Implementing a Python Stack

There are a couple of options when you're implementing a Python stack. This article won't cover all of them, just the basic ones that will meet almost all of your needs. You'll focus on using data structures that are part of the Python library, rather than writing your own or using third-party packages.

You'll look at the following Python stack implementations:

- `list`
- `collections.deque`
- `queue.LifoQueue`

Using `list` to Create a Python Stack

The built-in `list` structure that you likely use frequently in your programs can be used as a stack. Instead of `.push()`, you can use `.append()` to add new elements to the top of your stack, while `.pop()` removes the elements in the LIFO order:

```
>>>
>>> myStack = []

>>> myStack.append('a')
>>> myStack.append('b')
>>> myStack.append('c')

>>> myStack
['a', 'b', 'c']

>>> myStack.pop()
'c'
>>> myStack.pop()
'b'
>>> myStack.pop()
'a'

>>> myStack.pop()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
```

```
IndexError: pop from empty list
```

You can see in the final command that a `list` will raise an `IndexError` if you call `.pop()` on an empty stack.

`list` has the advantage of being familiar. You know how it works and likely have used it in your programs already.

Unfortunately, `list` has a few shortcomings compared to other data structures you'll look at. The biggest issue is that it can run into speed issues as it grows. The items in a `list` are stored with the goal of providing fast access to random elements in the `list`. At a high level, this means that the items are stored next to each other in memory.

If your 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.

There is a less serious problem as well. If you use `.insert()` to add an element to your stack at a position other than the end, it can take much longer. This is not normally something you would do to a stack, however.

The next data structure will help you get around the reallocation problem you saw with `list`.

Using `collections.deque` to Create a Python Stack

The `collections` module contains `deque`, which is useful for creating Python stacks. `deque` is pronounced "deck" and stands for "double-ended queue."

You can use the same methods on `deque` as you saw above for `list`, `.append()`, and `.pop()`:

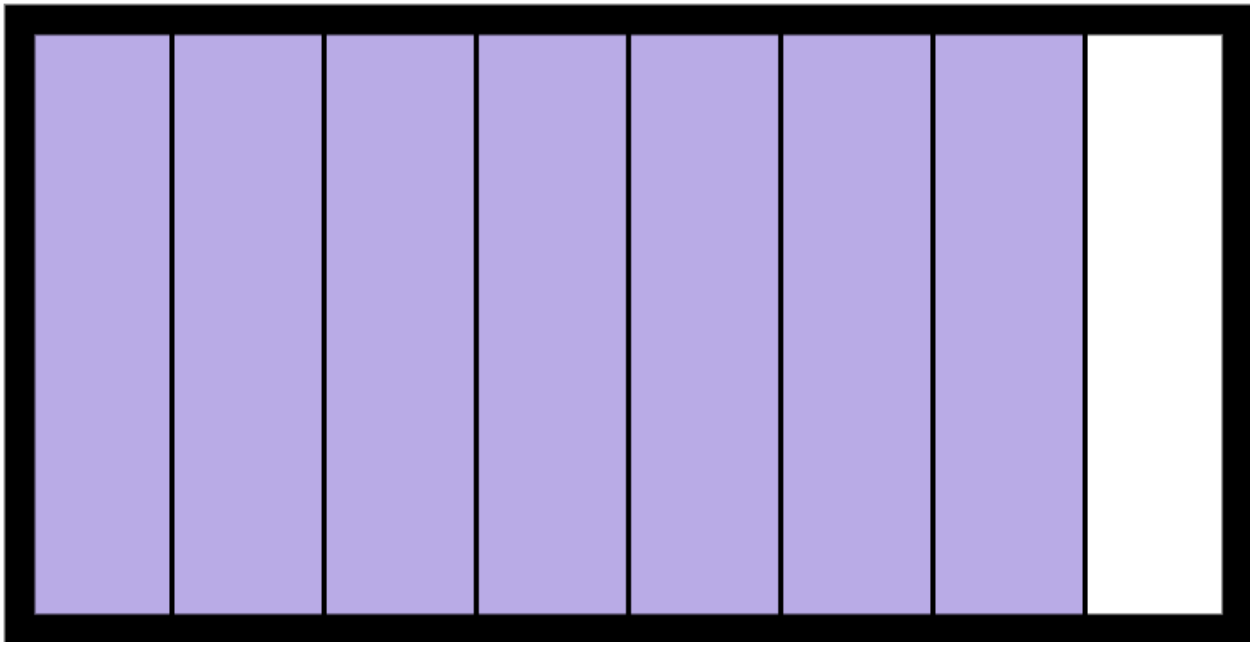
```
>>>
>>> from collections import deque
>>> myStack = deque()

>>> myStack.append('a')
```

```
>>> myStack.append('b')
>>> myStack.append('c')

>>> myStack
deque(['a', 'b', 'c'])

>>> myStack.pop()
'c'
>>> myStack.pop()
'b'
>>> myStack.pop()
'a'

>>> myStack.pop()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: pop from an empty deque
```

This looks almost identical to the `list` example above. At this point, you might be wondering why the Python core developers would create two data structures that look the same.

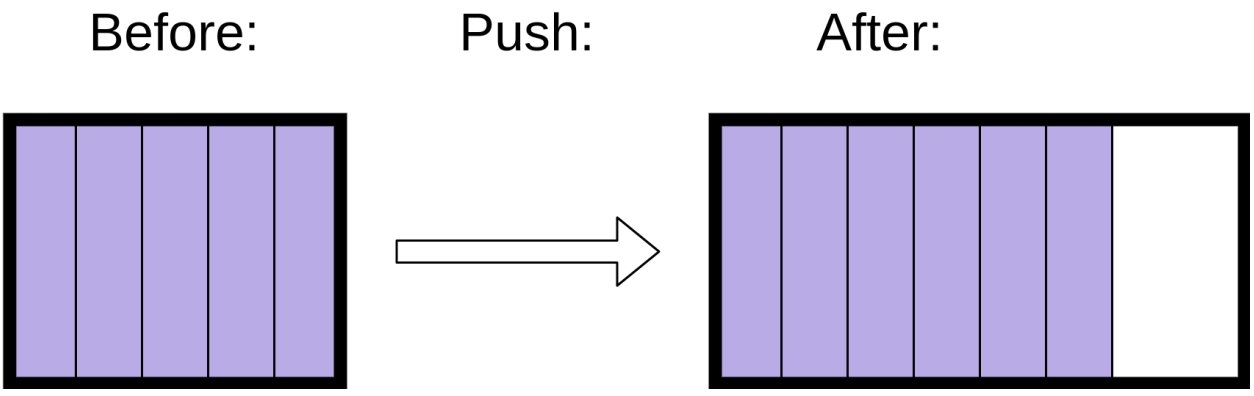
Why Have `deque` and `list`?

As you saw in the discussion about `list` above, it was built upon blocks of contiguous memory, meaning that the items in the list are stored right next to each other:



This works great for several operations, like indexing into the `list`. Getting `myList[3]` is fast, as Python knows exactly where to look in memory to find it. This memory layout also allows slices to work well on lists.

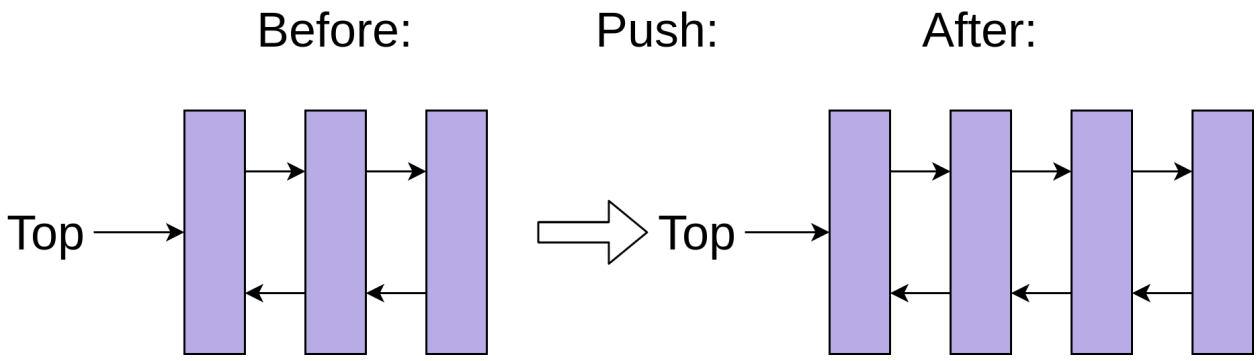
The contiguous memory layout is the reason that `list` might need to take more time to `.append()` some objects than others. If the block of contiguous memory is full, then it will need to get another block, which can take much longer than a normal `.append()`:



`deque`, on the other hand, is built upon a doubly linked list. In a linked list structure, each entry is stored in its own memory block and has a reference to the next entry in the list.

A doubly linked list is just the same, except that each entry has references to both the previous and the next entry in the list. This allows you to easily add nodes to either end of the list.

Adding a new entry into a linked list structure only requires setting the new entry's reference to point to the current top of the stack and then pointing the top of the stack to the new entry:



This constant-time addition and removal of entries onto a stack comes with a trade-off, however. Getting `myDeque[3]` is slower than it was for a list, because Python needs to walk through each node of the list to get to the third element.

Fortunately, you rarely want to do random indexing or slicing on a stack. Most operations on a stack are either `push` or `pop`.

The constant time `.append()` and `.pop()` operations make `deque` an excellent choice for implementing a Python stack if your code doesn't use threading.

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)

Python Stack => —----------

```
#list is a dynamic array and store data in Block ways in Memory that's why use=> deque form collections.



from collections import deque



stack = deque() # declear stack



dir(stack)

# print(dir(stack)) #list of fucntion of deque.



# print(len(stack)) # size of stack.







#Create/Add => --------------------

# stack.append('a')
```

```
# stack.append('b')

# stack.append('c')

# stack.append('d')

# stack.append('e')



# print(stack)



#Delete/pop=>--------------------

# stack.pop()

# stack.pop()

# stack.pop()

# stack.pop()

# stack.pop()

# stack.pop() #IndexError: pop from an empty deque

# print(stack)




# ----------------------------------Stack with Class
-------------------------------

from collections import deque
```

```
class Stack:

    def __init__(self):

        self.container = deque()



    def show(self):

        return print(f"Total Element of Stack=> {self.container}")



    def push(self,value):

        self.container.append(value)



    def pop(self,value):

        return self.container.pop(value)



    def peek(self): #show the value of last element not add/remove

        return print(f'Last Element of Stack=> {self.container[-1]}')



    def is_empty(self):

        return print(f'Stack is Empty=>{len(self.container)==0}')
```

```
    def size(self):
        return print(f'Size/Total Element of Stack=>
{len(self.container)}')




obj_stack = Stack()



obj_stack.push(1)

obj_stack.push(2)

obj_stack.push(3)

obj_stack.push(4)

obj_stack.push(5)

obj_stack.peek()

obj_stack.show()

obj_stack.size()

obj_stack.is_empty()
```