cases.

def \_\_init\_\_(self):

self.head = Node("head")

self.size = 0

# String representation of the stack

def \_\_str\_\_(self):

cur = self.head.next

out = ""

while cur:

out += str(cur.value) + "->"

cur = cur.next

return out[:-2]

# Get the current size of the stack

def getSize(self):

return self.size

# Check if the stack is empty

def isEmpty(self):

return self.size == 0

# Get the top item of the stack

def peek(self):

# Sanitary check to see if we

# are peeking an empty stack.

if self.isEmpty():

raise Exception("Peeking from an empty stack")

return self.head.next.value

# Push a value into the stack.

def push(self, value):

node = Node(value)

node.next = self.head.next

self.head.next = node

self.size += 1

# Remove a value from the stack and return.

def pop(self):

if self.isEmpty():

raise Exception("Popping from an empty stack")

remove = self.head.next

self.head.next = self.head.next.next

self.size -= 1

return remove.value

# Driver Code

if \_\_name\_\_ == "\_\_main\_\_":

stack = Stack()

for i in range(1, 11):

stack.push(i)

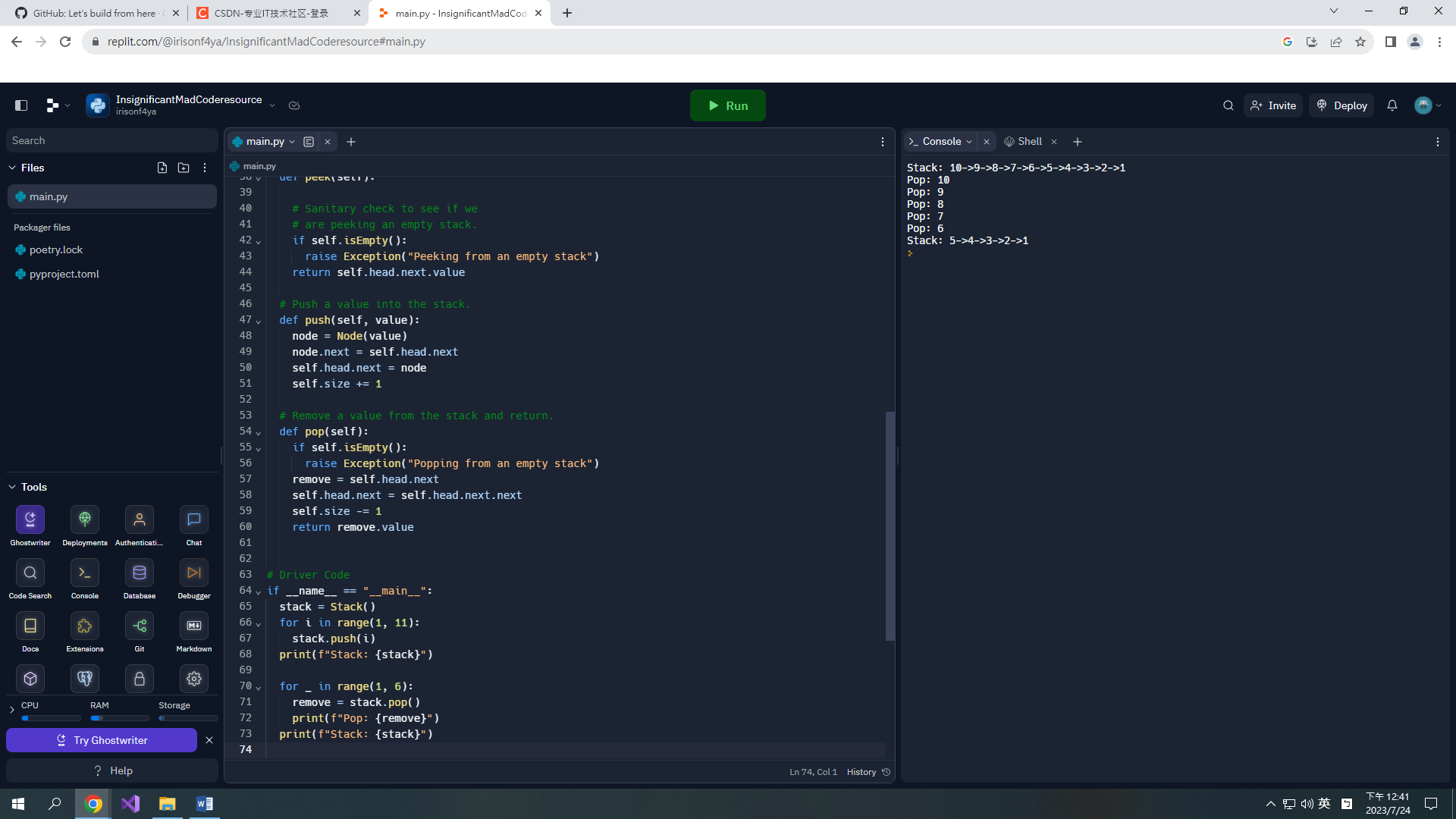
print(f"Stack: {stack}")

for \_ in range(1, 6):

remove = stack.pop()

print(f"Pop: {remove}")

print(f"Stack: {stack}")



**Output**

Stack: 10->9->8->7->6->5->4->3->2->1

Pop: 10

Pop: 9

Pop: 8

Pop: 7

Pop: 6

Stack: 5->4->3->2->1

### **Advantages of Stack:**

* Stacks are simple data structures with a well-defined set of operations, which makes them easy to understand and use.
* Stacks are efficient for adding and removing elements, as these operations have a time complexity of O(1).
* In order to reverse the order of elements we use the stack data structure.
* Stacks can be used to implement undo/redo functions in applications.

### **Drawbacks of Stack:**

* Restriction of size in Stack is a drawback and if they are full, you cannot add any more elements to the stack.
* Stacks do not provide fast access to elements other than the top element.
* Stacks do not support efficient searching, as you have to pop elements one by one until you find the element you are looking for.