Iterators and Generators

What's going on when you run a for loop in Python?

In [2]:
```
a = [1, 4, 5, 9]
for x in a :
    print (x ** 2)
```

```
1
16
25
81
```

What is happening is that under the hood, Python is creating an **iterator** -- an object that lets you step through the list one element at a time, returning each element as it goes. We can actually get to the underlying iterator of a list by calling the `__iter__` function:

In [20]:
```
i = a.__iter__()
print(i)
```

```
<list_iterator object at 0x10de87110>
```

An iterator object is one that implements the iterator **protocol**. A protocol is like an interface in Java or an abstract class in C++ -- it's basically a contract that an object must satisfy to be used in certain ways.

The iterator protocol says that an iterator needs to support two operations:

- `__iter__` that returns the iterator itself, and
- `__next__` that, as you might expect, returns the next element of whatever is being iterated over.

We need to define `__iter__` because `for` loops invoke `__iter__` on whatever you are iterating over. Python wants you to be able to use collections like lists or the iterators themselves in `for` loops.

In [21]:
```
print(i.__next__())
print(i.__next__())
print(i.__next__())
print(i.__next__())
```

```
1
4
5
9
```

When an iterator runs out of items (i.e., when `__next__` gets to the end of the collection), it raises a `StopIteration` exception.

> We have only briefly talked about exceptions in this class. **Exceptions** are a language construct that lets you break out of even very deep control flow when something "bad" happens (it doesn't have to be truly bad, like in the `StopIteration` case). You can then *catch* exceptions and do something (like end a `for` loop) when they are raised.

In [22]:
```
print(i.__next__()) #this will raise an exception
```

```
---------------------------------------------------------------------
----
StopIteration                             Traceback (most recent call l
ast)
<ipython-input-22-34bb15456ba2> in <module>
----> 1 print(i.__next__()) #this will raise an exception

StopIteration:
```

Because iterators are different objects, you can create *multiple* iterators from the same collection that each step over the data:

In [6]:
```
i1 = a.__iter__()
i2 = a.__iter__()
print(i1.__next__())
print(i1.__next__())
print(i2.__next__())
print(i2.__next__())
print(i1.__next__())
print(i1.__next__())
print(i2.__next__())
print(i2.__next__())
```

```
1
4
1
4
5
9
5
9
```

Actually, in code, the 'right' way to call `a.__iter__()` and `a.__next__()` and is to write `iter(a)` and `next(a)`, which is also simpler. For our purposes, both implementations do the same thing, though.

Let's write an iterator that starts at 1, returns numbers increasing by one, and stops after 20:

In [23]:
```
class MyNumbers:
    def __iter__(self):
        self.a = 1   #start the iterator at 1
        return self

    def __next__(self):
        if self.a <= 20:
            x = self.a    #define x as the current iterator
            self.a += 1   #increment self.a for next time
            return x      #return the current iterator
        else:
            raise StopIteration()   #this is how we stop a loop

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
    print(x)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Let's write an iterator that prints out every *other* element of a list. Note that we're going to use a trick: we'll keep a "normal" iterator for the list as part of our skip iterator:

In [8]:
```
class skipIterator :
    def __init__(self, inList) :
        self._inner = inList.__iter__()

    def __iter__(self) :
        return self    #Just need to return the iterator object

    def __next__(self) :
        self._inner.__next__() #skip one element
        return self._inner.__next__() #return the next
```

In [9]:
```
s = skipIterator([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
for x in s :
    print (x)
```

```
2
4
6
8
10
```

In the above example, why don't we need to raise StopException ourselves? The _inner iterator will raise the exception, and since we don't do anything special, that exception will propagate out of __next__ as if we raised it ourself.

Generators

Normally, an iterator needs a way to keep track of its "current" position in the data its iterating over, which can make for complicated code. Instead, we can use **generators** to do this kind of tracking automatically. A generator is a function that *yields* elements as it executes. A yield statement essentially returns a value from the function, but "pauses" the function where the yield was invoked.

Any function that has a `yield` statement in it automatically returns a generator object. It implements the iterator protocol (so you can use it in for loops). Calling `__next__` on a generator object executes the function until you get to a `yield` statement, then pauses and returns whatever is yielded. Calling `__next__` again just picks up execution at the yield statement and executes until the next `yield` .

To motivate the use of generators, let's start by writing an *iterator* that counts out the gaps between particular letters in a string. Note how we have to keep track of both how far along we are in the string as well as how long the current gap is:

In [29]:
```
class findDist :
    def __init__(self, tstr, char) :
        self.string = tstr
        self.char = char
        self.pos = 0

    def __iter__(self) :
        return self

    def __next__(self) :
        delta = 0
        if (self.pos == len(self.string)) :
            raise StopIteration() #at the end of the str
        while (self.string[self.pos] != self.char) :
            delta += 1      #keep track of current distance
            self.pos += 1   #keep track of where we currently are
        self.pos += 1       #important to skip over the char before star
ting the next count
        return delta
```

In [30]:
```
for i in findDist('abracadabra', 'a') :
    print (i)
```

```
0
2
1
1
2
```

Notice that this implementation will also result in an error whenever the final letter of `tstr` is not an `a`. To prevent this, we would need to add a `raise Stopiteration` clause in the `while` loop, so that it is checking whenever `self.pos` is incremented.

Now let's write the same thing using a generator. By writing a function with `yield`, we automatically get an iterator without having to write a class that implements the protocol:

In [31]:
```
def findDistYield(string, char) :
    delta = 0
    for c in string :
        if c == char :
            yield delta   #returns current delta and continues execution
            delta = 0
        else :
            delta += 1
```

In [32]:
```
for i in findDistYield('abracadabra', 'a') :
    print(i)
```

```
0
2
1
1
2
```

We can use iterators and generators to write iterators for new classes that we define. Consider a linked list class with an iterator:

In [14]:
```
class LinkedList :

    def __init__(self, init_val = None) :
        self.data = init_val
        self.next = None

    def insert(self, val) :
        newNode = LinkedList(self.data)
        newNode.next = self.next
        self.next = newNode
        self.data = val

    def insertList(self, vals) :
        for i in vals[::-1] :
            self.insert(i)

    class LinkedListIterator :
        def __init__(self, cur) :
            self.cur = cur

        def __iter__(self) :
            return self

        def __next__(self) :
            if (self.cur.data == None) :
                raise StopIteration
            else :
                ret = self.cur.data
                self.cur = self.cur.next
                return ret

    def __iter__(self) :
        return LinkedList.LinkedListIterator(self)
```

In [15]:
```
l = LinkedList()
l.insertList([1, 2, 3, 4, 5])
for x in l :
    print(x)
```

```
1
2
3
4
5
```

Notice that here we need an iterator subclass, called `LinkedListIterator`, to actually cycle through the values in the list. This part becomes much simpler with a generator, where we can just `yield` the data for the current element and move to the next one:

In [16]:
```
class LinkedList2 :

    def __init__(self, init_val = None) :
        self.data = init_val
        self.next = None

    def insert(self, val) :
        newNode = LinkedList(self.data)
        newNode.next = self.next
        self.next = newNode
        self.data = val

    def insertList(self, vals) :
        for i in vals[::-1] :
            self.insert(i)

    def __iter__(self) :
        cur = self
        while (cur.data != None) :
            yield cur.data
            cur = cur.next
```

In [17]:
```
l2 = LinkedList2()
l2.insertList([1, 2, 3, 4, 5])
for x in l2 :
    print(x)
```

```
1
2
3
4
5
```

Much shorter!

But note that `LinkedList` is a recursive type: its next pointer is another linked list. Could we do something even more clever with generators? Yes! We can yield the current element, then iterate over the rest of the list by invoking its generator.

In the code below, you can think of `yield from g`, where `g` is a generator, as implementing the statement `for v in g: yield v` (though they are not *exactly* equivalent in all cases).

In [18]:
```
class LinkedList3 :

    def __init__(self, init_val = None) :
        self.data = init_val
        self.next = None

    def insert(self, val) :
        newNode = LinkedList(self.data)
        newNode.next = self.next
        self.next = newNode
        self.data = val

    def insertList(self, vals) :
        for i in vals[::-1] :
            self.insert(i)

    def __iter__(self) :
        if self.data != None :
            yield self.data
            yield from self.next
```

In [19]:
```
l3 = LinkedList3()
l3.insertList([2, 4, 6, 8, 10])
for x in l3 :
    print(x)
```

```
2
4
6
8
10
```

Chaining Generators

We can chain generators together: by passing one generator to another and iterating over each, we can build a pipeline that passes data from one to the next. What's great about this is that the processing happens one element at a time (thanks to the `yield` statement) rather than fully building a list each time.

Let's first do this the normal way:

In [19]:
```
def square(vals) :
    return [v ** 2 for v in vals]

def negate(vals) :
    return [-1 * v for v in vals]

def div(vals) :
    return [v / 2 for v in vals]

negate(div(square([1, 2, 3, 4, 5])))
```

Out[19]:
```
[-0.5, -2.0, -4.5, -8.0, -12.5]
```

But what's happening is that we're creating a brand new list each time we call the next function in the chain. This can take a lot of memory, and a lot of time, if the lists are big. Let's now do the same thing with generators:

In [20]:
```
def ysquare(vals) :
    for v in vals :
        yield v ** 2

def ynegate(vals) :
    for v in vals :
        yield -1 * v

def ydiv(vals) :
    for v in vals :
        yield v / 2

ynegate(ydiv(ysquare([1, 2, 3, 4, 5])))
```

Out[20]:
```
<generator object ynegate at 0x10f429a20>
```

Note that this does not generate the list, since `ynegate` is a generator function. You have to iterate over it in order to get the values out of it. Luckily, `list` s can be constructed by passing them an iterator:

In [21]:
```
g = ynegate(ydiv(ysquare([1, 2, 3, 4, 5])))
list(g)
```

Out[21]:
```
[-0.5, -2.0, -4.5, -8.0, -12.5]
```

We get the same result, but each time the `list` constructor asks for the next element, each generator in the chain operates on just one additional item in the input list. The item is squared, then yielded to `ydiv` , then yielded to `ynegate` .

One final thing: just like we have list comprehensions as a fast way of building new lists, we have *generator expressions* as a fast way of building simple generators:

In [22]:
```
esquare = (v ** 2 for v in [1, 2, 3, 4, 5])
type(esquare)
```

Out[22]:
```
generator
```

It's probably useful to compare that to what the list comprehension would have looked like: `[v ** 2 for v in [1, 2, 3, 4, 5]]` . But since we used a generator expression, we created a generator that needs to be iterated over, rather than a new list. We can then keep the chain going:

In [23]:
```
ediv = (v / 2 for v in esquare)
enegate = (-1 * v for v in ediv)
```

In [24]:
```
list(enegate)
```

Out[24]:
```
[-0.5, -2.0, -4.5, -8.0, -12.5]
```

In [ ]:
```

```