CSC 148 Intro. to Computer Science

Lecture 4: Container implementation, Unit Test, Balanced Parentheses, Intro to Linked Lists

Amir H. Chinaei, Summer 2016

Office Hours: R 10-12 BA4222

ahchinaei@cs.toronto.edu
http://www.cs.toronto.edu/~ahchinaei/

Course page:
Review

❖ Last week
  - Composition and inheritance
  - Inheriting, extending, and overriding
  - Specific examples:
    - Shape: square, right angled triangle
    - Container: stack, sack

❖ Today
  - Container, Stack, and Sack implementation
  - Unit Test
  - Balanced Parenthesis
  - Introduction to linked lists
Recall

- Don't maintain documentation in two places, e.g. superclass and subclass, unless there's no other choice:
  - Inherited methods, attributes
    - no need to document again
  - extended methods
    - document that they are extended and how
  - overridden methods, attributes
    - document that they are overridden and how
**Stack/Sack definition**

- **Stack** contains items of various sorts. New items are added onto the top of the stack, items may only be removed from the top of the stack. It’s a LIFO structure.
- It's a mistake to try to remove an item from an empty stack, so we need to know if it is empty. We can tell how big a stack is.

- **Sack** contains items of various sorts. New items are added onto a random place in the sack, so the order items are removed from the sack is completely unpredictable.
- It's a mistake to try to remove an item from an empty sack, so we need to know if it is empty. We can tell how big a sack is.

Let’s revisit the API’s ….
Stack/Sack definition

- We noticed that there are several commonalities in the interface of a Stack and a Sack
  - i.e. the way a stack or sack is used by the client code
    - `s.__init__()`
    - `s.__str__()` e.g. `print(s)`
    - `s.__eq__()` e.g. `s == t`
    - `s.add()`
    - `s.remove()`
    - `s.is_empty()`

- so, we can abstract the commonalities in a higher level (super) class. Let’s name it Container
- and, develop the Container API ....
After developing the API, an important decision is

- which methods should be implemented, and
- which ones should be forced to be implemented by subclasses

```
s.__init__()
s.__str__()
s.__eq__()
s.add()
s.remove()
s.is_empty()
```

- What do you think? ....
A sample solution

- `__str__() is less subjective,`
- it can be implemented in Container
- Moreover,
- we chose to implement `__eq__() as well`
- we chose to force the implementation of the following methods to subclasses.
  
  ```python
  s.__init__()
  s.add()
  s.remove()
  s.is_empty()
  ```
- Note that these decisions depend on the project specification and our design goals
Testing

- We can use the command line to test if our newly developed data type (Stack, Sack, etc.) works the way we mean.

- Let’s do it ....

- Problems:
  - not organizing our tests
    - not being able to test large codes
  - not documenting our tests
    - not conforming with basic principles
  - not reusing our tests
    - not being able to do regression test
  - tedious to conduct independent tests
**unittest**

- A framework to setup test cases, run them independently from one another, document them, and reuse them when needed, …

- Extending `unittest.TestCase` is not essentially any different than extending any other class

- so, we develop a subclass:
  
  ```
  class myStackTestCase(unittest.TestCase):
  ```

- and override some special methods:
  ```
  setUp()
  tearDown()
  ```

- and follow some conventions:
  - `test???
  - `assert` statements
let’s see it in practice .....
A case study

- Let’s go back to the newly developed data types

- Balanced parentheses

- In some situations it is important that opening and closing parentheses match.

  - 12    good
  - (a5)  good
  - )a+b( bad
  - (ab(ca(d)ab))(d(a(b))cd(a)) good or bad?
Parenthesization

- Many computer programs (interpreters, compilers, calculators, etc.) need to evaluate such expressions

- Programs “see” one character at a time
(d(a(b)))cd(d(a))
(d(a(b))c d(a))
discussion .....
let’s move on to a new data type/structure
Motivation

- Regular Python lists are flexible and useful, but overkill in some situations:
  - they allocate large blocks of contiguous memory, which becomes increasingly difficult as memory is in use.

- Linked list nodes reserve just enough memory for the object value they want to refer to, a reference to it, and a reference to the next node in the list.
For now, we implement a linked list as objects (nodes) with a value and a reference to other similar objects.
class LinkedListNode:
    """
    Node to be used in linked list
    """

    === Public Attributes ===
    :param LinkedListNode next_: successor to this LinkedListNode
    :param object value: data this LinkedListNode represents

    def __init__(self, value, next_=None):
        """
        Create LinkedListNode self with data value and successor next_.
        """

        :param value: data of this linked list node
        :type value: object
        :param next_: successor to this LinkedListNode.
        :type next_: LinkedListNode|None
        :rtype: None

        self.value, self.next_ = value, next_
Next

- Midterm
- We continue with Linked List API and implementation