ITERATORS AND GENERATORS 11

DATA C88C

November 16, 2022

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.

Here is an example of using an iterator:

```
>>> lst = [5, 6, 7]
>>> lst_iterator = iter(lst)
>>> next(lst_iterator)
5
>>> next(lst_iterator)
6
>>> lst_iterator_2 = iter(lst)
>>> next(lst_iterator_2)
5
>>> next(lst_iterator)
7
>>> next(lst_iterator)
StopIteration
```

Notice that `lst_iterator` and `lst_iterator_2` each go through the list separately. At the time of creating `lst_iterator_2`, even though `lst_iterator` has gone through 5 and 6, `lst_iterator_2` restarts at the beginning with 5.

1.2 Iterators in For Loops

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
```

1.3 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".

Python has built-in functions called **`next`** and **`iter`** that call `__next__` and `__iter__` respectively. This is how we could use the `Naturals` iterator:

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

1.4 Questions

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

Solution: It can be helpful to refer back to the `iter` example on the page 2. Remember that calling `iter` returns something that you can call `next` on. The rest of the challenge in this problem is just keeping track of where you currently are in the sequence. Video walkthrough

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

Solution:
1

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

Solution:
3

```
>>> next(j)
```

Solution:
2

```
>>> next(i)
```

Solution:
StopIteration

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

```
>>> lst = ['data', 88, 'c']
>>> next(lst)
```

Solution:
Error

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

Solution:
'data'

```
>>> next(lst_iter)
```

Solution:
88

```
>>> next(iter(lst))
```

Solution:
'data'

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

Solution:
['c']

3. 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):
```

Solution:

```
        self.current = 0
        self.next = 1
```

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

Solution:

```
        old_current = self.current
        self.current = self.next
        self.next = old_current + self.current
        return old_current
```

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

4. 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)]
    >>> all_odd
    [-1, 1, 3, 5, 7]
    >>> for elem in Filter(all_odd, is_even):
    ...     print(elem) # No elements are even, so nothing is
       printed
    >>> s = Filter(naturals(), is_even)
    >>> next(s)
    2
    >>> next(s)
    4
    """
    def __init__(self, iterable, fn):
```

Solution:

```
        self.iterator = iter(iterable)
        self.fn = fn
```

```
    def __iter__(self):
```

Solution:

```
    return self
```

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

Solution:

```
    candidate = next(self.iterator)
    while not self.fn(candidate):
```

```
        candidate = next(self.iterator)
    return candidate
```

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
```

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.

2.2 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
```

`yield from` can be used in conjunction with other `yield` and `yield from` statements.

```
>>> 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]
```

Note: When the `list` function in Python receives an iterator, it calls the `next` function on the input until it raises a `StopIteration`. It puts each of the elements from the calls to `next` into a new list and returns it.

2.3 Using Generators in a Class

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.

2.4 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))
```

Solution:
4

```
>>> list(weird_gen(3))
```

Solution:
[3, 4]

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

Solution:
<generator object greeter at ...>

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

Solution:
hello!
5

```
>>> next(gen)
```

Solution:

```
goodbye!
hello!
5
```

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
    """
```

Solution:

```
    for elem in iterable:
        if fn(elem):
            yield elem
```

3. Write a generator function `gen_all_items` that takes a list of iterators and yields items from all of them in order.

```
def gen_all_items(lst):
    """
    >>> nums = [[1, 2], [3, 4], [[5, 6]]]
    >>> num_iters = [iter(lst) for lst in nums]
    >>> list(gen_all_items(num_iters))
    [1, 2, 3, 4, [5, 6]]
    """
```

Solution:

```
    for it in lst:
        yield from it
```

The `yield from` solution is nice and short. But this can also be done with just yield:

```
    for it in lst:
        for x in it:
            yield x
```

Notice that this function will not yield out of deep lists. That is, it will keep the brackets around deep lists and yield them together instead of one element at a time.

For an extra challenge, figure out how to yield deep list items! (so the example in the doctest would return [1, 2, 3, 4, 5, 6])

4. Write a generator function `combiner` that combines two input iterators using a given combiner function. The resulting iterator should have a size equal to the size of the shorter of its two input iterators.

```
def combiner(iterator1, iterator2, combiner):
    """
    >>> from operator import add
    >>> evens = combiner(iter(Naturals()), iter(Naturals()), add)
    >>> next(evens)
    0
    >>> next(evens)
    2
    >>> next(evens)
    4
    """
```

Solution:

```
    while True:
        yield combiner(next(iterator1), next(iterator2))
```

While this is the most compact solution, it may not be immediately obvious that we would arrive at this. It's acceptable to start with the "basic skeleton" of all generators:

```
    while True:
        <do some work here>
        yield <something>
        <do some other work here>
```

From this, we put in some basic steps:

- We want to fetch the next item from both our iterators.
- Then, we would want to combine them using our combiner function.
- Finally, we want to yield the result (be very careful not to return!).

```
    while True:
        n1 = next(iterator1)
        n2 = next(iterator2)
        result = combiner(n1, n2)
        yield result
```

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

```
>>> nats = iter(Naturals())
>>> doubled_nats = combiner(nats, nats, add)
>>> next(doubled_nats)
```

Solution: 1

```
>>> next(doubled_nats)
```

Solution: 5

Solution: The same naturals iterator has been fed into `combiner` twice. So the first yield will get the first two numbers out of naturals, the second yield will the the third and fourth numbers, and so on.

- 0 + 1 = 1
- 2 + 3 = 5

If you expected this to return 0 then 2, think about what would need to be changed in how we use `combiner`. Also, let's assume we don't want to change the behaviour of the `combiner` function.

Video walkthrough