Python Notes:

W1: Introduction to python

1. like in java where int, float has some limit to storage a particular range of the variable whereas in python there is no such issue of storing, No specific size of for storing any data type

2. a = 10
   id1 = id(a)
   b = a + 2-2
   id2 = id(b)
i'd give you unique identification to every number variable which is stored. Moreover, id is basically referencing = address so if you give10 variable same value then python will give reference to the same address only.
but the catch is this works only from -5,256 only before & after id changes

3. 10/3 = 3.33 #floatingpointdevison
    10//3 = 3 #interger division

4. it is not necessary to put else after if-else condition

5. it not even necessary to put else after if-elif-elif condition also

6. tip: in multiple nested loop outer loop will just maintain the range from where to where you have to go and inner loop will perform the function.

7. **Break statement** will come out of the loop which is just above it.

8. **while/for with else**:
the else part can be attached to while/for loops, but the else part will only work when while loop is terminated by the condition i.e condition is completed and then it will enter else part of while/for loop.
but if while loop contains a break statement then it will not go to else part of the while loop.

9.**continue keyword**: all the of the code below continue is skipped for that particular iteration.

```
for i in range(10):
    if i == 3:                    output→ 0,1,2,4,5,6,7,8,9
        continue
    print(i)
```

10. **return statement**- All the code below return is not executed at all in any case.

11. Variables inside the function are local variables because their scope is limited within the function, whereas global variables are defined outside the function and it is not necessary that global variables should be defined above the func, just define variables before calling the function.
so if you want to use the local variable outside the func,use return keyword to return values,further store the returned values from the function and print it.

12.
```
a = 14
def f():
   a=12
print(a)  #14
f()       #12
```
print(a) #14 - intuitively it should be 12 but it is not like that, python assumes that which you have defined is local to that variable so if you want ki jo changes function ke andar ho woh bahar bhi refect ho, then use global variable.

```
a=14
def f():
   global a
   a=12
print(a)        #14
f()             #12
print(a)        #12
```

LIST and its operation

13. li = [1,2,3,"cheerag"]
li[:] - will give all the element of the list
li[1:10] - output [1,2,3,"cheerag"] - it doesn't care whether you have the 10 elements or not, it will print up till 9 whatever you have.
List iteration shortcut : print(*li) → [1,2,3,"cheerag"]

14. li.insert(2,99) - > [1,2,99,3,cheerag]
li.insert(77,10) - > [1,2,9,3,"cheerag"] - > it doesn't matter whether you have 10 index or not if you have 10 index then it will place the element at its correct location else element will be stored at last location.

append vs extend in list
li = [1,2,3,"cheerag"] --now you want to store the multiple element inside list in one go
so if you do li.append([55,77,88])
Output : li = [1,2,3,"cheerag", [55,77,88]] - list store it as a single unit , so solve this problem we can use li.extend([55,77,88])
Output : li = [1,2,3,"cheerag",55,77,88]

Remove(works on element present in the list) vs POP(works on index)
li = [1,2,3,"cheerag",55,77,88,2]
li.remove(2) → li.remove only takes one args and the first 2 which is encountered is removed from the list. So, Output: [1,3,"cheerag",55,77,88,2]

Now we cannot remove elements from a specific index using the remove method , so the **pop method** does this for us.

li = [1,3,"cheerag",55,77,88,2]
li.pop(2) → it will remove cheerag from the list
li = [1,3,55,77,88,2]

And if you do li.pop only → it will remove last element from the list
li.pop()
Output : [1,3,55,77,88]

Note:
list actually stores the references of the element rather than list element , list element can be anywhere in the memory so it list will refer to those memory locations and it stores the references continuously.
So how this continuity is maintained : so suppose your list contain 3 elements and obviously it have the contiguous references of the elements , now for the 4 elements (behind the scene) contiguous of memory for around double the size of list is found in the memory, and the references is copied on the new memory location and now list(li) will point to new location now.
So every time the list crosses the element spaces , it will look for the spaces in the memory double the size of the current list and copy the reference over there.

elements = [int(x) for x in input().split()] - we can take a list as an input using this method.

Mutable vs Immutable

Immutable:(Not able to change value in the memory location rather change the reference)
x = 1 , here x pointing to reference where 1 is stored
x= 3 , now when x = 3 , x is started to point a new reference rather than changing the value of at the same location where it was previously pointing

a = x → a pointing to x which is 3 now
a = 5 → now will start pointing to new reference rather than changing the of 3 to 5

Mutable (change in one leads to change in memory of other)

Li = [1,2,4,5]
Li2 = Li → now li2 and li are pointing to same reference
Li2[3] = 10
Li =[1,2,4,10] so change in Li2 leads to change in Li also.

Question on mutable and immutable:

```
def change(li):
    li[1] = li[1] + 2
li = [1,2,3,4,5]
change(li)
print(li)
```

Output : [1,4,3,4,5] # cz list are mutable hence change in one reflects change in other list cz reference is same

```
def change(li):
    li = [3,4,5,6,7]
li = [1,2,3,4,5]
change(li)
print(li)
```

Output : [1,2,3,4,5] → now the reference is different so same list is printed

```
def change(li):
    li[1] = li[1] + 2
    li = [3,3,3,4,5]
li = [1,2,3,4,5]
change(li)
print(li)
```

Output : [1,4,3,4,5]

Searching and sorting techniques:

1.Linear search:

```
def linearSearch(n,arr,search):

        for i in range(n):
                if arr[i] == search:
                        print("Element found at {}th position".format(i+1))
                        break
        else:
        print("Element Not present in the List")


n = int(input())
listOfElement = [int(x) for x in input().split()]
search = int(input())
linearSearch(n,listOfElement,search)
```

2.Binary search (It always take sorted array as an input**)**

```
def binarySearch(start,end,arr,element):

        while start <= end :
                mid = (start + end) //2

                if arr[mid] > element: #left half
                        end = mid - 1

        #return binarySearch(start,end,arr,element)

                elif arr[mid] < element: #right half
                        start = mid +1
        #return binarySearch(start,end,arr,element)

                elif element == arr[mid]:
                        print(mid)
                        break
        else:
        print("-1")


n = int(input())
arr = [int(x) for x in input().split()]
x = int(input())
start = 0
end = n -1

binarySearch(start,end,arr,x)
```

3. **Selection Sort :** find the min element in the array and swap it the current pointer of the array.
i.e 1,2,5,7,0,8,9

so we will traverse the list by taking 1 as the min until we finish the list, as soon as number less than 1 is encountered we swap it with at number so 1 is swapped with 0 in this case and so on.

```
    def selectionSort(a,n):
        for i in range(n-1):
                min_index = i
                for j in range(i+1,n):
                        if a[min_index] > a[j]:
                        min_index = j

                a[i],a[min_index] = a[min_index],a[i]
         return a
#n = int(input())
```

```
n = 6
a = [13,24,39,0,1,2]
result = selectionSort(a,n)
for data in result:
        print(data,end = " ")
```

4.**Bubble Sort** - in the first pass we place the largest element at the last position and we do it till n-1 times to sort the entire array.

```
def bubbleSort(a,n):
        for i in range(n-1):
                for j in range(n-i-1):
                        if a[j] > a[j+1]:
                                a[j],a[j+1] = a[j+1],a[j]

        return a

n = 6
a = [13,24,39,0,1,2]
result = bubbleSort(a,n)
for data in result:
        print(data,end = " ")
```

Strings:

strings are the character of sequence.
but actually it's a sequence of Unicode
internally it stores "cheerag" in 0,1, so it's an encoding technique.

python uses Unicode coding.

we can initialize string in single, double, triplet quotes
adv of triplet over others is: multiline printing using triple quotes

string is immutable, you can only the change the reference of the string but you can't change the string variable, while lists are mutable in nature

```
s = 'cheerag'
s[0] = "s"
not possible
```

id changes every time the string concat happens.

s = "asd"
id(s)
s = s*3
id(s)
id will be different for both cases.

everytime slicing is done, a new reference is created for that string.

in and not in in strings -- both looks for the continuous substrings in the strings

eg: s = "Hello"
if 'el' in s: --> it will be a true statement because el is coming together and it continues.

comparison of 2 strings --> are on the basis of ASCII value till the last char of the string and if 2 strings are not the same then it will go for length.

a = "abcdef" == "abcd"
print(a)
Output : False

a = "abce" >= "abcdef"
print(a)
Output: True

so phele ASCII values check hoti hai last char tak , uske baad length check hoti hai.

string functions:
str.split() -- returns a list from where you have splitted the string.
by default the argument is space so it will split on every space it encounters and store it as the element of the list.

str = "hi,how,are,you"
str.split(","2)
output: ['hi', 'how', 'are,you']
so arg 2 tells how many splits you want of your list , in the above 2 is passed so it splits on first to delimiter(",") and save the rest of the list as a singleC element of the list.

TWO DIMENSIONAL LIST:

Jagged list : - since two dimensional lists are the list of lists , but lists can be of different size and shape , so jagged lists are list of lists where column size is not the same.
Eg:
Li = [[1,2,3,4], [7,7], [9,9,9,9,9,9,9]]

List comprehensions:

Syntax :

1.Li = [<output><for condition><if statement>]

eg:
li _=[1,,23,5,5,6,4]
li = [el **2 for el in li_ if el%2 == 0]

1. If and else statement in list comprehensions:

Li = [<output><if condition><else condition><for loops>]

Note:Suppose else is coming along with if statement then need to put both the statement just before the for loop condition and if the condition is only if statement only , then you can place if after for loop only
(if else sath mein hai toh for loop se phle aayege , and if akela hai toh for loop ke baad)

Multiple for loops in the list comprehensions :
list =[ele for i in li for j in li if i == j]

2. List of list generation using list comprehension:

li = [‘cheerag’,’shubham’,’Rahul’’]

li2d = [[j for j in i]for i in li]

Output : [['c', 'h', 'e', 'e', 'r', 'a', 'g'], ['s', 'h', 'u', 'b', 'h', 'a', 'm'], ['R', 'a', 'h', 'u', 'l']]

2D list input methods:

1, JAGGED LIST
n = 3   #rows
m = 4 #cols

newlist = [[int(j) for j in input().split()] for i in range(n)]
Here we have not made restriction in the column size , so columns can be different for every ith element.

Method 2 :

3,3 #m,n
1,2,3,4,5,6,7,8,9   #list

Output : [[1,2,3],[4,5,6],[7,8,9]]

inputString = input().split()
n,m = int(inputString[0]),int(inputString[1])
b = input().split()

```
New_list = [[ int(b[m*i+j])  for j in range(m)]for i in range(n)]
```
OR
```
n,m= list(map(int,input().split()))
b= [int(x) for x in input().split()]
New_list  = [[b[m*i+j] for j in range(m)]for i in range(n)]
```

Method 3 :
```
#input : 3 3 1 2 3 45 6 7 89 9 99

#now n ,m and list are given in the same line

inputString = input().split()
n,m = int(inputString[0]), int(inputString[1])
b = inputString[2:]
newList = [[int(b[m*i+j])for j in range(m)]for i in range(n)]
```

Printing 2-D list

Method 1 : Printing rows and column (not a good practise for jagged columns)

```
for i in range(n):
    for j in range(m):
        print(a[i][j])
```

Method 2 : this is used in case of jagged as well as normal printing of the 2-D list

```
for i in li:
    for j in i:
        print(j,end="")
    print()
```

Concept of JOIN:

join is the string method which is used to join anything with the string. We can join iterable like List, Tuple, String, Dictionary and Set , or we can say that with the help to join we can make them as strings.

Syntax : “<string to be joined>”.join(<iterable>)

eg: “011”.join([‘a’,b’,’c’]
Output : a011b011c

Note : Join doesn’t join the last element of the iterable in the string.

We can use the concept of join to print the 2-D array where we don't want space character after termination of the last column.

Method 3 :

```
li =[1,2,3,4,5,6,7,8]

for row in li:
        outputString = " ".join([str(ele) for ele in row])
        print(outputString)
```

1 2 3 4 5 6 7 8
Note: basically this method is used when you don't want space at last character.

Tuples, dictionary, and Set :

Tuples:

1. tuples: initialize with () ,
2. when you want to initialize multiple values to a single variable then tuples are used.
   ```
   a = 1,2
   print(a) → (1,2)
   ```
3. Indexing and slicing works exactly the same in case of tuple as of list
   Note:Strings, list & tuples are the ordered sequence.

List VS tuple :
list → mutable
tuple → Immutable (i.e once created cannot be changed or update or delete or add element)

so deleting entries in the tuples or immutable things is prohibited but you can entirely delete that thing from the memory.
eg: del a

Features of tuple :
a = (1,2,3,4)
b = (5,6,7)
c = a + b
output: (1,2,3,4,5,6,7)

d = (c,b)
output: ((1,2,3,4),(5,6,7)) → tuples of tuple is created

eg:
a = 1,2

```
b = (4,5)
d = (a,b)
print(d[0]) →  (1,2)
```

```
a = 1,2
b = (4,5)
d = a+b
print(d[2]) → 4
```

```
a = (“ab”,”abc”,”def”)
print(min(a)) →  ab
```

Variable length input output argument :
Suppose you don't know how many parameters will come from user , then you can use variable parameter using *any variable name (eg *kargs, *args, *more) to handle such scenario:

```
def multiply(a,b,c,*more):
    value = a*b*c
    for i in more:
        value = value * i
    return value

V = multiply(1,2,3,4,5)
print(V)
```

Output: 120

we can also return more than 1 values in python through return statement, and store that into 1 variable using tuple.

```
    def sum_multiply(a,b,*more):
            sum_value = a+b
            m_value = a*b
            for i in more:
                sum_value += i
                m_value*=i
            return sum_value,m_value

s_m = sum_multiply(2,3,4)       #here we are catching more than 1 value in s_m
print(s_m)
```

Output : (9,24)

Note: Agar alag values chaiye sabki toh return krte hue alag variables mein save kar skte hai.

Dictionaries :
created using {}, mutable in nature ,i.e. add,update,delete and other operations can be done.

creating dict using fromkeys method:
d.fromkeys([2,3,6])
Output → {2: None, 3: None, 6: None}

d.fromkeys([2,3,6],'s')
{2: 's', 3: 's', 6: 's'}

Accessing values in dict :

d1={"1":a,"2":b,"3":c}
d1["1"]
Output : a
If you want to access without a loop you need to give the exact same key name in order to access it's value.

We can also access dict value using get function.
d1.get(2) → b

so **get** and **[ ]** is similar only if element is present, if element is missing → [ ] generates an error while get suppress the error and return None.
also in case of get,
a.get(22,"key not present") → it will look for key 22 in a dict , if 22 is present then value is returned else our custom msg will be displayed.

Iteration over dictionary :
d.keys() --> will give list of keys
d.values --> will give list of values
d.items() --> gives pairs of keys and values

for i in d:
  print(i) → wil give all the keys , to access the corresponding values use d[i]

for i in d:
  print(i,d[i]) → will give keys and its values

directly checking keys if present or not using IN method
d = {1:'a',2:'b',3:'c'}
1 in d
output : True

'b' in d

output: False

Note : in method of dict doesn't work on the values , it only works on keys.

Dictionary : Update method

Update parameter mein passed dict mein and actual dict mein agar koi common values hoti hai toh woh passed dict ke items se replace ho jaati hai.
eg :
d = {1: 2, 'abc': 5, 'def': 7}
d1={1: 200, 'abcd': 5, 'defe': 7}
d.update(d1) Output → {1: 200, 'abc': 5, 'def': 7, 'abcd': 5, 'defe': 7}

Dictionary : Pop method vs delete method

dict.pop(key) → any key- value pair can be deleted using the pop method, take key as args.
del d[1] → key-value pair can be deleted, takes index as argument.

Clear() → clear all key-value pairs , but dict still exists in the memory.
del d2 → delete the dict from the memory.