Week 3 - Implementing the Stack

Updated Sept 25, 2:30pm

Now that we have some practice using the Stack ADT, let's think about how to actually implement it.

Here are two ways of implementing the Stack based on lists.

```
class EmptyStackError(Exception):
    """Exception used when calling pop on empty stack."""
    pass


class Stack:

    def __init__(self):
        """ (Stack) -> NoneType """
        self.items = []

    def is_empty(self):
        """ (Stack) -> bool """
        return len(self.items) == 0

    def push(self, item):
        """ (Stack, object) -> NoneType """
        self.items.append(item)

    def pop(self):
        """ (Stack) -> object """
        try:
            return self.items.pop()
        except IndexError:
            raise EmptyStackError


class Stack2:

    def __init__(self):
        """ (Stack) -> NoneType """
        self.items = []

    def is_empty(self):
        """ (Stack) -> bool """
        return len(self.items) == 0

    def push(self, item):
        """ (Stack, object) -> NoneType """
        self.items.insert(item, 0)

    def pop(self):
        """ (Stack) -> object """
        try:
```

```
            item = self.items[0]
            self.items = self.items[1:]
            return item
        except IndexError:
            raise EmptyStackError
```

Even though these two implementations seem to be conceptually the same (the first uses the back of the list, the second uses the front), it turns out their performance is quite different!

A simple time profiler

By making use of a built-in Python library, we can easily get rough estimates on how long our code takes to run. I've created a module timer.py to simplify the use even further, as illustrated in the following code:

```
def compare_times():
    """Compare the times of operations of the two stacks."""
    stack = Stack()
    stack2 = Stack2()

    for i in range(1, 10):
        for j in range(40000):
            stack.push(j)
        with Timer('Stack 1: push ' + str(i*40000)):
            for j in range(100):
                stack.push(1)
                stack.pop(1)

    for i in range(1, 10):
        for j in range(40000):
            stack2.push(j)
        with Timer('Stack 2: pop ' + str(i*40000)):
            for j in range(100):
                stack2.push(1)
                stack2.pop()
```

Running this function illustrates a stark difference between the two stacks. While the `Stack` class seems to take the **same** amount of time per operation regardless of how many items are on the stack, the `Stack2` class seems to have the amount of time **grow linearly** with the number of items. Put another way, the amount of time required in a `Stack2` operation is roughly proportional to the size of the stack!

Python array allocation

To understand why there's such a dramatic difference, we really need to understand how Python lists are stored in memory. Recall that variables in Python store *references* to objects. A Python list is special type of object whose items are stored in consecutive blocks of memory (or rather, the references to its items). This is what makes accessing list elements so fast: getting the *i*-th list item can be done just by calculating its address, based on the address where the list starts, and offset by *i* addresses.

But to preserve this quality, lists must always be contiguous; there can't be any "holes". This makes insertion and removal less efficient: for an item to be deleted, all items after it

have to be moved down one address, and similarly, for insertion all items are moved up one address. So in particular, it's much faster to add and remove at the end of the list than at its beginning!

David Liu (liudavid at cs dot toronto dot edu)
Come find me in BA4260
Site design by Deyu Wang (dyw999 at gmail dot com)