Iterators

Iterators

For loop is used a lot in Python. One can iterate over almost every type of collection.
How is this made possible?

```
for element in (1, 2, 3):
        print element
for element in (1, 2, 3):
        print element
for key in {'one':1, 'two':2}:
        print key
for char in "123":
        print char
for line in open("myfile.txt"):
        print line
```

Iterators

- When for statement is called, a method `iter` is called on the object.
- This returns an object on which, the method `next` is implemented (which can go through the items)
- `next` keeps on giving elements, one by one.
- When there are no more elements, an exception `StopIteration` is raised. (Loop stops)

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    it.next()
StopIteration
```

How to make Iterable Classes?

To make a collection (personal class) iterable:

- It needs to have the method `__iter__` implemented. This is the function which enables `iter` to be called.
- `__iter__` should return and object with `next` implemented.
- Example below.

```python
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)
```

```
>>> for char in Reverse('spam'):
...         print char
...
m
a
p
s
```

Advantages/Disadvantages

- When we have Iterator implemented on an object, the for loop would not copy the object. So, especially for large collections, this is advantageous.
- Troubles: When the list (collection) has to be changed, an iterator can lead to catastrophe.
- In case of lists, use slicing. (Example)

I want all the squares upto 121 (not single digit) and I want also every double digit square + 30.

```

>>> lis = (x**2 for x in range(4, 12))
>>>
>>>
>>> for i in lis:
...     if i < 100:
...         lis.append(i+30)
...
>>> lis
(16, 25, 36, 49, 64, 81, 100, 121, 46,
 55, 66, 79, 94, 111, 76, 85, 96, 109,
 124, 106, 115, 126)
>>>
>>>
```

```
>>> lis = (x**2 for x in range(4, 12))
>>> for i in lis[:]:
...         if i < 100:
...                 lis.append(i+30)
...
>>> lis
(16, 25, 36, 49, 64, 81, 100, 121, 46,
                55, 66, 79, 94, 111)
>>>
```

Applied/Used

- In for slices
- In list comprehensions, for expressions
- in If operators `if x in this`
- In almost all the collections.
- More efficient than copying.

In Dicts

- `iter(d)` gives an iterator over the keys of the dict
    1. `d.iterkeys`
    2. `d.itervalues`
    3. `d.iteritems`
- Iterators over `d.keys, d.values, d.items`
- No lists are created.

Generators

Generators

- Iterator creators (So to speak)
- Regular functions which return without returning.
- Uses `yield` statement
- Each call of `next` resumes from where it left off.
- State/Data values stored

```

def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

>>> for char in reverse('golf'):
...     print char
...
f
l
o
g
```

Generators ...

- Generators are the equivalent of class based Iterators
- `__iter__` and `next` are created automatically
- Saving the vales makes it easier. No need to separate initialization/storage of index.
- Automatic raising of Exception on termination.

Simulating Generators

- Can be simulated with normal functions.
  1. Start with an empty list.
  2. Fill in the list instead of the `yield` statement
  3. Then return an iterator of the list
  4. Same result

```python
def r(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]


def rS(data):
    _list = []
    for index in range(len(data)-1, -1, -1):
        _list.append(data[index])
    return iter(_list)
```

```
>>> import gs
>>> for x in gs.r('this is cool'):
...         print x,
...
l o o c   s i   s i h t
>>> for x in gs.rS('this is cool'):
...         print x,
...
l o o c   s i   s i h t
>>>
>>>
```

Socket Programming

Sockets

- API for inter process communication
- An integer, a thing called socket and methods for the same
- Different machines/processes
- Berkely
- In python as well

Server

1. create a socket
2. bind the socket to an address and port
3. listen for incoming connections
4. wait for clients
5. accept a client
6. send and receive data

```python
import socket

host = ''
port = 50000
backlog = 5
size = 1024
s = socket.socket(socket.AF_INET,
                  socket.SOCK_STREAM)
s.bind((host,port))
s.listen(backlog)
while 1:
    client, address = s.accept()
    data = client.recv(size)
    if data:
        client.send(data)
    client.close()
```

Client

1. create a socket
2. connect to the server
3. send and receive data

```python
import socket

host = 'localhost'
port = 50000
size = 1024
s = socket.socket(socket.AF_INET,
                  socket.SOCK_STREAM)
s.connect((host,port))
s.send('Hello, world')
data = s.recv(size)
s.close()
print 'Received:', data
————————
(sadanand@lxmayr10 \@ ˜)python    client.py
Received: Hello, world
(sadanand@lxmayr10 \@ ˜)
```

To Note

- In `recv`, one might not get all the data from the server in a single go. In such a case, a loop until data received in None is advised.
- If the server dies, then the client will hang (almost) (as good as)

A word about sockets

- Blocking Sockets: The socket is blocked until the request is satisfied. When the remote system writes on to it, the operation is complated and execution resumes.
- Non Blocking Sockets: Error conditions are to be handled properly. Doesn't wait for the remote system. It will be informed.

Zero Knowledge Proofs

Graphs

- Hamiltonian Path/Cycle
- Graph Isomorphism
- Going in the Cave

Problems

- Small client for HTTP
- Implement check-for-hamiltonian
- Infinite Iterator on a list
- .. . .

Forward

- Static methods
- Decorators
- Threading