Stack Using Array

**Implement a stack using an array**

In this notebook, we'll look at one way to implement a stack. First, check out the walkthrough for an overview, and then you'll get some practice implementing it for yourself.

Walkthrough

Below we'll go through the implementation step by step. Each step has a walkthrough and also a solution. We recommend that you first watch the walkthrough, and then try to write the code on your own.

When you first try to remember and write out the code for yourself, this effort helps you understand and remember the ideas better. At the same time, it's normal to get stuck and need a refresher—so don't hesitate to use the *Show Solution* buttons when you need them.

**Functionality**

Our goal will be to implement a Stack class that has the following behaviors:

1. push - adds an item to the top of the stack
2. pop - removes an item from the top of the stack (and returns the value of that item)
3. size - returns the size of the stack
4. top - returns the value of the item at the top of stack (without removing that item)
5. is\_empty - returns True if the stack is empty and False otherwise

**1. Create and initialize the Stack class**

First, have a look at the walkthrough:

Walkthrough

In [ ]:



​

In the cell below:

* Define a class named Stack and add the \_\_init\_\_ method
* Initialize the arr attribute with an array containing 10 elements, like this: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
* Initialize the next\_index attribute
* Initialize the num\_elements attribute

In [3]:



**class** Stack:

**def** \_\_init\_\_(self, size **=** 10):

self.arr **=** [0 **for** \_ **in** range(size)]

self.next\_index **=** 0

self.num\_elements **=** 0

Let's check that the array is being initialized correctly. We can create a Stack object and access the arr attribute, and we should see our ten-element array:

In [4]:



foo **=** Stack()

print(foo.arr)

print("Pass" **if** foo.arr **==** [0, 0, 0, 0, 0, 0, 0, 0, 0, 0] **else** "Fail")

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Pass

Show Solution

**2. Add the push method**

Next, we need to define our push method, so that we have a way of adding elements to the top of the stack.

Walkthrough

In [ ]:



​

Now give it a try for yourself. Here's are the key things to include:

* The method will need to have a parameter for the value that you want to push
* Remember that next\_index will have the index for where the value should be added
* Once you've added the value, you'll want to increment both next\_index and num\_elements

In [5]:



**class** Stack:

**def** \_\_init\_\_(self, initial\_size **=** 10):

self.arr **=** [0 **for** \_ **in** range(initial\_size)]

self.next\_index **=** 0

self.num\_elements **=** 0

**def** push(self,val):

self.arr[self.next\_index]**=**val

self.next\_index**+=**1

self.num\_elements**+=**1

Let's test it by creating a stack object and pushing an item onto the stack:

In [6]:



foo **=** Stack()

foo.push("Test!")

print(foo.arr)

print("Pass" **if** foo.arr[0] **==** "Test!" **else** "Fail")

['Test!', 0, 0, 0, 0, 0, 0, 0, 0, 0]

Pass

Show Solution

**3. Handle full capacity**

Great, the push method seems to be working fine! But we know that it's not done yet. If we keep pushing items onto the stack, eventually we will run out of room in the array. Currently, that will cause an Index out of range error. In order to avoid a stack overflow, we need to check the capacity of the array before pushing an item to the stack. And if the array is full, we need to increase the array size before pushing the new element.

Walkthrough

In [ ]:



​

First, define the \_handle\_stack\_capacity\_full method:

* Define an old\_arr variable and assign it the current (full) array
* Create a new (larger) array and assign it to arr.
* Iterate over the values in the old array and copy them to the new array.

Then, in the push method:

* Add a conditional to check if the array is full; if it is, call the \_handle\_stack\_capacity\_full

In [19]:



**class** Stack:

**def** \_\_init\_\_(self, initial\_size **=** 10):

self.arr **=** [0 **for** \_ **in** range(initial\_size)]

self.next\_index **=** 0

self.num\_elements **=** 0

**def** handle(self):

arr**=**[0 **for** \_ **in** range(len(self.arr)**\***2)]

**for** i **in** range(0,len(self.arr)):

arr[i]**=**self.arr[i]

self.arr**=**arr

**def** push(self, data):

**if** self.next\_index**==**len(self.arr):

self.handle()

self.arr[self.next\_index] **=** data

self.next\_index **+=** 1

self.num\_elements **+=** 1

We can test this by pushing items onto the stack until we exceed the original capacity. Let's try it and see if we get an error, or if the array size gets increased like we want it to.

In [20]:



foo **=** Stack()

foo.push(1)

foo.push(2)

foo.push(3)

foo.push(4)

foo.push(5)

foo.push(6)

foo.push(7)

foo.push(8)

foo.push(9)

foo.push(10) *# The array is now at capacity!*

foo.push(11) *# This one should cause the array to increase in size*

print(foo.arr) *# Let's see what the array looks like now!*

print("Pass" **if** len(foo.arr) **==** 20 **else** "Fail") *# If we successfully doubled the array size, it should now be 20.*

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Pass

Hide Solution

In [ ]:



**class** Stack:

**def** \_\_init\_\_(self, initial\_size **=** 10):

self.arr **=** [0 **for** \_ **in** range(initial\_size)]

self.next\_index **=** 0

self.num\_elements **=** 0

**def** push(self, data):

**if** self.next\_index **==** len(self.arr):

print("Out of space! Increasing array capacity ...")

self.\_handle\_stack\_capacity\_full()

self.arr[self.next\_index] **=** data

self.next\_index **+=** 1

self.num\_elements **+=** 1

​

**def** \_handle\_stack\_capacity\_full(self):

old\_arr **=** self.arr

​

self.arr **=** [0 **for** \_ **in** range( 2**\*** len(old\_arr))]

**for** index, element **in** enumerate(old\_arr):

self.arr[index] **=** element

**4. Add the size and is\_empty methods**

Next, we need to add a couple of simple methods:

* Add a size method that returns the current size of the stack
* Add an is\_empty method that returns True if the stack is empty and False otherwise

(This one is pretty straightforward, so there's no walkthrough—but there's still solution code below if you should need it.)

In [21]:



**class** Stack:

**def** \_\_init\_\_(self, initial\_size **=** 10):

self.arr **=** [0 **for** \_ **in** range(initial\_size)]

self.next\_index **=** 0

self.num\_elements **=** 0

**def** push(self, data):

**if** self.next\_index **==** len(self.arr):

print("Out of space! Increasing array capacity ...")

self.\_handle\_stack\_capacity\_full()

self.arr[self.next\_index] **=** data

self.next\_index **+=** 1

self.num\_elements **+=** 1

**def** size(self):

**return** self.num\_elements

​

**def** is\_empty(self):

**return** self.num\_elements **==** 0

​

**def** \_handle\_stack\_capacity\_full(self):

old\_arr **=** self.arr

​

self.arr **=** [0 **for** \_ **in** range( 2**\*** len(old\_arr))]

**for** index, value **in** enumerate(old\_arr):

self.arr[index] **=** value

Let's test the new methods:

In [22]:



foo **=** Stack()

print(foo.size()) *# Should return 0*

print(foo.is\_empty()) *# Should return True*

foo.push("Test") *# Let's push an item onto the stack and check again*

print(foo.size()) *# Should return 1*

print(foo.is\_empty()) *# Should return False*

0

True

1

False

Show Solution

**5. Add the pop method**

The last thing we need to do is add the pop method.

Walkthrough

In [ ]:



​

The method needs to:

* Check if the stack is empty and, if it is, return None
* Decrement next\_index and num\_elements
* Return the item that is being "popped"

In [25]:



**class** Stack:

**def** \_\_init\_\_(self, initial\_size **=** 10):

self.arr **=** [0 **for** \_ **in** range(initial\_size)]

self.next\_index **=** 0

self.num\_elements **=** 0

**def** push(self, data):

**if** self.next\_index **==** len(self.arr):

print("Out of space! Increasing array capacity ...")

self.\_handle\_stack\_capacity\_full()

self.arr[self.next\_index] **=** data

self.next\_index **+=** 1

self.num\_elements **+=** 1

**def** pop(self):

**if** self.is\_empty():

**return** **None**

**else**:

i**=**self.next\_index

self.next\_index **-=**1

self.num\_elements**-=**1

**return** self.arr[i**-**1]

​

**def** size(self):

**return** self.num\_elements

​

**def** is\_empty(self):

**return** self.num\_elements **==** 0

**def** \_handle\_stack\_capacity\_full(self):

old\_arr **=** self.arr

​

self.arr **=** [0 **for** \_ **in** range( 2**\*** len(old\_arr))]

**for** index, value **in** enumerate(old\_arr):

self.arr[index] **=** value

Let's test the pop method:

In [26]:



foo **=** Stack()

foo.push("Test") *# We first have to push an item so that we'll have something to pop*

print(foo.pop()) *# Should return the popped item, which is "Test"*

print(foo.pop()) *# Should return None, since there's nothing left in the stack*

0

None