02-iteration

February 3, 2017

1 Eager and Lazy Evaluation

- Eager - do tasks all at once
- Lazy - do tasks incrementally, on demand
- pros and cons to each method
- lazy advantages
  - don't need to store things until they are used
  - don't make more than you need
  - don't make things and then throw them away if not needed
  - can simulate infinite lists
- for loops can iterate over eager and lazy sequences

```
In [1]: # 'for' will loop over an existing list

        x = [3,5,'sadf', .343]
        for e in x:
            print(e)
```

```
3
5
sadf
0.343
```

```
In [2]: # range just evaluates to itself

        r = range(4)
        r
```

```
Out[2]: range(0, 4)
```

```
In [4]: # a range holds the values of the original args

        [r.start, r.stop]
```

```
Out[4]: [0, 4]
```

```
In [5]: # can make a list out of it

        list(range(4))

Out[5]: [0, 1, 2, 3]

In [6]: # for iterates over integers specified by range

        for x in range(4):
            print(x)

0
1
2
3


In [7]: # note: here range specifies the iteration,
        # but a million element list is never created

        total = 0
        for x in range(1000000):
            total += x
        total

Out[7]: 499999500000

In [8]: # here the list that range specifies IS created

        rl = list(range(4))

        print(rl)

        for x in rl:
            print(x)

[0, 1, 2, 3]
0
1
2
3
```

2 Iterator Protocol

- there is a general protocol for iterating over objects
- use 'iter' function to get an iterator from an object
  - not all objects have iterators - for example, int and float don't

- the 'iterator' may be the same object, or a different one
- some objects allow multiple iterators simultaneously
- call 'next' function repeatedly, to get the elements of the iteration
- when all elements have been produced, iterator will raise a 'StopIteration' error each time 'next' is called
- 'StopIteration' implies the iterator is 'exhausted' - discard it.
- why raise an error at the end of the iteration???
- for loops use iterator protocol

```
In [9]: x = [1,4]
        xi = iter(x)
        xi

Out[9]: <list_iterator at 0x1069766a0>

In [10]: # 1st value

         next(xi)

Out[10]: 1

In [11]: # 2nd value

         next(xi)

Out[11]: 4

In [13]: # done

         next(xi)


        ---------------------------------------------------------------------------

        StopIteration                             Traceback (most recent call last)

        <ipython-input-13-b14d902b2c22> in <module>()
          1 # done
          2
    ----> 3 next(xi)


        StopIteration:


In [12]: # error!

         next(x1)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-12-d401ec890df2> in <module>()
  1 # error!
  2
----> 3 next(x1)


NameError: name 'x1' is not defined
```

```
In [14]: # 'range' each iterator is a new obj - can have any number of them

         r = range(2)
         ri = iter(r)
         ri2 = iter(r)

         [r, ri, ri2, ri is r, ri is ri2]
```

```
Out[14]: [range(0, 2),
          <range_iterator at 0x106928cc0>,
          <range_iterator at 0x106928ae0>,
          False,
          False]
```

```
In [15]: next(ri)
```

```
Out[15]: 0
```

```
In [16]: [next(ri), next(ri2)]
```

```
Out[16]: [1, 0]
```

```
In [17]: # now ri is ahead of ri2, so it finishes first
         next(ri)
```

```
---------------------------------------------------------------------------

StopIteration                             Traceback (most recent call last)

<ipython-input-17-df74aee3c876> in <module>()
  1 # now ri is ahead of ri2, so it finishes first
----> 2 next(ri)


StopIteration:
```

```
In [18]: # one val left for ri2

         next(ri2)

Out[18]: 1

In [19]: # now ri2 is done

         next(ri2)


        ---------------------------------------------------------------------------

        StopIteration                             Traceback (most recent call last)

        <ipython-input-19-bf72214af584> in <module>()
          1 # now ri2 is done
          2
    ----> 3 next(ri2)


        StopIteration:
```

3 Generator Function

- one way to define an iterator
- a generator is defined by using a 'yield' statement inside a 'def'
- executing the function returns the iterator
- falling off the end of the function, or executing a 'return' statement, will terminate the generator.
- once a generator terminates, it is 'exhausted', and can not be used again
- calling 'next' on a generator will cause the generator to execute until it hits a 'yield' statement. The arg supplied to 'yield' will be returned by 'next'. The next time 'next' is called on the generator, the generator will resume executing on the statement following the yield.
- all local variable values are preserved between between 'next' calls to the generator

```
In [20]: # executing this function will return a generator

         def gf(n):
             for j in range(n):
                 yield(j)

In [21]: gf(5)

Out[21]: <generator object gf at 0x10694a518>

In [22]: # 'list' will run generator until it is exhausted.

         list(gf(5))
```

```
Out[22]: [0, 1, 2, 3, 4]

In [23]: # or can use returned generator explicitly via iteration protocol

         g = gf(2)
         g

Out[23]: <generator object gf at 0x10694ab48>

In [24]: next(g)

Out[24]: 0

In [25]: next(g)

Out[25]: 1

In [26]: # generator is finished - discard it

         next(g)


        ---------------------------------------------------------------------------
        StopIteration                             Traceback (most recent call last)

        <ipython-input-26-5e179d068219> in <module>()
          1 # generator is finished - discard it
          2
    ----> 3 next(g)


        StopIteration:


In [27]: # iterate over generator directly

         [j+10 for j in gf(3)]

Out[27]: [10, 11, 12]
```

4 A generator can represent an infinite sequence (sort of)

- eager approach can't work - not possible to make a list of all the even integers
- but in some sense lazy approach can represent that list with a generator, by supplying as many as are asked for

```
In [28]: def infinite(start, incr):
             e = start
             # this generator will never terminate
             while True:
                 yield(e)
                 e += incr

In [29]: # eg represents the positive even numbers

         eg = infinite(2,2)
         [next(eg) for j in range(5)]

Out[29]: [2, 4, 6, 8, 10]

In [30]: # a generator can use another generator

         def evenPowersOf2():
             eg = infinite(2,2)
             while True:
                 e = next(eg)
                 yield 2**e

         ep2 = evenPowersOf2()
         [next(ep2) for j in range(5)]

Out[30]: [4, 16, 64, 256, 1024]

In [32]: [next(ep2) for j in range(5)]

Out[32]: [4194304, 16777216, 67108864, 268435456, 1073741824]

In [33]: import operator

         # add series

         eg = infinite(2,2)
         g5 = infinite(5,5)

         # generators can use other generators

         def opgen(op, g1, g2):
             while True:
                 e1 = next(g1)
                 e2 = next(g2)
                 yield op(e1,e2)

         og = opgen(operator.add, eg, g5)

         [next(og) for j in range(5)]
```

```
Out[33]: [7, 14, 21, 28, 35]

In [34]: # subtract infinite series

         eg = infinite(2,2)
         g5 = infinite(5,5)
         og = opgen(operator.sub, eg, g5)

         [next(og) for j in range(5)]

Out[34]: [-3, -6, -9, -12, -15]
```

5 Yields do not have to be inside a loop

- fibonacci series is 1,1,2,3,5,8...
- f(0) = 1
- f(1) = 1
- f(n) = f(n-1) + f(n-2)

```
In [35]: def fibonacci():
             # easy way to handle the first two ones
             yield(1)
             yield(1)
             last = 1
             last2 = 1
             while True:
                 sum = last + last2
                 yield(sum)
                 last2 = last
                 last = sum

         f = fibonacci()

         for j in range(10):
             print( next(f))
```

```
1
1
2
3
5
8
13
21
34
55
```

6 Modifying a Running Generator

- can change generator state at any time

```
In [36]: def counter(maximum):
             cnt = 0
             while cnt < maximum:
                 # peculiar syntax
                 val = (yield cnt)
                 # If value provided, change counter
                 if val is not None:
                     cnt = val
                 else:
                     cnt += 1

In [37]: c = counter(1000)
         [next(c) for j in range(10)]

Out[37]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [38]: # change the 'cnt' variable that the generator saves
         # '(yield cnt)' in generator will return 300

         c.send(300)

         # generator continues from new value
         [next(c) for j in range(10)]

Out[38]: [301, 302, 303, 304, 305, 306, 307, 308, 309, 310]

In [39]: # the generator is nowhere near done, but we can terminate it

         c.close()

In [40]: # the generator is exhausted now

         next(c)


        ---------------------------------------------------------------------------

        StopIteration                             Traceback (most recent call last)

        <ipython-input-40-00f233f05b6b> in <module>()
          1 # the generator is exhausted now
          2
    ----> 3 next(c)


        StopIteration:
```

7 Generator Expression

- an expression that evaluates to a generator
- looks like a list comprehension, but with outer '()' instead of '[]'

```
In [41]: def ge(n):
             # can't return a 'def'
             return ( j**2 for j in range(2, n) if j != 3)

         g = ge(8)

In [42]: # pick first two manually

         next(g)

Out[42]: 4

In [43]: # skipped j == 3

         next(g)

Out[43]: 16

In [44]: # for gets the rest

         for j in g:
             print(j)

25
36
49
```

8 suppose want to sum 1,000,000 squares. . .

```
In [45]: # could do

         mil = 1000**2

         sq = [x**2 for x in range(mil)]
         sum(sq)

Out[45]: 333332833333500000

In [46]: # or

         total = 0
         for x in range(mil):
             total += x**2
         total
```

```
Out[46]: 333332833333500000

In [47]: # could use a generator

         # which way is better?

         sum(x**2 for x in range(mil))

Out[47]: 333332833333500000
```

9 A generator will finish if it calls a generator that finishes

```
In [48]: def g(n):
             for j in range(n):
                 yield j

         def g2(n):
             gen = g(n)
             while True:
                 yield next(gen)

In [49]: list(g2(3))

Out[49]: [0, 1, 2]

In [50]: # generate chars

         def chars(s):
             for c in s:
                 yield c

         cs = chars('larry')
         for c in cs:
             print(c)


l
a
r
r
y


In [51]: # 'yield from' will yield everything from its generator argument

         def gfrom(g):
             yield from g
```

```
gs = gfrom(chars('larry'))

for c in gs:
    print(c)
```

```
l
a
r
r
y
```