ECE 60146 Deep Learning **Name:** *Nikita Ravi (ravi30)*
Homework 1
01/13/2023

1 Introduction

The focus of this homework assignment was to gain some practice with some object oriented programming in python. The exercises involved inheritance and magic methods like the callable function for example.

2 Methodology

For this homework assignment, I created three classes called Sequence, Fibonacci, and Prime. The Sequence class was the base class and both Fibonacci and Prime are an extension of this class. The Sequence class behaved as an iterator, thus enabling the ability to iterate through the Fibonacci and Prime sequences generated based on the length provided. The following subsections dive into the purpose of each magic method.

2.1 ___init___ Function

The ___init___ method is similar to a constructor in Java or C++. This function is called everytime an object of the class is created to initialize the attributes of the object.

2.2 ___call___ Function

The ___call___ method is a callable function. A callable is any object that can be called like a function but has not been defined with a def statement.

2.3 ___len___ Function

The ___len___ method returns the length of the object called.

2.4 ___gt___ Function

The ___gt___ method is the greater than function which compares the attributes of the custom object and returns which one is greater. The function is called when executing the line object A > object B.

2.5 ___iter___ Function

A class must provide for an iterator for its instances to be iterable and this iterable must be returned by the ___iter___ method.

2.6 ___next___ Function

The ___next___ method returns the next element in the iterable object.

3 Implementation and Results

3.1 Task 1

Create a class named Sequence with an instance variable named array. The input parameter array is expected to be a list of numbers. This class will serve as the base class for the subclasses.

```
class Sequence(object):
    def __init__(self, array):
        self.array = array
```

Listing 1: Creating the Sequence class with init method

3.2 Task 2

Extend your Sequence class into a subclass called Fibonacci, with its ___init___ method taking in two input parameters: first_value and second_value. These two values will serve as the first two numbers in your Fibonacci sequence.

```
class Fibonacci(Sequence):
    def __init__(self, first_value, second_value):
        self.first_value = first_value
        self.second_value = second_value
        self.idx = -1
```

Listing 2: Creating the Fibonacci class with init method

3.3 Task 3

Further expand your Fibonacci class to make its instances callable. More specifically, after calling an instance of the Fibonacci class with an input parameter length, the instance variable array should store a Fibonacci sequence of that length and with the two aforementioned starting numbers. In addition, calling the instance should cause the computed Fibonacci sequence to be printed.

```
class Fibonacci(Sequence):
    def __init__(self, first_value, second_value):
        self.first_value = first_value
        self.second_value = second_value
        self.idx = -1

    def __call__(self, length):
        self.array = [0 for i in range(length)]
        self.array[0], self.array[1] = self.first_value, self.second_value

```

```
        idx = 2
        first, second = self.array[0], self.array[1]
        while(idx < length):
            next_value = first + second
            self.array[idx] = next_value

            first = second
            second = next_value
            idx += 1

        return self.array
```

Listing 3: Creating a callable function for Fibonacci

3.3.1 Results

Length	Results
5	1, 2, 3, 5, 8
8	1, 2, 3, 5, 8, 13, 21, 34

Table 1: Results of Fibonacci callable function with different lengths

3.4 Task 4

Modify your class definitions so that your Sequence instance can be used as an iterator. For example, when iterating through an instance of Fibonacci, the Fibonacci numbers should be returned one-by-one.

```
class Sequence(object):
    def __init__(self, array):
        self.array = array

    def get_number(self, i):
        return self.array[i]

    def __iter__(self):
        return self

    def __next__(self):
        self.idx += 1
        if(self.idx < len(self.array)):
            return self.array[self.idx]
        else:
            raise StopIteration

    next = __next__
```

Listing 4: Making the Sequence class into an iterator class

```
class Fibonacci(Sequence):
    def __init__(self, first_value, second_value):
        self.first_value = first_value
        self.second_value = second_value
        self.idx = -1

    def __call__(self, length):
        self.array = [0 for i in range(length)]
        self.array[0], self.array[1] = self.first_value, self.second_value

        idx = 2
        first, second = self.array[0], self.array[1]
        while(idx < length):
            next_value = first + second
            self.array[idx] = next_value

            first = second
            second = next_value
            idx += 1

        return self.array

    def __len__(self):
        return len(self.array)
```

Listing 5: Creating a ___len___ method for Fibonacci class

3.4.1 Results

Object with length	___len___ output	Iterator output
5	5	1, 2, 3, 5, 8
8	8	1, 2, 3, 5, 8, 13, 21, 34

Table 2: Results of the ___len___ method and Iterator class

3.5 Task 5

Make another subclass of the Sequence class named Prime. As the name suggests, the new class is identical to Fibonacci except that the array now stores consecutive prime numbers. Modify the class definition so that its instance is callable and can be used as an iterator.

```
class Prime(Sequence):
    def __init__(self):
        self.idx = -1

    def __call__(self, length):
        self.array = []
        idx = 0
        prime_number = 2
```

```
        while(idx < length):
            prime = True
            for div in range(2, prime_number):
                if(not prime_number % div):
                    prime = False

            if(prime):
                self.array.append(prime_number)
                idx += 1

            prime_number += 1

        return self.array

    def __len__(self):
        return len(self.array)
```

Listing 6: Creating the Prime class

3.5.1 Results

Objects with length	Callable output	___len___ output	Iterator output
5	2, 3, 5, 7, 11	5	2, 3, 5, 7, 11
8	2, 3, 5, 7, 11, 13, 17, 19	8	2, 3, 5, 7, 11, 13, 17, 19

Table 3: Results of Prime class

3.6 Task 6

Modify the base class Sequence such that two sequence instances of the same length can be compared by the operator $>$. Invoking $(A > B)$ should compare element-wise the two arrays and return the number of elements in A that are greater than the corresponding elements in B. If the two arrays are not of the same size, your code should throw a ValueError exception.

```
class Sequence(object):
    def __init__(self, array):
        self.array = array

    def get_number(self, i):
        return self.array[i]

    def __iter__(self):
        return self

    def __next__(self):
        self.idx += 1
        if(self.idx < len(self.array)):
```

```
            return self.array[self.idx]
        else:
            raise StopIteration

    next = __next__

    def __gt__(self, other):
        if(len(self.array) != len(other.array)):
            raise ValueError("Two arrays are not equal in length!")

        result = 0
        for x, y in zip(self.array, other.array):
            if(x > y):
                result += 1

        return result
```

Listing 7: Creating the ___gt___ method for Sequence class

3.6.1 Results

Fibonacci sequence	Prime sequence	___gt___ output
1, 2, 3, 5, 8, 13, 21, 34	2, 3, 5, 7, 11, 13, 17, 19	2
1, 2, 3, 5, 8, 13, 21, 34	2, 3, 5, 7, 11	ValueError: Two arrays are not equal in length!

Table 4: Results of the ___gt___ class