CS 88 Fall 2018 John Yang
Prof. David Culler Friday 2-4 PM

Iterators & Generators

Link to Discussion Slides: https://goo.gl/jgXzSM

At this point in CS 88, we have spent the last couple of weeks learning about object oriented programming and class design, specifically concepts like inheritance and abstract data types. Through variables and methods, we can now characterize our classes with a variety of attributes and functionality! Today, we're going to be learning about an additional set of methods that allows a data type to contain a collection of values that can be sequentially visited. This property is what we call an **iterable**!

1 Iterators

Note: Much of this document has been paraphrased and adopted from CS61A's discussion worksheet which discusses iterators and generators.

1.1 Terminology

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

In order for an object to be iterable, a special type of object called an **iterator** is required to actually retrieve the values from the iterable. Here, we introduce two new functions to help us accomplish these two tasks. The **iter** function creates an iterator over that iterable. The **next** function returns the current value of the iterable and shifts the iterator's pointer to the next value.

The relationship between an iterator and iterable is similar to a bookmark to a book. Similar to how a bookmark indicates the reader's position within the book content, the iterator indicates the user's current position within the collection of values from the iterable.

1.2 The Anatomy of an Iterator

Given the **next** and **iter** methods, how exactly does code like the following execute accordingly:

```
numList = [1, 2, 3]
for x in numList:
    print(x)
```

Perhaps the best explanation for how a Python compiler deconstructs and executes a for loop is if we rewrite the code in the form of a while loop:

```
items = iter(counts)
while True:
    try:
        i = next(items)
        print(i)
    except StopIteration:
        break
```

So what's happening? First, the **for** loop implicitly calls the **iter** function on the object being iterated over. This call returns the *iterator* object that will actually visit each value. Next, every iteration of the for loop is actually a call to the **next** method. From the *while True* loop, we can tell that the **next** method is called repeatedly until the StopIteration exception is raised, indicating that there are no more values in the iterable object's collection, so the for loop exits.

1.3 Problems

1.3.1 What Would Python Print?

```
>>> lst = ['c', 's', 8, 8]
>>> next(lst)
'TypeError: list object not an iterator'

>>> lstIter = iter(lst)
>>> next(lstIter)
c

>>> next(lstIter)
s

>>> next(lstIter)
8

>>> next(lstIter)
8

>>> next(lstIter)
StopIteration

>>> lstIter = iter(lst)
>>> next(lstIter)
c
```

1.3.2 A Custom Iterator

In the following problem, you have a class called transactions that has three different lists: store, items, and values. The **iter** function has already been implemented for you. Your task is to implement the **next** function that, on each call, returns the elements from the three lists at the same index.

```
class Transactions():
    stores = ["Walmart", "Target", "Walgreens"]
    items = ["toothpaste", "toothbrush", "floss"]
    values = [5, 15, 12]
    index = 0

    def __iter__(self):
        return self

    """
    Implement the next function so that it returns the elements from
    the three lists above at the same index.

    >>> transactions = Transactions()
    >>> transIter = iter(transactions)
    >>> next(transIter)
    'Walmart, toothpaste, 5'
    >>> next(transIter)
    'Target, toothbrush, 15'
    """

    def __next__(self):
        # Your code below!
        if self.index < min(len(self.stores), len(self.items), len(self.values)):
            value = self.stores[self.index] + ", " + self.items[self.index] + ", " + str(self.values[self.index])
            self.index += 1
            return value
        else:
            raise StopIteration
```

2 Generators

In the previous section, we learned about iterators in the context of a class that requires the implementation of the **next** and *iter* methods. With generators, we'll discover how the **yield** key word allows us to define methods that create iterators!

2.1 Terminology

A generator uses the **yield** command for reporting values. By using **yield**, a generator function will return an iterator when called. In other words, if **yield** is used within a function, Python automatically assumes that this function will create a generator.

Up until this point, we've been using **return** statements, so what distinguishes it from **yield**? We know that when a function is called, a separate frame is created (think back to Python Tutor)! If a **return** is executed, that method is frame is closed when the function exits. In other words, the frame disappears.

On the other hand, a **yield** statement *saves* the frame. When **next** is called, as opposed to starting from the beginning of the method, the execution continues from that **yield** statement. In fact, a generator function can have multiple **yield** statements, thus allowing us to create multiple stages of execution within the same method! Pretty neat stuff :)

The last thing to learn is the **yield from** keyword. The **yield from** yields every value from the iterator. In code, it's similar to doing the following:

for x **in** an$_i$*terator* :
    yield$x$

2.2 Problems

2.2.1 What Would Python Do?

```
>>> def powers(x):
>>>     value = x
>>>     while True:
>>>         yield value
>>>         value *= x
>>>         if value == 16
>>>             return value

>>> gen = powers(2)
>>> next(gen)
2

>>> next(gen)
4

>>> next(gen)
8
```

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

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

>>> gen = mystery(2)
>>> next(gen)
4
>>> next(gen)
StopIteration

>>> gen = mystery(9)
>>> next(gen)
9
>>> next(gen)
16
>>> next(gen)
StopIteration
```

At most how many times can **next**(gen) be called before an exception is thrown?
2