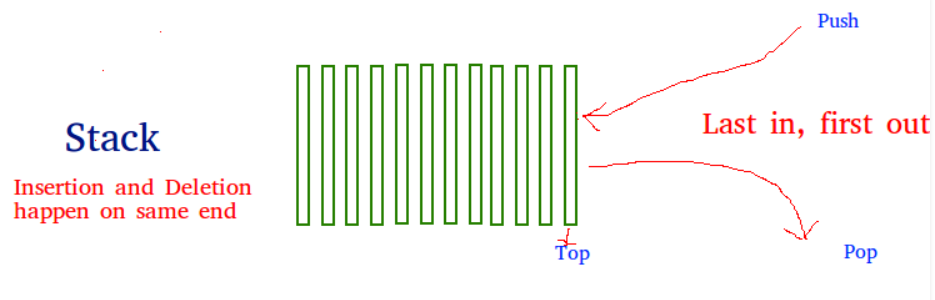
**Python Advanced Concepts(Selected - Part 1)**

**Overview**

|  |
| --- |
| The advanced features of any programming language are usually discovered through extensive experience. You’re coding up a complicated project and find yourself searching for something on the internet. In this section we will explore some of the (selected) Python advanced concepts. We will also explore modules that are often used in a Python project |

**What Is a Stack?**

A stack is a data structure that stores items in an Last-In/First-Out manner. This is frequently referred to as LIFO. This is in contrast to a queue, which stores items in a First-In/First-Out (**FIFO**) manner.



**Implementing a Stack**

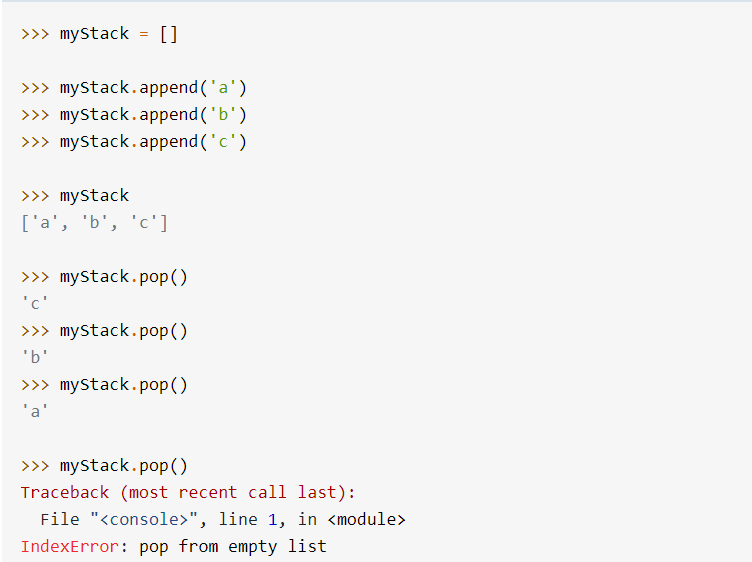
There are a couple of options available when you’re implementing a stack in Python. This module won’t cover all of them, just the basic ones that will meet almost all of your needs. You’ll focus on using data structures that are part of the Python library, rather than writing your own or using third-party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

We will look at the following Python stack implementations:

* list
* collections.deque

**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:



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 we mentioned with a 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()**:



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 the list above, it was built upon blocks of contiguous memory, meaning that the items in the list are stored right next to each other.

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**](https://en.wikipedia.org/wiki/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.

**Python Stacks: Which Implementation Should You Use?**

In general, you should use a deque if you’re not using **threading (** see [here for an explanation](https://realpython.com/intro-to-python-threading/) **)**. If you are using threading, then you should use a **LifoQueue** unless you’ve measured your performance and found that a small boost in speed for pushing and popping will make enough difference to warrant the maintenance risks.

***list*** may be familiar, but it should be avoided because it can potentially have memory reallocation issues. The interfaces for deque and list are identical, and deque doesn’t have these issues, which makes deque the best choice for your non-threaded Python stack.

You are now able to:

* Recognize when a stack would be a good data structure.
* Select which implementation is right for your problem.