Iterators and Generators

Manually Consuming an Iterable-1

```
with open('/etc/passwd') as f:
 try:
   while True:
       line = next(f) # Manually fetch the next line
       print(line, end='')
 except StopIteration:  # signals the end of iteration.
   pass
```

Manually Consuming an Iterable-2

```
with open('/etc/passwd') as f:
 try:
   while True:
      line = next(f,None)  # Manually fetch the next line
      if line is None:
         break
      print(line, end='')
```

Delegating Iteration

_children=[]
Node

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)
```

iter() returns `_children.__iter__()`

```
if __name__ == '__main__':
 root = Node(0)
 child1 = Node(1)
 child2 = Node(2)
 root.add_child(child1)
 root.add_child(child2)
 for ch in root:
   print(ch)
 # Outputs Node(1), Node(2)
```

__iter__() method forwards the iteration request to the internally held _children attribute.

Custom Iterator using Generator

frange()

```
def frange(start, stop, increment):
    x = start
    while x < stop:
        yield x
        x += increment
```

yield statement in a function turns it into a generator.

generators only run in response to iteration

```
for n in frange(0, 4, 0.5):
... print(n)
...
0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
>>> list(frange(0, 1, 0.125))
[0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875]
>>>
```

Custom Iterator using Generator

countdown()

```
>>> def countdown(n):
...     print('Starting to count from', n)
...     while n > 0:
...             yield n
...             n -= 1
...     print('Done!')
...
```

>>> # Run to first yield and emit a value

>>> next(c)

Starting to count from 3

3

>>> # Create the generator, notice no output appears

>>> c = countdown(3)

>>> c

<generator object countdown at 0x1006a0af0>

generator function only runs in response to next

for takes care of these details

Implementing the Iterator Protocol

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)
```

```
    def depth_first(self):
        yield self
        for c in self:
            yield from c.depth_first()

# Example
if __name__ == '__main__':
    root = Node(0)
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    child1.add_child(Node(3))
    child1.add_child(Node(4))
    child2.add_child(Node(5))

    for ch in root.depth_first():
        print(ch)
```

Manually Implementing the Iterator Protocol

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, other_node):
        self._children.append(other_node)

    def __iter__(self):
        return iter(self._children)

    def depth_first(self):
        return DepthFirstIterator(self)

class DepthFirstIterator(object):
    '''
    Depth-first traversal
    '''
    def __init__(self, start_node):
        self._node = start_node
        self._children_iter = None
        self._child_iter = None

    def __iter__(self):
        return self
```

Manually Implementing the Iterator Protocol

```
def __next__(self):
    # Return myself if just started; create an iterator for children
    if self._children_iter is None:
        self._children_iter = iter(self._node)
        return self._node

    # If processing a child, return its next item
    elif self._child_iter:
        try:
            nextchild = next(self._child_iter)
            return nextchild
        except StopIteration:
            self._child_iter = None
            return next(self)

    # Advance to the next child and start its iteration
    else:
        self._child_iter = next(self._children_iter).depth_first()
        return next(self)
```

Iterate Reverse

```
>>> a = [1, 2, 3, 4]
>>> for x in reversed(a):
...     print(x)
...
```


```
# Print a file backwards
f = open('somefile')
for line in reversed(list(f)):
    print(line, end='')
```

Reversed iteration can be customized on user-defined classes if they implement the __reversed__() method.

Iterate Reverse - overloaded

```
class Countdown:
    def __init__(self, start):
        self.start = start

    # Forward iterator
    def __iter__(self):
        n = self.start
        while n > 0:
            yield n
            n -= 1
```

It's no longer necessary to pull the data into a list and iterate in reverse on the list.

```
# Reverse iterator
def __reversed__(self):
    n = 1
    while n <= self.start:
        yield n
        n += 1
```

Iterating and Slicing !!

```
>>> def count(n):
...     while True:
...             yield n
...             n += 1
...
>>> c = count(0)
>>> c[10:20]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not subscriptable
```

Iterators are typically used for giving one data item after another.

What if we have to access a slice of them??

The generator here doesn't help!!!

Iterating and Slicing !!

```
>>> def count(n):
...     while True:
...             yield n
...             n += 1
...
>>> c = count(0)
>>> c[10:20]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not subscriptable
```

```
>>> # Now using islice()
>>> import itertools
>>> for x in itertools.islice(c, 10, 20):
...     print(x)
...
```

How islice() works

Iterators and generators can’t normally be sliced

because no information is known about their length (and they don’t implement indexing).

The result of islice() is an iterator that produces the desired slice items,

How? Consumes and discards all of the items up to the starting slice index.

Items are then produced by the islice object until the ending index has been reached

Note: islice() will consume data on the supplied iterator

Skipping the First Part of an Iterable - itertools.dropwhile()

```
>>> with open('/etc/passwd') as f:
...     for line in f:
...         print(line, end='')
...
##
# User Database
#
# Note that this file is consulted directly only when the system is running
# in single-user mode.  At other times, this information is provided by
# Open Directory.
...
##
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
...
>>>
```

Task:

Skip the comments at the beginning while reading

First cut solution

```
with open('/etc/passwd') as f:
    # Skip over initial comments
    while True:
        line = next(f, '')
        if not line.startswith('#'):
            break

    # Process remaining lines
    while line:
        # Replace with useful processing
        print(line, end='')
        line = next(f, None)
```

```
with open('/etc/passwd') as f:
    lines = (line for line in f if not line.startswith('#'))
```

Skipping the First Part of an Iterable - itertools.dropwhile()

```
>>> from itertools import islice
>>> items = ['a', 'b', 'c', 1, 4, 10, 15]
>>> for x in islice(items, 3, None):
...     print(x)
...
1
4
10
15
>>>
```

--> Items[3:]

```
>>> from itertools import dropwhile
>>> with open('/etc/passwd') as f:
...     for line in dropwhile(lambda line: line.startswith('#'), f):
...         print(line, end='')
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
...
>>>
```

Permutations and Combinations

```
>>> items = ['a', 'b', 'c']
>>> from itertools import permutations
>>> for p in permutations(items):
...     print(p)
...
('a', 'b', 'c')
('a', 'c', 'b')
('b', 'a', 'c')
('b', 'c', 'a')
('c', 'a', 'b')
('c', 'b', 'a')
>>>
```

```
>>> for p in permutations(items, 2):
...     print(p)
...
('a', 'b')
('a', 'c')
```

```
>>> from itertools import combinations
>>> for c in combinations(items, 3):
...     print(c)
...
('a', 'b', 'c')
>>> for c in combinations(items, 2):
...     print(c)
...
('a', 'b')
('a', 'c')
('b', 'c')
>>> for c in combinations(items, 1):
...     print(c)
...
('a',)
('b',)
('c',)
>>>
```