ITERATORS

PYTHON

FAST ENUMERATION


```
for a in [4,2,1,7,5,9]:
    print a
```

```
for a in "hello world":
    print a
```

```
myFile = open(”myFile.txt")
for currLine in myFile:
    print currLine
```

PYTHON

ITERATION PROTOCOL

Informator
A SOPRANO COMPANY

A simple for loop as this one:

```
for element in iterable:
    [DO SOMETHING]
```

is implemeted as:

```
while True:
    try:
        element = next(iter_obj)
        [DO SOMETHING]
    except StopIteration:
        break
```

PYTHON

ITERATION PROTOCOL

Informator
A SOPRANO COMPANY

PEP234 defines an iteration interface for objects in Python. (from year 2001)

For loops existed before so there must be alternative ways of handling iterations.

There are basically four ways of creating iterative ”functions”

- Create a function that Python can iterate over on its own (__getitem__)
- Create an iterator following the iteration interface (__iter__, __next__/next)
- Use a generator expression
- Create a generator

ITERATION PROTOCOL

Essentially, the protocol for iteration is as follows:

1. Check for an __iter__ method. (Then go for the new iteration protocol)
2. Otherwise, try calling the __getitem__ with successively larger integer values until it raises an IndexError.

Let us start with an ”old” __getitem__ iteration example.

PYTHON

__GETITEM__

Informator
A SOPRANO COMPANY

```
class oldIterTest():

    def __init__(self, text):
        self.text = text

    def __getitem__(self, index):
        result = self.text[index].upper()
        return result

a = oldIterTest('Hello world')

for x in a:
    print x
```

”This technique works for index-based stuff like sequences”

PYTHON

ITER( )

Informator
A SOPRANO COMPANY

The iter()-function is used to get an **iterator** from an **iterable** object.

```
myList = [1,3,5]
a = iter(myList)

print a.next()                    #  1
print a.next()                    #  3
print a.next()                    #  5
print a.next()                    #  StopIteration exception
```

next() returns the next element until it raises the **StopIteration** exception.

ITER( )

Before we move on:

```
a = iter(myList)
```

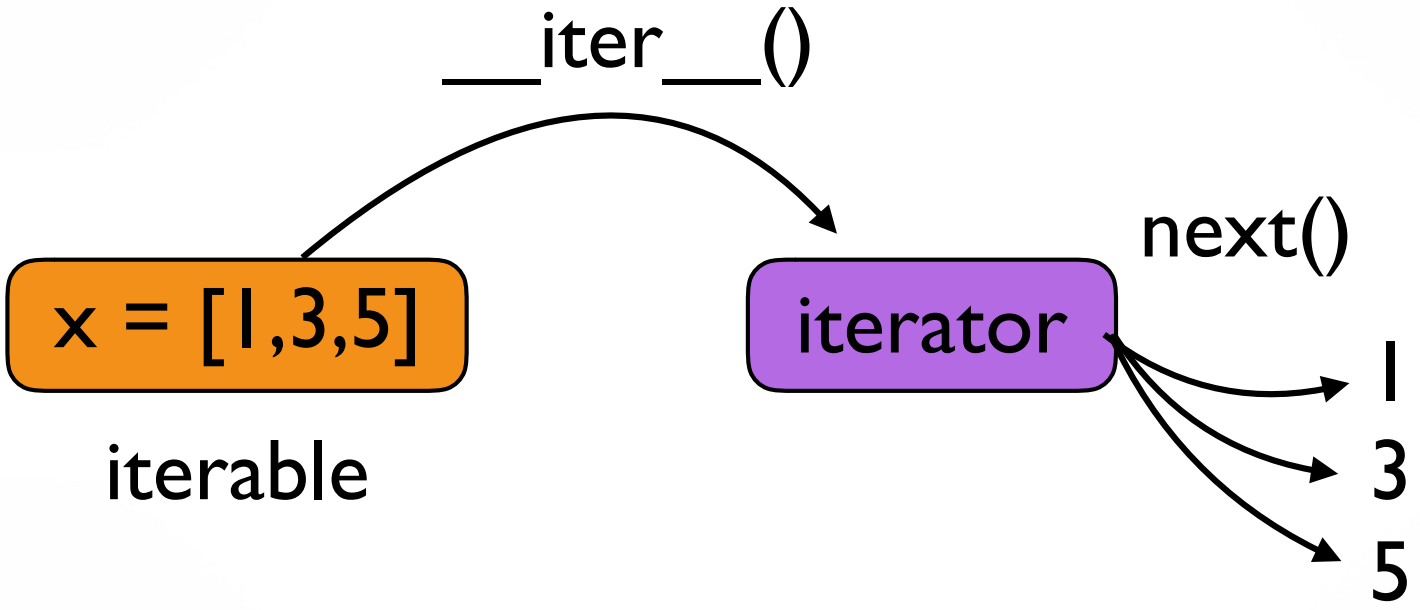
The code above is just an alternative way of writing:

```
a = myList.__iter__()
```

”myList” is an **iterable**

”a” is our binding to an **iterator**

ITERATION PROTOCOL



ITERABLES & ITERATORS

We are talking about two different things.

Iterables and **Iterators**

- Iterables return iterators using the __iter__() method
- Iterators implement the next() method __next__() in Python 3.x
- Iterators return self using the __iter__() method if a separate object

…. if a separate object ?? We save that situation for a few minutes.

ITERABLES & ITERATORS


```
class createRange(object):          # iterable/iterator in same obj

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

    def __iter__(self):                 # iterables return iterators
        return self

    def next(self):                     # iterators implements next()
        if self.i < self.n:
            i = self.i
            self.i += 2
            return i
        else:
            raise StopIteration()
```

PYTHON

EXERCISE

Informator
A SOFRANO COMPANY

Rewrite the class below so it uses the modern ”iterator interface” using the __iter__() and the next() method.

```
class oldIterTest():

    def __init__(self, text):
        self.text = text

    def __getitem__(self, index):
        result = self.text[index].upper()
        return result
```

EXTRA EXERCISE

Informator
A SOFRANO COMPANY

Rewrite the new version of your oldIterTest class so it iterates over the data backwards.

ITERABLES & ITERATORS

Often the **iterator** and the **iterable** is the same object.

But the **iterator** and the **iterable** could be two **different objects.**

Why do we want separate objects?

ITERABLES & ITERATORS

Let’s pretend we have a linked list as an iterable.

If the linked list also was an iterator we was stucked with having one iteration over the list at a time.

Using a separate iterator-object give you more possibilities and you don’t have to bother if another piece of the code is iterating the same data.

ITERABLES & ITERATORS

The **iterable:**

```
class newRange(object):

    def __init__(self, n):
        self.n = n

    def __iter__(self):
       return newRange_iterator(self.n)
```

And here's the **iterator**

ITERABLES & ITERATORS


```
class newRange_iterator(object):

    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()
```

PYTHON

EXERCISE

Create an iterable with a separate iterator.

Make sure you could do multiple iterations over the iterable at the same time.

Check your iterable by doing something like:

```
a = myIterable(10)
for x in a:
    print "Outer: " + x
    for z in a:
        print z
```

PYTHON

INFINITE LOOPS

Informator
A SOFRANO COMPANY

```
class InfiniteIteration:

    def __iter__(self):
        self.num = 1
        return self

    def __next__(self):
        num = self.num
        self.num += 2
        return num
        if self.num > 15:
            raise StopIteration
```