Lecture #17: Object-Oriented Versions of Types

Range Revisited

An iterator for Range:

```
class Range:
    """ A range of integers.
    >>> for p in Range(1, 3): print(p)
    1
    2
    """
    class _RangeIterator:
        def __init__(self, aRange):
            self._i, self._high = aRange._low, aRange._high

        def __next__(self):
            if self._i >= self._high:
                raise StopIteration
            else:
                self._i += 1; return self._i - 1

    def __init__(self, low, high):  self._low, self._high = low, high

    def __iter__(self):             return Range._RangeIterator(self)
```


Alternative Iterator

- It turns out that Python is sufficiently clever to do without an iterator, if the type is "tuple-like."
- The following is enough to allow **for** statements to work:

```
class Range:
    """ A range of integers."""
    def __init__(self, low, high):
        self._low, self._high = low, high

    def __len__(self):
        return min(self._low, sslf._high) - self._low

    def __getitem__(self, k):
        if k < 0:
            k = len(self) - k
        if 0 <= k < len(self):
            return self._low + k
        else:
            raise IndexError
```

- That is, Python can create an iterator using the `__getitem__` method, if it exists, which stops by raising `IndexError`.

RList Revisited

- Previously, we introduced `rlists`—recursive lists, aka *linked lists*.
- Here's a version in class form:

```
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        self._first = first
        self._rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self._first
        else:
            return self._rest[i-1]

    def __len__(self):
        return 1 + len(self._rest) # Whoa!  No base case!?
```


Linked Lists: Using the Iterator

- The iterator that Python creates from `__getitem__` is useful internally:

```
    def __str__(self):
        from io import StringIO
        r = StringIO()
        print("(", file=r, end="")
        sep = ""
        for p in self:
            print(sep + repr(p), file=r, end="")
            sep = ", "
        print(")", file=r, end="")
        return r.getvalue()

    def __repr__(self):
        return "Link({}, {})".format(repr(self._first), repr(self._rest))
```


Linked Lists: Fixing Performance

- Unfortunately, using `__getitem__` like this hides a performance problem.
- We have to redo the work to get to the next list item on each iteration.
- It would be better in this case to create an iterator.

```
class Link:
    ...
    def __iter__(self): return Link._iterator(self)

    class _iterator:
        def __init__(self, start):
            self._next_item = start

        # What else?
```