
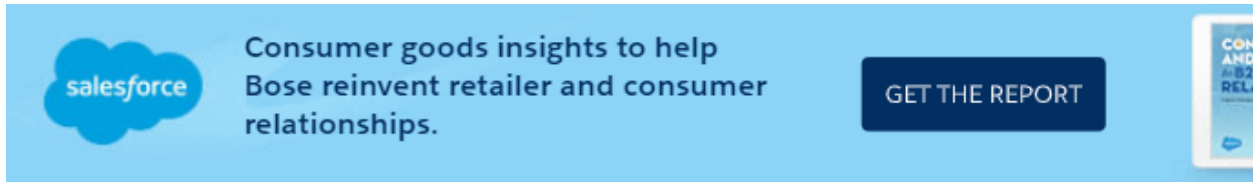


Stacks and Queues in Python

By  Marcus Sanatan (<https://twitter.com/marcussanatan>) • January 25, 2019 •
2 Comments (/stacks-and-queues-in-python/#disqus_thread)



Introduction

Data structures organize storage in computers so that we can efficiently access and change data. Stacks and Queues are some of the earliest data structures defined in computer science.

Simple to learn and easy to implement, their uses are common and you'll most likely find yourself incorporating them in your software for various tasks.

It's common for Stacks and Queues to be implemented with an Array or Linked List (</python-linked-lists/>). We'll be relying on the `List` data structure to accommodate both Stacks and Queues.

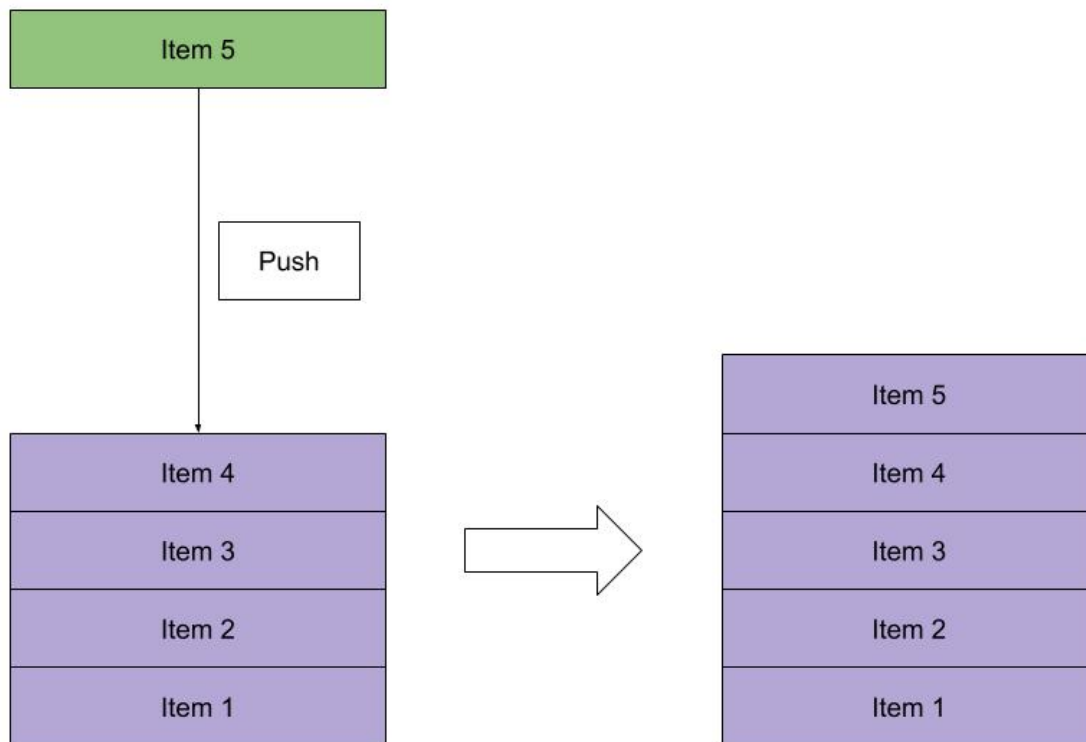
How do they Work?

Stack

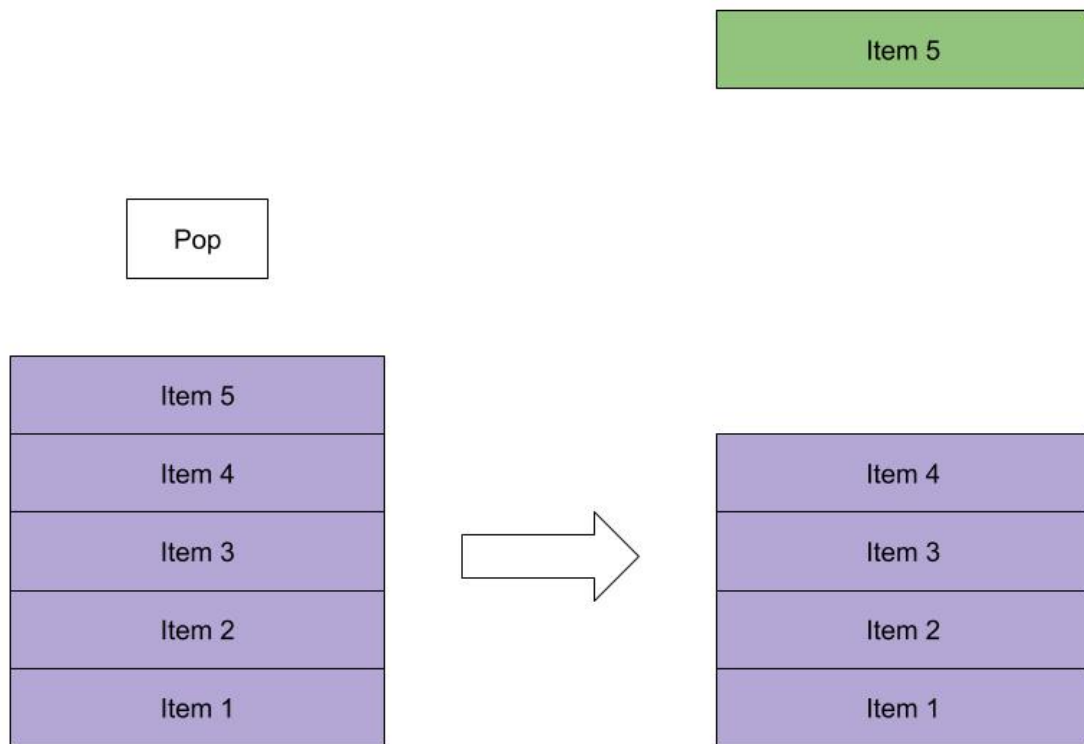
Stacks, like the name suggests, follow the **Last-in-First-Out** (LIFO) principle. As if stacking coins one on top of the other, the last coin we put on the top is the one that is the first to be removed from the stack later.

To implement a stack, therefore, we need two simple operations:

- push - adds an element to the top of the stack:



- pop - removes the element at the top of the stack:

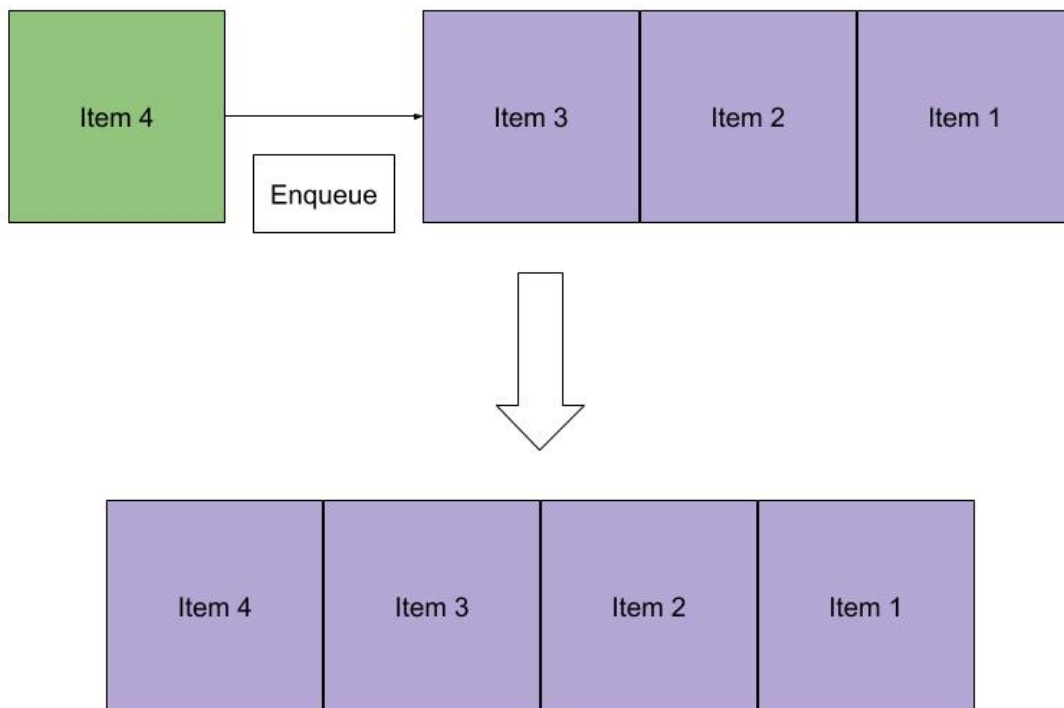


Queue

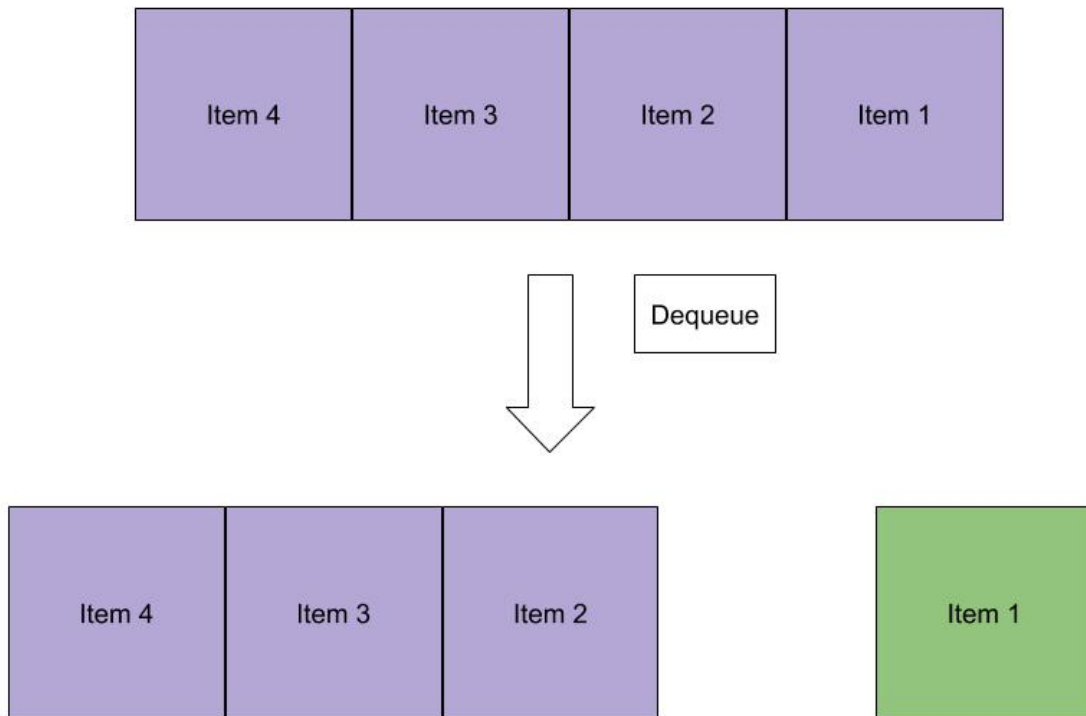
Queues, like the name suggests, follow the **First-in-First-Out** (FIFO) principle. As if waiting in a queue for the movie tickets, the first one to stand in line is the first one to buy a ticket and enjoy the movie.

To implement a queue, therefore, we need two simple operations:

- enqueue - adds an element to the end of the queue:



- `dequeue` - removes the element at the beginning of the queue:



Stacks and Queues using Lists

Python's built-in `List` data structure comes bundled with methods to simulate both *stack* and *queue* operations.

Let's consider a stack of letters:

```
letters = []

# Let's push some letters into our list
letters.append('c')
letters.append('a')
letters.append('t')
letters.append('g')

# Now let's pop letters, we should get 'g'
last_item = letters.pop()
print(last_item)

# If we pop again we'll get 't'
last_item = letters.pop()
print(last_item)

# 'c' and 'a' remain
print(letters) # ['c', 'a']
```

We can use the same functions to implement a Queue. The `pop` function optionally takes the index of the item we want to retrieve as an argument.

So we can use `pop` with the first index of the list i.e. `0`, to get queue-like behavior.

Consider a "queue" of fruits:

```
fruits = []

# Let's enqueue some fruits into our list
fruits.append('banana')
fruits.append('grapes')
fruits.append('mango')
fruits.append('orange')

# Now Let's dequeue our fruits, we should get 'banana'
first_item = fruits.pop(0)
print(first_item)

# If we dequeue again we'll get 'grapes'
first_item = fruits.pop(0)
print(first_item)

# 'mango' and 'orange' remain
print(fruits) # ['c', 'a']
```

Again, here we use the `append` and `pop` operations of the list to simulate the core operations of a queue.

Subscribe to our Newsletter

Get occasional tutorials, guides, and reviews in your inbox. No spam ever.

Unsubscribe at any time.

Stacks and Queues with the Deque Library

Python has a `deque` (pronounced 'deck') library that provides a sequence with efficient methods to work as a stack or a queue.

`deque` is short for *Double Ended Queue* - a generalized queue that can get the first or last element that's stored:

```
from collections import deque

# you can initialize a deque with a list
numbers = deque()

# Use append like before to add elements
numbers.append(99)
numbers.append(15)
numbers.append(82)
numbers.append(50)
numbers.append(47)

# You can pop like a stack
last_item = numbers.pop()
print(last_item) # 47
print(numbers) # deque([99, 15, 82, 50])

# You can dequeue like a queue
first_item = numbers.popleft()
print(first_item) # 99
print(numbers) # deque([15, 82, 50])
```

If you'd like to learn more about the `deque` library and other types of collections Python provides, you can read our [Introduction to Python's Collections Module \(/introduction-to-pythons-collections-module/\)](#) article.

Stricter Implementations in Python

If your code needed a stack and you provide a `List`, there's nothing stopping a programmer from calling `insert`, `remove` or other list functions that will affect the order of your stack! This fundamentally ruins the point of defining a stack, as it no longer functions the way it should.

There are times when we'd like to ensure that only valid operations can be performed on our data.

We can create classes that only exposes the necessary methods for each data structure.

To do so, let's create a new file called `stack_queue.py` and define two classes:

```
# A simple class stack that only allows pop and push operations
```

```
class Stack:
```

```
    def __init__(self):
        self.stack = []

    def pop(self):
        if len(self.stack) < 1:
            return None
        return self.stack.pop()

    def push(self, item):
        self.stack.append(item)

    def size(self):
        return len(self.stack)
```

```
# And a queue that only has enqueue and dequeue operations
```

```
class Queue:
```

```
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if len(self.queue) < 1:
            return None
        return self.queue.pop(0)

    def size(self):
        return len(self.queue)
```

The programmers using our `Stack` and `Queue` are now encouraged to use the methods provided to manipulate the data instead.

Examples

Imagine you're a developer working on a brand new word processor. You're tasked with creating an undo feature - allowing users to backtrack their actions till the beginning of the session.

A stack is an ideal fit for this scenario. We can record every action the user takes by pushing it to the stack. When the user wants to undo an action they'll pop it from the stack. We can quickly simulate the feature like this:

```
document_actions = Stack()

# The first enters the title of the document
document_actions.push('action: enter; text_id: 1; text: This is my favourite document')
# Next they center the text
document_actions.push('action: format; text_id: 1; alignment: center')
# As with most writers, the user is unhappy with the first draft and undoes the center alignment
document_actions.pop()
# The title is better on the left with bold font
document_actions.push('action: format; text_id: 1; style: bold')
```

Queues have widespread uses in programming as well. Think of games like Street Fighter or Super Smash Brothers. Players in those games can perform special moves by pressing a combination of buttons. These button combinations can be stored in a queue.

Now imagine that you're a developer working on a new fighting game. In your game, every time a button is pressed, an input event is fired. A tester noticed that if buttons are pressed too quickly the game only processes the first one and special moves won't work!

You can fix that with a queue. We can enqueue all input events as they come in. This way it doesn't matter if input events come with little time between them, they'll all be stored and available for processing. When we're processing the moves we can dequeue them. A special move can be worked out like this:

```
input_queue = Queue()

# The player wants to get the upper hand so pressing the right combination of buttons quickly
input_queue.enqueue('DOWN')
input_queue.enqueue('RIGHT')
input_queue.enqueue('B')

# Now we can process each item in the queue by dequeuing them
key_pressed = input_queue.dequeue() # 'DOWN'

# We'll probably change our player position
key_pressed = input_queue.dequeue() # 'RIGHT'

# We'll change the player's position again and keep track of a potential special move to perform
key_pressed = input_queue.dequeue() # 'B'

# This can do the act, but the game's logic will know to do the special move
```

Conclusion

Stacks and queues are simple data structures that allow us to store and retrieve data sequentially. In a stack, the last item we enter is the first to come out. In a queue, the first item we enter is the first to come out.

We can add items to a stack using the `push` operation and retrieve items using the `pop` operation. With queues, we add items using the `enqueue` operation and retrieve items using the `dequeue` operation.

In Python, we can implement stacks and queues just by using the built-in `List` data structure. Python also has the `deque` library which can efficiently provide stack and queue operations in one object. Finally, we've made our stack and queue classes for tighter control of our data.

There are many real-world use cases for stacks and queues, understanding them allows us to solve many data storage problems in an easy and effective manner.

📁 [python \(/tag/python/\)](/tag/python/), [data structures \(/tag/data-structures/\)](/tag/data-structures/)

🐦 (https://twitter.com/share?

text=Stacks%20and%20Queues%20in%20Python&url=https://stackabuse.com/stacks-and-queues-in-python/)

f (https://www.facebook.com/sharer/sharer.php?u=https://stackabuse.com/stacks-and-queues-in-python/)

g+ (https://plus.google.com/share?url=https://stackabuse.com/stacks-and-queues-in-python/)

in (https://www.linkedin.com/shareArticle?

mini=true%26url=https://stackabuse.com/stacks-and-queues-in-python/%26source=https://stackabuse.com)



(/author/marcus/)

About Marcus Sanatan (/author/marcus/)

🏠 Trinidad and Tobago

🐦 Twitter (https://twitter.com/marcussanatan)

🌐 Website (https://github.com/msanatan)

Web Dev|Games|Music|Art|Fun|Caribbean I love many things and coding is one of them!