PIC 16, Winter 2018

Lecture 3F: Iterators

Friday, January 26, 2018

Matt Haberland

UCLA

Announcements

- Assignment 3M due
- Office hours delayed today (1:30 p.m. – 2:00 p.m.) today.
- The unofficial waiting list is still quite long and the roster does seem to be stabilizing.

Intended Learning Outcomes

By the end of lecture, students are intended to be able to:

- implement iterator behavior in custom classes so they can be looped through using the same convenient syntax as built-in containers;
- use generator *functions* and generator expressions to create generator objects; and
- use generator *objects* as iterators (and iterables).

Activities

- Finish assignment 3M
- Work on assignment 3W
- Start assignment 3F
- Activity: write a function `my_for` that accepts:
  - A container
  - A function that accepts one input

```
my_for(container, f)
```

and:

  - asks the container for an iterator, then repeatedly
  - invokes the iterator's `next` method and
  - invokes the function on the element returned
  - until `StopIteration` is raised

mimicking the behavior of a `for` loop.

UCLA

```
for x in container:
    f(x)
```

```
my_for(container, f)
```

Containers, Iterators, and Interables

- "Iterators" have a `next` method that
  - returns successive objects on successive invocations, and
  - `raise` a `StopIteration` exception when there are no more objects to return
- "Containers" store multiple pieces of information
  - They always have a `__contains__` member function
  - They usually have an `__iter__` method that returns an iterator, and thus, they are said to be *iterable*
- `for` loops call a container's `__iter__` method to get an iterator.
- They call the iterator's `next` method to get the next element in the container.
- Usually, a container is not its own iterator, but it certainly can be. In this case, what would its `__iter__` method return?

How a for loop works on `a_list`

- `a_list`
  - has an `__iter__` method that returns `an_iterator`
- `an_iterator`
  - has a `next` method that:
    - returns successive elements of `a_list` each time it is invoked
    - raises a `StopIteration` exception when there are no more
-
```
for el in a_list:
    do_something_with(el)
```
  - invokes `a_list`'s `__iter__` method to get `an_iterator`
  - repeatedly:
    - invokes `an_iterator`'s `next` method to get `el`, and
    - `do(es)_something_with(el)`
  - terminates when a `StopIteration` is raised

iterator_improv

How a for loop works on `a_list`

- `a_list`
  - has an `__iter__` method that returns at iterator, `an_iterator`
- `an_iterator`
  - has a `next` method that:
    - returns successive elements of `a_list` each time it is invoked
    - raises a `StopIteration` exception when there are no more
-
```
for el in a_list:
    does_something_with(el)
```
  - invokes `a_list`'s `__iter__` method to get `an_iterator`
  - repeatedly:
    - invokes `an_iterator`'s `next` method to get `el`, and
    - `do(es)_something_with(el)`
  - terminates when a `StopIteration` is raised

fin

Generator Functions

- A generator function is a compact way of defining an iterator.
- A generator function looks like a function that `yields` objects rather than `returning` them.

```
def f():
    yield 1
    yield 2
    yield 3
```

- The funny thing is that calling a generator function doesn't immediately execute the code it contains.
- Actually, when you invoke a generator function, it `returns` an generator object.

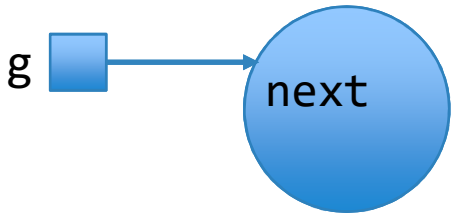
```
print f()
```

```
<generator object f at 0x0CF4B238>
```

Generator Objects

- A generator object is just an iterator defined by a generator function
  - What does that mean for it to be an iterator?
  - Hint: what method must this object have?
- The behavior of a generator object's `next` method is defined by the generator function
  - Code executes normally until a `yield` statement is reached
  - The `next` method `returns` whatever is `yielded`
  - Subsequent calls to `next` begin execution where previous calls left off
  - When there is no more code left, a `StopIteration` exception is raised

```
def f():                g = f()
    print 'hi',         print g.next()
    yield 1             g.next()
    yield 2             print g.next()
    yield 3             print g.next()
```




```
hi  1

3
Traceback (most recent call last):
<blah blah blah>
StopIteration
```

Generator Expressions

- This is an even more compact way of creating a generator object
- There's no need to think about all the details of generators when writing them
- Just think about writing a list comprehension, except:
  - this list comprehension won't immediately generate the list
  - instead, it will return a generator object
  - each time the generator object's `next` method is called, it will return one of the items that would have been in the list
- You write it exactly as you would a list comprehension, except that it's enclosed in parentheses
- Often generator expressions are used to create custom iterators for existing containers.

Why not a slice?

```
s = "supercalifragilisticexpialidocious"
for i in (s[i] for i in xrange(0,len(s),2):
    print i
```

xrange

- xrange works just like range, except
  - rather than returning a `list`, it returns a generator object and
  - the `next` method of the generator object returns elements of the range one by one.

```
for i in xrange(3):
    print i
```

```
0
1
2
```

- What is the advantage of xrange?
- When is it more appropriate than range? When is it not? Which case is more common?
- Accordingly, in Python 3, range works like Python 2's xrange

Are all iterators iterable?

- An iterator is an object with a `next` method
- An *iterable* object must have an `__iter__` method
- It is possible to write an iterator without an `__iter__` method

```
class Iterator:
    def __init__(self):
        self.i = -1

    def next(self):
        self.i += 1
        if self.i > 2:
            raise StopIteration()
        return self.i
```

```
class Iterable:
    def __iter__(self):
        return Iterator()

for i in [redacted]:
    print i
```

- So technically, according to these definitions, not all iterators are iterable
- But we can easily make an iterator iterable by giving it an `__iter__` method that returns `self`

Are generators iterable?

- To be called *iterable*, an object must have an `__iter__` method
- It turns out that all generator object automatically get an `__iter__` method (just like they have a `next` method defined by the generator function)
- Indeed, it returns `self`. Here's how you can tell:

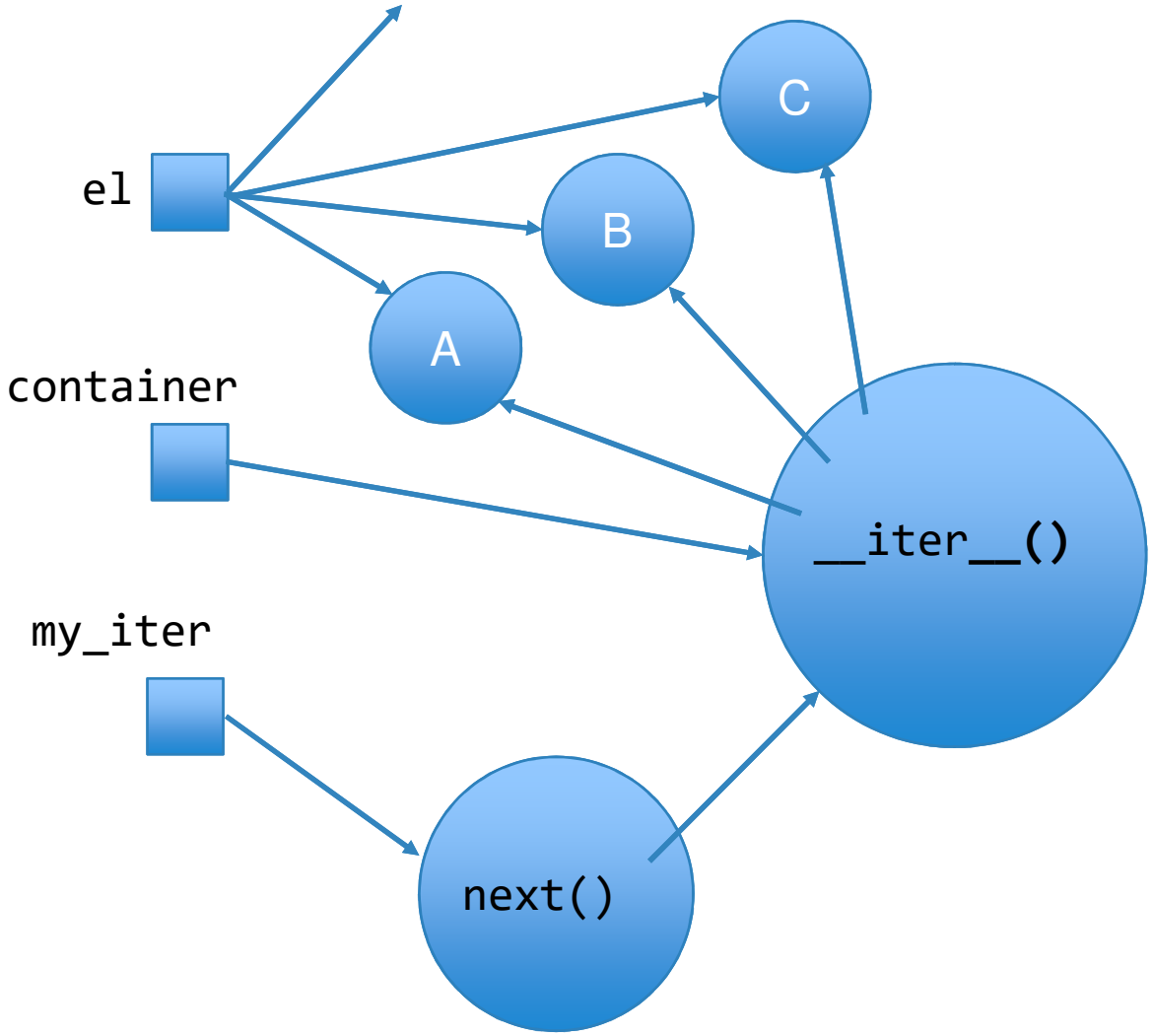
```
def f():
    yield 1
a = f()
b = a.__iter__()
c = iter(a)
print a is b is c
```

```
True
```

```
a = (x for x in [1])
b = iter(a)
c = a.__iter__()
print a is b is c
```

```
True
```

How a for loop works




```
for el in container:
    do_something_with(el)
```

1. built-in `iter` is invoked on the container, which effectively invokes the container's `__iter__` magic method
2. `__iter__` returns an iterator object
3. Until `StopIteration` is raised,
   - `el` gets whatever is returned by the iterator's next method
   - the contents of the loop are executed.