Teaching Strategies for Reinforcing Structural Recursion with Lists

Michael H. Goldwasser       David Letscher

Saint Louis University
I need some **volunteers** for today
For an object-oriented CS1, structural recursion can be more natural than functional recursion.

An object is composed of a basic shape and a (recursive) instance of the same class.
Pro: Graphics are fun and tangible.
Pro: Graphics are fun and tangible.

Con: recursive patterns are generally limited ("draw outer, draw rest"; "move outer, move rest")
Pro: Graphics are fun and tangible.

Con: recursive patterns are generally limited ("draw outer, draw rest"; "move outer, move rest")

Our goal is to provide a tangible presentation for a non-graphical example of structural recursion (namely purely-recursive lists).
Python supports a list class as a standard container.
Python supports a `list` class as a standard container. **Disclaimer**: the internal implementation is not truly recursive; it's an expandible array akin to Java’s `ArrayList` or C++’s `vector`.
Python supports a list class as a standard container. **Disclaimer:** the internal implementation is not truly recursive; its an expandible array akin to Java’s ArrayList or C++’s vector.

**Public Interface:** Our students are very familiar with use of this class and its menu of behaviors (we use lists from the opening weeks of CS1).
Python supports a list class as a standard container. **Disclaimer:** the internal implementation is not truly recursive; its an expandible array akin to Java’s ArrayList or C++’s vector.

**Public Interface:** Our students are very familiar with use of this class and its menu of behaviors (we use lists from the opening weeks of CS1). This allows us to decouple two potentially intertwined concepts:

1. the use of recursion
2. the abstraction of a container class
We rely on the familiar public interface by **precisely emulating** Python’s list class, including behaviors such as:

- `count(value)`
- `index(value)`
- `append(value)`
- `insert(index, value)`
- `remove(value)`
- `_len_()`
- `contains(value)`
- `getitem(index)`
- `setitem(index, value)`
- `repr()`

This allows us to sidestep the design issue of parameterizing the recursion.
Role Playing

Overview

Role Playing
Ground Rules
State Information
Message Passing
Point of View
Sequence Diagram
Local View

Variants

Mutators

Implementation

Conclusions
Role Playing

Classic activity for teaching object orientation.
Classic activity for teaching object orientation.

Classic activity for teaching (functional) recursion.
Role Playing

Classic activity for teaching object orientation.

Classic activity for teaching (functional) recursion.

Limited history for the combination of these ideas.
OurList Class: an instance will be 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 the None reference.

An empty list is a natural concept for our students because Python’s default list instance is empty.
Each actor is given a slip of paper that represents his/her state information.
Each actor is given a slip of paper that represents his/her state information.

Example: here is the slip currently held by Sharon.

<table>
<thead>
<tr>
<th>Sharon : OurList</th>
</tr>
</thead>
<tbody>
<tr>
<td>_head: 'E'</td>
</tr>
<tr>
<td>_rest: Per</td>
</tr>
</tbody>
</table>
Each actor is given a slip of paper that represents his/her state information.

<table>
<thead>
<tr>
<th>Matthew : OurList</th>
</tr>
</thead>
<tbody>
<tr>
<td>_head: None</td>
</tr>
<tr>
<td>_rest: None</td>
</tr>
</tbody>
</table>

Example: here is the slip currently held by Matthew.
We enforce strict “message passing” for all communication.

Activation records are sent inside a tennis ball.

<table>
<thead>
<tr>
<th>ACTIVATION RECORD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sent to:</td>
</tr>
<tr>
<td>Method:</td>
</tr>
<tr>
<td>Parameters (if any):</td>
</tr>
<tr>
<td>Please return to:</td>
</tr>
<tr>
<td>Return Value (if any):</td>
</tr>
</tbody>
</table>
We enforce strict “message passing” for all communication.

Activation records are sent inside a tennis ball.

<table>
<thead>
<tr>
<th>ACTIVATION RECORD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sent to:</td>
</tr>
<tr>
<td>Method:</td>
</tr>
<tr>
<td>Parameters (if any)</td>
</tr>
<tr>
<td>Please return to:</td>
</tr>
<tr>
<td>Return Value (if any)</td>
</tr>
</tbody>
</table>

Let’s get started with a call Errol .count('E')
# Errol’s Point of View

<table>
<thead>
<tr>
<th>Errol : OurList</th>
</tr>
</thead>
<tbody>
<tr>
<td>_head: 'F'</td>
</tr>
<tr>
<td>_rest: Sharon</td>
</tr>
</tbody>
</table>

## ACTIVATION RECORD

<table>
<thead>
<tr>
<th>Sent to:</th>
<th>Errol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method:</td>
<td>count</td>
</tr>
<tr>
<td>Parameters (if any):</td>
<td>'E'</td>
</tr>
<tr>
<td>Please return to:</td>
<td>Michael</td>
</tr>
<tr>
<td>Return Value (if any):</td>
<td></td>
</tr>
</tbody>
</table>
Errol ’s Point of View

Errol : OurList

ACTIVATION RECORD

| Sent to:   | Errol |
| Method:    | count |
| Parameters (if any): | 'E' |
| Please return to: | Michael |
| Return Value (if any): | |

ACTIVATION RECORD

| Sent to:   | Sharon |
| Method:    | count |
| Parameters (if any): | 'E' |
| Please return to: | Errol |
| Return Value (if any): | |
Sharon 's Point of View

Sharon : OurList

_ head: 'E'
_rest: Per

ACTIVATION RECORD

<table>
<thead>
<tr>
<th>Sent to:</th>
<th>Sharon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method:</td>
<td>count</td>
</tr>
<tr>
<td>Parameters (if any):</td>
<td>'E'</td>
</tr>
<tr>
<td>Please return to:</td>
<td>Errol</td>
</tr>
<tr>
<td>Return Value (if any):</td>
<td></td>
</tr>
</tbody>
</table>
Sharon ’s Point of View

Sharon : OurList

ACTIVATION RECORD

Sent to: Sharon
Method: count
Parameters (if any): 'E'
Please return to: Errol
Return Value (if any):

ACTIVATION RECORD

Sent to: Per
Method: count
Parameters (if any): 'E'
Please return to: Sharon
Return Value (if any):
Sharon’s Point of View

Sharon : OurList

ACTIVATION RECORD

Sent to: Sharon
Method: count
Parameters (if any): 'E'
Please return to: Errol
Return Value (if any): 

ACTIVATION RECORD

Sent to: Per
Method: count
Parameters (if any): 'E'
Please return to: Sharon
Return Value (if any): 1
Errol ’s Point of View

Errol : OurList

| _head: 'F' |
| _rest: Sharon |

ACTIVATION RECORD

<table>
<thead>
<tr>
<th>Sent to:</th>
<th>Errol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method:</td>
<td>count</td>
</tr>
<tr>
<td>Parameters (if any):</td>
<td>'E'</td>
</tr>
<tr>
<td>Please return to:</td>
<td>Michael</td>
</tr>
</tbody>
</table>

Return Value (if any): 

ACTIVATION RECORD

<table>
<thead>
<tr>
<th>Sent to:</th>
<th>Sharon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method:</td>
<td>count</td>
</tr>
<tr>
<td>Parameters (if any):</td>
<td>'E'</td>
</tr>
<tr>
<td>Please return to:</td>
<td>Errol</td>
</tr>
</tbody>
</table>

Return Value (if any): 2
Errol’s Point of View

Overview
Role Playing
Ground Rules
State Information
Message Passing
Point of View
Sequence Diagram
Local View
Variants
Mutators
Implementation
Conclusions

ACTIVATION RECORD

Sent to: Errol
Method: count
Parameters (if any): 'E'
Please return to: Michael
Return Value (if any): 2

ACTIVATION RECORD

Sent to: Sharon
Method: count
Parameters (if any): 'E'
Please return to: Errol
Return Value (if any): 2

Errol: OurList

_head: 'F'
_rest: Sharon
This is a sequence diagram. It shows the interactions and state changes among different entities. The entities are represented as rectangular boxes, and the arrows indicate the direction of message passing. The state information for each entity is shown within the box, typically indicating the head and rest of a list or the current state of a variable.
Sequence Diagram

Overview
Role Playing
Role Playing
Ground Rules
State Information
Message Passing
Point of View

Sequence Diagram
Local View

Variants
Mutators
Implementation
Conclusions

2007 OOPSLA Educators’ Symposium

Reinforcing Structural Recursion with Lists – 15 of 40
Sequence Diagram

- Errol : OurList
  - head: 'F'
  - rest: Sharon

- Sharon : OurList
  - head: 'E'
  - rest: Per

- Per : OurList
  - head: 'T'
  - rest: Dale

- Dale : OurList
  - head: 'E'
  - rest: Matthew

- Matthew : OurList
  - head: None
  - rest: None

count('E') -> count('E') -> count('E') -> count('E')
OurList
Matthew/B3 E
Dale
Per
head: rest:
head: rest:
head: rest:
head: rest:
head: rest:

count('E') count('E') count('E') count('E') count('E')

head: rest:
head: rest:
head: rest:
head: rest:
head: rest:

Errol : OurList
_Shrest: 'F'
_LIST: Sharon

Sharon : OurList
_Shrest: 'E'
_LIST: Per

Per : OurList
_Shrest: 'T'
_LIST: Dale

Dale : OurList
_Shrest: 'E'
_LIST: Matthew

Matthew : OurList
_Shrest: None
_LIST: None

2007 OOPSLA Educators’ Symposium
Reinforcing Structural Recursion with Lists – 15 of 40
OurList
Matthew
/B3
E
OurList
Dale
/T
/B3
Per
head:
rest:
OurList
Sharon
/F
/B3
Errol
head:
rest:

head:
rest:
count('E')
count('E')
count('E')
count('E')
count('E')
0
Sequence Diagram

Overview
Role Playing
Role Playing
Ground Rules
State Information
Message Passing
Point of View
Sequence Diagram
Local View
Variants
Mutators
Implementation
Conclusions

2007 OOPSLA Educators’ Symposium
Reinforcing Structural Recursion with Lists – 15 of 40
Sequence Diagram

Overview
Role Playing
Ground Rules
State Information
Message Passing
Point of View

Sequence Diagram
Local View
Variants
Mutators
Implementation

Conclusions

2007 OOPSLA Educators’ Symposium Reinforcing Structural Recursion with Lists – 15 of 40
OurList = Per

Sharon : OurList

_count_E_ = count('E')
_count_E_ = count('E')

head:

rest:

2

1
Variants

Overview
Role Playing
Variants
The index method
The __getitem__ method
Recursive Patterns
Mutators
Implementation
Conclusions
The index method

**Overview**

Role Playing

**Variants**

The index method

Recursive Patterns

Mutators

Implementation

Conclusions
The **index method**

Overview
Role Playing
Variants
The index method
Recursive Patterns
Mutators
Implementation
Conclusions

```
Errol: OurList
  _head: 'F'
  _rest: Sharon

index(T)

2007 OOPSLA Educators’ Symposium
Reinforcing Structural Recursion with Lists – 18 of 40
```
The index method

Overview
Role Playing
Variants
Recursive Patterns
Mutators
Implementation
Conclusions

The index method
The _getitem_ method

Errol : OurList
_head: 'F'
_rest: Sharon

index('T')

1

index('T')
The index method

Overview
Role Playing
Variants
The index method
Recursive Patterns
Mutators
Implementation
Conclusions

Errol : OurList
_head: 'F'
_rest: Sharon

index('T')

2

1

index('T')
The `index` method

- Variants
  - The `__getitem__` method

Recursive Patterns

Mutators

Implementation

Conclusions
The **index method**

```
Per : OurList
  _head: 'T'
  _rest: Dale
```

```
index('T')
```

```
0
```
The index method

Overview
Role Playing
Variants
The index method
The __getitem__ method
Recursive Patterns
Mutators
Implementation
Conclusions
The index method

Overview
Role Playing
Variants
Recursive Patterns
Mutators
Implementation
Conclusions

index('T')

Errol : OurList
head: 'F'
rest: Sharon

Sharon : OurList
head: 'E'
rest: Per

Per : OurList
head: 'T'
rest: Dale

Dale : OurList
head: 'E'
rest: Matthew

Matthew : OurList
head: None
rest: None

index('T')
### The index method

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>_head: 'F'</td>
<td>_head: 'E'</td>
<td>_head: 'T'</td>
<td>_head: 'E'</td>
<td>_head: None</td>
</tr>
<tr>
<td>_rest: Sharon</td>
<td>_rest: Per</td>
<td>_rest: Dale</td>
<td>_rest: Matthew</td>
<td>_rest: None</td>
</tr>
</tbody>
</table>

index('T')

- index('T')
- index('T')
- index('T')
The index method

Overview
Role Playing
Variants
The index method
Recursive Patterns
Mutators
Implementation
Conclusions

index('T')

index('T')

index('T')

0

index('T')
The index method

Overview
Role Playing
Variants
The index method
Recursive Patterns
Mutators
Implementation
Conclusions

OurList
Errol : OurList
head: 'F'
rest: Sharon
index('T')
index('T')

index('T') 0

Sharon : OurList
head: 'E'
rest: Per

Per : OurList
head: 'T'
rest: Dale

Dale : OurList
head: 'E'
rest: Matthew

Matthew : OurList
head: None
rest: None

2007 OOPSLA Educators’ Symposium
Reinforcing Structural Recursion with Lists – 20 of 40
The **index** method

**Overview**

**Role Playing**

**Variants**

**Recursive Patterns**

**Mutators**

**Implementation**

**Conclusions**

---

The index method

```
Errol : OurList
  _head: 'F'
  _rest: Sharon

Sharon : OurList
  _head: 'E'
  _rest: Per

Per : OurList
  _head: 'T'
  _rest: Dale

Dale : OurList
  _head: 'E'
  _rest: Matthew

Matthew : OurList
  _head: None
  _rest: None
```

```
index('T')
```

```
1
```

```
0
```

```
2
```

---

2007 OOPSLA Educators’ Symposium

Reinforcing Structural Recursion with Lists – 20 of 40
The index method

This differs from `count` because the recursion does not necessarily proceed to an empty list.
In Python, the operator syntax `data[2]` is implemented with a call to `data.__getitem__(2)`.

```
Errol : OurList
    _head: 'F'
    _rest: Sharon

__getitem__(2)
```

Overview
Role Playing
Variants
The index method
The `__getitem__` method
Recursive Patterns
Mutators
Implementation
Conclusions
In Python, the operator syntax `data[2]` is implemented with a call to `data._getitem__(2).`
In Python, the operator syntax `data[2]` is implemented with a call to `data.__getitem__(2)`.
In Python, the operator syntax `data[2]` is implemented with a call to `data.__getitem__(2)`.

```
Errol : OurList
  _head: 'F'
  _rest: Sharon
```

```
  __getitem__(2)
  __getitem__(1)
  'T'
```

```
  'T'
```
The `__getitem__` method

In Python, the operator syntax `data[2]` is implemented with a call to `data.__getitem__(2)`. Note: the parameter value changes during the recursion; the return value does not change.
## Recursive Patterns

<table>
<thead>
<tr>
<th>method</th>
<th>base case</th>
<th>parameters</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>empty</td>
<td>head</td>
<td>index</td>
</tr>
<tr>
<td><strong>len</strong></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>contains</strong></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>getitem</strong></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>setitem</strong></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>repr</strong></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>index</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>append</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Easiest:  __setitem__

It is a one-for-one change of data, without any structural change on the list.

(very similar pattern to  __getitem__ )
The **append** method

```
Errol : OurList
  _head: 'F'
  _rest:  Sharon

Sharon : OurList
  _head: 'E'
  _rest:  Per

Per : OurList
  _head: 'T'
  _rest:  Dale

Dale : OurList
  _head: 'E'
  _rest:  Matthew

Matthew : OurList
  _head: None
  _rest: None
```

append('S')
The append method

```
<table>
<thead>
<tr>
<th>Errol : OurList</th>
</tr>
</thead>
<tbody>
<tr>
<td>head: 'F'</td>
</tr>
<tr>
<td>rest: Sharon</td>
</tr>
</tbody>
</table>

append('S')

<table>
<thead>
<tr>
<th>Sharon : OurList</th>
</tr>
</thead>
<tbody>
<tr>
<td>head: 'E'</td>
</tr>
<tr>
<td>rest: Per</td>
</tr>
</tbody>
</table>

append('S')

<table>
<thead>
<tr>
<th>Per : OurList</th>
</tr>
</thead>
<tbody>
<tr>
<td>head: 'T'</td>
</tr>
<tr>
<td>rest: Dale</td>
</tr>
</tbody>
</table>

append('S')

<table>
<thead>
<tr>
<th>Dale : OurList</th>
</tr>
</thead>
<tbody>
<tr>
<td>head: 'E'</td>
</tr>
<tr>
<td>rest: Matthew</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Matthew : OurList</th>
</tr>
</thead>
<tbody>
<tr>
<td>head: None</td>
</tr>
<tr>
<td>rest: None</td>
</tr>
</tbody>
</table>
```
The **append** method

```
Errol : OurList
  _head: 'F'
  _rest: Sharon

Sharon : OurList
  _head: 'E'
  _rest: Per

Per : OurList
  _head: 'T'
  _rest: Dale

Dale : OurList
  _head: 'E'
  _rest: Matthew

Matthew : OurList
  _head: None
  _rest: None
```

```
append('S')
append('S')
append('S')
append('S')
```
The `append` method

```
Errol : OurList
  _head: 'F'
  _rest: Sharon

Sharon : OurList
  _head: 'E'
  _rest: Per

Per : OurList
  _head: 'T'
  _rest: Dale

Dale : OurList
  _head: 'E'
  _rest: Matthew

Matthew : OurList
  _head: None
  _rest: None
```

- `append('S')` is applied to each list.
- The diagram illustrates the flow of the `append` method.
The **append** method

- Errol : OurList
  - head: 'E'
  - rest: Sharon

- Sharon : OurList
  - head: 'E'
  - rest: Per

- Per : OurList
  - head: 'T'
  - rest: Dale

- Dale : OurList
  - head: 'E'
  - rest: Matthew

- Matthew : OurList
  - head: None
  - rest: None

append('S')

append('S')

append('S')

append('S')

append('S')

append('S')
The append method

```
Errol : OurList
  head: 'F'
  rest: Sharon

Sharon : OurList
  head: 'E'
  rest: Per

Per : OurList
  head: 'T'
  rest: Dale

Dale : OurList
  head: 'E'
  rest: Matthew

Matthew : OurList
  head: None
  rest: None
```

```
append('S')
append('S')
append('S')
append('S')
append('S')
```
The append method

Errol : OurList
  _head: 'F'
  _rest: Sharon

Sharon : OurList
  _head: 'E'
  _rest: Per

Per : OurList
  _head: 'T'
  _rest: Dale

Dale : OurList
  _head: 'E'
  _rest: Matthew

Matthew : OurList
  _head: None
  _rest: None

append('S')

append('S')

append('S')

append('S')

append('S')

append('S')

OurList()
The append method

```
Errol: OurList
  _head: 'F'
  _rest: Sharon

Sharon: OurList
  _head: 'E'
  _rest: Per

Per: OurList
  _head: 'T'
  _rest: Dale

Dale: OurList
  _head: 'E'
  _rest: Matthew

Matthew: OurList
  _head: None
  _rest: None

append('S')
```

```
Judy: OurList
  _head: None
  _rest: None
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```

```
append('S')
```
The append method

Errol : OurList
  _head: 'F'
  _rest: Sharon

append('S')

Sharon : OurList
  _head: 'E'
  _rest: Per

append('S')

Per : OurList
  _head: 'T'
  _rest: Dale

append('S')

Dale : OurList
  _head: 'E'
  _rest: Matthew

append('S')

Matthew : OurList
  _head: None
  _rest: None

append('S')

Judy : OurList
  _head: None
  _rest: None

OurList()

2007 OOPSLA Educators’ Symposium
Reinforcing Structural Recursion with Lists – 24 of 40
The **append** method

- **Errol**: OurList
  - head: 'F'
  - rest: Sharon

- **Sharon**: OurList
  - head: 'E'
  - rest: Per

- **Per**: OurList
  - head: 'T'
  - rest: Dale

- **Dale**: OurList
  - head: 'E'
  - rest: Matthew

- **Matthew**: OurList
  - head: None
  - rest: None

- **Judy**: OurList
  - head: None
  - rest: None

append('S')
append('S')
append('S')
append('S')
append('S')
The **append** method

```
Errol : OurList
    _head: 'F'
    _rest: Sharon

append('S') -->

Sharon : OurList
    _head: 'E'
    _rest: Per

append('S') -->

Per : OurList
    _head: 'T'
    _rest: Dale

append('S') -->

Dale : OurList
    _head: 'E'
    _rest: Matthew

append('S') -->

Matthew : OurList
    _head: None
    _rest: None

append('S') -->

Judy : OurList
    _head: None
    _rest: None
```

2007 OOPSLA Educators’ Symposium

Reinforcing Structural Recursion with Lists – 24 of 40
The append method

```
Errol : OurList
  _head: 'E'
  _rest: Sharon

append('S')

Sharon : OurList
  _head: 'E'
  _rest: Per

append('S')

Per : OurList
  _head: 'T'
  _rest: Dale

append('S')

Dale : OurList
  _head: 'E'
  _rest: Matthew

append('S')

Matthew : OurList
  _head: None
  _rest: None

append('S')

Judy : OurList
  _head: None
  _rest: None

append('S')

OurList()
```
The `append` method

Instructor can highlight the system’s memory management.
Arbitrary insertions and deletions can be performed (more on this in the conclusion...)
Implementation
class OurList:
    def __init__(self):
        self._head = None
        self._rest = None

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

Has a base case and a simple recursion
The count method

Has a base case and a non-trivial recursion

```python
def count(self, value):
    if self._isEmpty():
        return 0
    else:
        answer = self._rest.count(value)
        if self._head == value:  # additional match
            answer += 1
        return answer
```
The **__contains__** method

Has two distinct base cases

```python
def __contains__(self, value):
    if self.isEmpty():
        return False
    elif self._head == value:
        return True
    else:
        return value in self._rest  # implicit recursion
```
Conclusions

- Lots of fun
- Strategic challenges
- Varying recursive patterns
- Instills a local perspective
- Coherent transition to source code
- We have really used functional recursion as well as structural recursion.
Conclusions

- Lots of fun
- Strategic challenges
- Varying recursive patterns
- Instills a local perspective
- Coherent transition to source code
- We have really used functional recursion as well as structural recursion.
Conclusions

- Lots of fun
- Strategic challenges
  - Varying recursive patterns
  - Instills a local perspective
  - Coherent transition to source code
  - We have really used functional recursion as well as structural recursion.
Conclusions

- Lots of fun
- Strategic challenges
- Varying recursive patterns
- Instills a local perspective
- Coherent transition to source code
- We have really used functional recursion as well as structural recursion.
Conclusions

- Lots of fun
- Strategic challenges
- Varying recursive patterns
- Instills a local perspective
- Coherent transition to source code
- We have really used functional recursion as well as structural recursion.
Conclusions

- Lots of fun
- Strategic challenges
- Varying recursive patterns
- Instills a local perspective
- Coherent transition to source code
- We have really used functional recursion as well as structural recursion.
Conclusions

- Lots of fun
- Strategic challenges
- Varying recursive patterns
- Instills a local perspective
- Coherent transition to source code
- We have really used functional recursion as well as structural recursion.
Advanced Lessons

- **Ample opportunities for advanced lessons (time permitting)**
  - Error handling
  - Default parameter values
  - More complex recursive patterns
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)
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')
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:
        # reinsert our head as the front of the rest
        self._rest.insert(0, self._head)
        # and then store the new value here
        self._head = value
**The insert method (alternative)**

```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
```
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)
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)
Make use of other existing methods together with one recursive call.

```python
def reverse(self):
    if not self._isEmpty():
        self._rest.reverse()
        self._rest.append(self._head)
        self.remove(self._head)
```
OurList

\_head: 'F'
\_rest: Sharon

\textbf{sort}()