Lecture #16: Iterators, Generators

An Iterator Confusion

- The distinction between *iterators* (things with a `__next__` method) and *iterables* (things from which the `iter` function can construct an iterator) can be confusing, and sometimes downright incovenient.
- Suppose that `backwards(L)` returns an iterator object that returns the values in list L from last to first:

```
class backwards:
      def __init__(self, L):
          self._L = L, self._k = len(L) - 1

      def __next__(self):
          if self._k < 0: raise StopIteration
          else:
              self._k -= 1; return self._L[self._k + 1]
```

- The following won't work [why not?]:

```
for x in backwards(L):
    print(x)
```

An Iterator Convention

- Problem is that `for` expects an *iterable*, but a `backwards` is a pure `iterator`.
- This is awkward, so the usual fix is always to define iterator objects to have a trivial `__iter__` method on them:

```
class backwards:
      def __init__(self, L):
          self._L = L, self._k = len(L) - 1

      def __iter__(self):
          return self    # Now I am my own iterator

      def __next__(self):
          ...
```

- Iterators returned by Python library methods and other standard language constructs obey this convention.

Using `__getitem__` for Iterables

- When confronted with a type that does not implement `__iter__`, but does have a `__getitem__`, the `iter` function creates an iterator.
- This in itself is an example of *generic programming*!
- Conceptually:

```
class GetitemIterator:
    def __init__(self, anIterable):
        """An iterator over ANITERABLE, which must implement __getitem__.
           This iterator returns ANITERABLE[0], ANITERABLE[1], ... up
           to and not including the first index that causes an
           IndexError or StopIteration."""

        _________________________________

        _________________________________

    def __next__(self):

        ?
```

Using __getitem__ for Iterables (II)

A possible implementation:

```
class GetitemIterator:

    def __init__(self, anIterable):
        """An iterator over ANITERABLE, which must implement __getitem__.
           This iterator returns ANITERABLE[0], ANITERABLE[1], ... up
           to and not including the first index that causes an
           IndexError or StopIteration."""
        self._iterable = anIterable
        self._nextIndex = 0

    def __next__(self):
        try:
            v = self._iterable[self._nextIndex]
            self._nextIndex += 1
            return v
        except IndexError:
            raise StopIteration
```

Problem: Reconstruct the range class

- Want `Range(1, 10)` to give us something that behaves like a Python range, so that

  ```
  for x in Range(1, 10):
      print(x)
  ```

  prints 1–9.

  ```
  class Range:
       ???
  ```

Reconstructing Range (I)

```
class Range:
    def __init__(self, first, end, step=1):
        assert step != 0
        ??

    def __getitem__(self, k):
        ??

    def __iter__(self):
        return ??
```

Reconstructing Range (II)

```
class Range:
    def __init__(self, first, end, step=1):
        assert step != 0
        self._first, self._end, self._step = first, end, step

    def __getitem__(self, k):
        ??

    def __iter__(self):
        ??
```

Reconstructing Range (III)

```
class Range:
    def __init__(self, first, end, step=1):
        assert step != 0
        self._first, self._end, self._step = first, end, step

    def __getitem__(self, k):
        if k < 0:

            ________________________________
        if 0 <= k < self._len:

            return ________________________________
        else:

            ________________________________

    def __iter__(self):

        ________________________________
```

Reconstructing Range (IV)

```
class Range:
    def __init__(self, first, end, step=1):
        assert step != 0
        self._first, self._end, self._step = first, end, step

    def __getitem__(self, k):
        if k < 0:
            k += self._len
        if 0 <= k < self._len:
            return self._first + k * self._step
        else:
            raise IndexError

    def __iter__(self):

        ________________________________
```

Reconstructing Range (V)

```
class Range:
    def __init__(self, first, end, step=1):
        assert step != 0
        self._first, self._end, self._step = first, end, step

    def __getitem__(self, k):
        if k < 0:
            k += self._len
        if 0 <= k < self._len:
            return self._first + k * self._step
        else:
            raise IndexError

    def __iter__(self):

        return GetitemIterator(self)
```

Discussion

- An iterator represents a kind of "deconstruction" of a loop.
- Instead of writing a loop such as

```
x = 0             # Initialize iterator object, iterobj
while x < N:      # iterobj.__next__, part 1
    Do something using x
    x += 1        # iterobj.__next__, part 2
```

- ...we break it up as suggested by the comments.
- In some cases (e.g., iterators on trees), the result can be rather clumsy.
- Python provides a different, and generally clearer way to build these iterator objects: as *generators*.

Generators

- For a generator, one writes a function that produces in sequence all the desired values by means of `yield` statements.
- When such a function is called, it executes up to, but not including, the first `yield` and returns a *generator object*, which is a kind of iterator.
- Trivial example:

```
>>> def pairGen(x, y):
...     """A generator that yields X and then Y."""
...     yield x
...     yield y
>>> oneTwo = pairGen(1, 2)
>>> oneTwo
<generator object pairGen ...>
>>> oneTwo.__next__()
1
>>> oneTwo.__next__()
2
>>> oneTwo.__next__()
Traceback ... StopIteration
```

Generator Example: Alterative Implementation of GetitemIterator

```
>>> def GetitemIterator(iterable):
...     k = 0
...     while True:
...         try:
...             yield iterable[k]
...             k += 1
...         except IndexError:
...             return
>>> iterobj = GetitemIterator([1, 3, 7])
>>> iterobj.__next__()
1
>>> iterobj.__next__()
3
>>> for x in GetitemIterator([1, 3, 7]): print(x, end=" ")
1 3 7
```

RList Revisited

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

```
class Link:
    empty = ()

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

    def __getitem__(self, i):
        if i < 0:    # Negative indices count from the end.
            i += len(self)
        p = self     # Actually, could use self in place of p.
        while p is not empty and i > 0:
            p, i = p._rest, i - 1
        if p is empty:
            raise IndexError
        return p._first
```

Linked Lists: Using the Iterator

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

```
def __len__(self):
    c = 0
    for _ in self:
        c += 1
    return c

def __str__(self):
    from io import StringIO
    r = StringIO()    # A kind of file that builds a string in memory
    print("(", file=r, end="")
    sep = ""
    for p in self:    # This creates an iterator that uses __getitem__.
        print(sep + repr(p), file=r, end="")
        sep = ", "
    print(")", file=r, end="")
    return r.getvalue()
```

Linked Lists: Fixing Performance

- Unfortunately, the automatic use of `__getitem__` to create an iterator 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 a specialized iterator.

```
class Link:
    ...
    def __iter__(self):
        p = self
        while p is not Link.empty:
            yield p._first
            p = p._next
```

Iterating Over Trees

- Writing an iterator for a tree is tricky and leads to a rather complex implementation.
- But with a generator, it's pretty easy:

```
def preorderLabels(T):
    """Generate the labels of tree T in preorder (i.e., first the node
    label, then the preorder labels of the branches.)"""
    yield label(T)
    for child in branches(T):
        for label in preorderLabels(child):
            yield label
```

- A recursive generator!
- We can use `for` on `preorderLabels(child)` because Python makes all its generators into iterables, following the convention that iterators should implement a trivial `__iter__` method.

Facilitating Recursive Generators

- The loop in this last generator comes up with some frequency:

```
for label in preorderLabels(child):
    yield label
```

- We call the result of `preorderLabels(child)` a *subiterator*,
- There is a shorthand for this loop over a subiterator:

```
def preorderLabels(T):
    """Generate the labels of tree T in preorder (i.e., first the node
    label, then the preorder labels of the branches.)"""
    yield label(T)
    for child in branches(T):
        yield from preorderLabels(child)
```