```
f = open("test.txt") #Open a File
f = open("test.txt", 'r') #Open a File as Read-Only
f = open("test.txt", 'w') #Open a File to Write
f = open("test.txt", 'a') #Open a File to Append
f.close() #Close a File
f.write("write this line\n") #Write to a File
f.writelines(lines) #Write a list of lines
##Read a File
f.read() #reads to the EOF
f.read(n) #reads n characters Current File Position
with open(filename) as f:
    file_contents = f.read()
# the open_file object has automatically been closed.
f.tell() #Current file position
'''Change the File Position''' f.seek(n) #where n is new position.
f.readline() #Read a single line
list f.readlines() #Put all file lines into a
```

Some Basic python operators

```
** #Exponent
% #remainder
// #Integer division
/ #Normal division
| #bitwise OR, union of sets, updating dicts and counters
#Bitwise Operator
'0b{:04b}'.format(0b1100 & 0b1010) # '0b1000' and
'0b{:04b}'.format(0b1100 | 0b1010) # '0b1110' or
'0b{:04b}'.format(0b1100 ^ 0b1010) # '0b0110' exclusive or
'0b{:04b}'.format(0b1100 >> 2)    # '0b0011' shift right
'0b{:04b}'.format(0b0011 << 2)    # '0b1100' shift left

'''Lambda'''
add = lambda x, y: x + y
add(5, 3)
8
#Can be used with (list).sort(), sorted(), min(), max(), (heapq).nlargest,nsmallest(), map()
# a=3,b=8,target=10
min((b,a), key=lambda x: abs(target - x)) # 8
>>> ids = ['id1', 'id2', 'id30', 'id3', 'id22', 'id100']
>>> print(sorted(ids)) # Lexicographic sort
['id1', 'id100', 'id2', 'id22', 'id3', 'id30']
>>> sorted_ids = sorted(ids, key=lambda x: int(x[2:])) # Integer sort
>>> print(sorted_ids)
['id1', 'id2', 'id3', 'id22', 'id30', 'id100']
trans = lambda x: list(al[i] for i in x) # apple, a->0..
print(trans(words[0])) # [0, 15, 15, 11, 4]
#Lambda can sort by 1st, 2nd element in tuple

sorted([('abc', 121),('bbb',23),('abc', 148),('bbb', 24)], key=lambda x: (x[0],x[1]))
# [('abc', 121), ('abc', 148), ('bbb', 23), ('bbb', 24)]



def make_adder(n):
      return lambda x: x + n
>>> plus_3 = make_adder(3)
plus_3(4)  ###=7
input() #Inputs input in the form of string
```

Itertools

izip() returns an iterator that combines the elements of several iterators into tuples.

```
for i in izip([5, 6, 7], [14, 15, 16]):
    print(i),
```

```
# (5, 14) (6, 15) (7, 16)

itertools.accumulate(iterable[, func])
# Makes an iterator that returns the results of a function.
data = [1, 2, 3, 4, 5]
result = itertools.accumulate(data, operator.mul)
for each in result:
   print(each)
1, 2, 6, 24, 120

#Passing a function is optional:
result = itertools.accumulate(data)
1,3,5,7,9

itertools.combinations_with_replacement(iterable, r)
#Just like combinations(), but allows individual elements to be repeated more than once.

itertools.cycle(iterable) # cycles through an iterator endlessly.

itertools.chain(*iterables) #Take a series of iterables and return them as one long iterable.
>>> colors = ['red', 'orange', 'yellow', 'green', 'blue']
>>> shapes = ['circle', 'triangle', 'square', 'pentagon']
>>> result = itertools.chain(colors, shapes)
>>> for each in result:
>>>    print(each)
red orange yellow green blue circle triangle square pentagon
#####
itertools.compress(data, selectors)
>> shapes = ['circle', 'triangle', 'square', 'pentagon']
>>> selections = [True, False, True, False]
>>> result = itertools.compress(shapes, selections)
>>> for each in result:
>>>    print(each)
circle square

itertools.dropwhile(predicate, iterable)
#Make an iterator that drops elements from the iterable as long as the predicate is true; afterwards, returns every element.
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
>>> result = itertools.dropwhile(lambda x: x<5, data)
5 6 7 8 9 10 1

itertools.product(num_data, alpha_data) #Creates the cartesian products from a series of iterables.
itertools.repeat(object[, times])
itertools.starmap(function, iterable)
#Makes an iterator that computes the function using arguments obtained from the iterable.
itertools.zip_longest(*iterables, fillvalue=None)

import itertools
data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
list(itertools.accumulate(data)) # [3, 7, 13, 15, 16, 25, 25, 32, 37, 45]
list(itertools.accumulate(data, max))  # [3, 4, 6, 6, 6, 9, 9, 9, 9, 9]
cashflows = [1000, -90, -90, -90, -90]  # Amortize a 5% loan of 1000 with 4 annual payments of
90
list(itertools.accumulate(cashflows, lambda bal, pmt: bal*1.05 + pmt)) [1000, 960.0, 918.0,
873.9000000000001, 827.5950000000001]
for k,v in itertools.groupby("aabbbc")    # group by common letter
    print(k)                    # a,b,c
    print(list(v))              # [a,a],[b,b,b],[c,c]
```

Counter

```
from collections import Counter
import collections
count = Counter("hello") # Counter({'h': 1, 'e': 1, 'l': 2, 'o': 1})
count['l'] # 2
count['l'] += 1
count['l'] # 3
'''Get counter k most common in list of tuples'''
# [1,1,1,2,2,3]
# Counter  [(1, 3), (2, 2)]
def topKFrequent(self, nums: List[int], k: int) -> List[int]:
    if len(nums) == k:
        return nums
    return [n[0] for n in Counter(nums).most_common(k)] # [1,2]
elements()
'''lets you walk through each number in the Counter'''

def intersect(self, nums1: List[int], nums2: List[int]) -> List[int]:
    c1 = collections.Counter(nums1) # [1,2,2,1]
    c2 = collections.Counter(nums2) # [2,2]
    dif = c1 & c2                   # {2:2}
    return list(dif.elements())     # [2,2]
'''operators work on Counter'''

c = Counter(a=3, b=1)
d = Counter(a=1, b=2)
c + d # {'a': 4, 'b': 3}
c - d # {'a': 2}
c & d # {'a': 1, 'b': 1}
c | d # {'a': 3, 'b': 2}
c = Counter(a=2, b=-4)
+c # {'a': 2}
-c # {'b': 4}
'''Default Dict'''
d={}
print(d['Grapes'])# This gives Key Error
from collections import defaultdict
d = defaultdict(int) # set default
print(d['Grapes']) # 0, no key error
d = collections.defaultdict(lambda: 1)
print(d['Grapes']) # 1, no key error
from collections import defaultdict
dd = defaultdict(list)
dd['key'].append(1) # defaultdict(<class 'list'>, {'key': [1]})
dd['key'].append(2) # defaultdict(<class 'list'>, {'key': [1, 2]})
```

Zip

```
'''Combine two dicts or lists'''
s1 = {2, 3, 1}
s2 = {'b', 'a', 'c'}
list(zip(s1, s2)) # [(1, 'a'), (2, 'c'), (3, 'b')]
'''Traverse in Parallel'''

letters = ['a', 'b', 'c']
numbers = [0, 1, 2]
for l, n in zip(letters, numbers):
  print(f'Letter: {l}') # a,b,c
  print(f'Number: {n}') # 0,1,2
'''Empty in one list is ignored'''

letters = ['a', 'b', 'c']
numbers = []
for l, n in zip(letters, numbers):
  print(f'Letter: {l}') #
```

```
 print(f'Number: {n}') #
'''Compare characters of alternating words'''

for a, b in zip(words, words[1:]):
    for c1, c2 in zip(a,b):
        print("c1 ", c1, end=" ")
        print("c2 ", c2, end=" ")
'''Passing in * unpacks a list or other iterable, making each of its elements a separate argument.'''
a = [[1,2],[3,4]]
test = zip(*a)
print(test) # (1, 3) (2, 4)
matrix = [[1,2,3],[4,5,6],[7,8,9]]
test = zip(*matrix)
print(*test) # (1, 4, 7) (2, 5, 8) (3, 6, 9)
'''Useful when rotating a matrix'''

# matrix = [[1,2,3],[4,5,6],[7,8,9]]
matrix[:] = zip(*matrix[::-1]) # [[7,4,1],[8,5,2],[9,6,3]]
'''Iterate through chars in a list of strs'''

strs = ["cir","car","caa"]
for i, l in enumerate(zip(*strs)):
    print(l)
    # ('c', 'c', 'c')
    # ('i', 'a', 'a')
    # ('r', 'r', 'a')
'''Diagonals can be traversed with the help of a list'''

"""
[[1,2,3],
 [4,5,6],
 [7,8,9],
 [10,11,12]]
"""
def printDiagonalMatrix(self, matrix: List[List[int]]) -> bool:
    R = len(matrix)
    C = len(matrix[0])

    tmp = [[] for _ in range(R+C-1)]

    for r in range(R):
        for c in range(C):
            tmp[r+c].append(matrix[r][c])

    for t in tmp:
        for n in t:
            print(n, end=' ')
        print("")
"""
 1,
 2,4
 3,5,7
 6,8,10
 9,11
 12
"""
```

Linear Search

```
def linear_search(values, search_for):
    search_at = 0
    search_res = False
 # Match the value with each data element
```

```
while search_at < len(values) and search_res is False:
   if values[search_at] == search_for:
      search_res = True
else:
   search_at = search_at + 1
return search_res
l = [64, 34, 25, 12, 22, 11, 90]
print(linear_search(l, 12))
print(linear_search(l, 91))
```

Strings

```
name = 'Mosh'
message = f'Hi, my name is {name}'
message.upper() # to convert to uppercase
message.lower() # to convert to lowercase
message.title() # to capitalize the first letter of every word
message.find('p') # returns the index of the first occurrence of p (or -1 if not found)
message.replace('p', 'q')
#To check if a string contains a character (or a sequence of characters), we use the in operator:
contains = 'Python' in course
##Found from another source
str1.find('x')        # find first location of char x and return index
str1.rfind('x')       # find first int location of char x from reverse
#Parse a log on ":"
l = "0:start:0"
tokens = l.split(":")
print(tokens) # ['0', 'start', '0']
##Reverse works with built in split, [::-1] and " ".join()
s = "the sky  is blue"
def reverseWords(self, s: str) -> str:
  wordsWithoutWhitespace = s.split() # ['the', 'sky', 'is', 'blue']
  reversedWords = wordsWithoutWhitespace[::-1] # ['blue', 'is', 'sky', 'the']
  final = " ".join(reversedWords) # blue is sky the
##Manual split based on isalpha()

def splitWords(input_string) -> list:
  words = [] #
  start = length = 0
  for i, c in enumerate(input_string):
    if c.isalpha():
      if length == 0:
        start = i
        length += 1
      else:
        words.append(input_string[start:start+length])
        length = 0
  if length > 0:
    words.append(input_string[start:start+length])
  return words
##Test type of char

def rotationalCipher(input, rotation_factor):
  rtn = []
  for c in input:
    if c.isupper():
      ci = ord(c) - ord('A')
      ci = (ci + rotation_factor) % 26
      rtn.append(chr(ord('A') + ci))
    elif c.islower():
      ci = ord(c) - ord('a')
      ci = (ci + rotation_factor) % 26
      rtn.append(chr(ord('a') + ci))
    elif c.isnumeric():
```

```
    ci = ord(c) - ord('0')
    ci = (ci + rotation_factor) % 10
    rtn.append(chr(ord('0') + ci))
  else:
    rtn.append(c)
 return "".join(rtn)
#AlphaNumberic

isalnum()
##Get charactor index
print(ord('A')) # 65
print(ord('B')-ord('A')+1) # 2
print(chr(ord('a') + 2)) # c
##Replace characters or strings

def isValid(self, s: str) -> bool:
 while '[]' in s or '()' in s or '{}' in s:
  s = s.replace('[]','').replace('()','').replace('{}','')
 return len(s) == 0
##Insert values in strings

txt3 = "My name is {}, I'm {}".format("John",36) # My name is John, I'm 36
##Multiply strings/lists with *, even booleans which map to True(1) and False(0)

'meh' * 2 # mehmeh
['meh'] * 2 # ['meh', 'meh']
['meh'] * True #['meh']
['meh'] * False #[]
Find substring in string

txt = "Hello, welcome to my world."
x = txt.find("welcome")  # 7
startswith and endswith are very handy

str = "this is string example....wow!!!"
str.endswith("!!") # True
str.startswith("this") # True
str.endswith("is", 2, 4) # True
##Python3 format strings

name = "Eric"
profession = "comedian"
affiliation = "Monty Python"
message = (
    f"Hi {name}. "
    f"You are a {profession}. "
    f"You were in {affiliation}."
)
message
'Hi Eric. You are a comedian. You were in Monty Python.'
##Print string with all chars, useful for debugging

print(repr("meh\n"))    # 'meh\n'
```

Lists

```
numbers = [1, 2, 3, 4, 5]
numbers[0] # returns the first item
numbers[1] # returns the second item
numbers[-1] # returns the first item from the end
numbers[-2] # returns the second item from the end
numbers.append(6) # adds 6 to the end
```

```
numbers.insert(0, 6) # adds 6 at index position of 0
numbers.remove(6) # removes 6
numbers.pop() # removes the last item
numbers.clear() # removes all the items
numbers.index(8) # returns the index of first occurrence of 8
numbers.sort() # sorts the list
numbers.reverse() # reverses the list
numbers.copy() # returns a copy of the list
#Stacks are implemented with Lists. Stacks are good for parsing and graph traversal

test = [0] * 100 # initialize list with 100 0's
#2D
rtn.append([])
rtn[0].append(1) # [[1]]
#List Comprehension
number_list = [ x for x in range(20) if x % 2 == 0]
print(number_list) # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
#Reverse a list
ss = [1,2,3]
ss.reverse()
print(ss) #3,2,1
#Join list
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
list3 = list1 + list2 # ['a', 'b', 'c', 1, 2, 3]
```

Dictionaries

```
We use dictionaries to store
key/valuepairs.'''
customer = {
 "name": "John Smith",
 "age": 30,
 "is_verified": True
}
'''We can use strings or numbers to define keys. They should be unique. We can use
any types for the values.'''
customer["name"] # returns “John Smith”
customer["type"] # throws an error
customer.get("type", "silver") # returns “silver”
customer["name"] = "new name"
##Hashtables are implemented with dictionaries
d = {'key': 'value'}       # Declare dict{'key': 'value'}
d['key'] = 'value'         # Add Key and Value
{x:0 for x in {'a', 'b'}}   # {'a': 0, 'b': 0} declare through comprehension
d['key'])                # Access value
d.items()                 # Items as tuple list dict_items([('key', 'value')])
if 'key' in d: print("meh")  # Check if value exists
par = {}
par.setdefault(1,1)        # returns 1, makes par = { 1 : 1 }
par = {0:True, 1:False}
par.pop(0)               # Remove key 0, Returns True, par now {1: False}
for k in d: print(k)       # Iterate through keys
#Create Dict of Lists that match length of list to count votes
votes = ["ABC","CBD","BCA"]
rnk = {v:[0] * len(votes[0]) for v in votes[0]}
print(rnk) # {'A': [0, 0, 0], 'B': [0, 0, 0], 'C': [0, 0, 0]}
```

Random Module

```
import random
random.random() # returns a float between 0 to 1
```

```
random.randint(1, 6) # returns an int between 1 to 6
print(random.uniform(20, 60)) #return a random number with uniform distribution between 20 and 60
members = ['John', 'Bob','Mary']
leader = random.choice(members) # randomly picks an item
'''
weights is an optional parameter which is used to weigh the possibility for each value.
k is an optional parameter that is used to define the length of the returned list.
'''
mylist = ["apple", "banana", "mango"]
print(random.choices(mylist, weights = [10, 1, 1], k = 6))
'''Output: ['apple', 'banana', 'apple', 'apple', 'apple', 'banana']'''
```

Sorting

```
'''Python has built-in method for sorting a
list and it performs the sorting
in O(n * log(n)) Time. It modifies the original list and does not create or
return new list. We can even the order (descending) using the parameter
reverse'''
nums = [14, 70, 45, 20, 40, 89, 10]
nums.sort()
print(nums) # [10, 14, 20, 40, 45, 70, 89]
nums.sort(reverse=True)
print(nums) # [89, 70, 45, 40, 20, 14, 10]
chars = ['X', 'A','E', '1', '2']
chars.sort()
print(chars) # ['1', '2', 'A', 'E', 'X']
strings = ["py", "Py", "print", "Virtual", "venv", "pprint"]
strings.sort()
print(strings)
# ["py", "Py", "print", "Virtual", "venv", "pprint"]
```

Binary Tree

```
'''DFS Pre, In Order, and Post order Traversal

Preorder
encounters roots before leaves
Create copy
Inorder
flatten tree back to original sequence
Get values in non-decreasing order in BST
Post order
encounter leaves before roots
Helpful for deleting
Recursive'''

"""
    1
   / \
  2   3
 / \
4   5
"""
# PostOrder 4 5 2 3 1  (Left-Right-Root)
def postOrder(node):
  if node is None:
    return
  postorder(node.left)
  postorder(node.right)
  print(node.value, end=' ')
'''Iterative PreOrder'''

# PreOrder  1 2 4 5 3 (Root-Left-Right)
```

```
def preOrder(tree_root):
  stack = [(tree_root, False)]
  while stack:
    node, visited = stack.pop()
    if node:
      if visited:
        print(node.value, end=' ')
      else:
        stack.append((node.right, False))
        stack.append((node.left, False))
        stack.append((node, True))
'''Iterative InOrder'''

# InOrder   4 2 5 1 3 (Left-Root-Right)
def inOrder(tree_root):
  stack = [(tree_root, False)]
  while stack:
    node, visited = stack.pop()
    if node:
      if visited:
        print(node.value, end=' ')
      else:
        stack.append((node.right, False))
        stack.append((node, True))
        stack.append((node.left, False))
'''Iterative PostOrder'''

# PostOrder 4 5 2 3 1  (Left-Right-Root)
def postOrder(tree_root):
  stack = [(tree_root, False)]
  while stack:
    node, visited = stack.pop()
    if node:
      if visited:
        print(node.value, end=' ')
      else:
        stack.append((node, True))
        stack.append((node.right, False))
        stack.append((node.left, False))
'''Iterative BFS(LevelOrder)'''

from collections import deque
#BFS levelOrder 1 2 3 4 5
def levelOrder(tree_root):
  queue = deque([tree_root])
  while queue:
    node = queue.popleft()
    if node:
        print(node.value, end=' ')
        queue.append(node.left)
        queue.append(node.right)

def levelOrderStack(tree_root):
    stk = [(tree_root, 0)]
    rtn = []
    while stk:
        node, depth = stk.pop()
        if node:
            if len(rtn) < depth + 1:
                rtn.append([])
            rtn[depth].append(node.value)
            stk.append((node.right, depth+1))
```

```
            stk.append((node.left, depth+1))
    print(rtn)
    return True

def levelOrderStackRec(tree_root):
    rtn = []

    def helper(node, depth):
        if len(rtn) == depth:
            rtn.append([])
        rtn[depth].append(node.value)
        if node.left:
            helper(node.left, depth + 1)
        if node.right:
            helper(node.right, depth + 1)

    helper(tree_root, 0)
    print(rtn)
    return rtn
'''Traversing data types as a graph, for example BFS'''

def removeInvalidParentheses(self, s: str) -> List[str]:
    rtn = []
    v = set()
    v.add(s)
    if len(s) == 0: return [""]
    while True:
        for n in v:
            if self.isValid(n):
                rtn.append(n)
        if len(rtn) > 0: break
        level = set()
        for n in v:
            for i, c in enumerate(n):
                if c == '(' or c == ')':
                    sub = n[0:i] + n[i + 1:len(n)]
                    level.add(sub)
        v = level
    return rtn
'''Reconstructing binary trees

Binary tree could be constructed from preorder and inorder traversal
Inorder traversal of BST is an array sorted in the ascending order
Convert tree to array and then to balanced tree'''
def balanceBST(self, root: TreeNode) -> TreeNode:
    self.inorder = []

    def getOrder(node):
        if node is None:
            return
        getOrder(node.left)
        self.inorder.append(node.val)
        getOrder(node.right)

    # Get inorder treenode ["1,2,3,4"]
    getOrder(root)

    # Convert to Tree
    #       2
    #      1 3
    #        4
    def bst(listTree):
```

```
    if not listTree:
        return None
    mid = len(listTree) // 2
    root = TreeNode(listTree[mid])
    root.left = bst(listTree[:mid])
    root.right = bst(listTree[mid+1:])
    return root

  return bst(self.inorder)
```

Graph

```
'''Build an adjecency graph from edges list'''
# N = 6, edges = [[0,1],[0,2],[2,3],[2,4],[2,5]]
graph = [[] for _ in range(N)]
for u,v in edges:
  graph[u].append(v)
  graph[v].append(u)
# [[1, 2], [0], [0, 3, 4, 5], [2], [2], [2]]
'''Build adjecency graph from traditional tree'''
adj = collections.defaultdict(list)
def dfs(node):
  if node.left:
    adj[node].append(node.left)
    adj[node.left].append(node)
    dfs(node.left)
  if node.right:
    adj[node].append(node.right)
    adj[node.right].append(node)
    dfs(node.right)
dfs(root)
'''Traverse Tree in graph notation'''
# [[1, 2], [0], [0, 3, 4, 5], [2], [2], [2]]
def dfs(node, par=-1):
  for nei in graph[node]:
    if nei != par:
      res = dfs(nei, node)
dfs(0) # 1->2->3->4->5
```

Heapq

```
'''    1
      / \
     2   3
    / \ / \
   5  6 8  7
Priority Queue

Implemented as complete binary tree, which has all levels as full excepted deepest
In a heap tree the node is smaller than its children'''
def maximumProduct(self, nums: List[int]) -> int:
  l = heapq.nlargest(3, nums)
  s = heapq.nsmallest(3, nums)
  return max(l[0]*l[1]*l[2],s[0]*s[1]*l[0])
'''Heap elements can be tuples, heappop() frees the smallest element (flip sign to pop
largest)'''
def kClosest(self, points: List[List[int]], K: int) -> List[List[int]]:
    heap = []
    for p in points:
        distance = sqrt(p[0]* p[0] + p[1]*p[1])
        heapq.heappush(heap,(-distance, p))
        if len(heap) > K:
            heapq.heappop(heap)
    return ([h[1] for h in heap])
'''nsmallest can take a lambda argument'''
```

```
def kClosest(self, points: List[List[int]], K: int) -> List[List[int]]:
    return heapq.nsmallest(K, points, lambda x: x[0]*x[0] + x[1]*x[1])
'''The key can be a function as well in nsmallest/nlargest'''
def topKFrequent(self, nums: List[int], k: int) -> List[int]:
    count = Counter(nums)
    return heapq.nlargest(k, count, count.get)
'''Tuple sort, 1st/2nd element. increasing frequency then decreasing order'''
def topKFrequent(self, words: List[str], k: int) -> List[str]:
    freq = Counter(words)
    return heapq.nsmallest(k, freq.keys(), lambda x:(-freq[x], x))
```

Binary Search

```
def firstBadVersion(self, n):
    l, r = 0, n
    while l < r:
        m = l + (r-l) // 2
        if isBadVersion(m):
            r = m
        else:
            l = m + 1
    return l
"""
12345678
FFTTTTTT
"""
def mySqrt(self, x: int) -> int:
  def condition(value, x) -> bool:
    return value * value > x

  if x == 1:
    return 1

  left, right = 1, x
  while left < right:
    mid = left + (right-left) // 2
    if condition(mid, x):
      right = mid
    else:
      left = mid + 1

  return left - 1
binary search

Binary Search Tree
Use values to detect if number is missing

def isCompleteTree(self, root: TreeNode) -> bool:
    self.total = 0
    self.mx = float('-inf')
    def dfs(node, cnt):
        if node:
            self.total += 1
            self.mx = max(self.mx, cnt)
            dfs(node.left, (cnt*2))
            dfs(node.right, (cnt*2)+1)
    dfs(root, 1)
    return self.total == self.mx
Get a range sum of values

def rangeSumBST(self, root: TreeNode, L: int, R: int) -> int:
    self.total = 0
    def helper(node):
        if node is None:
            return 0
        if L <= node.val <= R:
```

```
            self.total += node.val
        if node.val > L:
            left = helper(node.left)
        if node.val < R:
            right = helper(node.right)
    helper(root)
    return self.total
```

Check if valid

```
def isValidBST(self, root: TreeNode) -> bool:
    if not root:
        return True
    stk = [(root, float(-inf), float(inf))]
    while stk:
        node, floor, ceil = stk.pop()
        if node:
            if node.val >= ceil or node.val <= floor:
                return False
            stk.append((node.right, node.val, ceil))
            stk.append((node.left, floor, node.val))
    return True
```