Morning----🡪

Problem1

create the series 1 2 2 3 3 5 5 7 8 11 13

#odd position---->1 2 3 5 8 13 This Fibonacci series N//2+1 time complexity O(n)

#Even position---->2 3 5 7 11 This is Prime series n/2-time complexity O (n square)

Code:

import math

def check\_if\_prime(num): #function to check prime number

    if(num<2):

        return False

    for i in range (2, math.ceil(math.sqrt(num))+1):

        if num % i == 0:

            return False

    return True

def nth\_prime(num): #function to get nth prime number

    j = 0

    if num == 1:

        j = 2

    elif num == 2:

        j = 3

    else:

        count = 2

        j = 4 #Number in J is checked if Prime or not

        while count<=num:

            if check\_if\_prime(j):

                count += 1

            if count==num:

              break

            j += 1

    return j

def nth\_fibo(n): #function to get nth Fibonacci number

    first= 1

    second = 2

    third=0

    if n==1:

        return first

    elif n==2:

        return second

    else:

        count=2

        while(count<n):

            third=first+second

            first = second

            second = third

            count+=1

        return third

n=int(input("Enter the number of terms:"))

i=1

while(i<=n):

    if (i%2 != 0):

        number=nth\_fibo ((i // 2) + 1)

    else:

        number=nth\_prime(i//2)

    print(number)

    i+=1

LIST SLICING

numbers = [2, 9, 7, 5, 3, 13, 19, 17, 29]  
print(numbers) # we get the complete list  
print(numbers[0]) # we get the 1st element  
print(numbers[-1]) # we get the last element  
Hence, we see that -ve indexing is possible in Python

print(numbers[-2]) # we get last but element from the list  
print(numbers[:]) # we get the complete list  
Here, we have written nothing before the colon and hence it is treated as 0, which means start from the beginning of the list. Here also have not written anything after the colon, which means it is treated as UP TO END OF THE LIST.

print(numbers[:3]) # Start from index 0 and access elements up to index 3-1 which is 2.  
print(numbers[1:-1]) # Start from index 1 (2nd element) and access elements up to last but element, because -1 is the index of last element and we must not include it.  
print(numbers[1:8:2]) # start from index 1 and access upto index 8-1 and increment each time by 2 elements. So o/p is [9, 5, 3, 19]  
print(numbers[::3]) # start from beginning and go up to end of the list with increment of 3  
print(numbers[8:1:-2]) #Start from index 8, go up to index 2 with decrement of 2 (or increment of -2)  
print(numbers[::-1]) # Since the increment is negative, we understand that we have to move in reverse (meaning from the end to towards the start). The area within the list we have to access is all the elements, because nothing is specified before and after the 1st colon

GLOBAL METHODS AND KEYWORDS:

numbers = [1, 2, 3, 4, 5]  
del numbers[1:3]  
print(numbers) # [1, 4, 5]  
del numbers[1]  
print(numbers) # [1, 5]  
del numbers[:] # we are deleting all the elements from the list  
print(numbers) # []  
numbers.insert(10, 0) # inserting the element 10 at index 0

INSERT ELEMENT AT REAR OF THE LIST: 3 different ways  
a.insert(len(a), x)  
a.append(x)  
a[len(a):] = [element]

Below is an error  
numbers[len(numbers):] = 3  
Type Error because we must assign only a list(itarable) not a primitive value

min () ---🡪used to find smallest number in a list

max ()-🡪used to find biggest number in a list

Sorted-🡪to get sorted list---while sorting original list is not modified, it makes a copy of the list in sorted order

len ()🡪to get length of list

eg:

numbers = [23, 7, 19, 41, 29, 3, 47]

**print**('Max number = ', **max**(numbers))

**print**('Min number = ', **min**(numbers))

**print**('Number of elements = ', **len**(numbers))

**print**('Sorted numbers = ', **sorted**(numbers))

**print**('Numbers = ', numbers)

numbers.**sort**()

**print**('Numbers = ', numbers)

name = 'Cambodia'

names = ['aizwal', 'imphal', 'shillong', 'agartala']

**print**(name)

**print**(name.**upper**())

**print**(name.**count**('a'))

**print**(name.**count**('A'))

**print**(name.**upper**().**count**('A'))

**print**(name.**upper**().**count**('a'))

**print**(name.**find**('o'))

**print**(name.**find**('dia'))

**print**(name.**find**('xx'))

afternoon**--🡪**

l1 = [3, 6, 10]

l2 = [1, 2, 4, 12]

if l1 > l2:

**print**('List1 is bigger')

else:

**print**('List2 is bigger')

if l1.**\_\_gt\_\_**(l2):

**print**('List1 is bigger')

else:

**print**('List2 is bigger')

'''

For readability purpose we use 6 R. oprs

> < >= <= != ==

However we can perform any of the 6 operations using:  > and ==

'''

list1 = []

list1 = **list**()

tuple1 = ()

tuple2 = **tuple**()

dictionary1 = { }

dictionary2 = **dict**()

set1 = **set**()

**print**(**type**(dictionary1))

**print**(**type**(set1))

**print**(**type**(tuple1))

def **varArgFunction1**(\*arguments):

**print**(arguments)

**print**(**type**(arguments)) *# tuple*

def **varArgFunction2**(\*arguments):

    for i in **range**(**len**(arguments)):

**print**(arguments[i])

**varArgFunction1**(1, 2, 4)

**varArgFunction1**()

**varArgFunction1**('list', 'tuple', 'set', 'dictionary')

**varArgFunction2**(1, 2, 4)

**varArgFunction2**()

**varArgFunction2**('list', 'tuple', 'set', 'dictionary')

''''

def varArgFunction2(\*arguments):

        print(arguments)

def varArgFunction2(\*arguments):

    for i in range(len(arguments)): #Loop with range() function

        print(arguments[i])

def varArgFunction2(\*arguments):

    for element in arguments: # for each loop. It accesses all elements of the tuple one by one

        print(arguments[i])

'''

def varArgFunction1 (\*arguments):

def **varArgFunction1**(\*arguments):

**print**(arguments)

*# arguments[1] = 22 # Type Error*

    arguments[5][0] = 20 *# even though the list is inside the tuple, we can modify it.*

**print**(arguments)

    arguments[5].append(50)

**print**(arguments)

**varArgFunction1**(1, 2, 4, 'list', 4.5, [2, 3, 5])

**eg:**

def **varArgFunction**(\*arguments): *# receives the data into a tuple. However, if it has objects like list or dictionary, then they will received by reference only.*

**print**(arguments)

    arguments[0].remove(10)

def **myFunction**(user\_given\_list): *# receives reference to the list numbers2*

    user\_given\_list.remove(35)

numbers1 = [**int**(num) for num in **input**().**split**(',')]

*# input: 10,20,30,40,50*

**print**(numbers1)

**varArgFunction**(numbers1)*# the list here is passed by reference*

**print**(numbers1)

numbers2 = **list**(**map**(**int**, **input**().**split**()))

**print**(numbers2)

**myFunction**(numbers2)

**print**(numbers2)

problem:

1.user gives data like this…………………………………………………………………….  
kerala-tiruvanantapuram karnataka-bengaluru tamilnadu-chennai  
You have to separate the states and store in the list states[] and also the separated capitals must be stored in capitals[]

2. Accept a string of space separated numbers and store them as a matrix (list of lists) of n rows.  
Now print the matrix row-wise

3. Accept N strings and count the number of Palindromes in it.

4. Accept N strings, and check how many of them possess the string X

5. Accept N main strings and N sub strings into lists and check create a list of numbers of 0s and 1s where 0 represents that the sub string at respective index is not a substring of the main string.

main\_list= ['andhra pradesh','kerala','Maharashtra','haryana']  
sub\_list  = ['pradesh', 'south', 'rashtra', 'punjab']

presence = [1, 0, 1, 0]