Week 3 - Stacks

Updated Sept 22, 7:30pm

Abstract Data Types

Because we've been both designing and implementing classes so far, we haven't stressed the difference between the public *interface* of a class and its internal *implementation* of that interface. However, you make use of this difference all the time in programming: whenever you use a built-in function, you only worry about what it does, not how it works. This is often true of the testing you've been doing: you test based on what the function is supposed to do, and not how it's implemented.

An **Abstract Data Type** is a particular kind of class that's defined purely by its interface rather than its implementation - that's what makes it *abstract*. You're already familiar with three abstract data types:

- **lists**: store elements sequentially. Access elements by index, add new elements to the list, determine if the list is empty
- **dictionaries**: store key-value pairs. Look up a value based on key, add/remove key-value pairs
- **files**: objects that you can open for reading in data or for writing data. Start reading from the beginning of a file, starting writing at the beginning of a file, skip to some part of a file.

As you saw in CSC108, you can do many, many useful things with lists and dictionaries - but this power is a double-edged sword. The more operations an interface offers, the harder it is to implement, and to use! This week, we're going to look at a much simpler ADT: the **Stack**.

Stack Redux

Recall from the exercise the following definition of a stack. Note that we've deliberately not included any implementation details, because that's not what's important when defining an ADT!

```
class Stack:

    def __init__(self):
        """ (Stack) -> NoneType
        Create a new empty stack.
        """

    def is_empty(self):
        """ (Stack) -> bool
        Return True if the stack has no items.
        """

    def push(self, item):
```

```
        """ (Stack, object) -> NoneType
        Add a new element to the top of the stack.
        """

    def pop(self):
        """ (Stack) -> object
        Remove and return the element at the top of the stack.
        Raise EmptyStackError is stack is empty.
        """
```

It might not seem like stacks are that powerful, but it's actually their simplicity that makes them easy to work with. You'll explore one interesting use of stacks on your first assignment, and the end of this section teases another use of stacks that's crucial to modern programming languages.

Size of a Stack

While stacks are used for storing data, and you can always tell when a stack has size 0, the interface doesn't include a method that tells you how many items are stored on the stack. Write a method that does this.

```
def size(stack):
    """ (Stack) -> int
    Return the number of elements stored in stack.
    """
```

Side note: mutation

Every class method you'll write in this course can be grouped into two categories: **non-mutating** and **mutating** methods. A method is **mutating** if it alters the internal state of an object, while **non-mutating** methods leave the object unchanged. For example, the `pop` and `push` methods are mutating, and the `is_empty` method is not.

Mutation is a fundamental concept in programming because one of the most frequent sources of errors for both beginners and experts alike is losing track of *how data changes* over the course of a program. Because of this, it is generally easier to reason about programs that do not use mutation; an entire style known **functional programming** holds as one of its central tenets that mutation should be used as little as possible.

Balanced parentheses (exercise)

Here's a classic interview question. A string of parentheses is *balanced* if every `'('` has a matching `')'` (and vice versa), and in every matching pair, the `'('` appears before the `')'`. For example, `'()(())'` and `'((()))'` are balanced, but `')('`, `'(()'`, and `'()()(()()'` are not.

Use a stack to write the following function.

```
def is_balanced(parens):
    """ (str) -> bool
    Return True if parens is a balanced string of parentheses.
    """
```

The Call Stack

Warning: I devoted more time in lecture to this topic than I had originally planned, so you are responsible for this material!

One of the most important concepts in computer science is the **function call stack**, which is the data structure used by the machine to keep track of function calls during the execution of a program. Consider the following simple example:

```
def f():
    x = g(1, 2)
    y = h()
    return x + y

def g(a, b):
    k(a)
    return a + b

def h():
    return 5

def k(a):
    return 16

>>> f()
8
```

The point of this example is not that `f()` returns `8` - you all could have told me that! - but to understand exactly what happens in between calling the function and getting the return value.

When `f` is called, the first thing that happens is it calls `g`, which in turn calls `k`. Then `k` returns, and `g` returns, and `h` gets called, etc. This is all consistent with your mental model of how Python programs work; what's really interesting is that this behaviour is exactly what a stack models!

The function call stack keeps track of all called functions that have yet to return, where the top of the stack is the function whose body is currently being executed. When a function is called, it gets **pushed to the top of the stack**, and it becomes the currently executing function. In the above example, first `f` gets pushed onto the stack, then `g`, and then `k`.

Now what happens when `k` completes its execution, and returns 10? Simple: it gets **popped off the stack**, and the next most recent function, `g`, returns to being the current execution function.

This notion of the call stack is more than just a metaphor: this literally is how machines keep track of information about function calls in memory at runtime! Of course, we're skipping several details here that are discussed much more thoroughly in future computer science courses. But do keep in mind that the function call stack is one of the most important concepts to understand about the execution of a program, and it relies on nothing more than our simple ADT.

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