Iterators, Generators

Iterators

An ITERABLE is:

- anything that can be looped over (i.e. you can loop over a string or file)
- anything that can appear on the right-side of a for-loop: `for x in iterable: ...`
- anything you can call with `iter()` have it return an ITERATOR: `iter(obj)`
- an object that defines `__iter__` that returns a fresh ITERATOR, or it may have a `__getitem__` method suitable for indexed lookup.

```
s = 'cat'
```

```
for si in s:
    print si
```

```
c
a
t
```

```
s.__getitem__
```

```
<method-wrapper '__getitem__' of str object at 0x10
```

```
n = [1,4,16,32]
```

```
for ni in n:
  print ni,
```

```
1 4 16 32
```

```
iter(n)
```

```
<listiterator at 0x10deebcd0>
```

```
n.__iter__
```

```
<method-wrapper '__iter__' of list object at 0x10d7
```

```
s.__iter__
```

```
---------------------------------------------------
-------------
AttributeError                              Traceback
nt call last)
```

```
for k in {"x": 1, "y": 2}:
    print k
```

```
y
x
```

```
for line in open("/Users/beerli/a.txt"):
    print line
```

```
The quick fox jumps of the lazy dog

The red fox is running away
```

```
>>> ",".join(["a", "b", "c"])
'a,b,c'
>>> ",".join({"x": 1, "y": 2})
'y,x'
>>> list("python")
['p', 'y', 't', 'h', 'o', 'n']
>>> list({"x": 1, "y": 2})
['y', 'x']
```

Many functions consume iterables

An ITERATOR is:

- an object with state that remembers where it is during iteration
- an object with a `__next__` method (Python 3; `next` before) that:
  - returns the next value in the iteration
  - updates the state to point at the next value
  - signals when it is done by raising `StopIteration`
- an object that is self-iterable (meaning that it has an `__iter__` method that returns self).

```
>>> x = iter([1, 2, 3])
>>> x
<listiterator object at 0x1004ca850>
>>> x.next()
1
>>> x.next()
2
>>> x.next()
3
>>> x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```python
niter = iter(n)
```

```python
for ni in niter:
    print ni
```

```
1
4
16
32
```

Iterators are implemented as classes. Here is an iterator that works like built-in `xrange` function.

```
class yrange:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def next(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

The `__iter__` method is what makes an object iterable. Behind the scenes, the *iter* function calls `__iter__` method on the given object.

The return value of `__iter__` is an iterator. It should have a `next` method and raise `StopIteration` when there are no more elements.

```
y = yrange(4)
```

```
y.next()
```

```
0
```

```
y.next()
```

```
1
```

```
y.next()
```

```
2
```

```
y.next()
```

```
3
```

```
y.next()
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-27-75a92ee8313a> in <module>()
----> 1 y.next()
```

How would we write a reverse iterator given our class yrange ?

```
class yrange:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def next(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

```
>>> it = reverse_iter([1, 2, 3, 4])
>>> it.next()
4
>>> it.next()
3
>>> it.next()
2
>>> it.next()
1
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print char
...
m
a
p
s
```

Generators

facilitate the construction of iterators

```
def zrange(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

http://anandology.com/python-practice-book/iterators.html

```python
z = zrange(3)
```

```python
z
```

```
<generator object zrange at 0x10def4500>
```

```python
z.next()
```

```
0
```

```python
z.next()
```

```
1
```

```python
z.next()
```

```
2
```

```python
z.next()
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-34-6c49e4c11a56> in <module>()
----> 1 z.next()

StopIteration:
```

So a generator is also an iterator. You don't have to worry about the iterator protocol.

The word "generator" is confusingly used to mean both the function that generates and what it generates.

We will use the word "generator" to mean the generated object and "generator function" to mean the function that generates it.

Can you think about how it is working internally?

When a generator function is called, it returns a generator object without even beginning execution of the function. When *next* method is called for the first time, the function starts executing until it reaches `yield` statement. The yielded value is returned by the `next` call.
The following example demonstrates the interplay between `yield` and call to `next` method on generator object.

http://anandology.com/python-practice-book/iterators.html

```
>>> def foo():
...     print "begin"
...     for i in range(3):
...         print "before yield", i
...         yield i
...         print "after yield", i
...     print "end"
...
>>> f = foo()
>>> f.next()
begin
before yield 0
0
>>> f.next()
after yield 0
before yield 1
1
>>> f.next()
after yield 1
before yield 2
2
>>> f.next()
after yield 2
end
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

http://anandology.com/python-practice-book/iterators.html

```python
def integers():
    """Infinite sequence of integers."""
    i = 0
    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())
```

```
[0, 1, 4, 9, 16]
```

```python
print take(6,integers())
```

```
[0, 1, 2, 3, 4, 5]
```

```python
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
```

```python
def fibonacci():
    last = 1
    secondlast = 0
    yield 1
    for i in integers():
        yield (secondlast + last)
        secondlast, last = last, secondlast + last
```

```python
print take(20,fibonacci())
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
```

5.3. Generator Expressions

Generator Expressions are generator version of list comprehensions. They look like list comprehensions, but returns a generator back instead of a list.

```
>>> a = (x*x for x in range(10))
>>> a
<generator object <genexpr> at 0x401f08>
>>> sum(a)
285
```

We can use the generator expressions as arguments to various functions that consume iterators.

```
>>> sum((x*x for x in range(10)))
285
```

When there is only one argument to the calling function, the parenthesis around generator expression can be omitted.

http://anandology.com/python-practice-book/iterators.html

Another example:
Lets say we want to find first 10 (or any n) pythagorian triplets.
A triplet `(x, y, z)` is called pythagorian triplet if `x*x + y*y == z*z`.

It is easy to solve this problem if we know till what value of *z* to test for.
But we want to find first n pythagorian triplets.

```
pyt = ((x, y, z) for z in integers() for y in xrange(1, z) for x in range(1, y) if x*x + y*y == z*z)
```

```
take(10, pyt)
```

```
[(3, 4, 5),
 (6, 8, 10),
 (5, 12, 13),
 (9, 12, 15),
 (8, 15, 17),
 (12, 16, 20),
 (15, 20, 25),
 (7, 24, 25),
 (10, 24, 26),
 (20, 21, 29)]
```

http://anandology.com/python-practice-book/iterators.html

5.3.1. Example: Reading multiple files

Lets say we want to write a program that takes a list of filenames as arguments and prints contents of all those files, like `cat` command in unix.

The traditional way to implement it is:

```
def cat(filenames):
    for f in filenames:
        for line in open(f):
            print line,
```

Now, lets say we want to print only the line which has a particular substring, like `grep` command in unix.

```
def grep(pattern, filenames):
    for f in filenames:
        for line in open(f):
            if pattern in line:
                print line,
```

Both these programs have lot of code in common. It is hard to move the common part to a function. But with generators makes it possible to do it.

http://anandology.com/python-practice-book/iterators.html

```
def readfiles(filenames):
    for f in filenames:
        for line in open(f):
            yield line

def grep(pattern, lines):
    return (line for line in lines if pattern in lines)

def printlines(lines):
    for line in lines:
        print line,

def main(pattern, filenames):
    lines = readfiles(filenames)
    lines = grep(pattern, lines)
    printlines(lines)
```

The code is much simpler now with each function doing one small thing. We can move all these functions into a separate module and reuse it in other programs.

http://anandology.com/python-practice-book/iterators.html

5.4. Itertools

The itertools module in the standard library provides lot of intersting tools to work with iterators.

Lets look at some of the interesting functions.

chain – chains multiple iterators together.

```
>>> it1 = iter([1, 2, 3])
>>> it2 = iter([4, 5, 6])
>>> itertools.chain(it1, it2)
[1, 2, 3, 4, 5, 6]
```

izip – iterable version of zip

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

http://anandology.com/python-practice-book/iterators.html

```
import itertools
```

```
dir(itertools)
```

```
['__doc__',
 '__file__',
 '__name__',
 '__package__',
 'chain',
 'combinations',
 'combinations_with_replacement',
 'compress',
 'count',
 'cycle',
 'dropwhile',
 'groupby',
 'ifilter',
 'ifilterfalse',
 'imap',
 'islice',
```

to scroll output; double click to hide

```
 'izip_longest',
 'permutations',
 'product',
 'repeat',
 'starmap',
 'takewhile',
 'tee']
```

```
help(itertools)
```

```
Help on module itertools:

NAME
    itertools - Functional tools for creating and using iterators.
```

ology.com/python-practice-book/iterators.html

Problem 2: Write a program that takes one or more filenames as arguments and prints all the lines which are longer than 40 characters.

Problem 3: Write a function `findfiles` that recursively descends the directory tree for the specified directory and generates paths of all the files in the tree.

Problem 4: Write a function to compute the number of python files (.py extension) in a specified directory recursively.

Problem 5: Write a function to compute the total number of lines of code in all python files in the specified directory recursively.

Problem 6: Write a function to compute the total number of lines of code, ignoring empty and comment lines, in all python files in the specified directory recursively.

Problem 7: Write a program `split.py`, that takes an integer `n` and a filename as command line arguments and splits the file into multiple small files with each having `n` lines.

http://anandology.com/python-practice-book/iterators.html