ECE60146: Homework 1

Alim Karimi

January 16, 2023

Question 1:

```
"""
Programming Tasks
1. Create a class named Sequence with an instance variable named array
as shown below:

"""

class Sequence (object): #note that object is a keyword. Every class is already by default a subclass of object
    def __init__( self , array ):
        self.array = array
```

Remarks for question 1: Copying what was given in assignment

Question 2:

```
"""
2. Now, 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):
        #Sequence.__init__(self, array) #initializing base class by initializing base class directly, but also
        #possible using the super method below!
        super(Fibonacci, self).__init__(array) #initialzing base class using super() method
        self.first_value = first_value
        self.second_value = second_value
```

Remarks for question 2: Derived class “Fibonacci” was created using the “super()” syntax. Constructor takes two values.

Question 3:

```
"""
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. Shown below is a demonstration
of the expected behaviour described so far:
1 FS = Fibonacci ( first_value =1 , second_value =2 )
2 FS ( length =5 ) # [1, 2, 3, 5, 8]
"""

class Fibonacci(Sequence):
    def __init__(self, first_value, second_value):
        self.first_value = first_value
        self.second_value = second_value
        #Sequence.__init__(self, [self.first_value, self.second_value]) #one way to initialize base class
        super(Fibonacci, self).__init__([self.first_value, self.second_value]) #how to do the line above, but with
        #super() syntax

    def __call__(self, length): #make Fibonacci instance callable with __call__
        for i in range(0, length - 2):
            self.array.append(self.array[-1] + self.array[-2]) #logic to compute value of FS
        print(self.array)
#Testing with reproduction:
FS = Fibonacci ( first_value =1 , second_value =2 )
FS(length = 5)
#Testing with own parameters:
FS = Fibonacci(3,5)
FS(length = 9)
```

```
[1, 2, 3, 5, 8]
[3, 5, 8, 13, 21, 34, 55, 89, 144]
```

Remarks for question 3: Fibonacci class is callable with __call__ and logic in the aforementioned method takes the length parameter and creates the Fibonacci sequence. Tested with own and lab's parameters.

Question 4:

```
"""
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.
The snippet below illustrates the expected behavior:
1 FS = Fibonacci ( first_value =1 , second_value =2 )
2 FS ( length =5 ) # [1, 2, 3, 5, 8]
3 print (len( FS ) ) # 5
4 print ([n for n in FS]) # [1, 2, 3, 5, 8]

"""

class Sequence (object): #note that object is a keyword. Every class is already by default a subclass of object
    def __init__( self , array ):
        self.array = array
        self.idx = -1 #start with -1 as the index for iteration, as __next__ when called will move the idx to 0
    def __iter__(self): #method so that a Sequence instance can be returned
        return self
    def __next__(self): #method so that a Sequence instance can be returned index by index
        self.idx = self.idx + 1
        if self.idx < len(self.array):
            return self.array[self.idx] #return the value in self.array held in the current element
        else:
            raise StopIteration
    def __len__(self):
        "This function is to return the length when len(FS) called"
        return len(self.array)

class Fibonacci(Sequence):
    def __init__(self, first_value, second_value):
        self.first_value = first_value
        self.second_value = second_value
        super(Fibonacci, self).__init__([self.first_value, self.second_value]) #how to do the line above, but with
        #super() syntax
    def __call__(self, length):   #make Fibonacci callable with __call__
        for i in range(0, length - 2):
            self.array.append(self.array[-1] + self.array[-2])
        print(self.array)
        return self.array
#Testing reproducing values:
FS = Fibonacci(first_value =1 , second_value =2)
FS(length = 5)
print(len(FS))
print ([n for n in FS])
#Testing with own parameters
FS = Fibonacci(first_value =2 , second_value =3)
FS(length = 9)
print(len(FS))
print([n for n in FS])
```

```
[1, 2, 3, 5, 8]
5
[1, 2, 3, 5, 8]
[2, 3, 5, 8, 13, 21, 34, 55, 89]
9
[2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Remarks for question 4: __iter__ and __next__ allow iterating through values of instantiated Fibonnaci object and the implementation in __len__ allows the printing of the length of the object. Tested with lab and own parameters.

Question 5:

```
"""
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.
What is shown below illustrates the expected behavior:
PS = Prime ()
PS ( length =8 ) # [2, 3, 5, 7, 11 , 13 , 17 , 19]
print (len( PS ) ) # 8
print ([n for n in PS]) # [2, 3, 5, 7, 11 , 13 , 17 , 19]

"""
class Prime(Sequence): #Prime inherts Sequence
    def __init__(self):
        #self.num = 1 #start with 1, it isn't a prime.
        self.idx = -1
        super(Prime, self).__init__([]) #initialize base class Sequence with an empty array
    def __call__(self, length):
        check_num = 2 #first possible prime is 2
        while (len(self.array) != length):
            prime = True #default is we want to add this to list. We want to find a condition where
            #number mod something other than itself and 1 is 0.
            if length == 1:
                self.array = [2]
            else:
                for x in range(2, check_num - 1):
                    if check_num % x == 0:
                        prime = False

            if (prime == True):
                self.array.append(check_num)
            check_num += 1 #go to the next integer and back to the top of the while loop. Check if that is a prime.

        print(self.array)
        return self.array

    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
#Testing reproducing values
PS = Prime()
PS(length = 8)
print (len(PS))
print([n for n in PS])
#Testing with own parameters
PS = Prime()
PS(length = 9)
print (len(PS))
print([n for n in PS])
```

```
[2, 3, 5, 7, 11, 13, 17, 19]
8
[2, 3, 5, 7, 11, 13, 17, 19]
[2, 3, 5, 7, 11, 13, 17, 19, 23]
9
[2, 3, 5, 7, 11, 13, 17, 19, 23]
```

Remarks for question 5: Sequence is a base class for Prime as well. Prime has a __call__, __iter__, and __next__ methods which allow Prime to have similar functionality to Fibonnaci. Logic is built in to check if every integer after 1 is a prime number until the amount of primes in the array is equal to the length parameter. Tested with lab and own parameters.

Question 6:

```
"""
6. Finally, 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. Shown below is an example:

FS = Fibonacci ( first_value =1 , second_value =2 )
FS ( length =8 ) # [1, 2, 3, 5, 8, 13 , 21 , 34]
PS = Prime ()
PS ( length =8 ) # [2, 3, 5, 7, 11 , 13 , 17 , 19]
print ( FS > PS ) # 2
PS ( length =5 ) # [2, 3, 5, 7, 11]
print ( FS > PS ) # will raise an error
# Traceback ( most recent call last ):
# ...
# ValueError : Two arrays are not equal in length !
"""

class Sequence (object): #note that object is a keyword. Every class is already by default a subclass of object
    def __init__( self , array ):
        self.array = array
        self.idx = -1
    def __iter__(self):
        return self
    def __next__(self):
        self.idx = self.idx + 1
        if self.idx < len(self.array):
            return self.array[self.idx]
        else:
            raise StopIteration
    def __len__(self):
        return len(self.array)

    def __gt__(self, other): #overload the > operator to count the number of elements in one array that are GT the
        if len(self.array) != len(other.array):
            raise ValueError('The arrays being compared need to be the same length')
        count_gt = 0
        for i in range(0, len(self.array)):
            if self.array[i] > other.array[i]:
                count_gt += 1
        return count_gt


class Fibonacci(Sequence):
    def __init__(self, first_value, second_value):
        self.first_value = first_value
        self.second_value = second_value
        #Sequence.__init__(self, [self.first_value, self.second_value]) #one way to initialize base class
        super(Fibonacci, self).__init__([self.first_value, self.second_value]) #how to do the line above, but with
    #make it callable with __call__
    def __call__(self, length):
        for i in range(0, length - 2):
            self.array.append(self.array[-1] + self.array[-2])
        print(self.array)
        return self.array

class Prime(Sequence): #Prime inherts Sequence
    def __init__(self):
        self.idx = -1
        super(Prime, self).__init__([]) #initialize base class Sequence with an empty array. Prime is a subclass o

    def __call__(self, length):
        check_num = 2 #first possible prime is 2
        self.array = []
        while (len(self.array) != length): #we want to check for primes until the length of the array is filled
            #with enough prime numbers
            prime = True #default is we want to add this to list. We want to find a condition where
            #number mod something other than itself and 1 is 0.
            if length == 1: #if the length is 1, then the only prime number we check is 2 (it is a prime) so we
                #add it to the array of prime numberes
                self.array = [2]
            else:    #check range of values from 2 to one less than the
                for x in range(2, check_num - 1):
                    if check_num % x == 0:
                        prime = False #if we find that a number that is not 1 or itself is divisible by the next n
                        #then the number we are checking is not a prime. Make the flag false.
            if (prime == True):
                self.array.append(check_num)
            check_num += 1 #go to the next integer and back to the top of the while loop. Check if that is a prime
        print(self.array)
        return self.array

    def __iter__(self):
        return self

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

Question 6, continued:

```
    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

#Testing reproducing values:
FS = Fibonacci( first_value =1 , second_value =2)
FS(length =8) # [1, 2, 3, 5, 8, 13 , 21 , 34]
PS = Prime()
PS(length =8) # [2, 3, 5, 7, 11 , 13 , 17 , 19]
print ( FS > PS ) # 2
PS(length =5) # [2, 3, 5, 7, 11]
print ( FS > PS ) # will raise an error

```

```
[1, 2, 3, 5, 8, 13, 21, 34]
[2, 3, 5, 7, 11, 13, 17, 19]
2
[2, 3, 5, 7, 11]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [18], in <cell line: 101>()
     99 print ( FS > PS ) # 2
    100 PS(length =5) # [2, 3, 5, 7, 11]
--> 101 print ( FS > PS )

Input In [18], in Sequence.__gt__(self, other)
     35 def __gt__(self, other): #overload the > operator to count the number of elements in one array that are GT th
e other
     36     if len(self.array) != len(other.array):
---> 37         raise ValueError('The arrays being compared need to be the same length')
     38     count_gt = 0
     39     for i in range(0, len(self.array)):

ValueError: The arrays being compared need to be the same length
```

```
#run part 6, but with my own parameters
FS = Fibonacci( first_value =1 , second_value =2)
FS(length =10)
PS = Prime()
PS(length =10)
print ( FS > PS )
PS(length =6) # [2, 3, 5, 7, 11, 13]
print ( FS > PS ) # will raise an error
```

```
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
4
[2, 3, 5, 7, 11, 13]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [19], in <cell line: 8>()
      6 print ( FS > PS )
      7 PS(length =6) # [2, 3, 5, 7, 11, 13]
----> 8 print ( FS > PS )

Input In [18], in Sequence.__gt__(self, other)
     35 def __gt__(self, other): #overload the > operator to count the number of elements in one array that are GT th
e other
     36     if len(self.array) != len(other.array):
---> 37         raise ValueError('The arrays being compared need to be the same length')
     38     count_gt = 0
     39     for i in range(0, len(self.array)):

ValueError: The arrays being compared need to be the same length
```

Remarks for question 6: The __gt__ method is overloaded and has logic built in to first check if the sequence lengths are the same (throws and error if they are not) and then does an

element wise comparison of the number of elements that are greater in the Fibonnaci sequence compared to the Prime sequence. Tested with my own and the lab's parameters.