CMSC 245: Principles of Programming Languages

Lab#8: Doing Leonardo Fibonacci's Numbers in Python – Using Iterators & Generators!

The Fibonacci Series is defined as shown below:

$$F(1) = 1$$
$$F(2) = 1$$
$$F(n) = F(n-2) + F(n-1)$$

Thus, you get the series:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

To compute the nth Fibonacci number, you can write a **recursive** Python function as shown below:

```
# fibr(n) - Compute the nth Fibonacci number, recursively
def fibr(n):
    if n==1 or n==2:
        return 1
    else:
        return fibr(n-2) + fibr(n-1)
```

Enter and run this function (please, use python-3.X) to print out some of the values above. Then, try:

```
>>> fibr(20)
6765
>>> fibr(30)
832040
>>> fibr(40)
102334155
>>> fibr(50)
```

You will notice that computing `fibr(40)`, or `fibr(50)` starts to take a long time! You also know why this is so: there are too many recursive calls O(2^n) (is a lose upper bound). You may have to kill the last call since it takes so much time. We need to speed this up!

One way to do this is to write an **iterative version**. Go ahead and think about it. Try writing one yourself.

OK, here is one version:

```
# fibi(n) - Compute the nth Fibonacci number, iteratively
def fibi(n):
    if n==1 or n==2:
        return 1
    fibMinus2 = 1
    fibMinus1 = 1
    i = 3
    while (i <= n):
        fibI = fibMinus2 + fibMinus1
        fibMinus2 = fibMinus1
        fibMinus1 = fibI
        i = i + 1
    return fibI
```

And, some runs:

```
>>> fibi(20)
6765
>>> fibi(30)
832040
>>> fibi(40)
102334155
>>> fibi(50)
12586269025
>>> fibi(60)
1548008755920
```

It didn't take that long for the computer to compute the 50th, or 60th Fibonacci number, did it? Because we are not using any stack space. This is a simple iteration. And runs in time proportional to n – the number you are trying to compute.

Iterators in Python

Iterators are 'hidden' features in Python that you have used quite often. For example,

```
>>> numbers = [1,2,3,4,5]
>>> for i in numbers:
      print(i)
1
2
3
4
5
```

In the above, the list (sequence), `numbers` already have an iterator defined for it. All sequences in Python do. As do many other objects that you have used (files, e.g.). The

above is making use of the iterator for lists. You can explicitly create an iterator and use it:

```
>>> i = iter(numbers)
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
>>> next(i)
4
>>> next(i)
5
```

That is, by passing the list, `numbers` to `iter()` (which is a predefined Python function), you are creating a new **iterator object**. This can only be done if the object provided to `iter()` implements the `__iter__()` and `__next__()` methods (which a list does). Then, as shown above, after the creation of the iterator (using `iter()`), subsequent calls using `next()` to the object returned by `iter()` will return the next element in the sequence. What happens when, after the last call to next (shown above), we try to call `next()` again? Remember, the list only had 5 elements. Try it and you will see this:

```
>>> next(i)
5
>>> next(i)
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    next(i)
StopIteration
>>>
```

A `StopIteration` exception is raised because there are no more objects to be returned. That is, the iteration is over. Thus, when you do the following iteration:

```
>>> l = [1,2,3,4,5]
>>> for i in l:
      print(i)
1
2
3
4
5
```

The iteration stops because of the exception. All of this is 'hidden' under the Python hood!

Now, we can learn to make use of it for our own purposes. Any object in Python can be made to be **iterable** (as long as it makes sense to do so) by defining two methods for it: `__iter__()`, and `__next__()`.

```
# PowerOfTwo - Iterable object to compute powers of 2
class PowerOfTwo:
    def __init__(self, max = 0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n <= self.max:
            result = 2 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration
```

Study the above class carefully. First look at the constructor: it takes an argument. This argument defines the limit on how big a power of 2 it will compute. For example, you can create an iterator object that computes powers of 2 up to 3:

```
>>> i = PowerOfTwo(3)
>>> j = iter(i)
>>> next(j)
1
>>> next(j)
2
>>> next(j)
4
>>> next(j)
8
```

Or, you can simply do:

```
>>> for i in PowerOfTwo(4):
      print(i)
1
2
4
8
16
>>>
```

That is, you have just defined an **iterable object** that returns subsequent values in a series defined by you!

And so, speaking of series, let's apply this to our Fibonacci series:

```
# fibS - Defines a Fibonacci iterator
class fibS:
    def __init__(self, max=1):
        self.max=max
        self.fibMinus2=1
        self.fibMinus1=1
        self.n=1

    def __next__(self):
        if self.n <= self.max:
            self.fibOld = self.fibMinus2
            self.fibN = self.fibMinus2 + self.fibMinus1
            self.fibMinus2 = self.fibMinus1
            self.fibMinus1 = self.fibN
            self.n += 1
            return self.fibOld
        else:
            raise StopIteration

    def __iter__(self):
        return self
```

As before, please study the above carefully and then try it:

```
>>> fib = fibS(50)
>>> for i in fib:
      print(i)
1
1
2
3
5
8
13
21
34
…
1134903170
1836311903
2971215073
4807526976
7778742049
12586269025
>>>
```

Generators

Given that defining an iterator object requires defining a new class with the `__next__()` and `__iter__()` functions, sometimes it is convenient to create iterators on the fly. For that purpose, Python has **generators- they are functions that generate iterators!**

Generators are functions. Instead of using a return statement to return a result, they use the **yield** statement in its place. Moreover, after the function returns a value (i.e. yields a value) it 'remembers' its state. So that, in the next iteration, it picks up right after the yield statement. Confusing? Well, look at the generator below:

```
# fibg() - A generator for Fibonacci sequence
def fibg():
    fibMinus2 = 1
    fibMinus1 = 1

    while True:
            fibOld = fibMinus2
            fibI = fibMinus2 + fibMinus1
            fibMinus2 = fibMinus1
            fibMinus1 = fibI

            if fibOld > 6000000:
                    return

            yield fibOld
```

We are stopping the iteration (by returning from the function) after the number computed becomes greater than 6 million. Once written you can use a generator just like an iterator:

```
>>> for i in fibg():
      print(i)
1
1
2
3
…
3524578
5702887
```

Task#1: Modify the generator above to stop iteration after computing the nth Fibonacci number: for example:

```
>>> for i in fibg2(5):
      print(i)
1
1
2
3
5
```

Task#2: Write a generator `primes(n)` that produces the first n prime numbers:

```
>>> x = 1
>>> for i in primes(100):
      print(f"x:\t{i}")
      x += 1
1: 2
2: 3
3: 5
…
99: 523
100: 541
```