CSC148 L5102
Introduction to Computer Science
Joe Lim
joe.lim@utoronto.ca
http://www.cdf.toronto.edu/~csc148/winter/

Outline
- Documentation, Special Methods for Inheritance
- Abstract Data Types (ADT)
- Implement ADTs with classes, inheritance

Avoid Duplicating Documentation
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
- See shape.py and square.py

Pycharm type hinting, redux
- Type hinting is new in the Python world, and to get the benefit of Pycharm’s inspector, some fussing may be needed...
- @type doesn’t play well with text describing an attribute, so switch to @param

Special Methods for Shape
- Class Shape needs __str__, __eq__, and do all its subclasses
- Although we could override this in each subclass, a bit of research shows another way

New Lists from Old
Suppose L is a list of the first hundred natural numbers:

\[
L = \text{list}(\text{range}(100))
\]
If we want a new list with the squares of all the elements of L, we could

\[
\text{New list} = []
\]
For \(x\) in L:
    new_list.append(x*x)
OR using list comprehension

\[
\text{New list} = [x^2 \text{ for } x \text{ in } L]
\]
Filtering with [...]

We can make sure the new list only uses specific elements of the old list—
$L = ["one", "two", "three", "four", "five", "six"]$

By adding a condition . . .

```python
new_list = [s * 3 for s in L if s <= "one"]
```

Notice that a comprehension can span several line, if that makes it easier to understand.

General Comprehension Pattern

```
[expression for name in iterable if condition]
```

Python expressions evaluate to values, name refers to each element of iterable (list, tuple, dictionary,...) in turn, and a condition evaluates to either True or False

See Code like Pythonista

List Comprehension

- Construct lists like a mathematician do
  - $S = \{x^2 : x \in [0 \ldots 9]\}$
  - $V = \{1, 2, 4, 8, \ldots, 2^{12}\}$
  - $M = \{x | x \in S \text{ and } x \text{ even}\}$

Do this in Python

More Exercises

Given a list of temperatures in Celsius, convert it to Fahrenheit.

```python
celsius = [39.2, 36.5, 37.3, 37.8]
fahrenheit = ??
```

More Exercises...

Cross Product of two sets

```python
colours = ["red", "green", "yellow", "orange"]
things = ["house", "car", "pepper"]
coloured_things = ??
```

One more...

Find all the prime numbers less than or equal to 50 using the Sieve of Eratosthenes.

- Generate a list of integers from 2 to 50
- First number in the list is 2; cross out every 2nd number in the list after it by counting up from 2 in increments of 2.
- Next number in the list after 2 is 3; cross out every 3rd number in the list after it by counting up from 3 in increments of 3, ... etc.
Common ADTs
In CS we recycle our intuition about the outside world as ADTs. We abstract the data and operations, and suppress the implementation.
- Sequences of items; can be added, removed, accessed by position
- Specialized list where we only have access to most recently added item
- Collection of items accessed by their associated keys

Abstraction
- Separates the what from the how
- Provides modularity
- Classes are the Python representation for ADTs
- ADT involves both data and operations on that data.

Stack Example
Try the Python Visualizer.

The calls to first and second are stored on a stack that defies gravity by growing downward.

Implementation Possibilities
The public interface of our Stack ADT should be constant, but inside we could implement it in various ways.
- Use a python list, which already has a pop method and an append method
- Use a python list, but add and remove from position 0
- Use a python dictionary with integer keys 0, 1, ..., keeping track of the last index used, and which have been removed

Stack Abstract Data Type

<table>
<thead>
<tr>
<th>Stack Operation</th>
<th>Stack Contents</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.isEmpty</td>
<td>()</td>
<td>True</td>
</tr>
<tr>
<td>s.push (4)</td>
<td>[4]</td>
<td></td>
</tr>
<tr>
<td>s.push ('dog')</td>
<td>[4, 'dog']</td>
<td></td>
</tr>
<tr>
<td>s.peek</td>
<td>[4, 'dog']</td>
<td>'dog'</td>
</tr>
<tr>
<td>s.push (True)</td>
<td>[4, 'dog', True]</td>
<td></td>
</tr>
<tr>
<td>s.size</td>
<td>[4, 'dog', True]</td>
<td>3</td>
</tr>
<tr>
<td>s.isEmpty</td>
<td>[4, 'dog', True]</td>
<td>False</td>
</tr>
<tr>
<td>s.push (8.4)</td>
<td>[4, 'dog', True, 8.4]</td>
<td></td>
</tr>
<tr>
<td>s.pop</td>
<td>[4, 'dog', True]</td>
<td>8.4</td>
</tr>
<tr>
<td>s.pop</td>
<td>[4, 'dog']</td>
<td>True</td>
</tr>
<tr>
<td>s.size</td>
<td>[4, 'dog']</td>
<td>2</td>
</tr>
</tbody>
</table>
Queue

- Like stacks, queues are linear collections
- Insertions restricted to one end called the rear
- Removals restricted to the other end called the front
- Support First-in, First-out (FIFO) protocol
- Supports two fundamental operations:
  - Enqueue – Add item the rear of a queue
  - Dequeue – Removes item from the front

Queue ADTs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Queue Content</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>q.enqueue(t)</td>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>q.enqueue(b)</td>
<td>a b</td>
<td>b</td>
</tr>
<tr>
<td>q.enqueue(c)</td>
<td>a b c</td>
<td>c</td>
</tr>
<tr>
<td>q.isEmpty()</td>
<td>a b c</td>
<td>False</td>
</tr>
<tr>
<td>len(q)</td>
<td>a b c</td>
<td>3</td>
</tr>
<tr>
<td>q.dequeue()</td>
<td>a b c</td>
<td>a</td>
</tr>
<tr>
<td>q.dequeue()</td>
<td>a b c</td>
<td>b</td>
</tr>
<tr>
<td>q.dequeue()</td>
<td>a b c</td>
<td>c</td>
</tr>
<tr>
<td>q.peek()</td>
<td>a b c</td>
<td>a</td>
</tr>
<tr>
<td>q.peek()</td>
<td>a b c</td>
<td>b</td>
</tr>
<tr>
<td>q.peek()</td>
<td>a b c</td>
<td>c</td>
</tr>
<tr>
<td>q.dequeue()</td>
<td>a b c</td>
<td>Exception</td>
</tr>
<tr>
<td>q.enqueue(d)</td>
<td>a b c d</td>
<td>d</td>
</tr>
</tbody>
</table>

Bag ADT

Here’s a description of a sack, which has similar features to a stack:

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. Take a few minutes to identify the main noun, verb, and attributes of the main noun to guide our class design. Remember to be flexible about alternate names and designs for the same class.

Testing

Use your docstring for testing as you develop, but use unit testing to make sure that your particular implementation remains consistent with your ADTs interface. Be sure to:

- Import the module unittest
- subclass unittest.TestCase for your tests, and begin each method that carries out a test with the string test
- Compose tests before and during implementation

Python unittest

- Unit testing framework
- Unit – smallest testable part of a program
- doctest vs unittest
  - test case are not defined inside the module
  - program documentation and test descriptions are separate from each other
  - Increase work to create test cases