ITERATORS AND GENERATORS 10

COMPUTER SCIENCE 61A

July 23, 2015

1 Iterators

An **iterator** is an object that tracks the position in a sequence of values. It can return an element at a time, and it is only good for one pass through the sequence. The following is an example of a class that implements Python's iterator interface. This iterator calculates all of the natural numbers one-by-one, starting from zero:

```
class Naturals():
    def __init__(self):
        self.current = 0

    def __next__(self):
        result = self.current
        self.current += 1
        return result

    def __iter__(self):
        return self
```

An iterator is an object that has a `__next__` and an `__iter__` method.

1.1 `__next__`

The `__next__` method checks if it has any values left in the sequence; if it does, it computes the next element. To return the next value in the sequence, the `__next__` method keeps track of its current position in the sequence. If there are no more values left to compute, it must raise an exception called `StopIteration`. This signals the end of the sequence.

Note: the `__next__` method defined in the `Naturals` class does *not* raise `StopIteration` because there is no "last natural number".

1.2 `__iter__`

The `__iter__` method returns an iterator object. If a class implements both a `__next__` method and an `__iter__` method, its `__iter__` method can simply return `self` as the class itself is an iterator. In fact, the Python docs require that all iterators' `__iter__` methods must return `self`.

1.3 Implementation

When defining an iterator, you should always keep track of current position in the sequence. In the `Naturals` class, we use `self.current` to save the position.

Iterator objects maintain state. Each successive call to `__next__` will return the next element, which may be different, so `__next__` is considered *non-pure*.

Python has built-in functions called **`next`** and **`iter`** that call `__next__` and `__iter__` respectively.

For example, this is how we could use the `Naturals` iterator:

```
>>> nats = Naturals()
>>> next(nats)
0
>>> next(nats)
1
>>> next(nats)
2
```

1.4 Questions

1. Define an iterator whose ith element is the result of combining the ith elements of two input iterators using some binary operator, also given as input. The resulting iterator should have a size equal to the size of the shorter of its two input iterators.

```
>>> from operator import add
>>> evens = IteratorCombiner(Naturals(), Naturals(), add)
>>> next(evens)
0
>>> next(evens)
2
>>> next(evens)
4
```

```
class IteratorCombiner(object):
    def __init__(self, iterator1, iterator2, combiner):



    def __next__(self):



    def __iter__(self):
```

2. What is the result of executing this sequence of commands?

```
>>> nats = Naturals()
>>> doubled_nats = IteratorCombiner(nats, nats, add)
>>> next(doubled_nats)

>>> next(doubled_nats)
```

1.5 Extra Question

1. Create an iterator that generates the sequence of Fibonacci numbers.

```
class FibIterator(object):
    def __init__(self):



    def __next__(self):



    def __iter__(self):
        return self
```

2 Iterables

An **iterable** object represents a sequence. Examples of iterables are lists, tuples, strings, and dictionaries. The iterable class must implement an `__iter__` method, which returns an iterator. Note that since all iterators have an `__iter__` method, they are all iterable.

In general, a sequence's `__iter__` method will return a new iterator every time it is called. This is because an iterator cannot be reset. Returning a new iterator allows us to iterate through the same sequence multiple times.

In the following example, we've defined a simple iterable `Range` class, which represents the integers from 0 to `stop`.

```
class Range:
    def __init__(self, stop):
        self.stop = stop

    def __iter__(self):
        return RangeIterator(self.stop)
```

```
class RangeIterator:
    def __init__(self, stop):
        self.current = 0
        self.stop = stop

    def __iter__(self):
        return self

    def __next__(self):
        curr = self.current
        if curr >= self.stop:
            raise StopIteration
        self.current += 1
        return curr
```

Iterables can be used in for loops and as arguments to functions that require a sequence (e.g. **map** and **zip**). For example:

```
>>> for n in Range(2):
...     print(n)
...
0
1
```

This works because the for loop implicitly creates an iterator using the `__iter__` method. Python then repeatedly calls **next** repeatedly on the iterator, until it raises `StopIteration`. In other words, the loop above is (basically) equivalent to:

```
range_iterator = iter(Range(2))
is_done = False
while not is_done:
    try:
        val = next(range_iterator)
        print(val)
    except StopIteration:
        is_done = True
```

2.1 Questions

1. What would Python display in an interactive session?

```
>>> range3 = Range(3)
>>> for i in range3:
...     print(i)
...



>>> list(range3)

>>> iterator3 = iter(range3)
>>> list(iterator3)

>>> list(iterator3)
```

2. To make the `Link` class iterable, implement the `LinkIterator` class.

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
    def __iter__(self):
        return LinkIterator(self)

class LinkIterator:
    def __init__(self, link):



    def __iter__(self):



    def __next__(self):
```

3 Generators

A **generator** function is a special kind of Python function that uses a `yield` statement instead of a **`return`** statement to report values. *When a generator function is called, it returns an iterable object.*

The following is a function that returns an iterator for the natural numbers:

```
def generate_naturals():
    current = 0
    while True:
        yield current
        current += 1
```

Calling `generate_naturals()` will return a generator object, which you can use to retrieve values.

```
>>> gen = generate_naturals()
>>> gen
<generator object gen at ...>
>>> next(gen)
0
>>> next(gen)
1
```

Think of a generator object as containing an implicit `__next__` method. This means, by definition, a generator object is an iterator.

3.1 yield

The `yield` statement is similar to a **`return`** statement. However, while a **`return`** statement closes the current frame after the function exits, a `yield` statement causes the frame to be saved until the next time `__next__` is called, which allows the generator to automatically keep track of the iteration state.

Once `__next__` is called again, execution resumes where it last stopped and continues until the next `yield` statement or the end of the function. A generator function can have multiple `yield` statements.

Including a `yield` statement in a function automatically tells Python that this function will create a generator. When we call the function, it returns a generator object instead of executing the the body. When the generator's `__next__` method is called, the body is executed until the first `yield` statement.

3.2 Implementation

Because generators are technically iterators, you can implement `__iter__` methods using them. For example:

```
class Naturals():
    def __init__(self):
        self.current = 0

    def __iter__(self):
        while True:
            yield self.current
            self.current += 1
```

`Naturals`'s `__iter__` method now returns a generator object. The behavior of `Naturals` is exactly the same as before:

```
>>> nats = Naturals()
>>> nats_iterator = iter(nats)
>>> next(nats_iterator)
0
>>> next(nats_iterator)
1
```

There are a couple of things to note:

- *No `__next__` method in `Naturals`.* `__iter__` only needs to return an iterator, and a generator is an iterator
- `nats` is a `Naturals` object and `nats_iterator` is a generator
- Generator objects are iterators, so they can be used in for loops

3.3 Questions

1. Define a generator that yields the sequence of perfect squares. The sequence of perfect squares looks like: $1, 4, 9, 16 \ldots$

```
def perfect_squares():
```

2. To make the `Link` class iterable, implement the `__iter__` method using a generator.

```
class Link:
    empty = ()

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

    def __iter__(self):
```

3.4 Extra Questions

1. Write a generator function that returns all subsets of the positive integers from 1 to `n`. Each call to this generator's `__next__` method will return a list of subsets of the set `[1, 2, ..., n]`, where `n` is the number of times `__next__` was previously called.

```
def generate_subsets():
    """
    >>> subsets = generate_subsets()
    >>> for _ in range(3):
    ...     print(next(subsets))
    ...
    [[]]
    [[], [1]]
    [[], [1], [2], [1, 2]]
    """
```