Content delivered in class_12_20-August-2016

- Chapter 8: Generators and Iterators
  - Iterators
  - Iteration (Iter) protocol
  - Generators
  - Generator Expressions
  - Itertools

Interview Questions Discussed

Interview Question 1: What is the difference between yield and return?

Assignments Given

Assignment 1: Try the iter() protocol for frozenset

Assignment 2: Try to get the dictionary pair in dictionary-keyiterator object

---

Iterators

- Process to iterate through all the elements of a collection

In [2]:
```
for i in range(7):
    print 'Hey ',
```

```
Hey  Hey  Hey  Hey  Hey  Hey  Hey
```

In [3]:
```
for i in [12, 23, 34, 54, 56]:
    print i,
```

```
12 23 34 54 56
```

In [4]:
```
print [char for char in 'Python Programming']
```

```
['P', 'y', 't', 'h', 'o', 'n', ' ', 'P', 'r', 'o', 'g', 'r', 'a', 'm', 'm',
 'i', 'n', 'g']
```

In [5]:
```
for key in {'a': 'Apple', 'b': 'Ball'}:
    print key,
```

```
a b
```

The default iterator for dictionary is keys()

In [6]:
```
d = {'a': 'Apple', 'b': 'Ball'}
for key, value in d.items():
    print key, value
```

```
a Apple
b Ball
```

Also, iterators can be used in other ways

In [7]:
```
'-'.join(['Date', 'Month', 'Year'])
```

Out[7]:
```
'Date-Month-Year'
```

In [8]:
```
'-'.join({'Date':8, 'Month':4, 'Year': 2016})
```

Out[8]:
```
'Date-Year-Month'
```

In [9]:
```
list('Programming')
```

Out[9]:
```
['P', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g']
```

In [10]:
```
list({'Date':8, 'Month':4, 'Year': 2016})
```

Out[10]:
```
['Date', 'Year', 'Month']
```

Iteration (Iter) protocol

iter() - takes an iterable object and retuns an iterator

next() - method call to retun elements from the iterator. Results in StopIterator error, if the elements are not present

In [11]:
```
li = iter([12,23,34])  # list iterator
print li, type(li)
```

```
<listiterator object at 0x047723F0> <type 'listiterator'>
```

In [12]:
```
l = [12, 23, 45, 56]
print l, type(l)

li = iter(l)
print li, type(li)
```

```
[12, 23, 45, 56] <type 'list'>
<listiterator object at 0x04772450> <type 'listiterator'>
```

In [22]:
```
print dir(li)
```

```
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__
hash__', '__init__', '__iter__', '__length_hint__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subc
lasshook__', 'next']
```

In [13]:
```
print li.next()
```

```
12
```

In [14]:
```
print li.next()
```

```
23
```

In [15]:
```
print li.next()
```

```
45
```

In [16]:
```
print li.next()
```

```
56
```

In [17]:
```
print li.next()   # As there are no more values in it
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-17-6c5692cfc806> in <module>()
----> 1 print li.next()

StopIteration:
```

In [18]:
```
t = (12, 23, 45, 56)
print t, type(t)

ti = iter(t)         # tuple iterator
print ti, type(ti)
```

```
(12, 23, 45, 56) <type 'tuple'>
<tupleiterator object at 0x04852490> <type 'tupleiterator'>
```

In [19]:
```
ti.next()
```

Out[19]:
```
12
```

In [21]:
```
print dir(ti)
```

```
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__
hash__', '__init__', '__iter__', '__length_hint__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subc
lasshook__', 'next']
```

In [23]:
```
s = {12,23,34}
print s, type(s)

si = iter(s)         # set iterator
print si, type(si)
```

```
set([34, 12, 23]) <type 'set'>
<setiterator object at 0x047EFA80> <type 'setiterator'>
```

In [24]:
```
print si.next()
```

```
34
```

In [25]:
```
print dir(si)
```

```
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__
hash__', '__init__', '__iter__', '__length_hint__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subc
lasshook__', 'next']
```

Assignment 1: Try the iter() protocol for frozenset

In [26]:
```
d = {'a':12, 'b':23, 'c':34}
print d, type(d)

di = iter(d)         # dictionary iterator
print di, type(di)
```

```
{'a': 12, 'c': 34, 'b': 23} <type 'dict'>
<dictionary-keyiterator object at 0x047F1D50> <type 'dictionary-keyiterator'>
```

In [27]:
```
print di.next()
```

```
a
```

In [28]:
```
print dir(di)
```

```
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__
hash__', '__init__', '__iter__', '__length_hint__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subc
lasshook__', 'next']
```

Assignment 2: Try to get the dictionary pair in dictionary-keyiterator object

Generators

- It simplifies the creation of iterators
- It is a function that returns a sequence of results, rather than a single result.
- If a function uses the 'yield' keyword, it creates a generator object.
- yield is different from return.

In [29]:
```
def count(n):
    print "Stating to count!"
    i = 0
    while i<n:
        yield i
        i+=1
    #return     # PEP8 strongly discourages usage of yield and retun, in same
 function
```

In [30]:
```
c = count(3)
```

In [31]:
```
print c
```

```
<generator object count at 0x047EF8A0>
```

In [32]:
```
c.next()
```

```
Stating to count!
```

Out[32]:
```
0
```

In [33]:
```
print c.next()
print c.next()
```

```
1
2
```

In [34]:
```
c.next()     # because there are no more values
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-34-b7f29180fded> in <module>()
----> 1 c.next()     # because there are no more values

StopIteration:
```

This function doesn't get executed when the function call is made; but executed when the next() method call is done.

In [35]:
```
def foo():
    print "Start the function!"
    for i in range(3):
        print "before yield", i
        yield i
        print "after yield", i
    print "end of function "
```

In [36]:
```
f = foo()
```

In [37]:
```
f.next()
```

```
Start the function!
before yield 0
```

Out[37]:
```
0
```

In [38]:
```
f.next()
```

```
after yield 0
before yield 1
```

Out[38]:
```
1
```

In [39]:
```
f.next()
```

```
after yield 1
before yield 2
```

Out[39]:
```
2
```

Interview Question 1: What is the difference between yield and return?

yield will halt the execution, untill the next next() method is encountered. Where as *return* will return the result at only, and won't go back to the function

Generator function terminates by calling either *return* or by raising StopIteration error.

It is not recommended to place both yield and return for the same function.

In [40]:
```
def yrange(n):
    i = 0
    while i<n:
        yield i
        i+=1
```

In [41]:
```
y = yrange(3)
```

In [42]:
```
print y, type(y)
```

```
<generator object yrange at 0x04857788> <type 'generator'>
```

In [43]:
```
y.next()
```

Out[43]:
```
0
```

In [44]:
```
y.next()
```

Out[44]:
```
1
```

In [45]:
```
def integers():
    """Infinite sequence of integers."""
    i = 1
    while True:
        yield i
        i = i + 1

def squares():
    for i in integers():
        yield i * i

def take(n, seq):
    """Returns first n values from the given sequence."""
    seq = iter(seq)
    result = []
    try:
        for i in range(n):
            result.append(seq.next())
    except StopIteration:
        pass
    return result

print take(5, squares()) # prints [1, 4, 9, 16, 25]
```

```
[1, 4, 9, 16, 25]
```

In [46]:
```
#!/usr/bin/python
# Purpose: To make a Fibonacci generator.
# fibGenerator.py

def fibonacci(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1


if __name__ == '__main__':
    fib10 = fibonacci(10)
    for i in fib10:
        print i,
```

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

Generator Expressions

- tuple comprehension
  - It is generator version of list comprehension.
- List comprehension creates a sequence that contains the resulting data. Generator expression creates a generator that knows how to produce data on demand.
- Generator Expression (GE) improves performance and memory usage
- GE creates objects, which can't be indexed.

```
syntax:
    (expression for item1 in iterable1
                for item2 in iterable2
                for item3 in iterable3
                .
                .
                .
                for itemN in iterableN
                if condition)
```

In [47]:
```
b = (10*i for i in [1,2,3,4])
```

In [48]:
```
print b, type(b)
```

```
<generator object <genexpr> at 0x04699A30> <type 'generator'>
```

In [49]:
```
b.next()
```

Out[49]: 10

In [50]:
```
c = list(b)   # Generator expression to list conversion
```

In [51]:
```
sum(i*i for i in range(10))
```

Out[51]: 285

Itertools

- chain - chains multiple iterators together
- izip - iterable version of zip
- product - computes the cartesian product of input iterables

```
product(A,B) is same as ((x,y) for x in A for y in B)
```

In [52]:
```
import itertools
```

In [53]:
```
li1 = iter([1,2,3])
li2 = iter([4,5,6])
a = itertools.chain(li1, li2)
```

In [54]:
```
a, list(a)
```

Out[54]: (<itertools.chain at 0x4864090>, [1, 2, 3, 4, 5, 6])

In [55]:
```
list(itertools.chain(['ABC', 'DEF']))
```

Out[55]: ['ABC', 'DEF']

In [56]:
```
list(itertools.chain.from_iterable(['ABC', 'DEF']))
```

Out[56]: ['A', 'B', 'C', 'D', 'E', 'F']

Assignment 1: Try to do multi-dimensional list to single dimensional list, or list flattening, using itertools

In [57]:
```
for x, y in itertools.izip(["a", "b", "c"], [1, 2, 3]):
        print x,y
```

```
a 1
b 2
c 3
```

In [58]:
```
list(itertools.izip_longest('abcd', 'ABCD', fillvalue='-'))
```

Out[58]: [('a', 'A'), ('b', 'B'), ('c', 'C'), ('d', 'D')]

In [59]:
```
list(itertools.izip_longest('abcd', 'AB', fillvalue='-'))
```

Out[59]: [('a', 'A'), ('b', 'B'), ('c', '-'), ('d', '-')]

In [60]:
```
list(itertools.izip_longest('ab', 'ABCD', fillvalue='-'))
```

Out[60]:
```
[('a', 'A'), ('b', 'B'), ('-', 'C'), ('-', 'D')]
```

In [61]:
```
print list(itertools.product([1,2,3], repeat = 2))
```

```
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

In [62]:
```
print list(itertools.product([1,2,3], repeat = 1))
```

```
[(1,), (2,), (3,)]
```

In [63]:
```
print list(itertools.product([1,2,3], repeat = 0))
```

```
[()]
```

In [64]:
```
print list(itertools.product([1,2,3], repeat = 3))
```

```
[(1, 1, 1), (1, 1, 2), (1, 1, 3), (1, 2, 1), (1, 2, 2), (1, 2, 3), (1, 3, 1),
 (1, 3, 2), (1, 3, 3), (2, 1, 1), (2, 1, 2), (2, 1, 3), (2, 2, 1), (2, 2, 2),
 (2, 2, 3), (2, 3, 1), (2, 3, 2), (2, 3, 3), (3, 1, 1), (3, 1, 2), (3, 1, 3),
 (3, 2, 1), (3, 2, 2), (3, 2, 3), (3, 3, 1), (3, 3, 2), (3, 3, 3)]
```

In [65]:
```
print list(itertools.product([1,2,3],[3,4]))
```

```
[(1, 3), (1, 4), (2, 3), (2, 4), (3, 3), (3, 4)]
```

In [66]:
```
s = [[1,2,3],[3,4,5]]
print list(itertools.product(*s))
```

```
[(1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 3), (3, 4), (3, 5)]
```

In [67]:
```
print list(itertools.product(s))
```

```
[([1, 2, 3],), ([3, 4, 5],)]
```

In [68]:
```
s = [(1,2,3),[3,4,5]]   # non-homegeneous list
print list(itertools.product(*s))
```

```
[(1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 3), (3, 4), (3, 5)]
```

In [69]:
```
s = [(1,2,(45,78,9),3),[3,4,[33, 44],5]]  # multi-dimensional list
print list(itertools.product(*s))
```

```
[(1, 3), (1, 4), (1, [33, 44]), (1, 5), (2, 3), (2, 4), (2, [33, 44]), (2,
 5), ((45, 78, 9), 3), ((45, 78, 9), 4), ((45, 78, 9), [33, 44]), ((45, 78,
 9), 5), (3, 3), (3, 4), (3, [33, 44]), (3, 5)]
```

In [70]:
```
t = ((1,2,(45,78,9),3),[3,4,[33, 44],5])  # multi-dimensional tuple
print list(itertools.product(*t))         # displaying as a list
```

```
[(1, 3), (1, 4), (1, [33, 44]), (1, 5), (2, 3), (2, 4), (2, [33, 44]), (2,
 5), ((45, 78, 9), 3), ((45, 78, 9), 4), ((45, 78, 9), [33, 44]), ((45, 78,
 9), 5), (3, 3), (3, 4), (3, [33, 44]), (3, 5)]
```

In [71]:
```
print tuple(itertools.product(*t))        # displaying as a tuple
```

```
((1, 3), (1, 4), (1, [33, 44]), (1, 5), (2, 3), (2, 4), (2, [33, 44]), (2,
 5), ((45, 78, 9), 3), ((45, 78, 9), 4), ((45, 78, 9), [33, 44]), ((45, 78,
 9), 5), (3, 3), (3, 4), (3, [33, 44]), (3, 5))
```

In [72]:
```
list(itertools.permutations('AB',2))
```

Out[72]:
```
[('A', 'B'), ('B', 'A')]
```

In [73]:
```
list(itertools.combinations('AB',2))
```

Out[73]:
```
[('A', 'B')]
```

In [74]:
```
list(itertools.combinations_with_replacement('AB',2))
```

Out[74]:
```
[('A', 'A'), ('A', 'B'), ('B', 'B')]
```

In [75]:
```
list(itertools.permutations('ABC',2))
```

Out[75]:
```
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
```

In [76]:
```
list(itertools.combinations('ABC',2))
```

Out[76]:
```
[('A', 'B'), ('A', 'C'), ('B', 'C')]
```

In [77]:
```
list(itertools.combinations_with_replacement('ABC',2))
```

Out[77]:
```
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
```

In [78]:
```
list(itertools.compress('ABCDEF', [1,0,1,0,1,1]))
```

Out[78]:
```
['A', 'C', 'E', 'F']
```

In [79]:
```
list(itertools.compress('ABCDEF', [1,0,1,0,1,1,1]))
```

Out[79]:
```
['A', 'C', 'E', 'F']
```

In [80]:
```
list(itertools.compress('ABCDEF', [0, 1,0,1,0,1,1,1]))
```

Out[80]:
```
['B', 'D', 'F']
```