Teaching Strategies for Reinforcing Structural Recursion with Lists

Michael H. Goldwasser David Letscher

Dept. Mathematics and Computer Science
Saint Louis University
221 North Grand Blvd
St. Louis, Missouri 63103-2007

goldwamh@slu.edu letscher@slu.edu

Abstract

Recursion is an important concept in computer science and one that possesses beauty and simplicity, yet many educators describe challenges in teaching the topic. Kim Bruce champions the early use of *structural recursion* in an object-oriented introductory programming course as a more intuitive concept than traditional (functional) recursion. He uses many graphical examples for motivation (e.g., nested boxes, a ringed bullseye, fractals), providing concreteness to the recursive concept. Internally, most of those examples are disguised forms of a basic recursive list pattern. Recursive lists are important in and of themselves and a mainstay within the functional programming paradigm. However, further challenges exist in providing a tangible presentation for pure lists when disassociated from a graphical structure.

We describe an active-learning exercise in which students play the roles of distinct objects that together comprise the structure of a single, recursive list. This activity establishes intuition that we later use when developing a complete implementation of a recursive list class. Our approach demonstrates a rich set of recursive patterns involving several distinct forms of a base case and varied use of parameters and return values.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—recursion; E.1 [*Data Structures*]: lists, stacks and queues; K.3.2 [*Computers and Education*]: Computer and Information Science Education—computer science education

General Terms Algorithms, Design, Languages

Keywords Active learning, lists, recursion, role-playing

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-786-5/07/0010.

1. Introduction

In a recent pair of papers Sanders *et al.* study various mental models of recursion that students develop [6, 10]. The second of those papers opens with the statement,

> "Students struggle to understand recursion and we need to find good ways to teach the concept."

They analyze students' performances in comprehending and demonstrating the trace of a recursive process. The study is organized around aspects of the flow of control as it is passed forward to new activations, reaches base cases, and is passively returned to the caller at the conclusion of an activation.

In this paper, we outline specific strategies used in the classroom to bring concreteness to recursion in the context of an object-oriented introductory course. Our approach is a novel combination of three separate ideas from the literature: the use of structural recursion before functional recursion, the use of role-playing in the classroom, and the use of purely recursive lists as a robust example that demonstrates a variety of recursive patterns.

Structural Recursion. The first piece of our strategy is inspired by Kim Bruce's keynote address at SIGCSE 2005 in which he champions the use of structural recursion in an object-oriented CS1 course; those views are described more fully in a paper co-authored with Danyluk and Murtaugh [4]. With traditional functional recursion, there exists the classic paradox of an executing function calling "itself." The state of the recursive process is encapsulated within multiple activation records. When using structural recursion in an object-oriented framework, an object does not call itself. Instead it invokes a method on some other, tangible object (that just so happens to belong to the same class). Each object has its own state information and typically each has a single active method at any one time.

Bruce *et al.* provide additional concreteness by leveraging use of their `objectdraw` graphics package [3]. Most of their examples involve natural recursive structures such as a ringed target or a fractal-based broccoli, for which the structural components are tangibly visible to a student.

Purely Recursive Lists. While we appreciate the use of graphical recursions, our goal is to carry that concreteness to non-graphical recursive structures. In particular, we consider the classic example of a purely recursive list. This structure is a mainstay of functional programming languages such as LISP [8] and Scheme [1], yet equally elegant in an object-oriented language. Furthermore, lists can be used to demonstrate a wonderful mix of recursive patterns. The natural base case is at the end of the list for some methods, while at the beginning for others. For some methods, the parameters and return values are passed unchanged from level to level of the recursion, while for others those values are adjusted to fit the circumstances.

Though the strategies we describe in this paper are easily applied to any object-oriented language, our presentation is based on the use of Python in CS1. Our exploration of lists leverages our students' familiarity with the built-in list class (Python's analogy of Java's `ArrayList` and C++'s `vector`). Though Python's official list class is not really recursive, we use its public interface as a model when designing and implementing our own version. Since our students use lists from the beginning of the course, the menu of behaviors is established as common knowledge by the time we introduce recursion. This allows us to decouple two potentially intertwined concepts: (1) the use of recursion; (2) the abstraction of a container class. Another great advantage of using Python's list class as our model is the large number of behaviors to mimic. We can describe many of the behaviors in the classroom while leaving others as individual exercises for students.

Role-playing. The final piece of our strategy involves the use of role-playing in the classroom. In an object-oriented context, role-playing typically takes a form in which each student portrays an individual object [2]. Students then interact with each other using the supported methods. This formality helps to reinforce the concepts of individual state, of public interfaces, and of flow of control.

Role-playing is also commonplace when portraying (functional) recursion. For example a student who is asked to compute the value of $n!$ can query another student who is charged with reporting the value of $(n-1)!$. In such an exercise, students are portraying individual function activations rather than objects.

The idea of combining these two role-playing styles has received little attention. Levine comes closest to connecting the issues [7]. He discusses problems students have comprehending the independence of state between multiple instances of the same class, and separately discusses difficulties understanding the independence of state implicit with multiple activations of the same recursive function. He continues by suggesting the use of role-playing for strengthening the students' mental models. Yet the only recursive examples he gives involve drawing images using purely functional recursions.

The one clearly documented example we find for object-oriented role-playing with structural recursion is given by Colgate's unofficial AP Computer Science web page [9]; this suggests a role-play of the broccoli example from Bruce *et al.* Yet that material is rather limited as the recursive pattern for all behaviors has a straightforward form in which the action is invoked directly on each subcomponent.

Our Contribution. In the remainder of this paper, we describe classroom strategies for teaching recursive thinking in an object-oriented framework. This begins with an active-learning activity in which students cooperate to perform a live simulation of a recursive list. We then show how the intuitions established by that exercise can be used as the basis of an actual implementation.

2. An Active Learning Exercise

The instructor begins the activity by describing a high-level design for a class we name, OurList. Each instance has two data members: _head which represents the first data element, and _rest which is itself a list of remaining elements. We model an empty list using an instance that has **None** as the head and **None** as the rest. All other lists have both a head and a rest. Thus, a list of length one has an empty list instance as its rest (as opposed to **None**). Fortunately, motivating the role of an empty list is easy, since this is the default state for Python's built-in list class.

After this introduction the instructor seeks volunteers. Each actor is assigned to portray a list instance. We do not precisely script the students' behaviors. Our goal is to have students develop their own recursive thinking to accomplish each task. We reinforce the view of a list's internal state by handing each participant a piece of paper similar to the one shown in Figure 1. This slip of paper represents that object's individual state information, namely the current value of the head and the name of the person who represents the rest of the list. The instructor typically has templates for these slips ready beforehand, filling in the actual names after identifying volunteers.

Participants do not initially see the global view. However, the instructor takes care to establish the initial configuration so that the collective group comprises a single list, such as the one shown in Figure 2. The instructor initiates a query to the student who represents the beginning of the full list. Though we happen to use the signature of Python's methods, this activity is really language-neutral. A typical example is to ask a question such as, "how many times does the element `'E'` occur on the list?"

Figure 1: One student's view of the object state.

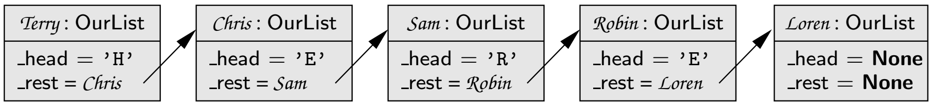

Figure 2: Five instances configured to model the underlying recursive structure for the list, `['H', 'E', 'R', 'E']`.

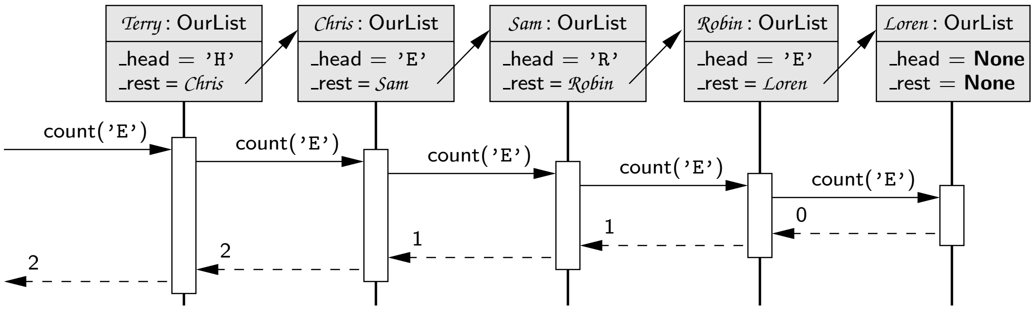

Figure 3: The complete trace of count('E') upon list `['H', 'E', 'R', 'E']`.

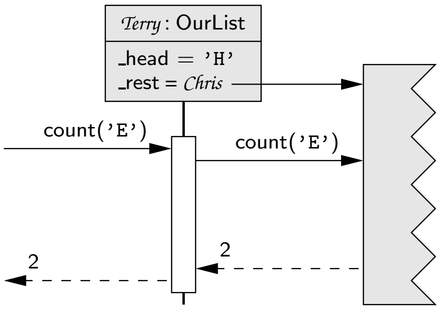

Figure 4: A local view of the initial call to count('E') upon list `['H', 'E', 'R', 'E']`.

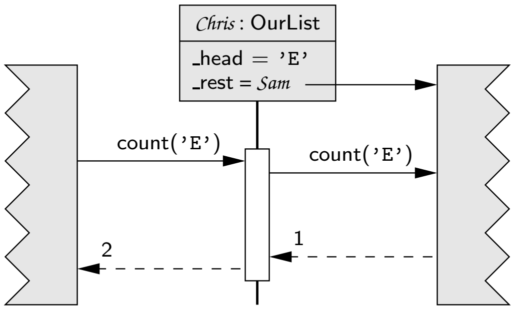

Figure 5: A local view of a secondary call to count('E') upon sublist `['E', 'R', 'E']`.

From a global perspective, we expect the recursion to unfold as shown in Figure 3. This style of diagram can also be used to recap the overall sequence of events that occur. However, we want each student to develop a local perspective, working through the puzzle using only the information which is readily available from that perspective. For example, the first student should experience the activity as shown in Figure 4. Once hearing the news that the rest of the list contains two E's, this student is able to answer the original question posed by the instructor, namely that there are two E's in the overall list. Of course this same process is being used recursively by all students. So the view of the second student in this process looks like Figure 5.

Though we do not put together a formal script, we can envision the interactions between students as a conversation similar to the one shown in Figure 6. Yet to develop the local view, *we forbid any public discussion in the classroom*, instead enforcing a strict message-passing protocol. The only communication that can take place is when a student invokes a presumed method upon the "rest" of his or her list. Taken to an extreme, we use a tennis ball with a seam cut open to represent the flow of control. No one may do anything unless holding the ball. To initiate a method call, a student writes the name of the method and any needed parameter values on a slip of paper (the activation record), inserts that paper into the tennis ball, and passes it to the appropriate individual. The caller then waits until the ball is eventually returned, with any expected return value written on the original slip of paper. Figure 7 shows the activation record for the call that Terry invokes on Chris during our simulation of count. In this case, Chris will eventually return control to Terry, sending the result 2 (which is the number of times E occurs on the list `['E', 'R', 'E']`). Terry will use this information to respond to the original activation that had been formally invoked by the instructor.

We do this activity without any previous discussion of the algorithm, nor examination of source code. Students must work through the puzzle as they go, determining her own personal responsibilities and the proper use of the parameters and return values. At each intermediate level the student

Instructor	→	*Terry*	How many E's are in your list?
Terry	→	*Chris*	How many E's are in your list?
Chris	→	*Sam*	How many E's are in your list?
Sam	→	*Robin*	How many E's are in your list?
Robin	→	*Loren*	How many E's are in your list?
Robin	←	*Loren*	There are 0.
Sam	←	*Robin*	There are 1.
Chris	←	*Sam*	There are 1.
Terry	←	*Chris*	There are 2.
Instructor	←	*Terry*	There are 2.

Figure 6: The "script" for counting the number of E's in the list `['H', 'E', 'R', 'E']`.

must consider "what question was asked of me?" separately from "what question might I ask of another?"

The instructor can provide a play-by-play of the action for the whole group, perhaps even drawing the global sequence that unfolds, as in Figure 3. But the activity should instill a local view of recursion, as in Figures 4 and 5, that becomes important when transitioning to code.

Variety of recursive patterns. As the class masters one behavior, another can be introduced. We draw the students' attention to various recursive patterns we encounter. Significant design aspects are the following: (1) whether the recursion extends all the way to the end of the list, (2) whether the parameter value remains the same at each level of the recursion, and (3) whether the return value remains the same at each level of the recursion.

In the case of counting, the recursion always proceeds to the empty list; there is no way to determine the full count without querying the rest of a list. The parameter for each successive call is the same as the original question (i.e., how many times does the element `'E'` occur?). However, the return value given by the sublist is not necessarily the correct return value for the larger context (as the current head may account for another match).

A very different recursive pattern is seen when asking whether the list contains a given value. Two separate base cases exist. When checking containment with an empty list, the answer is surely **False**. Alternatively, if the head of the list matches the given target value, **True** can be returned without any need for a subsequent call. The recursion proceeds only when neither of those cases occurs.

Another style is seen when asking, "what value is at index i on your list?" (Python's __getitem__ method). For this behavior, the parameter values change from level to level. For example if someone requests the element at index 5 of a list, that student will need to request the item at *index 4* of the sublist. The base case for this example is also different. The primary base case occurs when the index parameter is 0. Such a question can be immediately answered by returning the head element (without need for a successive call). Technically, there is also an error condition

ACTIVATION RECORD	
Sent to:	*Chris*
Method:	count
Parameters (if any):	`'E'`
Please return to:	*Terry*
Return Value (if any):	

Figure 7: An activation record initially sent to Chris from Terry. In our example, Chris will (eventually) return this to Terry with an answer of 2.

that can be reached if requesting any element from an empty list. Note that once an answer is returned by a recursive call, that precise value is subsequently returned up the line.

The overall set of methods in the standard list interface provides a rich combination of such recursive patterns. Figure 8 provides a summary for the most common behaviors.

Mutators and Memory Management. To this point, we have not considered any methods that mutate the list. Indeed, it seems to be more natural to have students gain familiarity through use of accessors. For the classroom activity, we take advantage of the fact that the instructor can simply configure the initial state of a sample list. But eventually, it is good to explore several ways in which a list can be changed. This also brings up the opportunity for several additional lessons regarding the underlying memory management.

The first example we consider with our students is the append method, which is responsible for adding a new element to the end of the list. This has a relatively straightforward recursion. In general, a student can add a new value to the end of her list by simply requesting that it be appended to the rest of her list. The base case occurs when appending onto an empty list. Recall that an empty list is represented by an instance with no head and no rest. Though it is tempting to have that student simply take the new element as the head and consider the job done, this violates the integrity of our conventions. The student who initially represented the empty list should become a list with length one, thus having the inserted value as the head and a newly instantiated empty list as the rest. Figure 9 shows an example of such a change. In the classroom, a new actor must be recruited for the role of a newly instantiated list. The instructor can play the role of the system, by "instantiating" another student and offering a slip of paper to record that instance's state information.

	base case			parameters			return value		
method	empty	head	index	same	vary	none	same	vary	none
__len__	✓					✓		✓	
__contains__	✓	✓		✓			✓		
__getitem__	✓		✓		✓		✓		
__setitem__	✓		✓		✓				✓
__repr__	✓					✓		✓	
count	✓			✓				✓	
index	✓	✓		✓				✓	
append	✓			✓					✓
insert	✓		✓		✓				✓
remove	✓	✓		✓					✓

Figure 8: A survey of the various recursive patterns demonstrated by OurList methods.

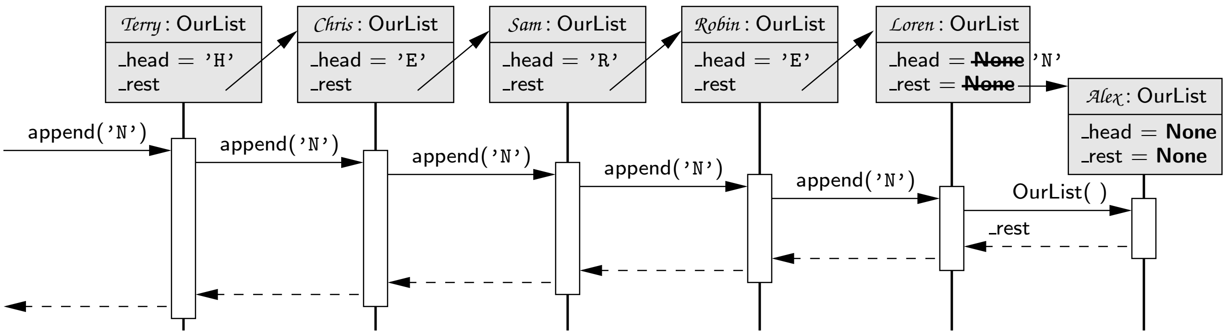

Figure 9: A call to append('N') upon list ['H', 'E', 'R', 'E'].

Another instructive example is the remove method, used to remove the leftmost occurrence of a given value. This behavior requires even more care than append. Typically students see a natural recursion based on the intuition that "if I don't have the desired value, then I should just tell the rest of my list to remove it"; in this case, the leftmost occurrence of the rest is the leftmost occurrence of the original. The challenge occurs when a student has been asked to remove a value that matches the head. For concreteness, consider the execution of remove('R') on the list as originally portrayed in Figure 2. If the remove request passes from Terry, to Chris and then to Sam, what is Sam to do? Setting his head to **None** does not suffice since it leaves him linked in the larger list. Sam may wish to remove himself from the larger context, but this would require changing the state of Chris' _rest attribute; there is no formal interface for Sam initiating such an interaction.

Instead, we lead students through a strategy in which Sam takes over the element at the head of the rest of the list (Robin in this case), and then intentionally splices that next person out of the game. Simulating this efficiently requires a break of the public encapsulation. Sam must take knowledge of Robin's _head and _rest values to accomplish this. To maintain accuracy in our analogy, we do not let Sam start using Robin's slip of paper, but instead to ask Robin what those values. In essence, the algorithm we describe corresponds to the following implementation.

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

Gaining knowledge of **self**._rest._rest requires "private" access, as there is no public interface for accessing that information. This example also brings up an opportunity to discuss garbage collection if desired. Since Robin is no longer a necessary actor, the instructor can play the role of the garbage collector and reclaim Robin's state.

3. Transitioning to an Implementation

The broader goal of our role-playing activity is to build a natural intuition for recursive processes. More tangibly, we want the local perspective from that activity to translate naturally into a true implementation of the process. So after gaining comfort with role-playing, we begin to sprinkle in more explicit discussion of source code to mirror the activity.

In this section, we introduce the basics of our implementation, revisiting several of the behaviors discussed in Section 2. We begin our class definition by providing a constructor that produces an initially empty list.

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

Because recognition of an empty list serves as a base case for almost all of our other methods, we chose to introduce a private utility function, _isEmpty, to improve legibility.

```
  def _isEmpty(self):
    return self._rest is None
```

When beginning the role-play, we relied on the instructor artificially configuring a non-empty list. For testing our implementation, we would like to be able to build interesting lists, so we typically begin by providing the append method. The implementation is quite straightforward and consistent with the intuition established when role-playing.

```
  def append(self, value):
    if self._isEmpty( ):
      self._head = value        # we have one element
      self._rest = OurList( )   # followed by empty list
    else:
      self._rest.append(value)  # pass it on
```

With this method in tact, we can create a longer list and then start to develop the various accessors, one-by-one. For example our implementation of count is written as follows.

```
  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
```

We walk through this code drawing a clear connection to the earlier activity. If counting occurrences on an empty list, the answer is surely zero. Otherwise we begin by getting the recursive count on the rest of the list. However that answer is not necessarily the proper answer for the current call. So the subsequent conditional is used to check for an additional match at the head.

The __contains__ method demonstrates a form with two distinct base cases and a third recursive case.

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

The first base case is triggered when reaching the end of a list and the second when matching the head. Otherwise we revert to a recursive call, with the result of that call blindly returned as the result of the current call.

Another interesting example is the __getitem__ method, which is used to retrieve the element at a specified index. Our implementation is as follows.

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

Notice in the recursive case, the parameterization changes. Retrieving the i^{th} element of a given list is equivalent to retrieving the $(i-1)^{\text{th}}$ element of the rest of the list. When i==0, the desired element is the head. For completeness, we mimic the built-in list by throwing an IndexError that is triggered when the index is out of bounds. To see how this works, we might role-play how this code works on a hypothetical call to access index seven on a list that has length five. A more complete implementation of our class follows, in Figure 10.

The most intricate of the methods shown is insert, which blends the technique of append when at the end of the list, and a shifting technique similar to that of remove, when inserting in the middle of the list. The implementation of repr produces a string representation of the entire list, carefully mimicking the literal form used by Python's built-in list class. By including that method, we can more easily demonstrate the state of our list when developing and testing the other methods.

Even our "complete" implementation does not yet include every feature of the built-in class. A nice benefit of using the Python list class as the model for our own implementation is the richness of the methods that are to be supported. As an instructor, we can present many typical examples to our students yet leave others as exercises.

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

  def _isEmpty(self):
    return self._rest is None

  def __len__(self):
    if self._isEmpty( ):
      return 0
    else:
      return 1 + self._rest.__len__( )

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

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

  def __setitem__(self, i, value):
    if self._isEmpty( ):
      raise IndexError('index out of range')
    elif i == 0:
      self._head = value
    else:
      self._rest.__setitem__(i−1, value)

  def __repr__(self):
    if self._isEmpty( ):
      return '[ ]'
    elif self._rest._isEmpty( ):
      return '[' + self._head.__repr__( ) + ']'
    else:
      return '[' + self._head.__repr__( ) + ', ' \
             + self._rest.__repr__( )[1:]

  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

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

  def append(self, value):
    if self._isEmpty( ):
      self._head = value        # we have one element
      self._rest = OurList( )   # followed by empty list
    else:
      self._rest.append(value)  # pass it on

  def insert(self, index, value):
    if self._isEmpty( ):          # "append" to end
      self._head = value
      self._rest = OurList( )
    elif index == 0:              # new element goes here!
      shift = OurList( )
      shift._head = self._head
      shift._rest = self._rest
      self._head = value
      self._rest = shift
    else:                         # insert recursively
      self._rest.insert(index−1, value)

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

Figure 10: Code four OurList class.

4. Conclusion

In this paper, we have presented a role-playing activity for the classroom, in which students work together to portray a purely recursive list class. The goal in doing so is to instill in our students an accurate mental model for recursion that can then be relied upon when implementing various recursive patterns. We have been very pleased with the activity in our own class, yet we are unable to offer a formal study of their effectiveness given our limited enrollments. Describing the strategies here may lead to a wider adoption by other educators and subsequently, a formal examination by educational researchers as to the effectiveness of the exercise.

We wish to emphasize that Python's `list` class makes a wonderful model for this activity due to the wide range of behaviors. There are many interesting challenges if trying to fully mimic the behavior of the built-in list class. Optional parameters can be supported for the index method. By default it is used to determine the index of the leftmost occurrence of a given value, but it should accept an optional starting index for the search as well as an optional ending index. There is also a method pop(i) which removes and returns the element with index i, yet by default pop( ) removes and returns the last element of the list. We explored the use of exceptions for error handling when presenting __getitem__, however that implementation is slightly flawed. It exposes the recursion to the original caller when the exception is raised from deep within the call stack. Avoiding the exposure of the recursion requires catching and re-raising such an exception at each level.

We also note that all of the methods we demonstrated relied on a single recursive call with the original signature. More general techniques are required to implement the reverse method. This can be explored through role-playing, with a burden on each student to determine a viable strategy. A group discussion could also be used to try to develop a solution to the puzzle. One approach is to recursively reverse the rest of the list and then rely upon other other methods to reposition the original head, as in,

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

Similar explorations are possible if adding support for the sort method in a recursive framework.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 2nd edition, 1996.

[2] S. K. Andrianoff and D. B. Levine. Role playing in an object-oriented world. In *Proc. 33rd SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 121–125, Covington, Kentucky, Feb. 27–Mar. 3, 2002.

[3] K. B. Bruce, A. Danyluk, and T. Murtaugh. A library to support a graphics-based object-first approach to CS 1. In *Proc. 32nd SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 6–10, Charlotte, North Carolina, Feb. 2001.

[4] K. B. Bruce, A. Danyluk, and T. Murtaugh. Why structural recursion should be taught before arrays in CS1. In *Proc. 36th SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 246–250, St. Louis, Missouri, Feb. 2005. ACM Press.

[5] M. L. Dorf. Backtracking the rat way. In *Proc. 23rd SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 272–275, Kansas City, Missouri, Mar. 1992.

[6] T. Götschi, I. Sanders, and V. Galpin. Mental models of recursion. In *Proc. 34th SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 346–350, Reno, Nevada, Feb. 2003.

[7] D. B. Levine. Helping students through multiplicities. *J. Computing Sciences in Colleges*, 15(5):285–291, 2000.

[8] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.

[9] C. Nevison. Advanced placement computer science: Unofficial web page. Accessed in March 2007 at http://cs.colgate.edu/APCS/Java/RolePlays/JavaRolePlays.htm.

[10] I. Sanders, V. Galpin, and T. Götschi. Mental models of recursion revisited. In *Proc. 11th Annual Conf. on Innovation and Technology in Computer Science (ITiCSE)*, pages 138–142, Bologna, Italy, June 2006.