ITERATORS AND GENERATORS 11

COMPUTER SCIENCE 88

November 17, 2021

1 Iterator

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
```

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.

While an iterable contains values that can be iterated over, we need another type of object called an **iterator** to actually retrieve values contained in an iterable. Calling the **`iter`** function on an iterable will create an iterator over that iterable. 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.

We have already been using iterables to go through the elements of a sequence. This happens all the time in for loops. For example:

```
>>> for n in [1, 2, 3]:
...     print(n)
...
1
2
3
```

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:

```
iterator = iter([1, 2, 3])
try:
    while True:
        n = next(iterator)
        print(n)
except StopIteration:
    pass
```

1.1 Questions

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

```
>>> lst = [6, 1, "a"]
>>> next(lst)


>>> lst_iter = iter(lst)
>>> next(lst_iter)


>>> next(lst_iter)
>>> next(iter(lst))

>>> [x for x in lst_iter]
```

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

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



    def __next__(self):




    def __iter__(self):
        return self
```

3. Implement an iterator class called `Filter`. The `__init__` method for `Filter` takes an iterable and a one-argument function `fn` that either returns True or False. The `Filter` iterator only contains elements of the iterable for which the predicate function `fn` returns True. Do not use a generator in your solution.

```
class Filter :
    """
    >>> is_even = lambda x: x % 2 == 0
    >>> for elem in Filter(range(5) , is_even):
    ...     print(elem)
    0
    2
    4
    >>> all_odd = (2*y-1 for y in range (5))
    >>> for elem in Filter(all_odd, is_even):
    ...     print(elem) # No elements are even !
    >>> s = Filter(naturals(), is_even)
    >>> next(s)
    2
    >>> next(s)
    4
    """
    def __init__(self, iterable, fn):







    def __iter__( self ):




    def __next__( self ):
```

2 Generator

Generators can be used to create iterators as well. Generators use a `yield` statement instead of `return`. When a generator function is called, the body of the function is not evaluated. Instead, an iterator is created and is the return value of the function call. The elements of this iterator are the yielded values of the function. For extra fun, `yield from` lets generators yield multiple values at once.

```
>>> square = lambda x: x*x
>>> def many_squares(s):
...     for x in s:
...         yield square(x)
...     yield from [square(x) for x in s]
...     yield from map(square, s)
...
>>> list(many_squares([1, 2, 3]))
[1, 4, 9, 1, 4, 9, 1, 4, 9]
```

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. 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 body. When the generator's **`next`** method is called, the body is executed until the next `yield` statement is executed.

2.1 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
    """
```