# **CSC148** *Intro. to Computer Science*

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

Amir H. Chinaei, Winter 2016

Office Hours: W 16:00–17:45 BA4222

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

*Course webpage:*
*http://www.cdf.toronto.edu/~csc148h/winter*



## 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

- A stack contains items of various sorts. New items are added on to 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.

- A sack contains items of various sorts. New items are added on to 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 ….



## Container

- 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.
  - 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 they 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:
  e.g. `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(a))



(d(a(b))cd(a))



discussion …..

- as Alfred mentioned: one solution is to use a counter c=0. If see a "(", c = c+1; if see a "), c = c-1; If at any time, c is negative, return False; also at the end, if c != 0, return False; otherwise, return True. Nice, but, not scalable to "{}", "[]", etc.
- as Jessie mentioned: we should ignore non-relevant characters: a, b, etc, …
- and, as Edi mentioned: we can use a stack s initially empty. If see a "(", add it to s; if see a ")", remove from s. If at any time, we are about to remove from and empty s, return False; also at the end, if s is not empty, return False; otherwise, return True. Nice, and scalable!



let's move on to a new data type/structure



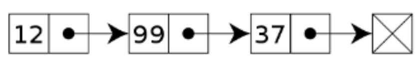
## Motivation

- 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 no de in the list



## Linked List

- For now, we implement a linked list as objects (nodes) with a value and a reference to other similar objects



## A Node class

```
class LinkedListNode:
    """
    Node to be used in linked list

    === 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 LinkedListNode self: this LinkedListNode
        @param object value: data of this linked list node
        @param LinkedListNode|None next_: successor to self
        @rtype: None
        """
        self.value, self.next_ = value, next_
```



## Next

- Midterm
- We continue with Linked List API and implementation