**PYTHON WEEK 4 ITERATORS AND GENERATOR**

**Iterators**

In any programming environment, it will always be useful to iterate. By now you have probably noticed that most container objects can be looped over using a for statement:

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, end='')

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the for statement calls iter() on the container object. The function returns an iterator object that defines the method \_\_next\_\_() which accesses elements in the container one at a time. When there are no more elements, \_\_next\_\_() raises a StopIteration exception which tells the for loop to terminate. You can call the \_\_next\_\_() method using the next() built-in function; this example shows how it all works:

 >>>

>>> s = 'abc'

>>> it = iter(s)

>>> it

<iterator object at 0x00A1DB50>

>>> next(it)

'a'

>>> next(it)

'b'

>>> next(it)

'c'

>>> next(it)

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

next(it)

StopIteration

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an \_\_iter\_\_() method which returns an object with a \_\_next\_\_() method. If the class defines \_\_next\_\_(), then \_\_iter\_\_() can just return self:

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**

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the yield statement whenever they want to return data. Each time next() is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

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

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the \_\_iter\_\_() and \_\_next\_\_() methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like self.index and self.data.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise StopIteration. In combination, these features make it easy to create iterators with no more effort than writing a regular function.