ITERATORS AND GENERATORS 11

DATA C88C

April 8, 2024

1 Iterators

1.1 Introduction

An **iterable** is a data type which contains a collection of values which can be processed one by one sequentially. Some examples of iterables we've seen include lists, tuples, strings, and dictionaries. In general, any object that can be iterated over in a **`for`** loop can be considered an iterable.

Often we want to access the elements of an iterable, one at a time. We find ourselves writing `lst[0]`, `lst[1]`, `lst[2]`, and so on. It would be more convenient if there was an object that could do this for us, so that we don't have to keep track of the indices.

This is where **iterators** come in. Given an iterable, we can call the **`iter`** function on that iterable to return a new iterator object. Each time we call **`next`** on the iterator object, it gives us one element at a time, just like we wanted. Each iterator keeps track of its position within the iterable. Calling the **`next`** function on an iterator will give the current value in the iterable and move the iterator's position to the next value.

In this way, the relationship between an iterable and an iterator is analogous to the relationship between a book and a bookmark - an iterable contains the data that is being iterated over, and an iterator keeps track of your position within that data.

Once an iterator has returned all the values in an iterable, subsequent calls to **`next`** on that iterable will result in a `StopIteration` exception. In order to be able to access the values in the iterable a second time, you would have to create a second iterator.

1.2 Writing an Iterator Class

As a reminder, 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
```

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.

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. In the `Naturals` class, we use `self.current` to save the position.

If there are no more values left to compute, the `__next__` method must raise an exception called `StopIteration`. This signals the end of the sequence. The `__next__` method defined in the `Naturals` class above does *not* raise `StopIteration` because there is no "last natural number".

1.3 Questions

1. What would Python display? If a StopIteration Exception occurs, write `StopIteration`, and if another error occurs, write `Error`.

```
>>> lst = [[1, 2]]
>>> i = iter(lst)
>>> j = iter(next(i))
>>> next(j)

>>> lst.append(3)
>>> next(i)

>>> next(j)

>>> next(i)
```

2. Create an iterator that generates the sequence of Fibonacci numbers. The Fibonacci sequence starts with 0 and 1, and then all subsequent numbers are formed by adding the two previous numbers together. The first ten numbers of the Fibonacci sequence are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

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



    def __next__(self):









    def __iter__(self):
        return self
```

2 Generators

2.1 Introduction

Generators can be used to create iterators as well. Generators are functions that use a `yield` statement instead of `return`. When a generator function is called, the body of the function is not evaluated yet. Instead, a generator object, which is a type of iterator,is created and is the return value of the function call. The elements of this iterator are the yielded values of the function.

```
>>> square = lambda x: x*x
>>> def get_squares(s):
...     for x in s:
...         yield square(x)
>>> square_iter = get_squares([1, 2, 3])
>>> next(square_iter)
1
>>> next(square_iter)
4
>>> next(square_iter)
9
>>> next(square_iter)
StopIteration
```

2.2 Yielding From an Iterable

When `yield from` is called on an iterator, it will `yield` every value from that iterator. It's similar to doing the following:

```
for x in an_iterator:
    yield x
```

2.3 Questions

1. What would Python display? If a StopIteration Exception occurs, write `StopIteration`, or if another error occurs, write `Error`.

```
>>> def weird_gen(x):
...     if x % 2 == 0:
...         yield x * 2
...     else:
...         yield x
...         yield from weird_gen(x - 1)
>>> next(weird_gen(2))


>>> list(weird_gen(3))


>>> def greeter(x):
...     while x % 2 != 0:
...         print('hello!')
...         yield x
...         print('goodbye!')
>>> greeter(5)

>>> gen = greeter(5)
>>> next(gen)



>>> next(gen)
```

2. Implement a generator function called `filter(iterable, fn)` that only yields elements of iterable for which fn returns True.

```
def filter(iterable, fn):
    """
    >>> is_even = lambda x: x % 2 == 0
    >>> list(filter(range(5), is_even))
    [0 , 2 , 4]
    >>> all_odd = [2*y-1 for y in range (5)]
    >>> list(filter(all_odd, is_even))
    []
    >>> s = filter(naturals(), is_even)
    >>> next(s)
    2
    >>> next(s)
    4
    """
```