Implementing Lists and Dictionaries in Python

Lists are sequences, like tuples. The Python language does not give us access to the implementation of lists, only to the sequence abstraction and the mutation methods we have introduced in this section. To overcome this language-enforced abstraction barrier, we can develop a functional implementation of lists, again using a recursive representation. This section also has a second purpose: to further our understanding of dispatch functions.

We will implement a list as a function that has a recursive list as its local state. Lists need to have an identity, like any mutable value. In particular, we cannot use `None` to represent an empty mutable list, because two empty lists are not identical values (e.g., appending to one does not append to the other), but `None is None`. On the other hand, two different functions that each have`empty_rlist` as their local state will suffice to distinguish two empty lists.

Our mutable list is a dispatch function, just as our functional implementation of a pair was a dispatch function. It checks the input "message" against known messages and takes an appropriate action for each different input. Our mutable list responds to five different messages. The first two implement the behaviors of the sequence abstraction. The next two add or remove the first element of the list. The final message returns a string representation of the whole list contents.

```
>>> def mutable_rlist():
        """Return a functional implementation of a
mutable recursive list."""
        contents = empty_rlist
        def dispatch(message, value=None):
            nonlocal contents
            if message == 'len':
                return len_rlist(contents)
            elif message == 'getitem':
                return getitem_rlist(contents, value)
            elif message == 'push_first':
                contents = rlist(value, contents)
            elif message == 'pop_first':
                f = first(contents)
                contents = rest(contents)
                return f
            elif message == 'str':
                return str(contents)
        return dispatch
```

We can also add a convenience function to construct a functionally implemented recursive list from any built-in sequence, simply by adding each element in reverse order.

```
>>> def to_mutable_rlist(source):
        """Return a functional list with the same
contents as source."""
        s = mutable_rlist()
        for element in reversed(source):
            s('push_first', element)
        return s
```

In the definition above, the function `reversed` takes and returns an iterable value; it is another example of a function that uses the conventional interface of sequences.

At this point, we can construct a functionally implemented lists. Note that the list itself is a function.

```
>>> s = to_mutable_rlist(suits)
>>> type(s)
<class 'function'>
>>> s('str')
"('heart', ('diamond', ('spade', ('club', None))))"
```

In addition, we can pass messages to the list `s` that change its contents, for instance removing the first element.

```
>>> s('pop_first')
'heart'
>>> s('str')
```

```
"('diamond', ('spade', ('club', None)))"
```

In principle, the operations `push_first` and `pop_first` suffice to make arbitrary changes to a list. We can always empty out the list entirely and then replace its old contents with the desired result.

Message passing. Given some time, we could implement the many useful mutation operations of Python lists, such as `extend`and `insert`. We would have a choice: we could implement them all as functions, which use the existing messages `pop_first`and `push_first` to make all changes. Alternatively, we could add additional `elif` clauses to the body of `dispatch`, each checking for a message (e.g., `'extend'`) and applying the appropriate change to `contents` directly.

This second approach, which encapsulates the logic for all operations on a data value within one function that responds to different messages, is called message passing. A program that uses message passing defines dispatch functions, each of which may have local state, and organizes computation by passing "messages" as the first argument to those functions. The messages are strings that correspond to particular behaviors.

One could imagine that enumerating all of these messages by name in the body of `dispatch` would become tedious and prone to error. Python dictionaries, introduced in the next section, provide a data type that will help us manage the mapping between messages and operations.

Implementing Dictionaries. We can implement an abstract data type that conforms to the dictionary abstraction as a list of records, each of which is a two-element list consisting of a key and the associated value.

```
>>> def dictionary():
        """Return a functional implementation of a
dictionary."""
        records = []
        def getitem(key):
            for k, v in records:
                if k == key:
                    return v
        def setitem(key, value):
            for item in records:
                if item[0] == key:
                    item[1] = value
                    return
            records.append([key, value])
        def dispatch(message, key=None, value=None):
            if message == 'getitem':
                return getitem(key)
            elif message == 'setitem':
                setitem(key, value)
            elif message == 'keys':
                return tuple(k for k, _ in records)
            elif message == 'values':
                return tuple(v for _, v in records)
        return dispatch
```

Again, we use the message passing method to organize our implementation. We have supported four messages: `getitem`,`setitem`, `keys`, and `values`. To look up a value for a key, we iterate through the records to find a matching key. To insert a value for a key, we iterate through the records to see if there is already a record with that key. If not, we form a new record. If there already is a record with this key, we set the value of the record to the designated new value.

We can now use our implementation to store and retrieve values.

```
>>> d = dictionary()
>>> d('setitem', 3, 9)
>>> d('setitem', 4, 16)
>>> d('getitem', 3)
9
>>> d('getitem', 4)
16
>>> d('keys')
(3, 4)
>>> d('values')
(9, 16)
```

This implementation of a dictionary is *not* optimized for fast record lookup, because each response to the message `'getitem'` must iterate through the entire list of `records`. The built-in dictionary type is considerably more efficient.

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/objects.html#implementing-lists-and-dictionaries