Have you heard of stacks and wondered what they are? Do you have the general idea but are wondering how to implement a Python stack? You've come to the right place!

In this tutorial, you'll learn:

- How to recognize when a stack is a good choice for data structures
- How to decide which implementation is best for your program
- What extra considerations to make about stacks in a threading or multiprocessing environment

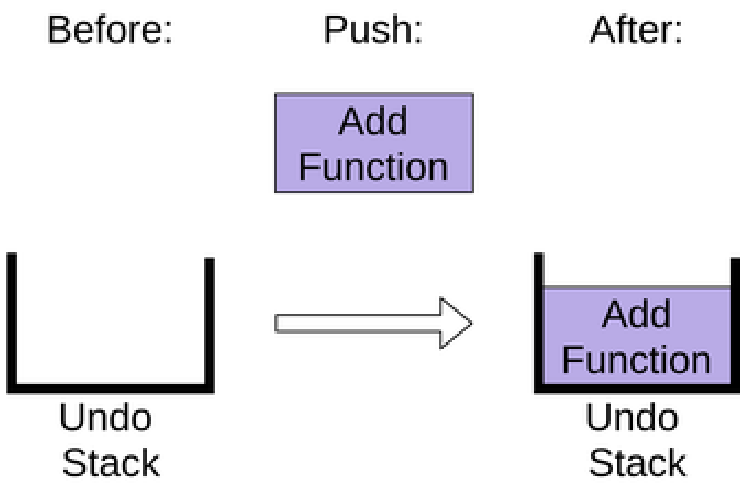
This tutorial is for Pythonistas who are comfortable running scripts, know what a `list` is and how to use it, and are wondering how to implement Python stacks.

What Is a Stack?

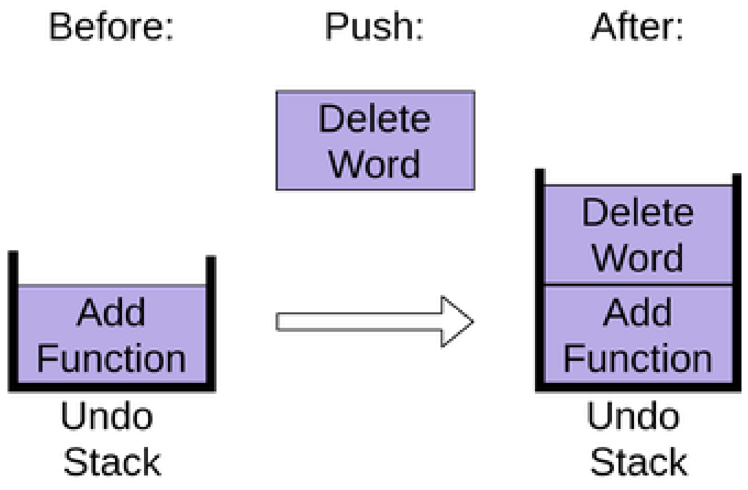
A stack is a data structure that stores items in an Last-In/First-Out manner. This is frequently referred to as LIFO. This is in contrast to a queue, which stores items in a First-In/First-Out (FIFO) manner.

It's probably easiest to understand a stack if you think of a use case you're likely familiar with: the *Undo*feature in your editor.

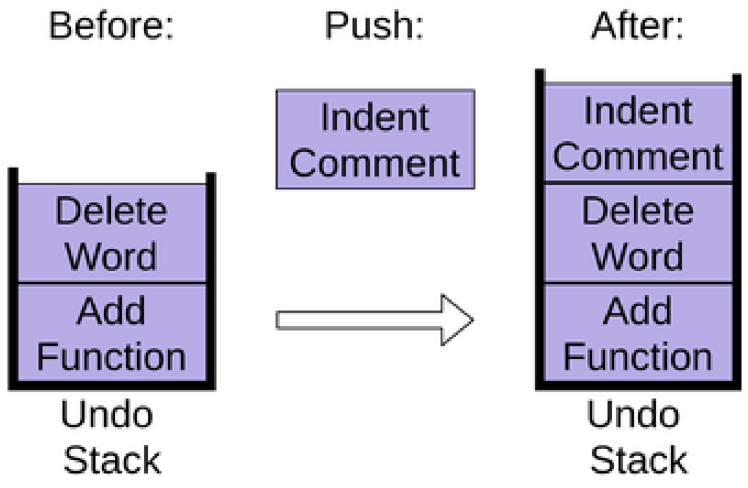
Let's imagine you're editing a Python file so we can look at some of the operations you perform. First, you add a new function. This adds a new item to the undo stack:



You can see that the stack now has an *Add Function* operation on it. After adding the function, you delete a word from a comment. This also gets added to the undo stack:

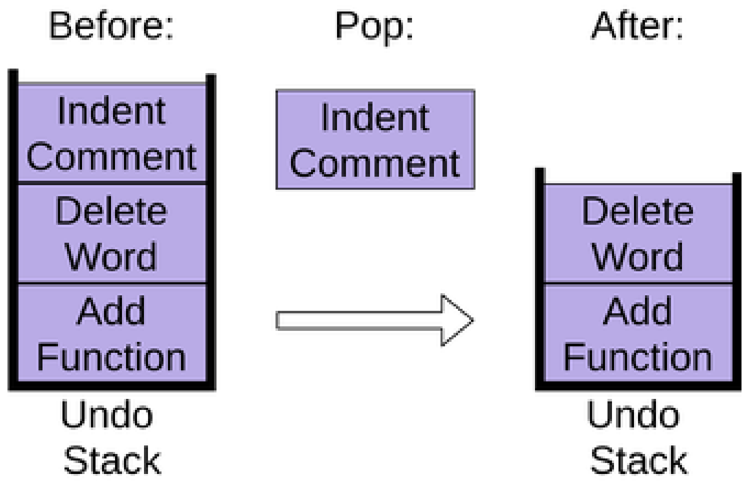


Notice how the *Delete Word* item is placed on top of the stack. Finally you indent a comment so that it's lined up properly:



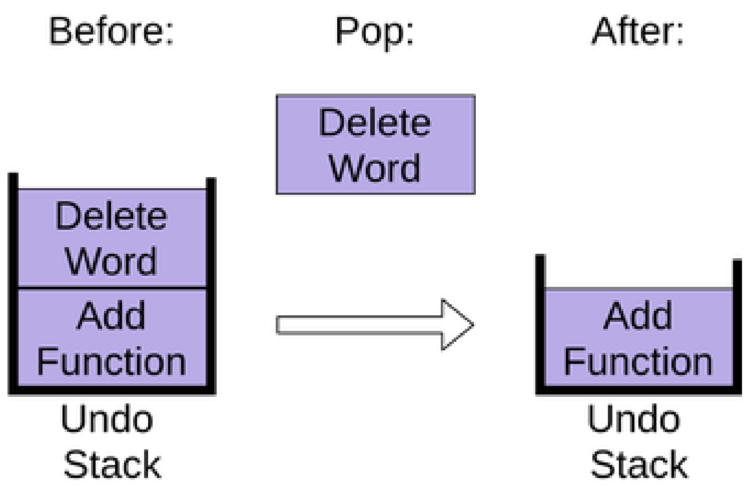
You can see that each of these commands are stored in an undo stack, with each new command being put at the top. When you're working with stacks, adding new items like this is called `push`.

Now you've decided to undo all three of those changes, so you hit the undo command. It takes the item at the top of the stack, which was indenting the comment, and removes that from the stack:



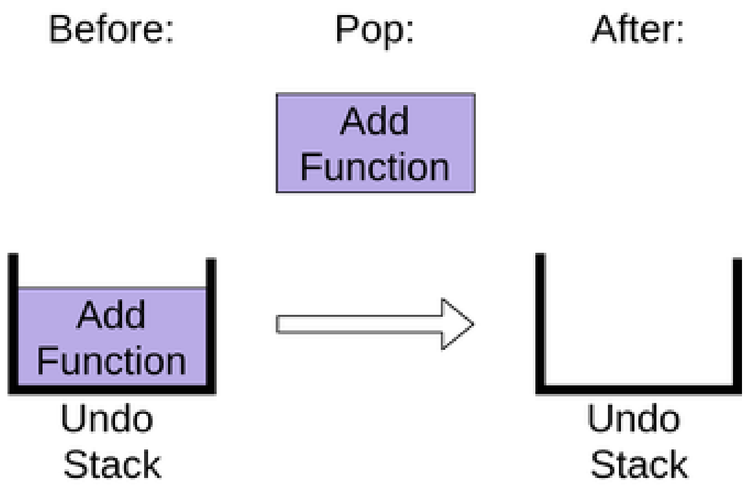
Your editor undoes the indent, and the undo stack now contains two items. This operation is the opposite of `push` and is commonly called `pop`.

When you hit undo again, the next item is popped off the stack:



This removes the *Delete Word* item, leaving only one operation on the stack.

Finally, if you hit *Undo* a third time, then the last item will be popped off the stack:



The undo stack is now empty. Hitting *Undo* again after this will have no effect because your undo stack is empty, at least in most editors. You'll see what happens when you call `.pop()` on an empty stack in the implementation descriptions below.

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`

Remove ads

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.”

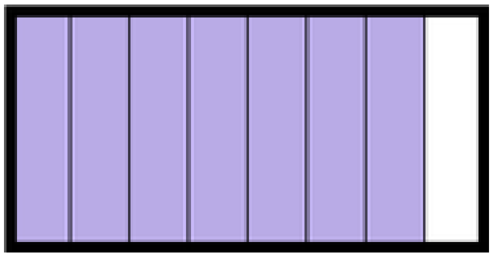
You can use the same methods on `deque` as you saw above for `list`, `.append()`, and `.pop()`:

```
>>> 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.