Stack

This lesson will introduce you to the stack data structure and its implementation in Python which we'll use in the problems throughout this chapter.

We'll cover the following

In this lesson, we are going to consider the stack data structure and its implementation in Python.

In subsequent lessons, we'll provide specific problems where a stack is particularly useful. We'll be using the implementation that we develop in this lesson to solve those problems.

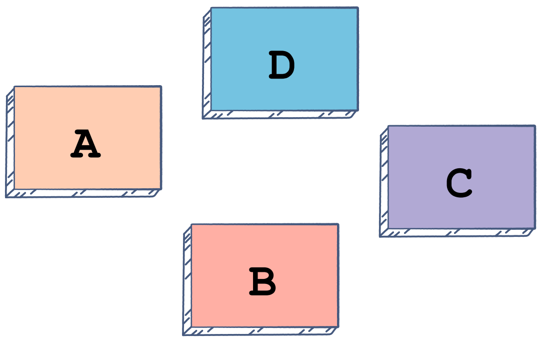
What is a stack?

First of all, let me describe what a stack is. Based on the name, it should be a relatively familiar concept.

Let's assume that we have four books with the following riveting titles:

- A
- B
- C
- D

At the moment, these books are strewn out all over the floor and we want to stack them up neatly.



Four books are lying all over the place.

1 of 5

Now we have a nice neat stack of books! If we want to retrieve a book from this stack, we can take the book on top. Taking a book from the bottom is a bit precarious and we don't want to topple the entire stack. Therefore, we'll take down the top book on the stack and read it or do whatever we want to do with it.

Let's say we want to take **Book A**. Right now, it is at the bottom of the stack, so we need to take **Book D**, put it down, then do the same for **Book C** and **Book B**, and then we can access **Book A**.

This is the main idea of a stack. The data structure stack is very similar to a physical stack that you'd most likely be familiar with. The stack data structure allows us to place any programming artifact, variable or object on it, just as our example stack allowed us to put books in it.

Stack Operations

`Push`

The operation to insert elements in a stack is called **push**. When we *push* the book on a stack, we put the book on the previous *top* element which means

that the new book becomes the *top* element. This is what we mean when we use the *push* operation, we *push* elements onto a stack. We insert elements onto a stack and the last element to be pushed is the new *top* of the stack.

Pop

There is another operation that we can perform on the stack, popping. Popping is when we take the top book of the stack and put it down. This implies that when we remove an element from the stack, the stack follows the *First-In, Last Out* property. This means that the top element, the last to be inserted, is removed when we perform the pop operation.

`Push` and `Pop` are two fundamental routines that we'll need for this data structure.

Peek

Another thing that we can do is view the top element of the stack so we can ask the data structure: "What's the top element?" and it can give that to us using the *peek* operation. Note that the *peek* operation does not remove the *top* element, it merely returns it.

We can also check whether or not the stack is empty, and a few other things too, that will come along the way as we implement it.

Now I'm going to create a stack class, and the constructor of the class is going to initialize a Python list. The Python list has a lot of things that we're after, and it's going to make it easier for us to tweak Python's implementation of the list to be adapted to what we would expect to see in a stack, namely, push/pop.

```
"""
Stack Data Structure.
"""
class Stack():
    def __init__(self):
        self.items = []
```

I'm defining a class variable that I'm calling `items`, and I'm assigning it to an empty list. `self.items` is created when we create a stack object so now let's

create the `push` method:

```
"""
Stack Data Structure.
"""
class Stack():
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)
```

`push` is going to be a member of this class, and it's going to take an `item` as an argument. In our example, that item is the book. We are putting or pushing the name of the book onto the top of the stack.

`append` is a built-in method for a Python list that adds the `item` to the end of the list. This is just what we want to do for our stack in the push method.

```
"""
Stack Data Structure.
"""
class Stack():
    def __init__(self):
        self.items = []

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

    def pop(self):
        return self.items.pop()
```

The other method we need to implement is `pop`. Since we're basing this off a list, Python makes this very easy as well. There is an implicit `pop` method that returns the last element inserted to the list.

Another thing that we're going to do is add a few more methods, but before that, I'm going to do a test drive.

```
"""
Stack Data Structure.

"""
class Stack():
    def __init__(self):
        self.items = []

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

    def pop(self):
        return self.items.pop()

    def get_stack(self):
        return self.items

myStack = Stack()
myStack.push("A")
myStack.push("B")
print(myStack.get_stack())
myStack.push("C")
print(myStack.get_stack())
myStack.pop()
print(myStack.get_stack())
```

We make a method called `get_stack()`. This will return the `items` list, which forms the core of the stack. Then I define a stack object, `myStack`, on **line 17** and push `A` and `B` onto the stack, **lines 18-19**. After these operations, I can print `myStack` and the output is `['A', 'B']`. `A` is at the bottom of the stack, and `B` is on the top. Then we push `C` on **line 21** and the output is `['A', 'B', 'C']`, which implies `C` is the top.

We understand the core fundamentals of the stack. A few of the other methods we can include in this are helpful, namely, we could have a method called `is_empty`. It will return whether or not the stack is empty.

```
"""
Stack Data Structure.
"""
class Stack():
    def __init__(self):
        self.items = []

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

    def pop(self):
        return self.items.pop()
    def is_empty(self):
        return self.items == []

    def get_stack(self):
        return self.items

myStack = Stack()
print(myStack.is_empty())
```

We get `True` here because there’s nothing on the stack at this point. If we push something and call `is_empty()`, we should get `False`. You can try it out yourself.

```
"""
Stack Data Structure.
"""
class Stack():
    def __init__(self):
        self.items = []

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

    def pop(self):
        return self.items.pop()

    def is_empty(self):
        return self.items == []

    def peek(self):
        if not self.is_empty():
            return self.items[-1]

    def get_stack(self):
        return self.items

myStack = Stack()
myStack.push("A")
myStack.push("B")
myStack.push("C")
myStack.push("D")
print(myStack.peek())
```

Now we have the peek operation on **line 17** which tells us the topmost element of the stack.

If `peek` is called on the stack in the above code, it should return `D`. `peek` checks if the `stack` is not empty on **line 18** and if it isn't, it returns the last element of the list, which in this case, is the top element of the stack. We return `self.items[-1]` on **line 19** which is the last item in the list for Python. Hence, at first, we get `D` as the element that Python thinks is on the top of the stack.

> Note that you can also check whether the stack is empty or not in the methods implemented previously. However, to keep things simple, this step has been omitted from the other methods.

Go ahead and create a stack object. You don't have to limit yourself to letters or strings or anything like that, you could also push numbers. Just play around with this data structure and get a sense of how it works. I'll see you in the next lesson.