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 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.

- A 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
    - `__init__()`
    - `__str__()`: e.g. `print(s)`
    - `__eq__()`: e.g. `s == t`
    - `add()`
    - `remove()`
    - `is_empty()`
  - so, we can abstract the commonalities in a higher level (super) class. Let’s name it Container
  - and, develop the Container API …. ….
**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:
  - `__init__()`
  - `add()`
  - `remove()`
  - `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
  - Tediou 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

---

**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(cd(ab))(cd(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.
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.

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

```python
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

    __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