Iterators and Generators 6

Computer Science 61A

March 2, 2017

1 Iterators

An **iterator** is an object that tracks the position in a sequence of values in order to provide sequential access. It returns elements one at a time and is only good for one pass through the sequence. The following is an example of a class that implements Python's iterator interface using two special methods `__next__` and `__iter__`. 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
```

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, Python specifies that an iterator's `__iter__` method should 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 in the sequence. Since this element may be different from the previous one, `__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 Iterables

An **iterable** object is any container that can be processed sequentially. 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.

1.5 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)
```

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

```
def gen_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 = gen_naturals()
>>> gen
<generator object gen at ...>
>>> next(gen)
0
>>> next(gen)
1
```

2.1 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 next yield statement is executed.

2.2 __iter__

We can make our own classes iterable using the __iter__ method, which returns an iterator object. Because generators are technically iterators, you can implement __iter__ methods using them. For example:

```
class Naturals():
    def __iter__(self):
        current = 0
        while True:
            yield current
            current += 1
```

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

```
>>> nats = Naturals()
>>> nats_iterator1 = iter(nats)
>>> next(nats_iterator1)
0
>>> next(nats_iterator1)
1
>>> nats_iterator2 = iter(nats)
>>> next(nats_iterator2)
0
```

In this example, we can iterate over the same object more than once by calling **iter** multiple times. Note that nats is an iterable object and the nats_iterator's are generators.

2.3 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. 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]]
    """
```

3 Nonlocal Practice

1. The bathtub below simulates an epic battle between Finn and Kylo Ren over a populace of rubber duckies. Fill in the body of `ducky` so that all doctests pass.

```
def bathtub(n):
    """
    >>> annihilator = bathtub(500) # the force awakens...
    >>> kylo_ren = annihilator(10)
    >>> kylo_ren()
    490 rubber duckies left
    >>> rey = annihilator(-20)
    >>> rey()
    510 rubber duckies left
    >>> kylo_ren()
    500 rubber duckies left
    """
    def ducky_annihilator(rate):
        def ducky():









        return ducky
    return ducky_annihilator
```

2. (Fall 2013) Draw the environment diagram that results from the following code:

```
def miley(ray):
    def cy():
        def rus(billy):
            nonlocal cy
            cy = lambda: billy + ray
            return [1, billy]
        if len(rus(2)) == 1:
            return [3, 4]
        else:
            return [cy(), 5]
    return cy()[1]

billy = 6
miley(7)
```