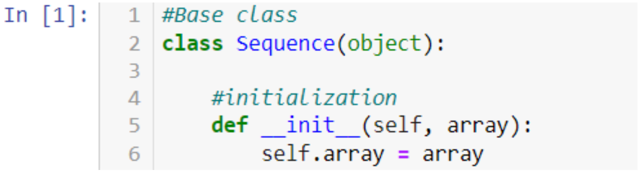
SPRING 2023 ECE 60146 – Homework 1

Sahithi Kodali – 34789866
kodali1@purdue.edu

This homework involves programming tasks of the Python Object-Oriented programming concepts. Further, the implementation and results are discussed in this report.

Implementation and Results:

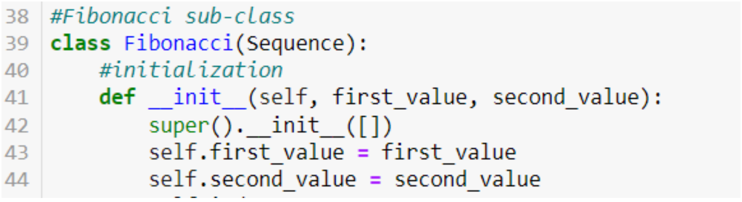
1. Created a class *Sequence* with an instance variable *array* which acts as the base class as shown in Figure 1.



```
In [1]:
#Base class
class Sequence(object):

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

Figure 1

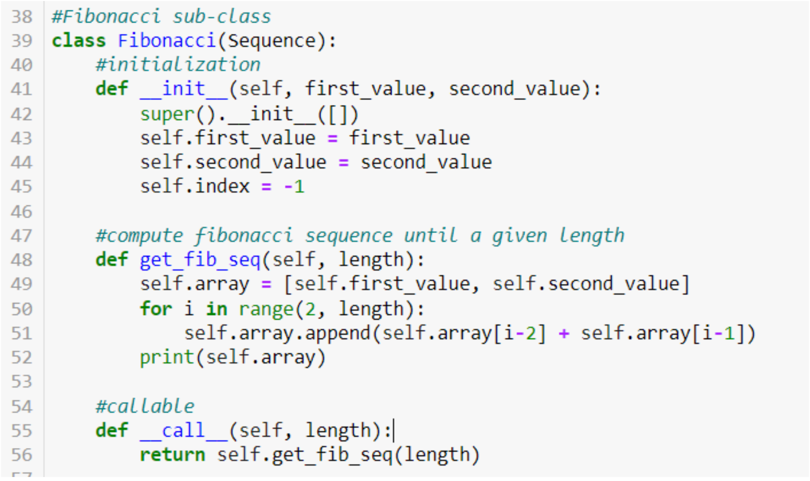
2. A subclass *Fibonacci* extending the base class *Sequence* is written with the parameters *first_value* and *second_value* initialized in the *__init__* method of the class. These parameters will be the first two numbers of the *Fibonacci* sequence as shown in Figure 2.



```
#Fibonacci sub-class
class Fibonacci(Sequence):
    #initialization
    def __init__(self, first_value, second_value):
        super().__init__([])
        self.first_value = first_value
        self.second_value = second_value
```

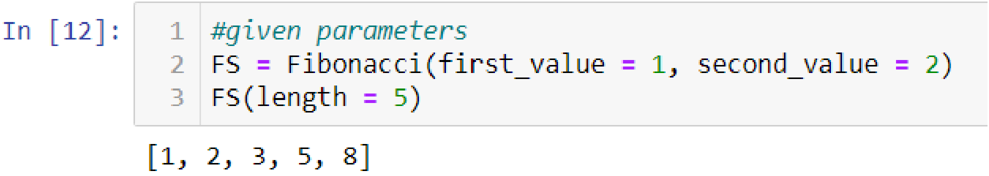
Figure 2

3. A method *get_fib_seq* has been written that takes the parameter length and prints an array of *Fibonacci* sequence of given length. Fibonacci of a number can be computed as the sum of the previous two numbers with first two values given as values one and two. To make the instance callable, a *__call__* method that calls the *get_fib_seq* method has been defined as shown in Figure 3. The results for the given and own parameters are shown in Figure 4 and 5 respectively.



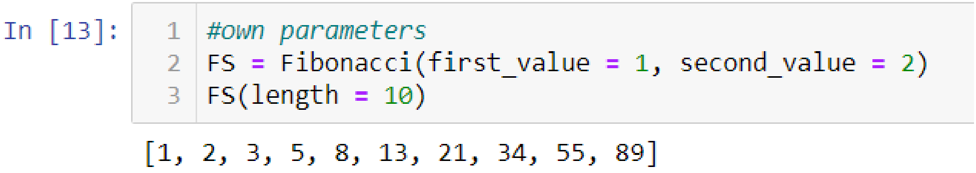
```
#Fibonacci sub-class
class Fibonacci(Sequence):
    #initialization
    def __init__(self, first_value, second_value):
        super().__init__([])
        self.first_value = first_value
        self.second_value = second_value
        self.index = -1

    #compute fibonacci sequence until a given length
    def get_fib_seq(self, length):
        self.array = [self.first_value, self.second_value]
        for i in range(2, length):
            self.array.append(self.array[i-2] + self.array[i-1])
        print(self.array)

    #callable
    def __call__(self, length):
        return self.get_fib_seq(length)
```

Figure 3



```
In [12]: #given parameters
         FS = Fibonacci(first_value = 1, second_value = 2)
         FS(length = 5)

[1, 2, 3, 5, 8]
```

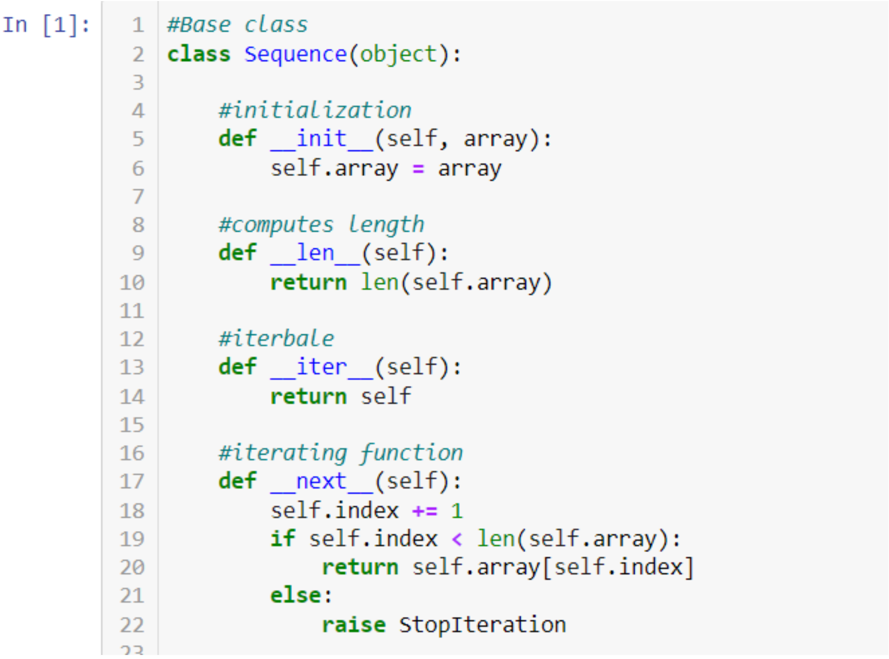
Figure 4: For given parameters



```
In [13]: #own parameters
         FS = Fibonacci(first_value = 1, second_value = 2)
         FS(length = 10)

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

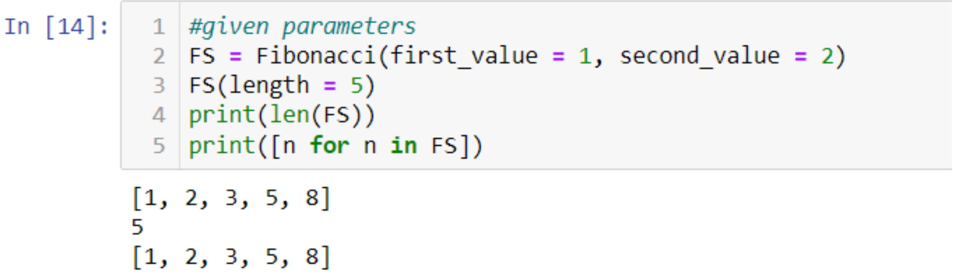
Figure 5: For own parameters

4. To make the instances of the subclass of *Sequence* class iterable, the method __*iter*__ is used to initialize iteration and the method __*next*__ performs the iteration task by printing every number in the sequence instance. Further, the method __*len*__ is used to compute the sequence instance length as shown in Figure 6. The results for the given and own parameters are shown in Figure 7 and 8 respectively.



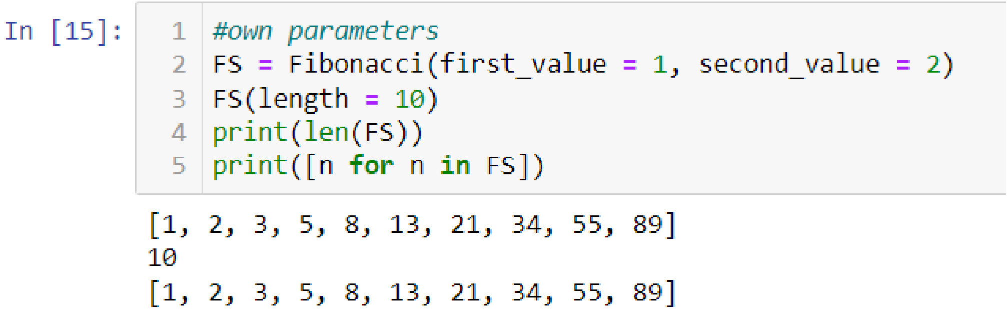
```
In [1]: #Base class
        class Sequence(object):

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

            #computes length
            def __len__(self):
                return len(self.array)

            #iterbale
            def __iter__(self):
                return self

            #iterating function
            def __next__(self):
                self.index += 1
                if self.index < len(self.array):
                    return self.array[self.index]
                else:
                    raise StopIteration
```

Figure 6



```
In [14]: #given parameters
         FS = Fibonacci(first_value = 1, second_value = 2)
         FS(length = 5)
         print(len(FS))
         print([n for n in FS])

[1, 2, 3, 5, 8]
5
[1, 2, 3, 5, 8]
```

Figure 7: For given parameters

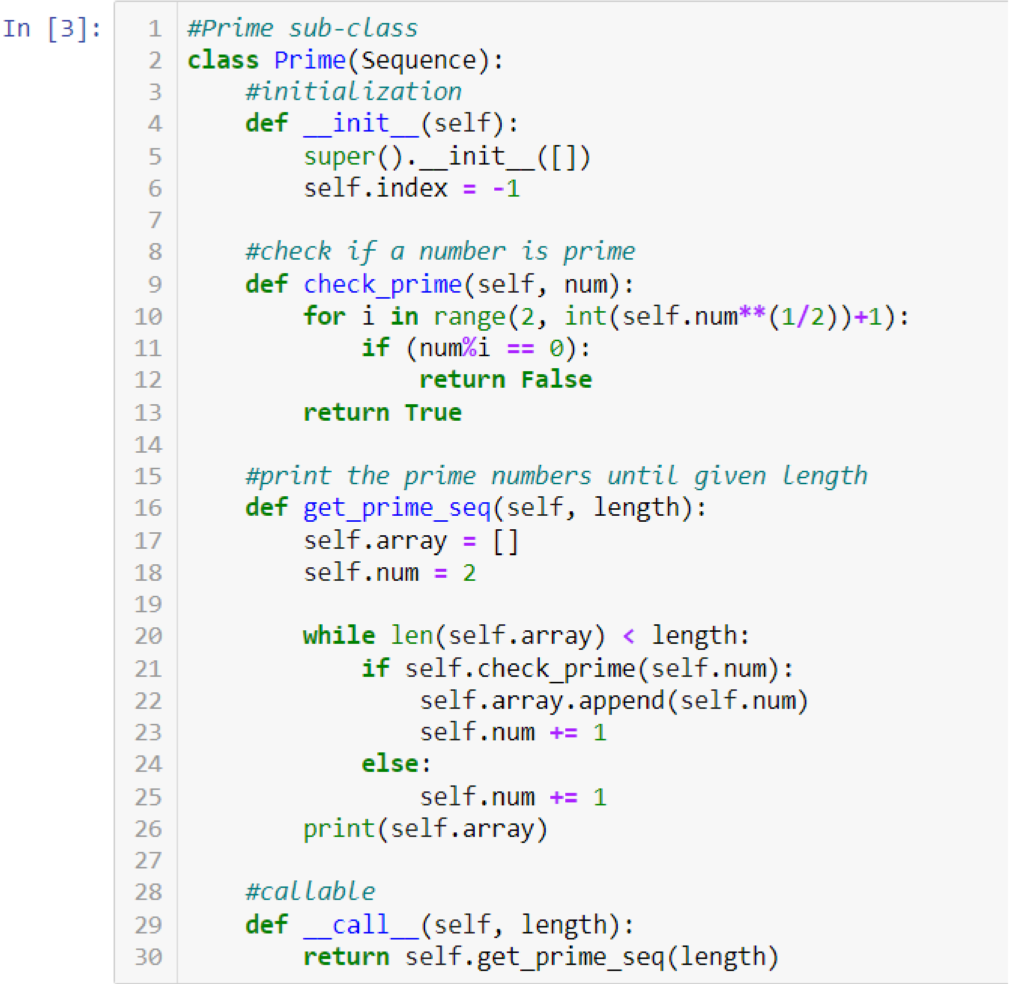


```
In [15]:
#own parameters
FS = Fibonacci(first_value = 1, second_value = 2)
FS(length = 10)
print(len(FS))
print([n for n in FS])
```

```
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
10
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

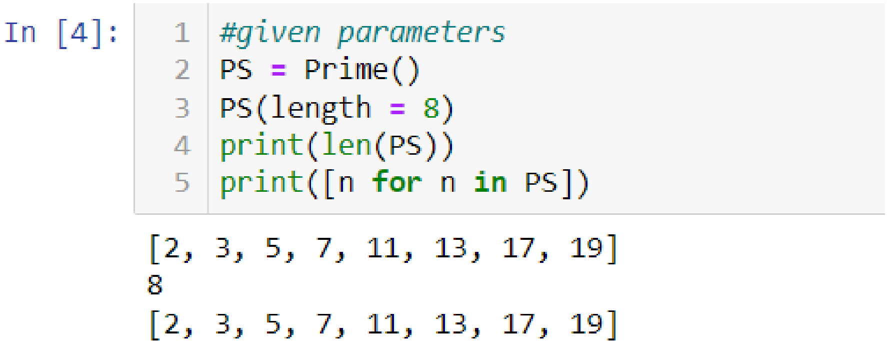
Figure 8: For own parameters

5. Similar to the step 3 and 4 above, a method *get_prime_seq* is written to print the first given length number of primes which invokes *check_prime* method that checks if a number is prime or not. The given number is divided by numbers from 2 to the square root of given numbers; if the remainder is zero, the number is not a prime else it is a prime. To make the instance callable, a *__call__* method that calls the *get_prime_seq* method has been defined and the methods *__iter__* and *__next__* defined in *Sequence* class makes the instance iterable as shown in Figure 9. The results for the given and own parameters are shown in Figure 10 and 11 respectively.



```
In [3]:
#Prime sub-class
class Prime(Sequence):
    #initialization
    def __init__(self):
        super().__init__([])
        self.index = -1

    #check if a number is prime
    def check_prime(self, num):
        for i in range(2, int(self.num**(1/2))+1):
            if (num%i == 0):
                return False
        return True

    #print the prime numbers until given length
    def get_prime_seq(self, length):
        self.array = []
        self.num = 2

        while len(self.array) < length:
            if self.check_prime(self.num):
                self.array.append(self.num)
                self.num += 1
            else:
                self.num += 1
        print(self.array)

    #callable
    def __call__(self, length):
        return self.get_prime_seq(length)
```

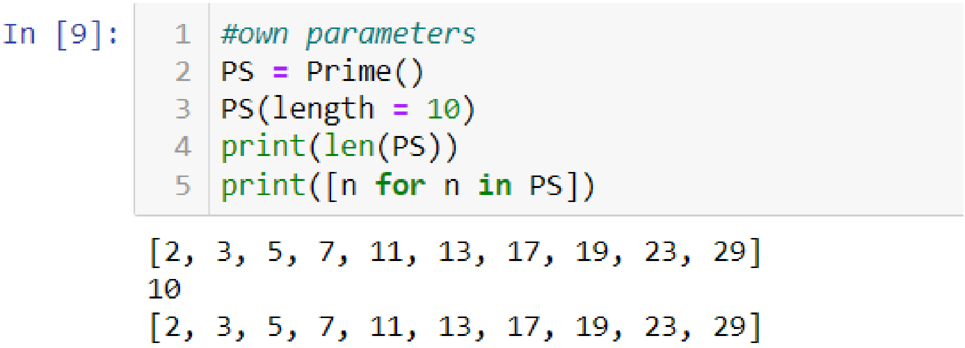
Figure 9



```
In [4]:
#given parameters
PS = Prime()
PS(length = 8)
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]
```

Figure 10: For given parameters

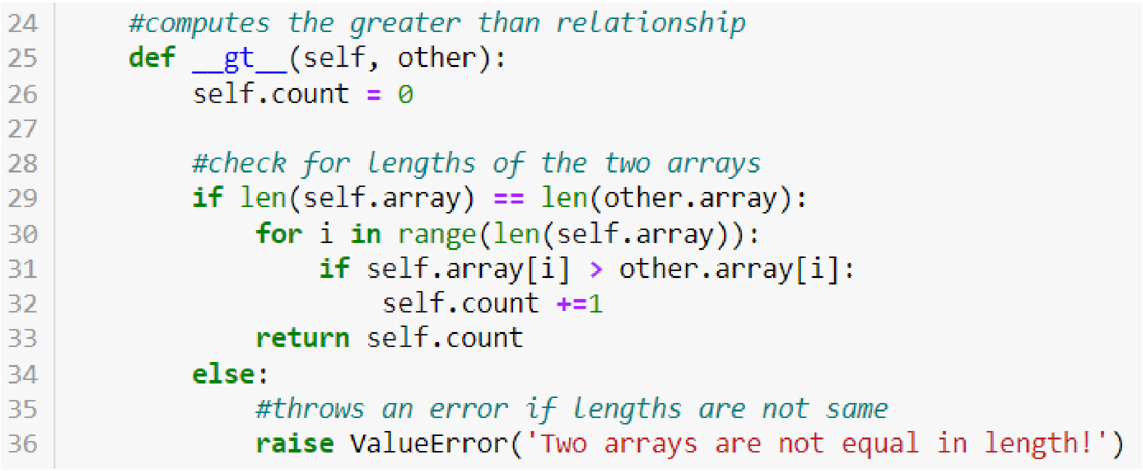


```
In [9]:
#own parameters
PS = Prime()
PS(length = 10)
print(len(PS))
print([n for n in PS])
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
10
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

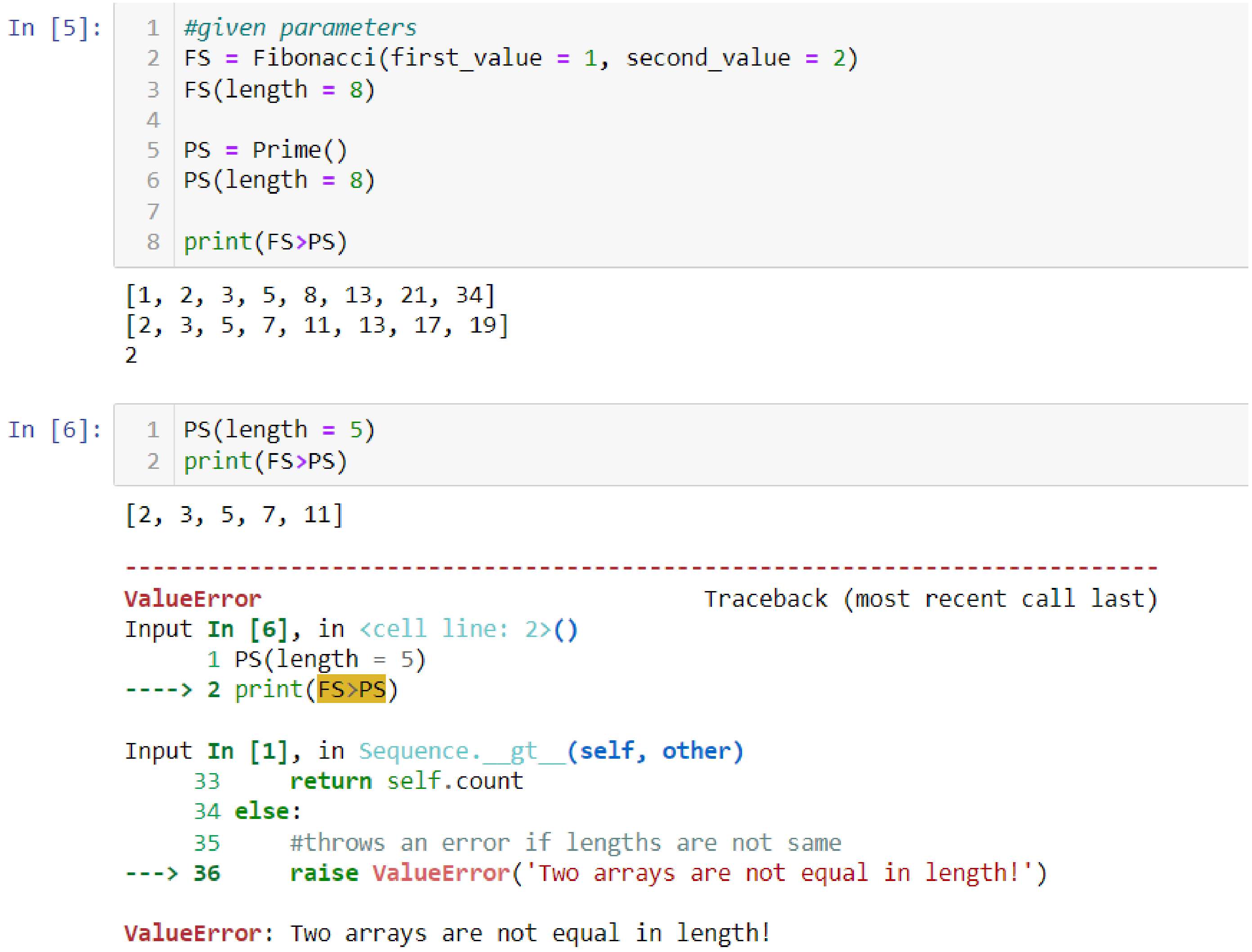
Figure 11: For own parameters

6. A method *__gt__* that overrides the inbuilt *__gt__* function is written in *Sequence* class such that if the lengths of arrays returned by the two instances is same, a parameter *count* initialized to zero is incremented by 1 for every number greater than the corresponding number in the other array. If the lengths of both the arrays are not same, a *'ValueError'* is thrown stating the reason for error as shown in Figure 12. The results for the given and own parameters are shown in Figure 13 and 14 respectively.



```
    #computes the greater than relationship
    def __gt__(self, other):
        self.count = 0

        #check for lengths of the two arrays
        if len(self.array) == len(other.array):
            for i in range(len(self.array)):
                if self.array[i] > other.array[i]:
                    self.count +=1
            return self.count
        else:
            #throws an error if lengths are not same
            raise ValueError('Two arrays are not equal in length!')
```

Figure 12



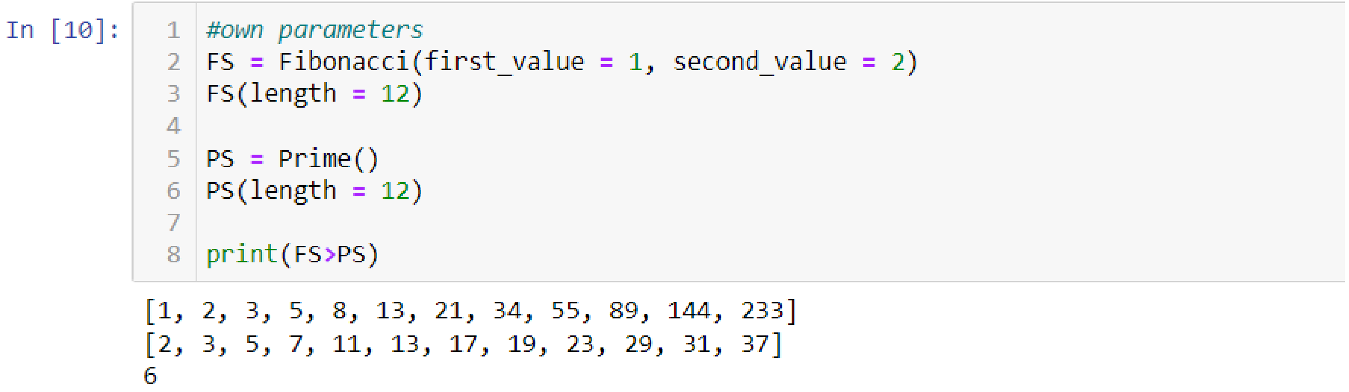
```
In [5]:
#given parameters
FS = Fibonacci(first_value = 1, second_value = 2)
FS(length = 8)

PS = Prime()
PS(length = 8)

print(FS>PS)
```

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

```
In [6]:
PS(length = 5)
print(FS>PS)
```

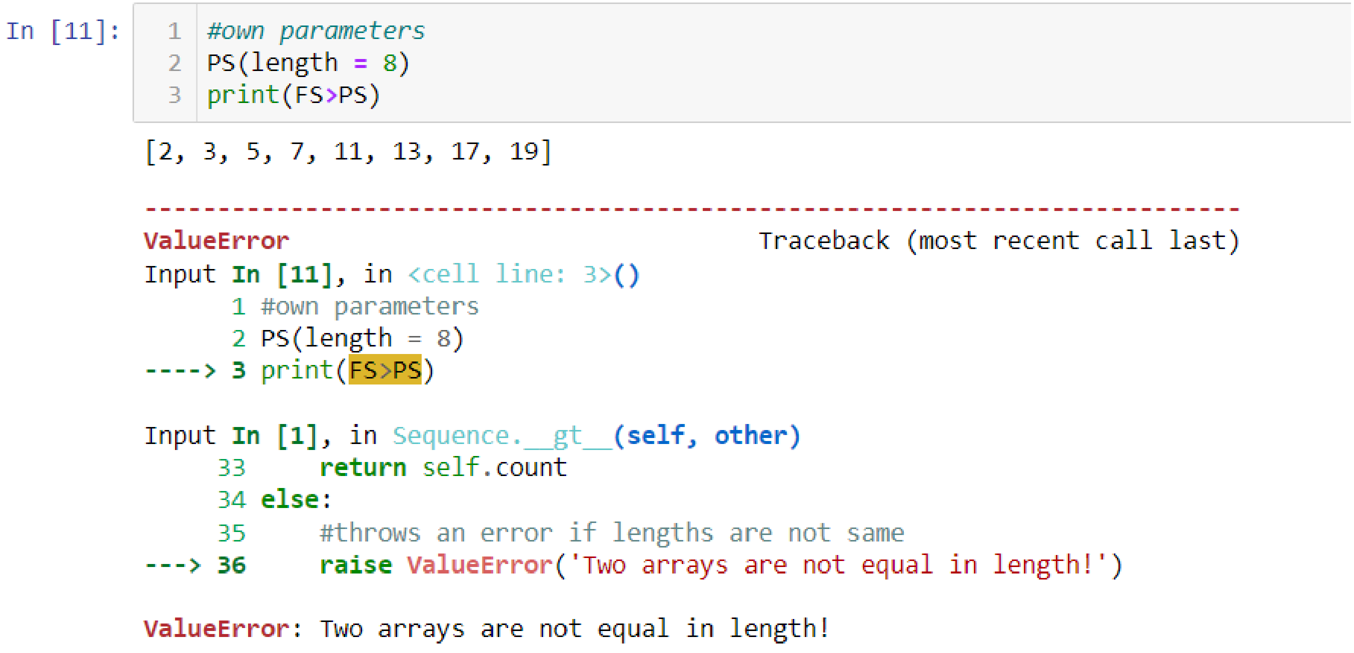
```
[2, 3, 5, 7, 11]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [6], in <cell line: 2>()
      1 PS(length = 5)
----> 2 print(FS>PS)

Input In [1], in Sequence.__gt__(self, other)
     33     return self.count
     34 else:
     35     #throws an error if lengths are not same
---> 36     raise ValueError('Two arrays are not equal in length!')

ValueError: Two arrays are not equal in length!
```

Figure 13: For given parameters



```
In [10]:
#own parameters
FS = Fibonacci(first_value = 1, second_value = 2)
FS(length = 12)

PS = Prime()
PS(length = 12)

print(FS>PS)
```

```
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
6
```

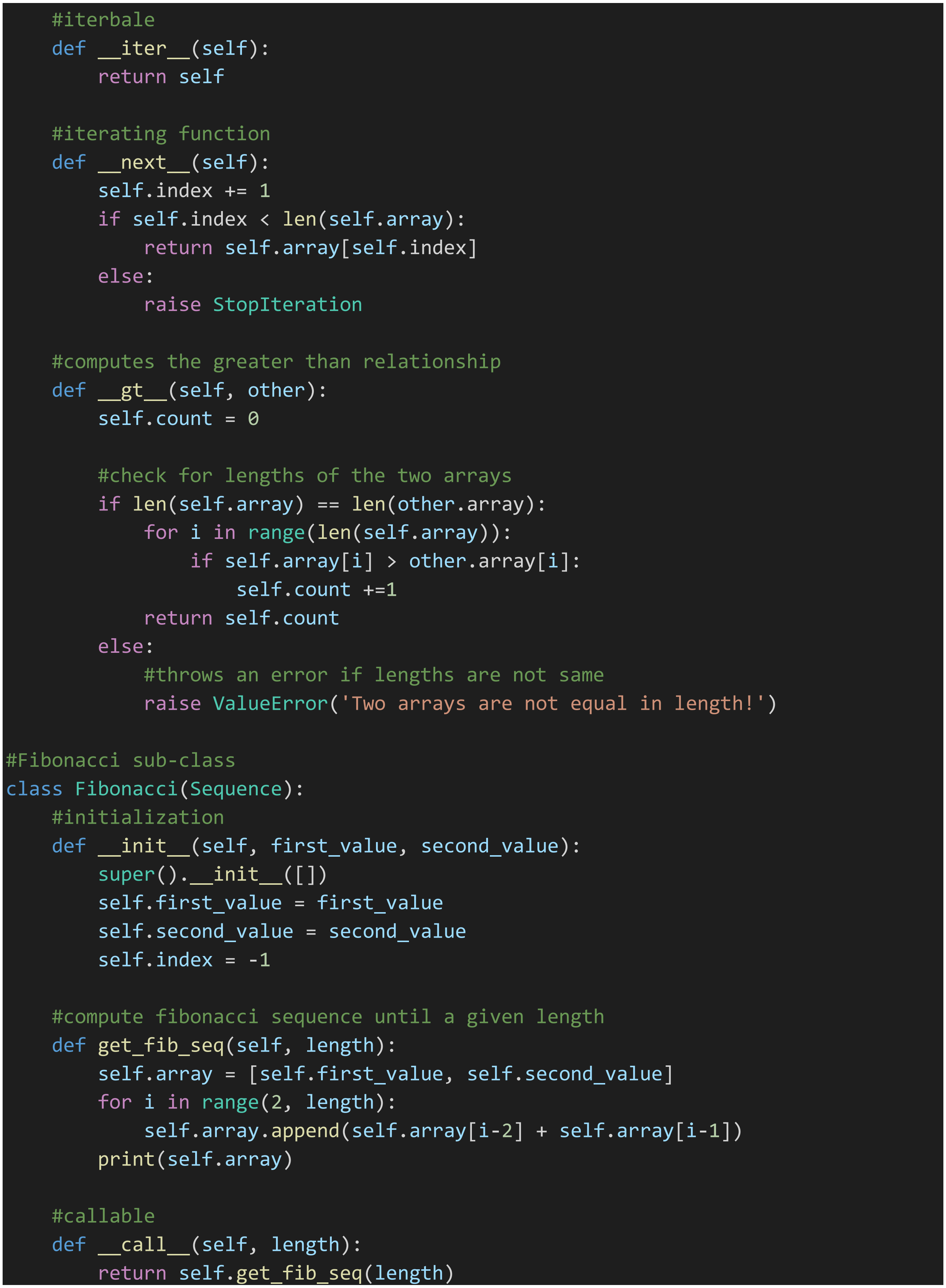


```
In [11]: #own parameters
         PS(length = 8)
         print(FS>PS)

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

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [11], in <cell line: 3>()
      1 #own parameters
      2 PS(length = 8)
----> 3 print(FS>PS)

Input In [1], in Sequence.__gt__(self, other)
     33     return self.count
     34 else:
     35     #throws an error if lengths are not same
---> 36     raise ValueError('Two arrays are not equal in length!')

ValueError: Two arrays are not equal in length!
```

Figure 14: For own parameters

Source Code:

```
#!/usr/bin/env python
# coding: utf-8

# <h2><center>ECE60146 Deep Learning</center></h2>
# <h3><center>Homework - 1 </center></h3>
# <h3><center>Sahithi Kodali - 34789866</center></h3>
# <h3><center>kodali1@purdue.edu</center></h3>

# In[1]:


#Base class
class Sequence(object):

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

    #computes length
    def __len__(self):
        return len(self.array)
```



```
    #iterbale
    def __iter__(self):
        return self

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

    #computes the greater than relationship
    def __gt__(self, other):
        self.count = 0

        #check for lengths of the two arrays
        if len(self.array) == len(other.array):
            for i in range(len(self.array)):
                if self.array[i] > other.array[i]:
                    self.count +=1
            return self.count
        else:
            #throws an error if lengths are not same
            raise ValueError('Two arrays are not equal in length!')

#Fibonacci sub-class
class Fibonacci(Sequence):
    #initialization
    def __init__(self, first_value, second_value):
        super().__init__([])
        self.first_value = first_value
        self.second_value = second_value
        self.index = -1

    #compute fibonacci sequence until a given length
    def get_fib_seq(self, length):
        self.array = [self.first_value, self.second_value]
        for i in range(2, length):
            self.array.append(self.array[i-2] + self.array[i-1])
        print(self.array)

    #callable
    def __call__(self, length):
        return self.get_fib_seq(length)
```

```
# In[2]:


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


# In[3]:


#own parameters
FS = Fibonacci(first_value = 1, second_value = 2)
FS(length = 10)
print(len(FS))
print([n for n in FS])


# In[4]:


#Prime sub-class
class Prime(Sequence):
    #initialization
    def __init__(self):
        super().__init__([])
        self.index = -1

    #check if a number is prime
    def check_prime(self, num):
        for i in range(2, int(self.num**(1/2))+1):
            if (num%i == 0):
                return False
        return True

    #print the prime numbers until given length
    def get_prime_seq(self, length):
        self.array = []
        self.num = 2

        while len(self.array) < length:
            if self.check_prime(self.num):
                self.array.append(self.num)
                self.num += 1
            else:
                self.num += 1
        print(self.array)

    #callable
    def __call__(self, length):
        return self.get_prime_seq(length)


# In[5]:


#given parameters
PS = Prime()
PS(length = 8)
print(len(PS))
print([n for n in PS])


# In[6]:


#own parameters
PS = Prime()
PS(length = 10)
print(len(PS))
print([n for n in PS])


# In[7]:


#given parameters
FS = Fibonacci(first_value = 1, second_value = 2)
FS(length = 8)

PS = Prime()
PS(length = 8)

print(FS>PS)


# In[8]:
```

```
PS(length = 5)
print(FS>PS)


# In[9]:


#own parameters
FS = Fibonacci(first_value = 1, second_value = 2)
FS(length = 12)

PS = Prime()
PS(length = 12)

print(FS>PS)


# In[10]:


#own parameters
PS(length = 8)
print(FS>PS)
```

****