Iterators & Generators

Intro2CS – week 10
Iterators

• Objects that repeatedly return new values

```python
x = MySimpleIterator()
print(next(x))
print(next(x))
print(next(x))
print(next(x))
```

• Built-in function `next()` calls `__next__()` on the iterator
Example

class MySimpleIterator:
    def __init__(self):
        self.__counter = 0

    def __next__(self):
        self.__counter += 1
        return self.__counter - 1

x = MySimpleIterator()
print(next(x))
print(next(x))
print(next(x))
print(next(x))
Iterators and Iterables

- Another built-in function: `iter()` creates an iterator

```python
In[2]: names = ['Alice', 'Bob', 'Charlie']
In[3]: x = iter(names)
In[4]: next(x)
Out[4]: 'Alice'
In[5]: next(x)
Out[5]: 'Bob'
In[6]: next(x)
Out[6]: 'Charlie'
In[7]: next(x)
Traceback (most recent call last):
  File "C:\WinPython-64bit-3.3.5.0\python-3.3.5.amd64\lib\site-packages\code_obj", line 1, in <module>
next(x)
StopIteration
```
• Lists are “iterable”: we can create iterators for them

• For loops work on anything that is iterable

```python
name_list = ["Jack","Jill","John","Julia"]

for item in name_list:
    print(item)
```
Iterables and iterators

Returns an iterator that goes over this object.

Makes the class “iterable”.

Iterators are also “iterable”.

Returns the next item.
name_list = ["Jack","Jill","John","Julia"]

for item in name_list:
    print(item)

for item in iter(name_list):
    print(item)

• What for loops really do:
  – Get an iterable object
  – Create an iterator for it
  – Repeatedly call next() on it (until StopIteration is raised)
Primes Iterator

class PrimesIter():
    def __init__(self, max_num):

    def __iter__(self):

    def __next__(self):

    def is_prime(num):

def is_prime(num):
    for i in range(2, int(num**(1/2)) + 1):
        if num % i == 0:
            return False
    return True
class PrimesIter():
    def __init__(self, max_num):
        self.__max = max_num
        self.__cur = 1

    def __iter__(self):
        return self
This is the way to notify that we are out of values.
Generators

• The easiest way to create iterators

```python
def simple_generator():
    yield "my"
    yield "simple"
    yield "generator"

my_iter = simple_generator()
print(next(my_iter))
print(next(my_iter))
print(next(my_iter))
```

Similar to “return”, but next time will continue from where we stopped
A generator for primes

```python
def primes_iter(max_val):
    for i in range(2, max_val):
        if is_prime(i):
            yield i

print([i for i in primes_iter(100)])
```
• Let’s also add an iterator to the Stack class, so that we can run over it with loops. (The iterator here doesn’t remove items)

We create the iterator using a generator:

```python
def __iter__(self):
    cur = self.__head
    while cur:
        yield cur.data
        cur = cur.next
```
A Tree Iterator

class TreeNode:
    def __init__(self, data=None, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right

    def __iter__(self):
        yield self.data
        if self.left:
            for item in self.left: yield item
        if self.right:
            for item in self.right: yield item
my_tree = TreeNode("a", TreeNode("b"),
                 TreeNode("c"))

for item in my_tree:
    print(item)

• Will print:
  a
  b
  c
def power_set_iter(full_set):
    if len(full_set) == 0:
        yield set()
    else:
        item = full_set.pop()
        for subs in power_set_iter(full_set):
            yield subs | {item}
        yield subs

for i in power_set_iter({1,2,3,4}):
    print(i, end=" ")