An Introduction to Object-Oriented Programming

Michael H. Goldwasser       David Letscher

March 26, 2009

The Object-Oriented Paradigm
Data and Behaviors

All programming languages provide means for
✓ storing data
✓ performing operations

Object-oriented languages support
a closer pairing of data and behaviors.

Each individual object has
➤ its own internal state (data representation)
➤ a set of supported behaviors (operations)

Classes help define objects with similar structure.

---

Terminology

An object is an instance of a class (spot is a Dog)

<table>
<thead>
<tr>
<th>Various synonyms</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
</tr>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>fields</td>
</tr>
<tr>
<td>data members</td>
</tr>
<tr>
<td>instance variables</td>
</tr>
<tr>
<td>state information</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>breed</td>
</tr>
<tr>
<td>weight</td>
</tr>
<tr>
<td>hairColor</td>
</tr>
<tr>
<td>tailLength</td>
</tr>
</tbody>
</table>
Object Interactions

**spot.fetch(slippers)**

![Diagram of object interactions]

Caller (jane) invokes method upon callee (spot)

**Accessors**: do not affect the callee’s state

**Mutators**: may affect the callee’s state

Case Study: Television

**Brainstorm:**

Let’s develop a model for a typical television

What state information is stored internally?

What outward behaviors are supported?

Parameters? Return values?
**Case Study: Television**

**volume** and **channel** are both attributes.
How are they treated differently in our design?

How does support of “mute” affect our design?

Advanced features?

---

**Television Design (SPOILER ALERT)**

<table>
<thead>
<tr>
<th>Television</th>
</tr>
</thead>
<tbody>
<tr>
<td>brand</td>
</tr>
<tr>
<td>model</td>
</tr>
<tr>
<td>powerOn</td>
</tr>
<tr>
<td>togglePower()</td>
</tr>
<tr>
<td>volume</td>
</tr>
<tr>
<td>muted</td>
</tr>
<tr>
<td>channel</td>
</tr>
<tr>
<td>prevChan</td>
</tr>
<tr>
<td>channelUp()</td>
</tr>
<tr>
<td>channelDown()</td>
</tr>
<tr>
<td>setChannel(number)</td>
</tr>
<tr>
<td>jumpPrevChan()</td>
</tr>
</tbody>
</table>

---
Encapsulation

Design the public interface for using an object … but treat implementation details as private!

If revisions break backward compatibility with the public interface, it can be a major ordeal! 😊

Inheritance

```
Person
  name
  birthdate
  phoneNumber
  curSchedule

Student
  advisor
  transcript
  declaredMajor

  getMajor()
  declareMajor(major)
  enroll(course)
  drop(course)
  getGrade(course)
  assignGrade(course, grade)
  hasTaken(course)
  checkGraduation()
  getGpa()

Professor
  department
  office
  officeHours

  getOffice()
  setOffice(room)
  getDepartment()
  getOfficeHours()
  addHours(time)
  removeHours(time)
  assignGrade(course, student, grade)
```

PyCon 2009  An Introduction to Object-Oriented Programming – 9 of 110

PyCon 2009  An Introduction to Object-Oriented Programming – 10 of 110
Using Objects

Python’s built-in classes

Everyone who has used Python is already familiar with the use of objects from various classes.

We will review some of these, in order to reinforce object-oriented principles.

- list class (mutable)
- tuple class (immutable)
- str class (immutable)
- int, long, bool classes (immutable)
Objects and Identifiers

```python
groceries = list()  # equivalently, groceries = []
guests = list()
guests.append('Chip')
groceries = guests
```

### Calling methods

**Components of the calling syntax:**

```
groceries.append('bread')
```

- **object**
- **method**
- **parameters**
Operator Syntax

Operators are really hidden method calls

<table>
<thead>
<tr>
<th>Call</th>
<th>Equivalent method call</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val in data</code></td>
<td>data.<em>contains</em>(val)</td>
</tr>
<tr>
<td><code>data[i]</code></td>
<td>data.<em>getitem</em>(i)</td>
</tr>
<tr>
<td><code>data[i] = val</code></td>
<td>data.<em>setitem</em>(i, val)</td>
</tr>
<tr>
<td><code>dataA + dataB</code></td>
<td>dataA.<em>add</em>(dataB)</td>
</tr>
<tr>
<td><code>dataA == dataB</code></td>
<td>dataA.<em>eq</em>(dataB)</td>
</tr>
</tbody>
</table>

Some built-in functions are hooks for method calls

<table>
<thead>
<tr>
<th>Call</th>
<th>Equivalent method call</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(data)</code></td>
<td>data.<em>len</em>( )</td>
</tr>
</tbody>
</table>

Accessors vs. Mutators

Accessors:

- `len(data)`
- `val in data`
- `data.count(val)`
- `data.index(val)`

Mutators:

- `data.append(val)`
- `data.insert(i, val)`
- `data.extend(sequence)`
- `data.remove(val)`
- `data.pop(i)`
- `data.sort()`
- `data.reverse()`
Python’s `str` class

- Immutable class
  
  Once an instance is created, its state cannot be changed

- This implies that every method is an accessor.
  
  Some methods generate a new string instance
  
  Example: `person.lower()`

Assignment Semantics

An assignment binds an identifier; it does not alter the state of any object.

Example: `person = person.lower()`

- `person.lower()` creates a new string instance that is a downcased version of the original
- The assignment binds `person` to the new string
List Iteration

```python
guests = ['Carol', 'Alice', 'Bob']
for person in guests:
    print 'Hello my name is', person
```

Hello my name is Carol
Hello my name is Alice
Hello my name is Bob

Defining Your Own Classes
**General Indentation Pattern**

```python
class Point:
    def __init__(self):
        body
    def getX(self):
        body
    def setX(self, val):
        body
```

**Constructor**

```python
class Point:
    def __init__(self, initialX=0, initialY=0):
        self._x = initialX
        self._y = initialY
```

- Internal name for the constructor is `__init__`
- “Keyword” `self` refers to the new instance. Explicit formal parameter, but not an actual parameter for caller.
- Qualified `self._x` is an instance variable
- Optional parameters allows initial state to be set or default to (0,0)
**S I P O I S F P O S I A I C I D U O**

---

**External vs. Internal View of self**

```python
def setX(self, val):
    self._x = val
```

```python
corner = Point()
corner.setX(side)
```

---

**Accessors and Mutators**

```python
def getX(self):
    return self._x
```

```python
def setX(self, val):
    self._x = val
```

- Provide means to check and set the value of the x coordinate (similar methods for y)
- Keeps the member variable _x "private" (although Python allows external access)
- Un-Pythonic (?)
More Interesting Behaviors

```python
def scale(self, factor):
    self._x *= factor
    self._y *= factor
```

```python
def distance(self, other):
    dx = self._x - other._x
    dy = self._y - other._y
    return sqrt(dx*dx + dy*dy)
```

- Local variables (e.g., dx) are unqualified
- Can access member data of another instance using a qualified syntax

Improving our class

```python
def normalize(self):
    mag = self.distance(Point( ))
    if mag > 0:
        self.scale(1/mag)
```

- Calls `self.distance` and `self.scale` are invoked on the current instance.
- Call to `Point()` constructs an unnamed instance
Special Methods

By default, Python does not know how to display an instance of a user-defined class.

```python
>>> topLeft = Point(5,20)
>>> print topLeft
<Point.Point instance at 0x386968>
```

Precise format depends on system, and also whether this is an “old-style” or “new-style” class.

Displayed string format designated with `__str__` method

```python
def __str__(self):
    return '<' + str(self._x) + ', ' + str(self._y) + '>
```

```python
>>> topLeft = Point(5,20)
>>> print topLeft
<5, 20>
>>> topLeft
<_main_.Point instance at 0x386968>
```

To get the interpreter to display the point in the second case define the `__repr__` method.
Operator Overloading

```python
>>> bottomRight = Point(45,60)
>>> offset = Point(5,5)
>>> corner = bottomRight + offset
TypeError: unsupported operand types for +
```

Syntax `bottomRight + offset` is evaluated internally as `bottomRight.__add__(offset)`

Our implementation:

```python
def __add__(self, other):
    return Point(self._x + other._x, self._y + other._y)
```

Polymorphism

An operation’s ability to perform a behavior differently depending on the context is called **polymorphism**.

```python
def __mul__(self, operand):
    if isinstance(operand, (int, float)): # Scalar product
        return Point(self._x * operand, self._y * operand)
    elif isinstance(operand, Point): # Dot product
        return Point(self._x * operand._x + self._y * operand._y)
```

Use `isinstance` to perform runtime type-checking
For additional practice, we implement a `Television` class based on our earlier high-level design.

```python
class Television:
    def __init__(self):
        self._powerOn = False
        self._muted = False
        self._volume = 5
        self._channel = 2
        self._prevChan = 2
```

```python
def togglePower(self):
    self._powerOn = not self._powerOn

def toggleMute(self):
    if self._powerOn:
        self._muted = not self._muted
```

Notice that the power button always has effect, but the mute button only works when power is on.
Television Class

```python
def volumeUp(self):
    if self._powerOn:
        if self._volume < 10:
            self._volume += 1
        self._muted = False
    return self._volume

def channelUp(self):
    if self._powerOn:
        self._prevChan = self._channel
        if self._channel == 99:
            self._channel = 2
        else:
            self._channel += 1
    return self._channel

def setChannel(self, number):
    if self._powerOn:
        if 2 <= number <= 99:
            self._prevChan = self._channel
            self._channel = number
    return self._channel

def jumpPrevChannel(self):
    if self._powerOn:
        incoming = self._channel
        self._channel = self._prevChan
        self._prevChan = incoming
    return self._channel
```
Class-level Attributes

class Television:
    # class-level attributes will be shared by all
    _minVolume = 0
    _maxVolume = 10
    _minChannel = 2
    _maxChannel = 99

def __init__(self):
    self._powerOn = False
    self._muted = False
    self._volume = (Television._minVolume + Television._maxVolume) // 2
    self._channel = Television._minChannel
    self._prevChan = Television._minChannel

Fraction Class

As an example of an immutable class, we develop a Fraction class.

class Fraction:
    def __init__(self, numerator=0, denominator=1):
        if denominator == 0:  # fraction is undefined
            self._numerator = 0
            self._denominator = 0
        else:
            factor = gcd(abs(numerator), abs(denominator))
            if denominator < 0:  # want to divide through
                factor = -factor  # by negated factor
            self._numerator = numerator // factor
            self._denominator = denominator // factor
### Arithmetic Methods

```python
def __add__(self, other):
    return Fraction(self.numer * other.denom + self.denom * other.numer, 
                    self.denom * other.denom)

def __sub__(self, other):
    return Fraction(self.numer * other.denom - self.denom * other.numer, 
                    self.denom * other.denom)

def __mul__(self, other):
    return Fraction(self.numer * other.numer, self.denom * other.denom)

def __div__(self, other):
    return Fraction(self.numer * other.denom, self.denom * other.numer)
```

### Comparison Methods

```python
def __lt__(self, other):
    return self.numer * other.denom < self.denom * other.numer

def __eq__(self, other):
    return self.numer == other.numer and self.denom == other.denom
```

### Type Conversion Methods

```python
def __float__(self):
    return float(self.numer) / self.denom

def __int__(self):
    return int(float(self))  # convert to float, then truncate
Good Software Practices

Mastermind Game

PyCon 2009
General principles:

- **Top-down design**
- **Bottom-up implementation**
- **Modularity**

Note: initial version will have a text interface.
### Top Level Design

#### Mastermind

- `Mastermind(inputManager, outputManager)`

#### Score

- `Score(numBlack, numWhite)`
- `getNumBlack()`
- `getNumWhite()`

#### Pattern

- `Pattern(numPegs)`
- `__len__()`
- `getPegColor(index)`
- `setPegColor(index, colorID)`
- `compareTo(otherPattern)`
- `randomize(numColors)`

#### TextInput

- `TextInput(colorNames)`
- `queryLengthOfPattern()`
- `queryNumberOfColors()`
- `queryNumberOfTurns()`
- `queryNewGame()`
- `enterGuess()`

#### TextOutput

- `TextOutput(colorNames)`
- `startGame(lengthOfPattern, maxNumberOfTurns)`
- `displayTurn(guess, result)`
- `announceVictory(secret)`
- `announceDefeat(secret)`

---

### Naming Conventions

**Pattern** # class name

**TextOutput** # class name (camel–case)

**guess** # instance of Pattern class

**announceVictory** # method name (verb)

---

Our conventions vary from Python distribution. Some odd naming within Python libraries:

- `time.time()` # current time
- `random.random()` # randomly generated float
- `random.Random()` # instance of Random class
class Score:
    """A score for a single turn from game of Mastermind.
    
    A "black" component designates the number of pegs that are
    exact matches for the answer. A "white" component counts
    pegs that are correctly colored but not well positioned.
    """
    def __init__(self, numBlack, numWhite):
        """Create score with given black and white components."""
        self.numBlack = numBlack
        self.numWhite = numWhite

    def getNumBlack(self):
        """Return the black component of the score."""
        return self.numBlack

    def getNumWhite(self):
        """Return the white component of the score."""
        return self.numWhite
Encapsulation

“Private” vs. “Public” in Python

- single-underscored names (latent support for privacy)
- double-underscored names (name-mangling)
- getters/setters vs. properties

Unit Testing

- Test individual classes before moving to a higher level of implementation

```python
class Pattern:
    # implementation here

if __name__ == '__main__':
    # unit test here
```

- Can use the `doctest` module for regression testing.
Inheritance

Augmentation

Inheritance can be used to add new behaviors.

For example, `class DeluxeTV(Television) that stores a list of favorite channels.`

New methods:
- `addToFavorites`
- `removeFromFavorites`
- `jumpToFavorite`
from Television import Television

class DeluxeTV(Television):
    """A television that maintains a set of favorite channels."""

def __init__(self):
    """Creates a new DeluxeTV instance.

    The set of favorite channels is initially empty."

    Television.__init__(self)           # parent constructor
    self._favorites = []

    def addToFavorites(self):
        """Adds the current channel to the list of favorites, if not present.

        If power is off, there is no effect."

        if self._powerOn and self._channel not in self._favorites:
            self._favorites.append(self._channel)

    def removeFromFavorites(self):
        """Removes the current channel from the list of favorites, if present.

        If power is off, there is no effect."

        if self._powerOn and self._channel in self._favorites:
            self._favorites.remove(self._channel)
Specialization

Can override inherited method to alter behavior.

class SortedSet(list):
    '''Stores elements in increasing order.'''

Many methods can be used without change:
    __contains__, __getitem__, __len__,
    __eq__, index, pop, remove

But others will not work as desired. For example, the list.insert method allows user to place element in arbitrary position.

SortedSet Class

```python
def __init__(self, initial=None):
    list.__init__(self)          # parent constructor
    if initial:
        self.extend(initial)

def indexAfter(self, value):
    walk = 0
    while walk < len(self) and value >= self[walk]:
        walk += 1
    return walk

def insert(self, value):
    if value not in self:      # new signature
        place = self.indexAfter(value)
    list.insert(self, place, value) # avoid duplicates
```

PyCon 2009
We want other methods work slightly differently but with same interface.

```python
def append(self, object):
    self.insert(object)  # our version!

def extend(self, other):
    for element in other:
        self.insert(element)  # our version!
```

There is no need to do any sorting in the `sort` method and reversing should not be allowed.

```python
def sort(self):
    pass  # it's already sorted

def reverse(self):
    raise RuntimeError('SortedSet cannot be reversed')

def __setitem__(self, index, object):
    raise RuntimeError('not supported by SortedSet')
```
Multiple Inheritance

Can be useful ...

but added complexity.

To Inherit or Not To Inherit

?
Intermission

A Deeper Understanding of the Management of Objects
Deeper Understanding

Important to have knowledge of core issues:

- Objects and identifiers
- Aliasing
- Objects that reference other objects
- Information passing
- Containers

Instantiation

```python
>>> mySavings = Account()
>>> myChecking = Account()
>>> id(mySavings)
402656
>>> id(myChecking)
403336
```

Deeper Understanding

Object Orientation
Using Objects
Defining Classes
Software Practices
Inheritance
Intermission
Object Management
Instantiation
Aliasing
Equivalence Testing
Garbage Collection
Referencing Objects
Deeper Aliasing
Copying Objects
The copy module
Information Passing
Default Parameters
Python’s Internals
Structural Recursion

Deeper Understanding

Object Orientation
Using Objects
Defining Classes
Software Practices
Inheritance
Intermission
Object Management
Instantiation
Aliasing
Equivalence Testing
Garbage Collection
Referencing Objects
Deeper Aliasing
Copying Objects
The copy module
Information Passing
Default Parameters
Python’s Internals
Structural Recursion
>>> myDebit = myChecking

>>> id(mySavings)
402656

>>> id(myChecking)
403336

>>> id(myDebit)
403336

>>> mySavings.deposit(100.0)

>>> myChecking.deposit(300.0)

>>> myDebit.withdraw(50.0)

>>>
Equivalence Testing (is vs. ==)

Equivalence Testing (is vs. ==)

>>> mySavings is myChecking
False
>>> myDebit is myChecking
True
>>> myChecking is myChecking
True
>>> myChecking is not mySavings
True

Equivalence Testing (is vs. ==)

>>> myDebit == myChecking
True
>>> mySavings == myChecking
False
>>> mySavings.deposit(150)
>>> mySavings == myChecking
False
>>> mySavings is myChecking
False
Objects that Reference Other Objects

Let’s revise our portrayal once again.

Bits for value 0.0 are not stored within the account. **_balance** is a reference to a float instance.
A **list** is a list of references.
**Tuples are Immutable (but contents ...)**

```python
defrozenset = (mySavings, myChecking)
defrozenset[0].withdraw(100) #not really frozen
```

**Deeper Forms of Aliasing**

Model: a couple that shares a portfolio.
Deeper Forms of Aliasing

Result of `spouseAssets.append(spouseRetirement)`

Model: separate portfolios (but shared accounts)
Deeper Forms of Aliasing

Result of `spouseAssets.append(spouseRetirement)`

[Diagram showing object relationships]

Copying Objects

A “shallow” copy of a `DeluxeTV`. What’s wrong with this picture?

[Diagram showing object relationships]
The copy module

- `copy.copy(x)`
  produces a shallow copy of object x
- `copy.deepcopy(x)`
  produces a deep copy of object x

Information Passing

```python
def verifyBalance(account, threshold):
    meetsThreshold = account.getBalance() >= threshold
    return meetsThreshold
```

```
mySavings = Account()
claimedAssets = 200.0
qualifies = verifyBalance(mySavings, claimedAssets)
```
Default Parameters

What’s wrong with the following constructor?

```
class DeluxeTV:
    def __init__(self, favorites = []):
        Television.__init__(self)
        self._favorites = favorites
```

Default Parameters

What’s wrong with the following method declaration for a list class?

```
def pop(self, index = len(self) - 1):
```

Solution?
Default Parameters

What’s wrong with the following method declaration for a list class?

```python
def pop(self, index = len(self) - 1):
```

Solution?

```python
def pop(self, index = None):
    if index is None:
        index = len(self) - 1
```

Python’s Internal Mechanisms
Name Resolution

- Dictionary based: key = identifier, value = value of corresponding object
- Note: everything is an object
- Several levels of **namespaces**: global, local, ...
- Name resolution searches for the identifier until it finds the name and retrieves its value.
- If identifier is not found a NameError or AttributeError occurs.

Global Namespace

```python
>>> print globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__doc__': None}
```

1. Stores all identifiers introduced in the top-level scope, including global variables, functions and classes.
2. __name__ having the value main indicates that the interpreter was started without loading a source file.
Global Namespace

```python
processor = 8086

def jam():
    print 'strawberry'

class room:
    def __init__(self):
        self._capacity = 30

locals()  
>>> demo(bon giorno)
Local dictionary is {'x': 10, 'param': 'bon giorno'}
```

Local Namespace

Stores identifiers introduced in a function.

```python
def demo(param = 'hello'):
    """An example of a function."""
    x = len(param)
    print 'Local dictionary is', locals()

>>> demo('bon giorno')
Local dictionary is {'x': 10, 'param': 'bon giorno'}
```
State of each class stored in an internal dictionary

```python
>>> myTv = Television()
>>> print vars(myTv)
{'_prevChan': 2, '_volume': 5, '_channel': 2,
'_powerOn': False, '_muted': False}
>>> print vars(Television)
{'_minChannel': 2, '_maxVolume': 10,
'_volumeDown': <function volumeDown at 0x7f4de04878c0,
'_volumeUp': <function volumeUp at 0x7f4de0487848>,
... }
```

- Name resolution for member data or member function starts with the instance-level dictionary.
- If identifier is not found it proceeds to class-level dictionary.
- If the class is derived from another, then it checks the class-level dictionary of each parent class.
### Modules

- Global dictionary stores all loaded modules, include the special module `__builtins__`.
- When searching at global scope it checks this dictionary if identifier is not found.

```python
>>> import math
>>> print globals()
{'__builtins__': <module '__builtin__' (built-in)>,
 '__name__': '__main__', '__doc__': None,
 'math': <module 'math' from '/usr/lib/python2.5/lib-dynload/math.so'>}
```

Use of qualified name accesses correct module, e.g. `math.sqrt(5)`

```python
>>> from math import sqrt, pi
>>> print globals()
{'__builtins__': <module '__builtin__' (built-in)>,
 '__name__': '__main__', 'pi': 3.1415926535897931,
 '__doc__': None, 'sqrt': <built-in function sqrt>}
```
Structural Recursion

Emulating Python’s List Class

What if Python’s list class did not exist?

We can implement our own version, emulating behaviors such as:

- count(value)
- index(value)
- append(value)
- insert(index, value)
- remove(value)
- __len__( )
- __contains__(value)
- __getitem__(index)
- __setitem__(index, value)
- __repr__( )
Recursive Design

We use recursion to implement OurList class. An instance is represented recursively using two attributes:

- **.head**: a reference to the first element (if any)
- **.rest**: a reference to a secondary list with all remaining elements (if any)

Our base case is an **empty list**, represented with both **.head** and **.rest** set to **None**.

**Disclaimer:** Python’s built-in list is not recursive; its an expandable array of references.

The Constructor

class OurList:

```python
def __init__(self):
    self._head = None
    self._rest = None

def _isEmpty(self):  # a private utility
    return self._rest is None
```

**Note:** This version always constructs an empty list. Our complete version allows for arbitrary sequence.
The **count** Method

A typical sequence diagram:

![Sequence Diagram](attachment:image.png)

Local View

Coding intuition is based on a local view:
The **count** method

Has a base case and a non-trivial recursion

```python
def count(self, value):
    if self._isEmpty():
        return 0  # we do not have any
    else:
        answer = self._rest.count(value)
        if self._head == value:
            answer += 1  # an additional match
    return answer
```

The **__contains__** method

val in data  \(\implies\)  data. **__contains__**(val)
This differs from `count` because the recursion does not necessarily proceed to an empty list.
Implementation

Two distinct base cases

def __contains__(self, value):
    if self._isEmpty():
        return False
    elif self._head == value:
        return True
    else:
        return value in self._rest  # recurse

The __getitem__ method

data[2] is evaluated as data.__getitem__(2)

Note: the parameter value changes during the recursion; the return value does not change.
Implementing **`__getitem__`**

```python
def __getitem__(self, i):
    if self._isEmpty():
        raise IndexError('index out of range')
    elif i == 0:
        return self._head
    else:
        return self._rest._getitem_(i−1)
```

**Error handling**

```python
def __getitem__(self, i):
    if self._isEmpty():
        raise IndexError('index out of range')
    elif i == 0:
        return self._head
    else:
        try:
            return self._rest._getitem_(i−1)
        except IndexError:
            raise IndexError('index out of range')
```
The index method

def index(self, value):
    if self.isEmpty():
        raise ValueError('x not in list')
    elif self.head == value:
        return 0
    else:  # look in remainder of the list
        return 1 + self.rest.index(value)

Mutators: append

PyCon 2009 An Introduction to Object-Oriented Programming – 106 of 110
The **append** method

Has a base case and a simple recursion

```python
def append(self, value):
    if self._isEmpty():
        self._head = value  # we have one item
        self._rest = OurList()  # followed by empty list
    else:
        self._rest.append(value)  # recurse
```

The **insert** method

```python
def insert(self, index, value):
    if self._isEmpty():  # "append" to end
        self._head = value
        self._rest = OurList()
    elif index > 0:  # insert recursively
        self._rest.insert(index - 1, value)
    else:  # new item goes here!
        shift = OurList()
        shift._head = self._head
        shift._rest = self._rest
        self._head = value
        self._rest = shift
```
The remove method

def remove(self, value):
    if self._isEmpty():
        raise ValueError('value not in list')
    elif self._head == value:
        self._head = self._rest._head # private
        self._rest = self._rest._rest # private
    else:
        self._rest.remove(value)

Default parameters

def pop(self, index=None):
    if self._isEmpty():
        raise IndexError('pop from empty list')
    else:
        if index is None:
            index = len(self) - 1
        if index == 0:
            answer = self._head
            self._head = self._rest._head
            self._rest = self._rest._rest
            return answer
        else:
            return self._rest.pop(index - 1)