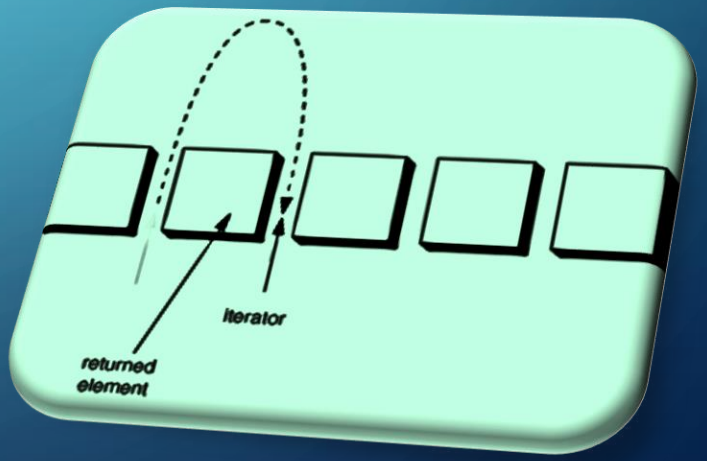
ITERATORS AND GENERATORS

__ITER__, __NEXT__, __GETITEM__



Iterators

- Iterators are objects that permit iteration over a collection.
- Some iteratables are simple objects and not a collection.
- An iterable object contains a ‘__iter__’ method that returns an iterable.
- An iterator may also contain ‘__next__’. Next moves to the ‘next’ position in the iterable.

Automatic Usage

- Generally it’s not necessary to call __next__ and __iter__ directly.
- Python will call them for you automatically if you use list or ‘for’ comprehensions.
- If you need to directly call them, use the built-in functions ‘next’ and ‘iter’ functions.

__iter__

- In general, the __iter__ function always returns ‘self’ because ‘self’ is contains the iterable collection.

```
def __iter__(self):
    return self
```

__next__

- The __next__ returns the next element in the sequence.

```
def __next__(self):
    y = self.n
    self.n += 1
    return y
```

__getitem__

- The __getitem__ is Python's way of overloading the [ ] operator.

```
def __getitem__(self,index):
        self.index += 1
        return self.items[index]
```

StopIteration

- Python relies on Exceptions to stop an iteration. When a the end of the sequence is reached, raise a StopIteration() exception:

```
if self.count > self.limit:
    raise StopIteration()
```

IndexError

- If __getitem__ is used and the end of a sequence is reached, raise a IndexError() exception.

```
if self.index > self.limit:
    raise IndexError()
```

Generators

- Generators are functions that create sequences of results.
- Generators maintain their local state so it can iterate to the next position.
- The generator state is maintained using the ‘yield’ keyword.

Yield

- Yield is a keyword that is used like return, except the function will return a generator.
- When a generator is started, it runs until a yield statement is hit.
- It then returns the object.
- The generator will resume iteration using the previous state of the iterator.

Example

```
def createGenerator():
        mylist = range(3)
        for m in mylist:
                yield m*m

Output:  0, 1, 4
```

How FOR works

- For loop:

```
for element in iterable:
    # do something with element
```

- Implementation:

```
iter_obj = iter(iterable)
while True:
    try:
        element = next(iter_obj)
    except StopIteration:
        break
```

Summary

- Iteration is a process implying iterables (implementing the __iter__() method) and iterators (implementing the __next__() method).
- Iterators are objects that let you iterate on iterables.