ECE 60146 Deep Learning Homework 1

Mehmet Berk Sahin, sahinm@purdue.edu, 34740048

January 14, 2023

1 Introduction

In this homework, the basics of Python Object-Oriented programming are covered by implementing child and parent classes to use inheritance and by generating arbitrary iterators. In Python, all classes inherit object classes. Thus, all classes that are implemented are children of that class. Other than that I define a parent class, *Sequence*, and define its two child class, *Fibonacci* and *Prime*.

2 Methodology

I defined a parent class, *Sequence*, to represent the sequence objects. It holds the sequence entered by the user via input parameter *array* as a list variable, *self.array*. To compare any two sequences (Sequence objects), I overwrite to ">" operator by overwriting "__gt__()" magic method. Invoking $A > B$ makes an element-wise comparison between two arrays and returns the number of elements in A that are greater than the corresponding elements in B. If the two arrays have different lengths, the function throws a **ValueError** exception.

In the assignment, we used different types of sequences which are *Fibonacci* and *Prime* numbers. Since they are a kind of sequence, they inherit the *Sequence* class and add new features, which are the properties of their types, on it. *Sequence* class has two instance variables: first and second values. They are the initial values of the Fibonacci sequence. Knowing these two values one can generate a Fibonacci sequence with arbitrary length because the rule is the following:

$$F_n = F_{n-1} + F_{n-2} \tag{1}$$

where F_n is the n^{th} element of the sequence. 3 methods of *Fibonacci* class have been overwritten. "__call__(self, length)" method makes the *Fibonacci* object callable and it creates a *Fibonacci* sequence with the two initial variables and the given length. It saves the sequence as a list to *self.array* instance variable and prints the list. "__iter__(self)" makes the *Fibonacci* object iterable. Lastly, "__len__(self)" returns the length of the sequence when "len()" function is called on a *Sequence* instance.

Prime represents a sequence that consists of prime numbers. It has one instance variable, which is *self.array* and it keeps the sequence as a list in the memory. It has two instance methods that have been overwritten. "__call__(length)" makes the *Prime* object callable. Given the length, it generates a sequence consisting of prime numbers. It saves the sequence as a list to *self.array* and prints the sequence. "__iter__()" returns an iterable for the sequence.

Since *Prime* and *Fibonacci* classes' objects are iterable, and since they are both kinds of Sequence, I designed an iterator class, which is *SeqIterable*, for *Prime* and *Fibonacci* sequences. Since it is an iterator class, overwriting ”__iter__()” and ”__next__()” is sufficient. The former returns an iterable. The latter defines the way iteration occurs. It returns the next element in the sequence, in other words, the list.

In the assignment, we are asked to reproduce the outputs for each of the provided snippets in the homework with the given parameters and to produce the correct outputs with the input parameters of my choice. Thus, I added a prompt that asks the user to display to run the results in the homework or to run the results with the inputs of my choice. If the user enters 0, the former one will occur. If the user enters 1, the latter will occur. Otherwise, the program will ask the user until it enters a valid choice.

3 Implementation and Results

In this part, I will show screenshots of the relevant parts of my code to explain how I solve the question. Note that, I shorten the doc string of the classes to make the figures not too large. In the source code, you can see the full version.

Question 1

This question does not require any output. Sequence class asked in question 1 is implemented.

```python
class Sequence(object):
    """A class to represent a sequence...."""

    def __init__(self, array):
        self.array = array
```

Figure 1: Sequence class in Q1

This class will serve as the base class for the subclasses later in the upcoming questions.

Question 2

Sequence class created in question 1 has been extended into a subclass called *Fibonacci* with the initializer asked in the question. It can be seen in Figure 2.

```python
class Fibonacci(Sequence):
    """A class to represent Fibonacci sequences...."""

    def __init__(self, first_value, second_value):
        super(Fibonacci, self).__init__([])
        self.first_value = first_value
        self.second_value = second_value
```

Figure 2: Fibonacci class asked in Q2

Question 3

Fibonacci class is extended to make its instances *callable*. In particular, after *Fibonacci* object is initialized, it can be called with an input parameter *length* to create a sequence and store that sequence in an instance variable array. This sequence starts with the initial values entered at the initialization of the object. To achieve this, *Fibonacci* class is updated as follows: "__call__(self,

```
class Fibonacci(Sequence):
    """A class to represent Fibonacci sequences...."""

    def __init__(self, first_value, second_value):
        super(Fibonacci, self).__init__([])
        self.first_value = first_value
        self.second_value = second_value

    def __call__(self, length):
        fib_seq = []
        for i in range(length):
            if i == 0:
                fib_seq.append(self.first_value)
            elif i == 1:
                fib_seq.append(self.second_value)
            else:
                new_element = fib_seq[-1] + fib_seq[-2] # Fibonacci rule
                fib_seq.append(new_element)
        # Save the result and print it
        self.array = fib_seq
        print(self.array)
```

Figure 3: Fibonacci class updated for Q3

length)" magic method is overwritten to make the *Fibonacci* instances callable. It puts the first two elements to the list first, then it creates Fibonacci elements until the length limit is reached. To reproduce the results in the assignment, I wrote the same code as in the homework (Figure 4). Result is given in Figure 5 and it is correct. Also, I tried the code with the input parameters of

```
print("-"*10 + "Result of question 3" + "-"*10)
FS = Fibonacci(first_value=1, second_value=2)
FS(length=5)
print("-"*40)
```

Figure 4: Code snippet in Q3

```
----------Result of question 3----------
[1, 2, 3, 5, 8]
----------------------------------------
```

Figure 5: Result of code snippet in Q3

my choice as in Figure 6. Result of that code snippet can be seen in Figure 7. As it can be seen

```python
# Q3
print("-" * 10 + "Result of question 3" + "-" * 10)
FS = Fibonacci(first_value=1, second_value=5)
FS(length=7)
print("-" * 40)
```

Figure 6: Code snippet in Q3 with the parameters of my choice

```
----------Result of question 3----------
[1, 5, 6, 11, 17, 28, 45]
----------------------------------------
```

Figure 7: Result of the code snippet in Q3 with the parameters of my choice

from the output, the length of the sequence is equal to the length parameter. And the elements of the sequence obey the Fibonacci rule in equation (1).

Question 4

In this part, we are asked to modify the *Sequence* definition so that instances of that class can be used as an *iterator*. To achieve this, I update my *Sequence* class definition as in Figure 8. To make

```python
class Sequence(object):
    """A class to represent a sequence...."""

    def __init__(self, array):
        self.array = array

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

    def __iter__(self):
        return SeqIterable(self)

class SeqIterable(object):
    """Iterator class for Sequence class...."""
    def __init__(self, seq_obj):
        self.items = seq_obj.array
        self.index = -1

    def __iter__(self):
        return self

    def __next__(self):
        self.index += 1 # initialize index as 0
        # make sure index is in the range of the sequence
        if self.index < len(self.items):
            return self.items[self.index]
        else:
            raise StopIteration
```

Figure 8: Sequence class updated for Q4 and Iterator class has been added

class instances iterable, one should overwrite the "__iter__(self)" magic method. It should return an iterator for the *Sequence* class. To achieve this, one should define a separate iterator class for *Sequence*. So, I defined *SeqIterable* as an iterator to *Sequence* class. It has two instance variables: the list for the sequence and the index which indicates the iterator's current location. I started it from -1 because the instance is not used as an iterator yet. Since the instance of *SeqIterable* is iterable, I need to overwrite ”__iter__(self)” and it returns itself as an iterable. ”__next__(self)” controls how iteration is performed. It returns the next element in the sequence. If it reaches the end of the sequence, it stops the iteration. The code snippet for question 4 in the assignment is implemented in Figure 9. Result of this is given in Figure 10.

```
FS = Fibonacci(first_value=1, second_value=2)
FS(length=5)
print(len(FS))
print([n for n in FS])
```

Figure 9: Code snippet in Q4

```
----------Result of question 4----------
[1, 2, 3, 5, 8]
5
[1, 2, 3, 5, 8]
----------------------------------------
```

Figure 10: Result of the code snippet in Q4

Result is exactly the same as given in the homework, it is correct. Lastly, I tried the same code snippet with the parameters of my choice. The result and the implementation can be seen in Figure 11 and 12. As can be seen in the figures above result is correct. The length of the output sequences

```
FS = Fibonacci(first_value=1, second_value=1)
FS(length=8)
print(len(FS))
print([n for n in FS])
```

Figure 11: Code snippet in Q4 with the parameters of my choice

```
----------Result of question 4----------
[1, 1, 2, 3, 5, 8, 13, 21]
8
[1, 1, 2, 3, 5, 8, 13, 21]
----------------------------------------
```

Figure 12: Results of the code snippet in Q4 with the parameters of my choice

is the same as the length parameter and elements of the list obey the Fibonacci rule.

Question 5

This question asks us to create another subclass of the *Sequence* class named *Prime*. It is identical to *Fibonacci* except that it stores consecutive prime numbers. Also, its instances are *callable* and can be used as an *iterator*. Its implementation can be seen below: ”__call__(self, length)” method is

```python
class Prime(Sequence):
    """A class to represent sequence of prime numbers...."""

    def __init__(self):
        super(Prime, self).__init__([])

    def __call__(self, length):
        prime_seq = [] # sequence of prime numbers
        if length >= 1:
            prime_seq.append(2) # smallest prime number
        for i in range(length-1):
            new_num = prime_seq[-1] # continue for the prime number search from the last
            primeFound = False # flag for whether prime number is found
            while not primeFound:
                primeFound = True
                new_num += 1
                for prime in prime_seq:
                    # prime is found if the below statement is never true
                    if new_num % prime == 0:
                        primeFound = False
                        break
            # out of while means we found prime and we can add it to seq.
            prime_seq.append(new_num)
        self.array = prime_seq # save the prime sequence
        print(self.array)
```

Figure 13: Prime class asked in Q5

very similar to the one in *Fibonacci*. It creates a sequence consisting of consecutive prime numbers with a given length. If the length is more than 1, first the smallest prime number is added to the list. Then, other prime numbers are searched. The idea is beginning from the last number on the list, each number's primeness is questioned by dividing it by the previous prime number that is kept in the list and checking whether the remainder is 0. If any division results in 0 remainder, then that number is not prime. Otherwise, it is a prime number and added to the list. This is done under the ”__call__(self, length)” method. The code snippet for question 5 and its result can be seen in Figure 14 and 15.

```python
PS = Prime()
PS(length=8)
print(len(PS))
print([n for n in PS])
```

Figure 14: Code snippet in Q5

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

Figure 15: Result of the code snippet in Q5

The results are true because we want to see the first 8 prime numbers as a sequence and at the output, we have 8 prime numbers. Between the elements of the sequence, there is not any prime number missing. Results are the exactly the same given in the assignment. Lastly, I implement the same code snippet with the parameters of my choice. Implementation and the results can be seen 16 and 17.

```
PS = Prime()
PS(length=10)
print(len(PS))
print([n for n in PS])
```

Figure 16: Code snippet in Q5 with the parameters of my choice

```
----------Result of question 5----------
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
10
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
----------------------------------------
```

Figure 17: Results of the code snippet in Q5 with the parameters of my choice

As can be seen in the figures above, the results are correct. We expect to see the 10 smallest prime numbers consecutively as an output. Indeed, this is what we see. There is not any number that is divisible by a number other than itself and 1 in the sequence. And there is not any missing prime number.

Question 6

In this part, I update the *Sequence* class such that two *Sequence* objects having the same length can be compared by the operator $>$. For instance, invoking $A > B$ compares element-wise the two arrays and returns the number of elements in A that are greater than the corresponding elements in B. If the length of the arrays is not equal to each other, then the code throws a **ValueError** exception. The final updated version of *Sequence* class can be seen in Figure 18. Note that the ">"

```python
class Sequence(object):
    """A class to represent a sequence...."""

    def __init__(self, array):
        self.array = array

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

    def __iter__(self):
        return SeqIterable(self)

    def __gt__(self, other):
        """..."""

        counter = 0
        if len(self.array) != len(other):
            raise ValueError("Two arrays are not equal in length!")
        else:
            for idx in range(len(self.array)):
                if self.array[idx] > other.array[idx]:
                    counter += 1 # count the true values
        return counter
```

Figure 18: Final implementation of Sequence class asked in Q6

operator corresponds to "__gt__(self, other)" magical method so I overwrite it to achieve the asked result. If the lengths are not equal, it raises an **ValueError**. Else, it compares the corresponding elements in two arrays. One should note that if the comparison is $A > B$, *self* represents A and *other* represents B. They are both *Sequence* objects. The code snippet for question 6 and its result can be seen in Figure 19 20, and 21.

```python
FS = Fibonacci(first_value=1, second_value=2)
FS(length=8)
PS = Prime()
PS(length=8)
print(FS > PS)
PS(length=5)
print(FS > PS)
```

Figure 19: Code snippet in Q6

```
----------Result of question 6----------
[1, 2, 3, 5, 8, 13, 21, 34]
[2, 3, 5, 7, 11, 13, 17, 19]
2
[2, 3, 5, 7, 11]
```

Figure 20: Result of the code snippet in Q6

As can be seen in the figures, the results are correct. Prime and Fibonacci sequences that are generated with respect to the given length parameter have the correct length and their elements obey the prime rule and Fibonacci rule respectively. Also, the number of elements that satisfies (A > B) is found to be 2, which is the same in the assignment. All the outputs are exactly the same. Lastly, when we try to compare two arrays with different lengths, the code raises a **ValueError** as in Figure 21.

```
Traceback (most recent call last):
  File "/Users/berksahin/PycharmProjects/ECE60146/Homework1.py", line 203, in <module>
    print(FS > PS)
  File "/Users/berksahin/PycharmProjects/ECE60146/Homework1.py", line 47, in __gt__
    raise ValueError("Two arrays are not equal in length!")
ValueError: Two arrays are not equal in length!
```

Figure 21: Error Result in Q6

Lastly, I implement the same code with the parameters of my choice. Implementation and the results can be seen in Figure 22 and 23.

```python
FS = Fibonacci(first_value=0, second_value=2)
FS(length=4)
PS = Prime()
PS(length=4)
print(FS > PS)
PS(length=0)
print(FS > PS)
```

Figure 22: Code snippet in Q6 with the parameters of my choice

```
----------Result of question 6----------
[0, 2, 2, 4]
[2, 3, 5, 7]
0
[]
Traceback (most recent call last):
  File "/Users/berksahin/PycharmProjects/ECE60146/Homework1.py", line 240, in <module>
    print(FS > PS)
  File "/Users/berksahin/PycharmProjects/ECE60146/Homework1.py", line 47, in __gt__
    raise ValueError("Two arrays are not equal in length!")
ValueError: Two arrays are not equal in length!
```

Figure 23: Results of the code snippet in Q6 with the parameters of my choice

4 Lessons Learned

In this homework, I practiced the basics of Python Object-Oriented programming. I learned how to use inheritance between parent and child classes. I learned how to create an iterator class to make an instance of a class iterable. I learned how to use the attributes of the parent class and how to create instance methods. I learned how to throw exceptions with arbitrary messages.

```

class Sequence(object):
    """
    A class to represent a sequence.

    Attributes
    ----------
    array : list
        sequence as a list

    Methods
    -------
    __gt__(other):
        Performs element-wise > between arrays. And returns the number of true values.
    __iter__():
        Creates a iterator for Fibonacci object.
    __len__():
        Print the length of the Fibonacci sequence.
    """

    def __init__(self, array):
        self.array = array

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

    def __iter__(self):
        return SeqIterable(self)

    def __gt__(self, other):
        """
        Performs element-wise > operation on the elements of arrays. Returns
        the number of True values.

        Parameters
        ----------
        other : Sequence, required
            another Sequence object for comparison

```

```
        Returns
        -------
        Number of True values after the comparison.
        """

        counter = 0
        if len(self.array) != len(other):
            raise ValueError("Two arrays are not equal in length!")
        else:
            for idx in range(len(self.array)):
                if self.array[idx] > other.array[idx]:
                    counter += 1 # count the true values
        return counter

class SeqIterable(object):
    """
    Iterator class for Sequence class.

    Attributes
    ----------
    items : list
        sequence of prime numbers as a list
    index : int
        index that iterator follows in the sequence

    Methods
    -------
    __iter__():
        Returns an iterator
    __next__():
        Returns the element in the sequence at the current index

    """

    def __init__(self, seq_obj):
        self.items = seq_obj.array
        self.index = -1
```

```python

    def __iter__(self):
        return self

    def __next__(self):
        self.index += 1 # initialize index as 0
        # make sure index is in the range of the sequence
        if self.index < len(self.items):
            return self.items[self.index]
        else:
            raise StopIteration

class Fibonacci(Sequence):
    """
    A class to represent Fibonacci sequences.

    Attributes
    ----------
    first_value : int
        first number of the Fibonacci sequence
    second_value : int
        second number of the Fibonacci sequence
    array : list
        sequence as a list

    Methods
    -------
    __call__(length):
        Creates a Fibonacci sequence with the given length and print it.
    """

    def __init__(self, first_value, second_value):
        super(Fibonacci, self).__init__([])
        self.first_value = first_value
        self.second_value = second_value

    def __call__(self, length):
        fib_seq = []
        for i in range(length):
```

```
            if i == 0:
                fib_seq.append(self.first_value)
            elif i == 1:
                fib_seq.append(self.second_value)
            else:
                new_element = fib_seq[-1] + fib_seq[-2] # Fibonacci rule
                fib_seq.append(new_element)
        # Save the result and print it
        self.array = fib_seq
        print(self.array)

class Prime(Sequence):
    """
    A class to represent sequence of prime numbers.

    Attributes
    ----------
    array : list
        sequence of prime numbers as a list

    Methods
    -------
    __call__(length):
        Generates a sequence with the given length and consists of prime numbers.
    """

    def __init__(self):
        super(Prime, self).__init__([])

    def __call__(self, length):
        prime_seq = [] # sequence of prime numbers
        if length >= 1:
            prime_seq.append(2) # smallest prime number
        for i in range(length-1):
            new_num = prime_seq[-1] # continue for the prime number search from the last
            primeFound = False # flag for whether
```

```
prime number is found
            while not primeFound:
                primeFound = True
                new_num += 1
                for prime in prime_seq:
                    # prime is found if the below statement is never true
                    if new_num % prime == 0:
                        primeFound = False
                        break
            # out of while means we found prime and we can add it to seq.
            prime_seq.append(new_num)
        self.array = prime_seq # save the prime sequence
        print(self.array)

# ask user to either reproduce the results in the snippets or
# produce results with the parameters of my choice
choice = int(input("Reproduce the results in the snippet (enter 0), run author's inputs (enter 1): "))

while not ((choice == 1) or (choice == 0)):
    print("Your answer is invalid. Enter a valid answer.")
    choice = int(input("Reproduce the results in the snippet (enter 0), run author's inputs (enter 1): "))
# reproduce the results in the code snippets of the assignment
if choice == 0:
    print("Results with the parameters given in the homework.")
    # Q3
    print("-"*10 + "Result of question 3" + "-"*10)
    FS = Fibonacci(first_value=1, second_value=2)
    FS(length=5)
    print("-"*40)
```

```

    # Test for Fibonacci Sequence (Q4)
    print("-" * 10 + "Result of question 4" + "-" * 10)
    FS = Fibonacci(first_value=1, second_value=2)
    FS(length=5)
    print(len(FS))
    print([n for n in FS])
    print("-"*40)

    # Test for Prime Sequence (Q5)
    print("-" * 10 + "Result of question 5" + "-" * 10)
    PS = Prime()
    PS(length=8)
    print(len(PS))
    print([n for n in PS])
    print("-"*40)

    # Test for Comparison Operator (Q6)
    print("-" * 10 + "Result of question 6" + "-" * 10)
    FS = Fibonacci(first_value=1, second_value=2)
    FS(length=8)
    PS = Prime()
    PS(length=8)
    print(FS > PS)
    PS(length=5)
    print(FS > PS)
    print("-"*40)
else:
    # produce the results with the parameters of my choice
    print("Results with the parameters of my choice. ")
    # Q3
    print("-" * 10 + "Result of question 3" + "-" * 10)
    FS = Fibonacci(first_value=1, second_value=5)
    FS(length=7)
    print("-" * 40)
```

```

    # Test for Fibonacci Sequence (Q4)
    print("-" * 10 + "Result of question 4" + "-" * 10)
    FS = Fibonacci(first_value=1, second_value=1)
    FS(length=8)
    print(len(FS))
    print([n for n in FS])
    print("-" * 40)

    # Test for Prime Sequence (Q5)
    print("-" * 10 + "Result of question 5" + "-" * 10)
    PS = Prime()
    PS(length=10)
    print(len(PS))
    print([n for n in PS])
    print("-" * 40)

    # Test for Comparison Operator (Q6)
    print("-" * 10 + "Result of question 6" + "-" * 10)
    FS = Fibonacci(first_value=0, second_value=2)
    FS(length=4)
    PS = Prime()
    PS(length=4)
    print(FS > PS)
    PS(length=0)
    print(FS > PS)
    print("-" * 40)

```