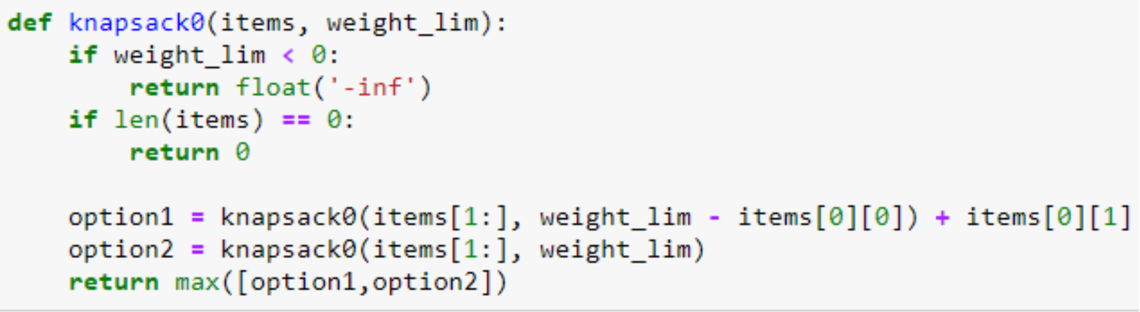
# תוכן עניינים:

## Contents

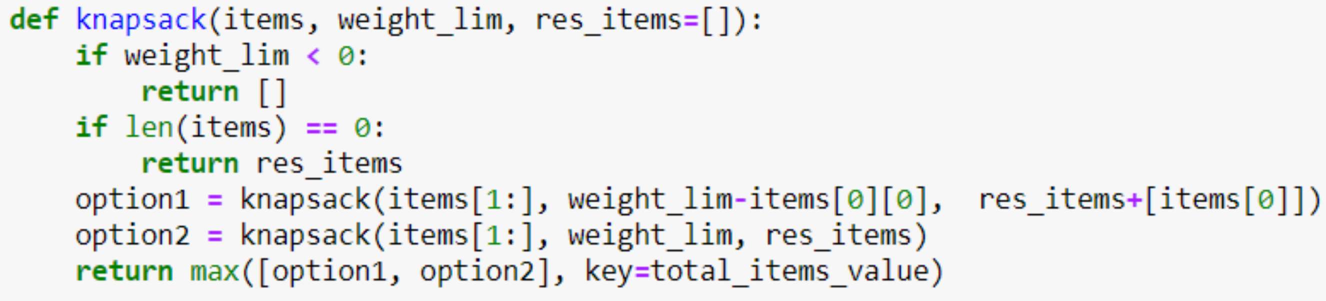
## בעיית הגנב

גרסה א : מה המקסימום שאפשר לקחת ?



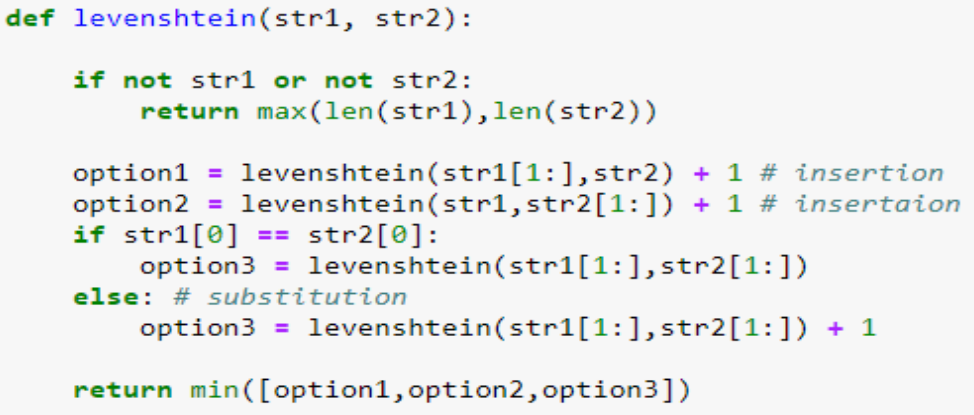
```python
def knapsack0(items, weight_lim):
    if weight_lim < 0:
        return float('-inf')
    if len(items) == 0:
        return 0

    option1 = knapsack0(items[1:], weight_lim - items[0][0]) + items[0][1]
    option2 = knapsack0(items[1:], weight_lim)
    return max([option1,option2])
```

גרסה מתקדמת : מה הרשימה של המקסימום שאפשר לקחת?



```python
def knapsack(items, weight_lim, res_items=[]):
    if weight_lim < 0:
        return []
    if len(items) == 0:
        return res_items
    option1 = knapsack(items[1:], weight_lim-items[0][0],  res_items+[items[0]])
    option2 = knapsack(items[1:], weight_lim, res_items)
    return max([option1, option2], key=total_items_value)
```

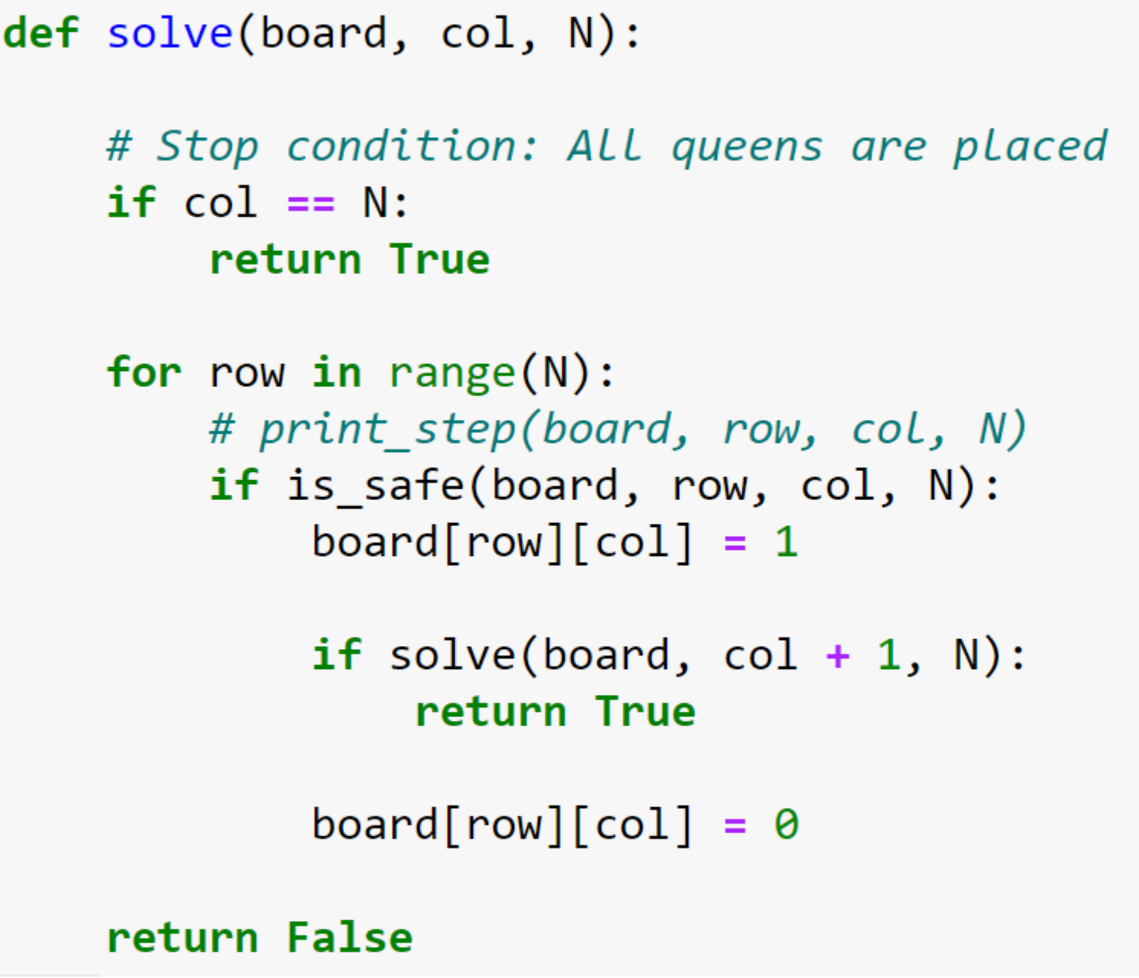
## מרחק



```python
def levenshtein(str1, str2):

    if not str1 or not str2:
        return max(len(str1),len(str2))

    option1 = levenshtein(str1[1:],str2) + 1 # insertion
    option2 = levenshtein(str1,str2[1:]) + 1 # insertaion
    if str1[0] == str2[0]:
        option3 = levenshtein(str1[1:],str2[1:])
    else: # substitution
        option3 = levenshtein(str1[1:],str2[1:]) + 1

    return min([option1,option2,option3])
```

## לוינשטיין:

## בעיית N המלכות:



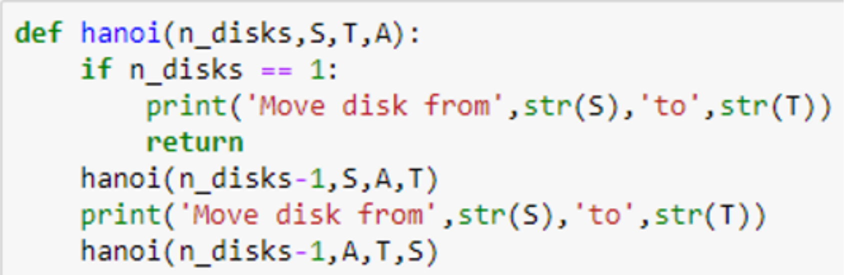
```python
def solve(board, col, N):

    # Stop condition: All queens are placed
    if col == N:
        return True

    for row in range(N):
        # print_step(board, row, col, N)
        if is_safe(board, row, col, N):
            board[row][col] = 1

            if solve(board, col + 1, N):
                return True

            board[row][col] = 0

    return False
```

גרסה מהאינטרנט:

```python
#Number of queens
print ("Enter the number of queens")
N = int(input())#chessboard
#NxN matrix with all elements 0
board = [[0]*N for _ in range(N)]
def is_attack(i, j):
    #checking if there is a queen in row or column
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonals
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False
def N_queen(n):
    #if n is 0, solution found
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            '''checking if we can place a queen here or
not
            queen will not be placed if the place is
being attacked
            or already occupied'''
            if (not(is_attack(i,j))) and
(board[i][j]!=1):
                board[i][j] = 1
                if N_queen(n-1)==True:
                    return True
                board[i][j] = 0
    return False
N_queen(N)
for i in board:
    print (i)
```

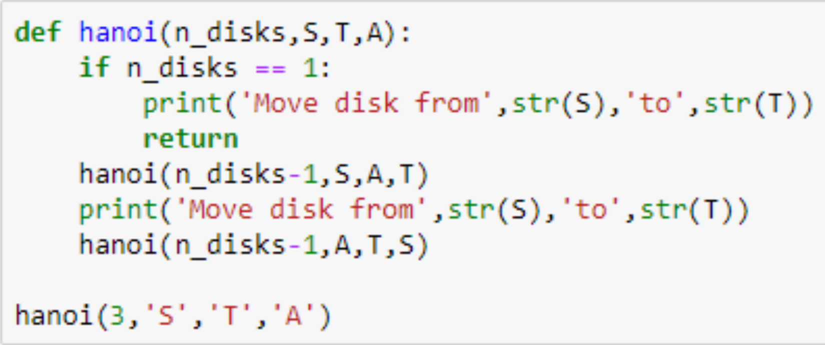
## מגדלי האנוי:

```
def TowerOfHanoi(n , source, destination, auxiliary):
    if n==1:
        print "Move disk 1 from source",source,"to
destination",destination
        return
    TowerOfHanoi(n-1, source, auxiliary, destination)
    print "Move disk",n,"from source",source,"to
destination",destination
    TowerOfHanoi(n-1, auxiliary, destination, source)
```



```python
def hanoi(n_disks,S,T,A):
    if n_disks == 1:
        print('Move disk from',str(S),'to',str(T))
        return
    hanoi(n_disks-1,S,A,T)
    print('Move disk from',str(S),'to',str(T))
    hanoi(n_disks-1,A,T,S)
```

1. If there is only a single disk – easy!
2. Move n-1 disks from S to A (T is the auxiliary rode)
3. Move disk from S to T
4. Move n-1 disks from A to T (S is the auxiliary rode)



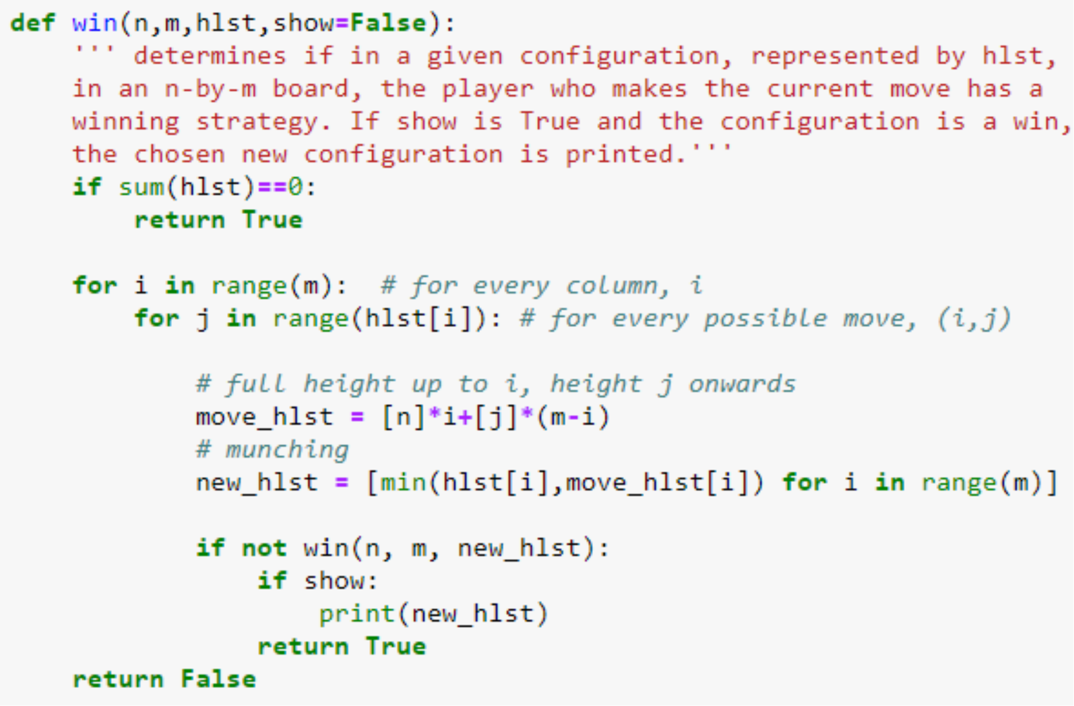
```python
def hanoi(n_disks,S,T,A):
    if n_disks == 1:
        print('Move disk from',str(S),'to',str(T))
        return
    hanoi(n_disks-1,S,A,T)
    print('Move disk from',str(S),'to',str(T))
    hanoi(n_disks-1,A,T,S)

hanoi(3,'S','T','A')
```

# יוסי וליאורה בלי ועם ממואיזציה

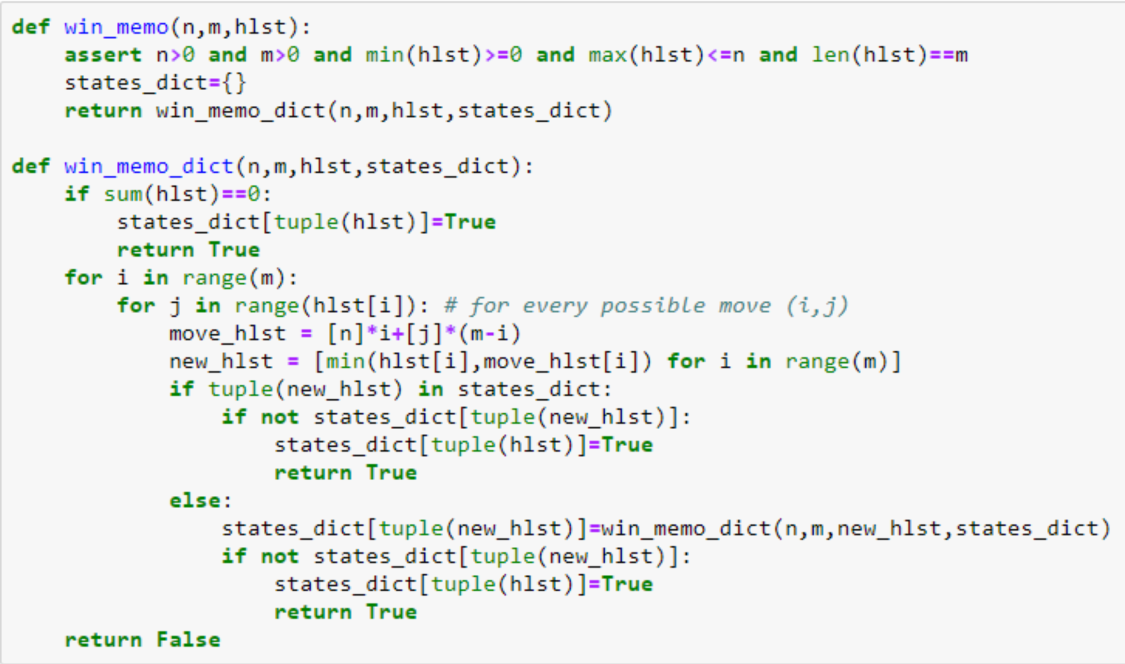
```
def optimize_flowers_selection(flowers, budget):
     return rec_max_beauty(flowers,budget,0)

def rec_max_beauty(lst_flowers,budget,est_val):
     if len(lst_flowers) == 0 or budget == 0: # check if
got to end or no more money left
            return (est_val,budget)
      curr_flower = lst_flowers[0] # save the first current
flower
      option1 =(0,0)   # default if it wont get in the if
condition
      if curr_flower[2] <= budget: # if there is money lef
in the budget and its ok to take another flower
            option1 = rec_max_beauty(lst_flowers[1:],budget-
curr_flower[2],est_val+curr_flower[1]) # take another
flower
      option2 = rec_max_beauty(
lst_flowers[1:],budget,est_val) # dont take another
```

```
flower
    if option1[0] > option2[0]: # check who has more est
value
        return option1
    elif option1[0] == option2[0]:
        if option1[1] > option2[1]:
            return option1
        return option2
    return option2


"""
Q2
"""
mem = dict() # create dictionary to be memory
def get_plants_to_buy_faster(flowers, budget):
    return rec_max_beauty_mem(flowers,budget,0,mem)

def rec_max_beauty_mem(lst_flowers,budget,est_val,mem):
    key = (len(lst_flowers),budget,est_val) # key that
save the situation
    if key in mem:
        return mem[key]
    if len(lst_flowers) == 0 or budget == 0: # if there
are no more flowers left or no money
        mem[key] = (est_val,budget)
    else: # mean the key is not in the dict
        my_flower = lst_flowers[0]  # save the current
first flowers
        optionA =(0,0)
        if my_flower[2] <= budget: # check if there is
money left to take the flower
            optionA =
rec_max_beauty_mem(lst_flowers[1:],budget-
my_flower[2],est_val+my_flower[1],mem)
        optionB =
rec_max_beauty_mem(lst_flowers[1:],budget,est_val,mem)
        if optionA[0] > optionB[0]: # check who has more
est value and update the key value
            mem[key]= optionA
        elif optionA[0] == optionB[0]:
            if optionA[1] > optionB[1]:
                mem[key] = optionA
            else:
                mem[key] = optionB
        else:
            mem[key]= optionB
    return mem[key]
```

# טבלת שוקולד עם ובלי ממואיזציה



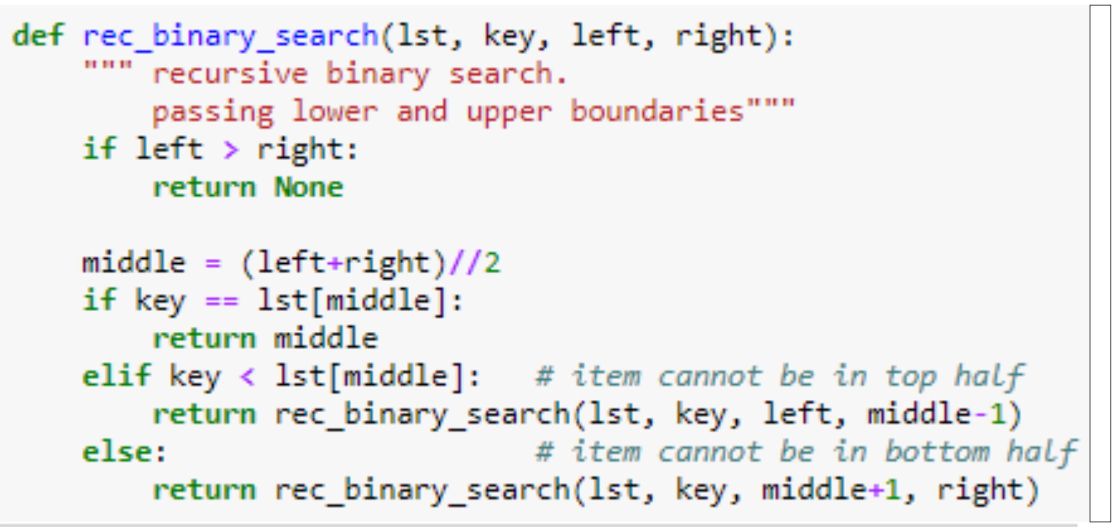
```
def win(n,m,hlst,show=False):
    ''' determines if in a given configuration, represented by hlst,
    in an n-by-m board, the player who makes the current move has a
    winning strategy. If show is True and the configuration is a win,
    the chosen new configuration is printed.'''
    if sum(hlst)==0:
        return True

    for i in range(m):  # for every column, i
        for j in range(hlst[i]): # for every possible move, (i,j)

            # full height up to i, height j onwards
            move_hlst = [n]*i+[j]*(m-i)
            # munching
            new_hlst = [min(hlst[i],move_hlst[i]) for i in range(m)]

            if not win(n, m, new_hlst):
                if show:
                    print(new_hlst)
                return True
    return False
```



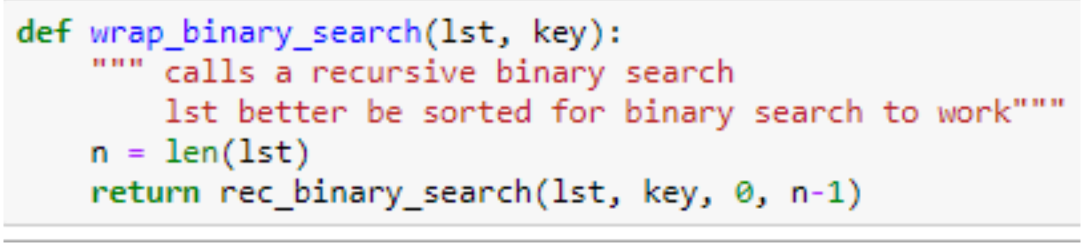
```
def win_memo(n,m,hlst):
    assert n>0 and m>0 and min(hlst)>=0 and max(hlst)<=n and len(hlst)==m
    states_dict={}
    return win_memo_dict(n,m,hlst,states_dict)

def win_memo_dict(n,m,hlst,states_dict):
    if sum(hlst)==0:
        states_dict[tuple(hlst)]=True
        return True
    for i in range(m):
        for j in range(hlst[i]): # for every possible move (i,j)
            move_hlst = [n]*i+[j]*(m-i)
            new_hlst = [min(hlst[i],move_hlst[i]) for i in range(m)]
            if tuple(new_hlst) in states_dict:
                if not states_dict[tuple(new_hlst)]:
                    states_dict[tuple(hlst)]=True
                    return True
            else:
                states_dict[tuple(new_hlst)]=win_memo_dict(n,m,new_hlst,states_dict)
                if not states_dict[tuple(new_hlst)]:
                    states_dict[tuple(hlst)]=True
                    return True
    return False
```

# מיונים וחיפושים

חיפוש בינארי:

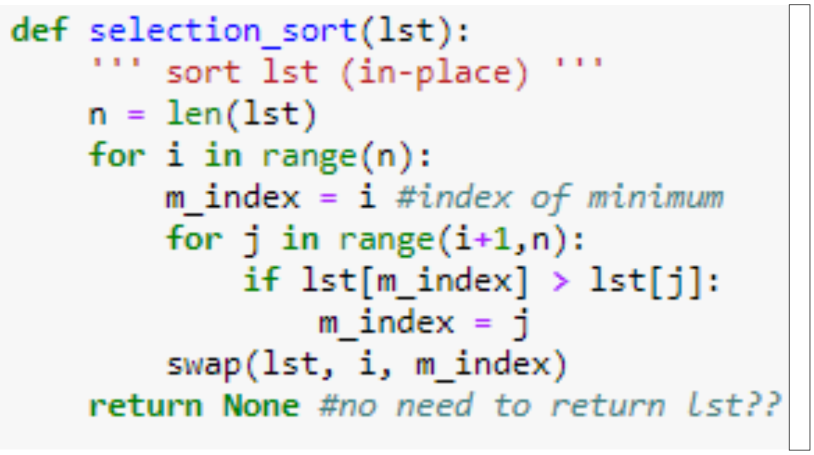


```
def rec_binary_search(lst, key, left, right):
    """ recursive binary search.
        passing lower and upper boundaries"""
    if left > right:
        return None

    middle = (left+right)//2
    if key == lst[middle]:
        return middle
    elif key < lst[middle]:    # item cannot be in top half
        return rec_binary_search(lst, key, left, middle-1)
    else:                      # item cannot be in bottom half
        return rec_binary_search(lst, key, middle+1, right)
```

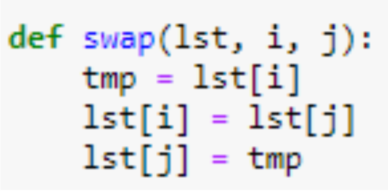


```
def wrap_binary_search(lst, key):
    """ calls a recursive binary search
        lst better be sorted for binary search to work"""
    n = len(lst)
    return rec_binary_search(lst, key, 0, n-1)
```

selection sort

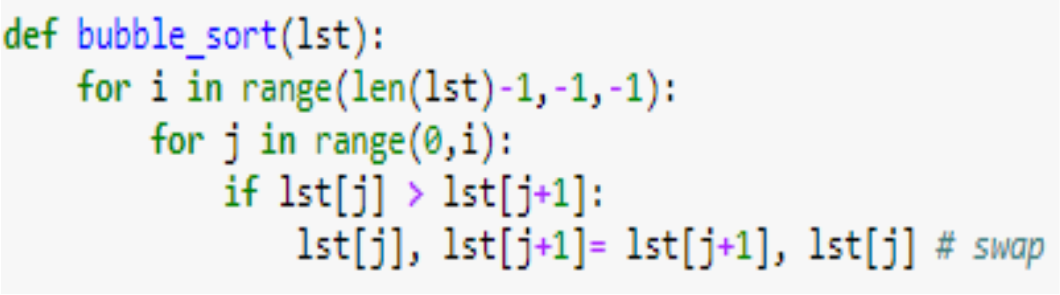


```
def selection_sort(lst):
    ''' sort lst (in-place) '''
    n = len(lst)
    for i in range(n):
        m_index = i #index of minimum
        for j in range(i+1,n):
            if lst[m_index] > lst[j]:
                m_index = j
        swap(lst, i, m_index)
    return None #no need to return lst??
```



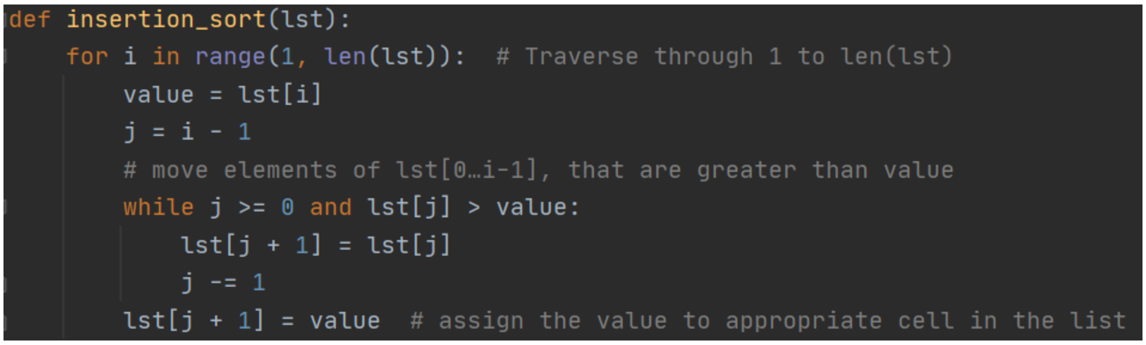
```
def swap(lst, i, j):
    tmp = lst[i]
    lst[i] = lst[j]
    lst[j] = tmp
```

buuble sort



```
def bubble_sort(lst):
    for i in range(len(lst)-1,-1,-1):
        for j in range(0,i):
            if lst[j] > lst[j+1]:
                lst[j], lst[j+1]= lst[j+1], lst[j] # swap
```

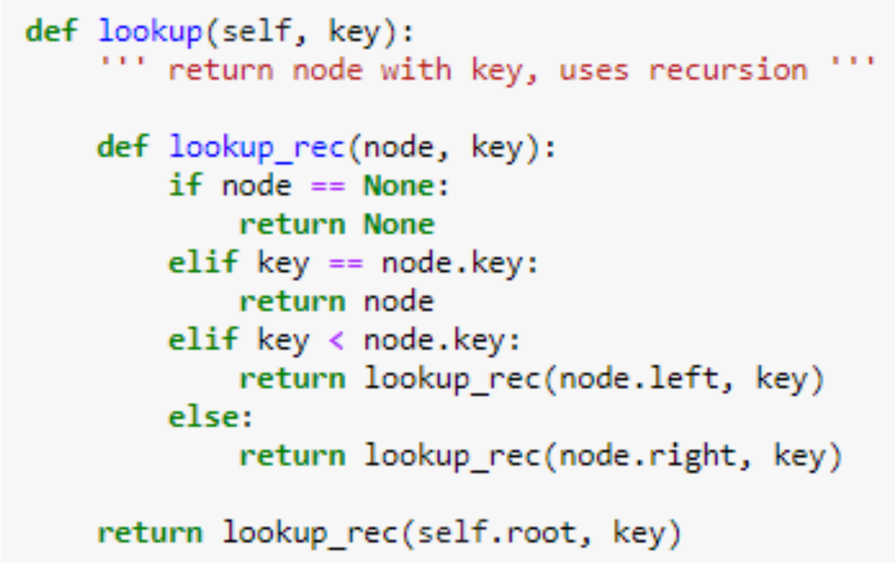
insertion sort



```python
def insertion_sort(lst):
    for i in range(1, len(lst)):  # Traverse through 1 to len(lst)
        value = lst[i]
        j = i - 1
        # move elements of lst[0…i-1], that are greater than value
        while j >= 0 and lst[j] > value:
            lst[j + 1] = lst[j]
            j -= 1
        lst[j + 1] = value  # assign the value to appropriate cell in the list
```

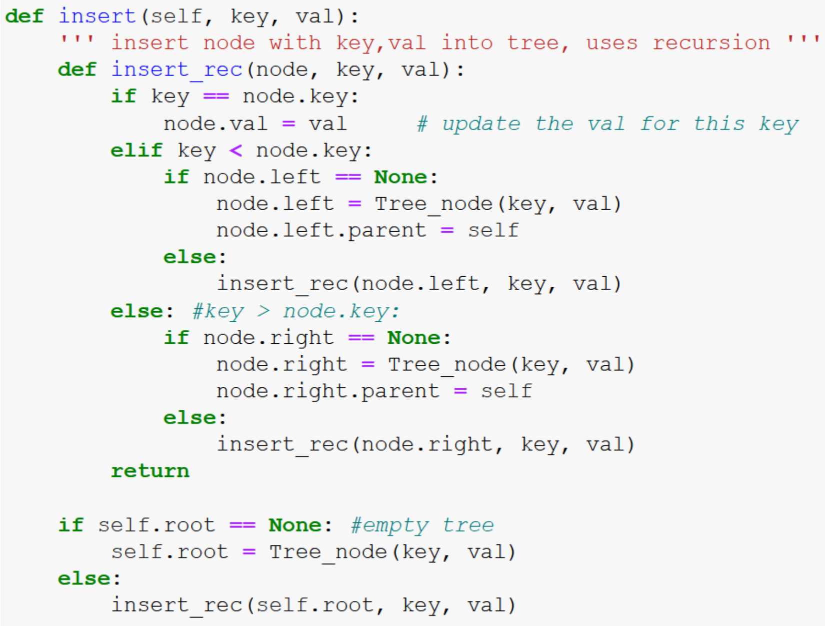
# עצים ועצים בינארי

## חיפוש ערך מסוים בעץ:



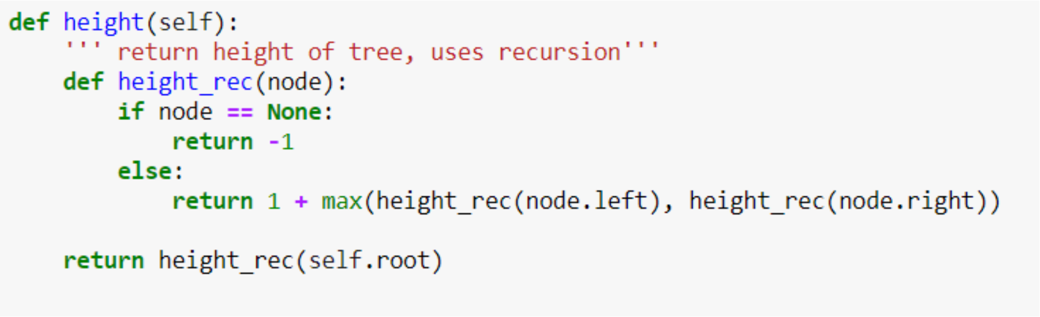
```python
def lookup(self, key):
    ''' return node with key, uses recursion '''

    def lookup_rec(node, key):
        if node == None:
            return None
        elif key == node.key:
            return node
        elif key < node.key:
            return lookup_rec(node.left, key)
        else:
            return lookup_rec(node.right, key)

    return lookup_rec(self.root, key)
```

## הכנסה לעץ :



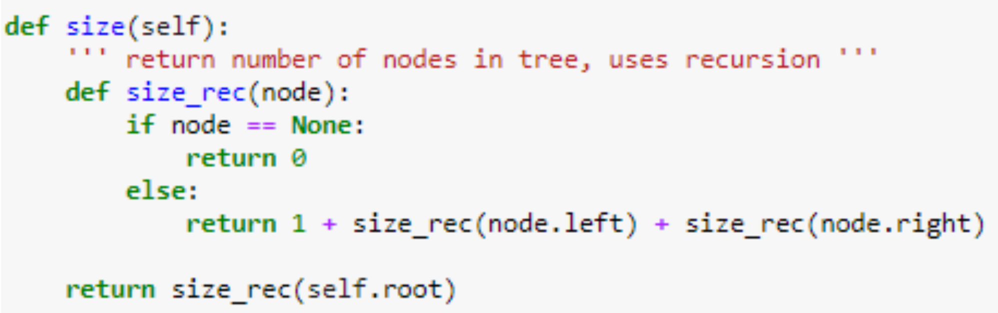
```python
def insert(self, key, val):
    ''' insert node with key,val into tree, uses recursion '''
    def insert_rec(node, key, val):
        if key == node.key:
            node.val = val     # update the val for this key
        elif key < node.key:
            if node.left == None:
                node.left = Tree_node(key, val)
                node.left.parent = self
            else:
                insert_rec(node.left, key, val)
        else: #key > node.key:
            if node.right == None:
                node.right = Tree_node(key, val)
                node.right.parent = self
            else:
                insert_rec(node.right, key, val)
        return

    if self.root == None: #empty tree
        self.root = Tree_node(key, val)
    else:
        insert_rec(self.root, key, val)
```

## גובה העץ:



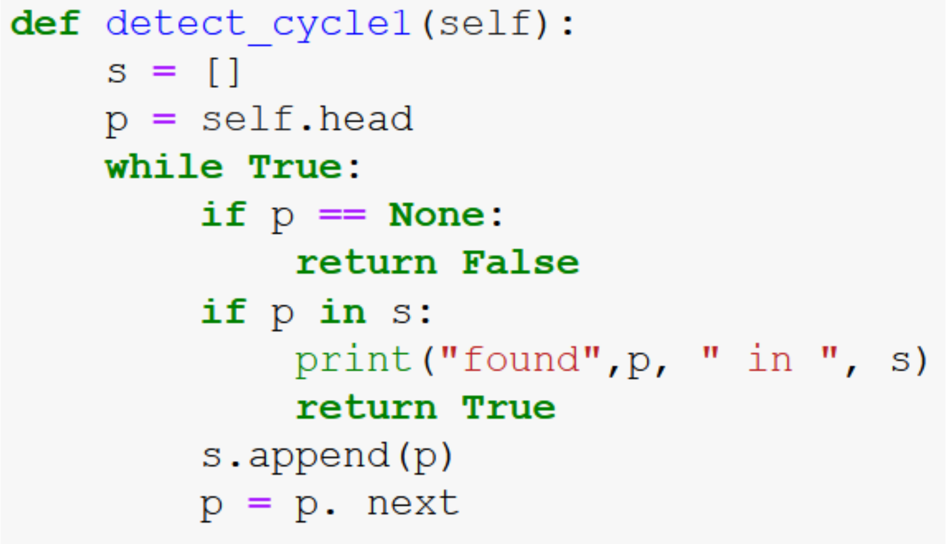
```python
def height(self):
    ''' return height of tree, uses recursion'''
    def height_rec(node):
        if node == None:
            return -1
        else:
            return 1 + max(height_rec(node.left), height_rec(node.right))

    return height_rec(self.root)
```

## גודל העץ



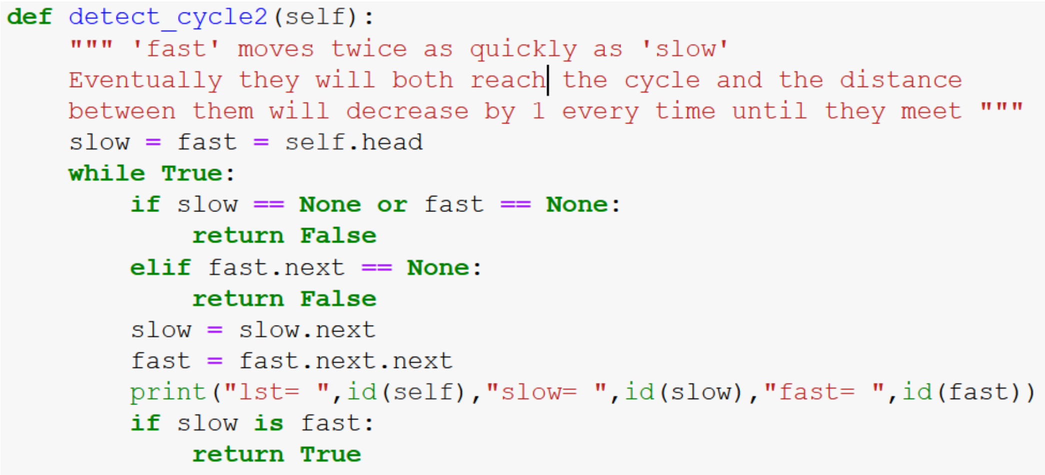
```python
def size(self):
    ''' return number of nodes in tree, uses recursion '''
    def size_rec(node):
        if node == None:
            return 0
        else:
            return 1 + size_rec(node.left) + size_rec(node.right)

    return size_rec(self.root)
```

## צב וארנב גרסה א



```python
def detect_cycle1(self):
    s = []
    p = self.head
    while True:
        if p == None:
            return False
        if p in s:
            print("found",p, " in ", s)
            return True
        s.append(p)
        p = p. next
```

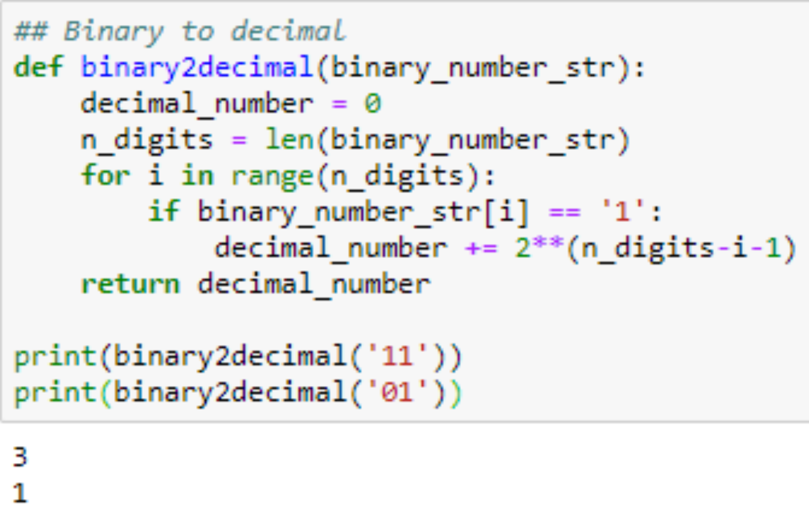
## צב וארנב גרסה ב



```python
def detect_cycle2(self):
    """ 'fast' moves twice as quickly as 'slow'
    Eventually they will both reach the cycle and the distance
    between them will decrease by 1 every time until they meet """
    slow = fast = self.head
    while True:
        if slow == None or fast == None:
            return False
        elif fast.next == None:
            return False
        slow = slow.next
        fast = fast.next.next
        print("lst= ",id(self),"slow= ",id(slow),"fast= ",id(fast))
        if slow is fast:
            return True
```

# מספרים בינארים

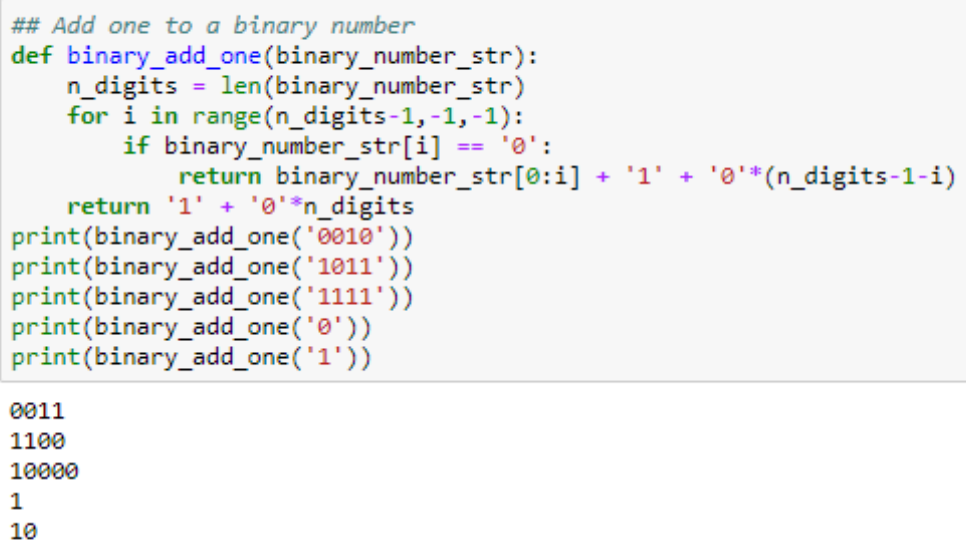
## מבינארי לדצימלי:



```python
## Binary to decimal
def binary2decimal(binary_number_str):
    decimal_number = 0
    n_digits = len(binary_number_str)
    for i in range(n_digits):
        if binary_number_str[i] == '1':
            decimal_number += 2**(n_digits-i-1)
    return decimal_number

print(binary2decimal('11'))
print(binary2decimal('01'))
```

```
3
1
```

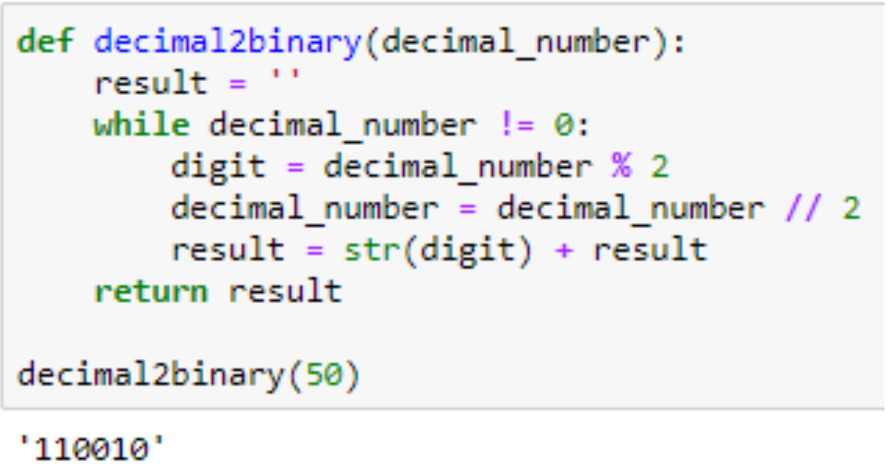
## הוספת אחד למספר בינארי



```python
## Add one to a binary number
def binary_add_one(binary_number_str):
    n_digits = len(binary_number_str)
    for i in range(n_digits-1,-1,-1):
        if binary_number_str[i] == '0':
            return binary_number_str[0:i] + '1' + '0'*(n_digits-1-i)
    return '1' + '0'*n_digits
print(binary_add_one('0010'))
print(binary_add_one('1011'))
print(binary_add_one('1111'))
print(binary_add_one('0'))
print(binary_add_one('1'))
```

```
0011
1100
10000
1
10
```

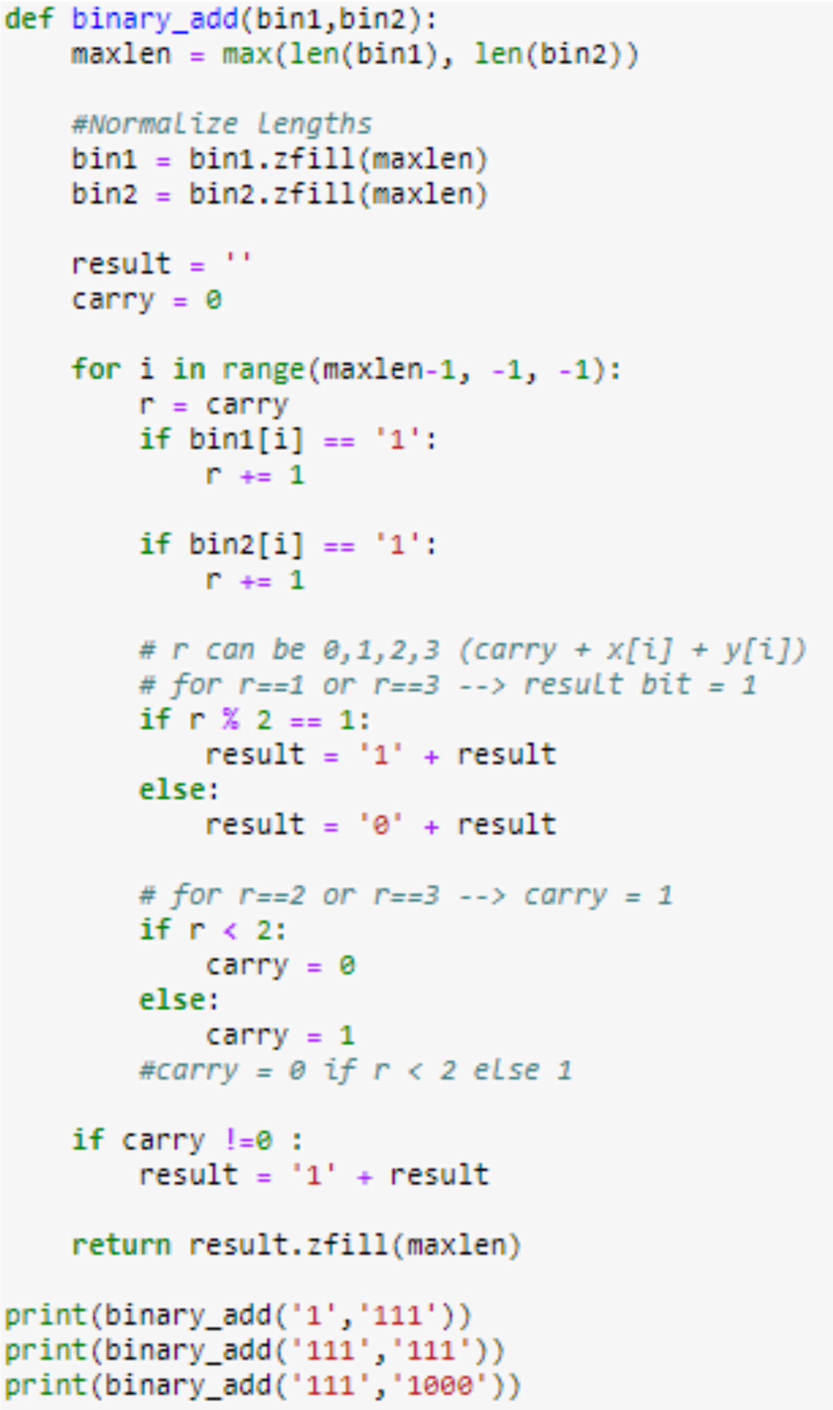
## מדצימלי לבינארי



```
def decimal2binary(decimal_number):
    result = ''
    while decimal_number != 0:
        digit = decimal_number % 2
        decimal_number = decimal_number // 2
        result = str(digit) + result
    return result

decimal2binary(50)
```

```
'110010'
```

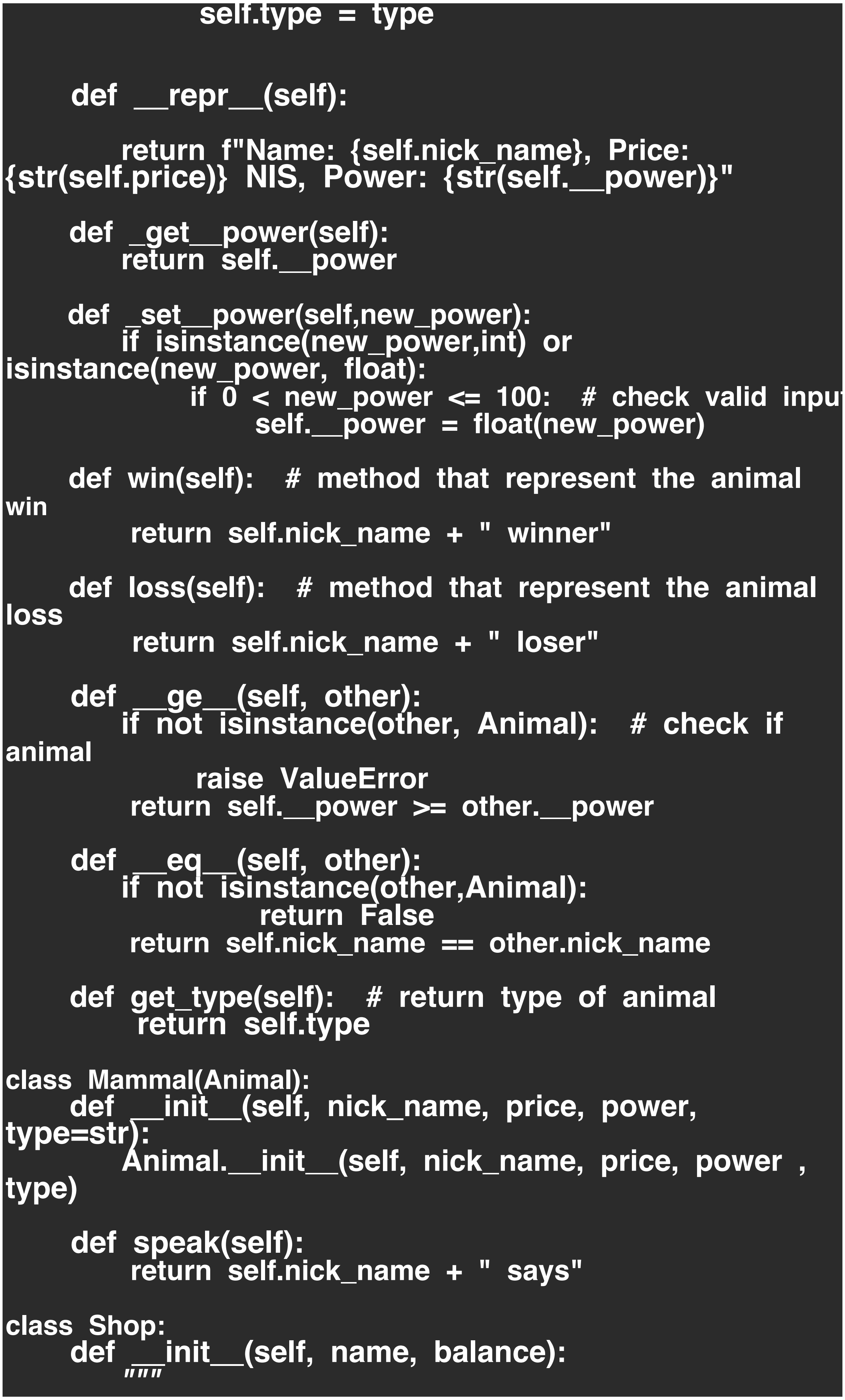
## חיבור שני מספרים בינאריים



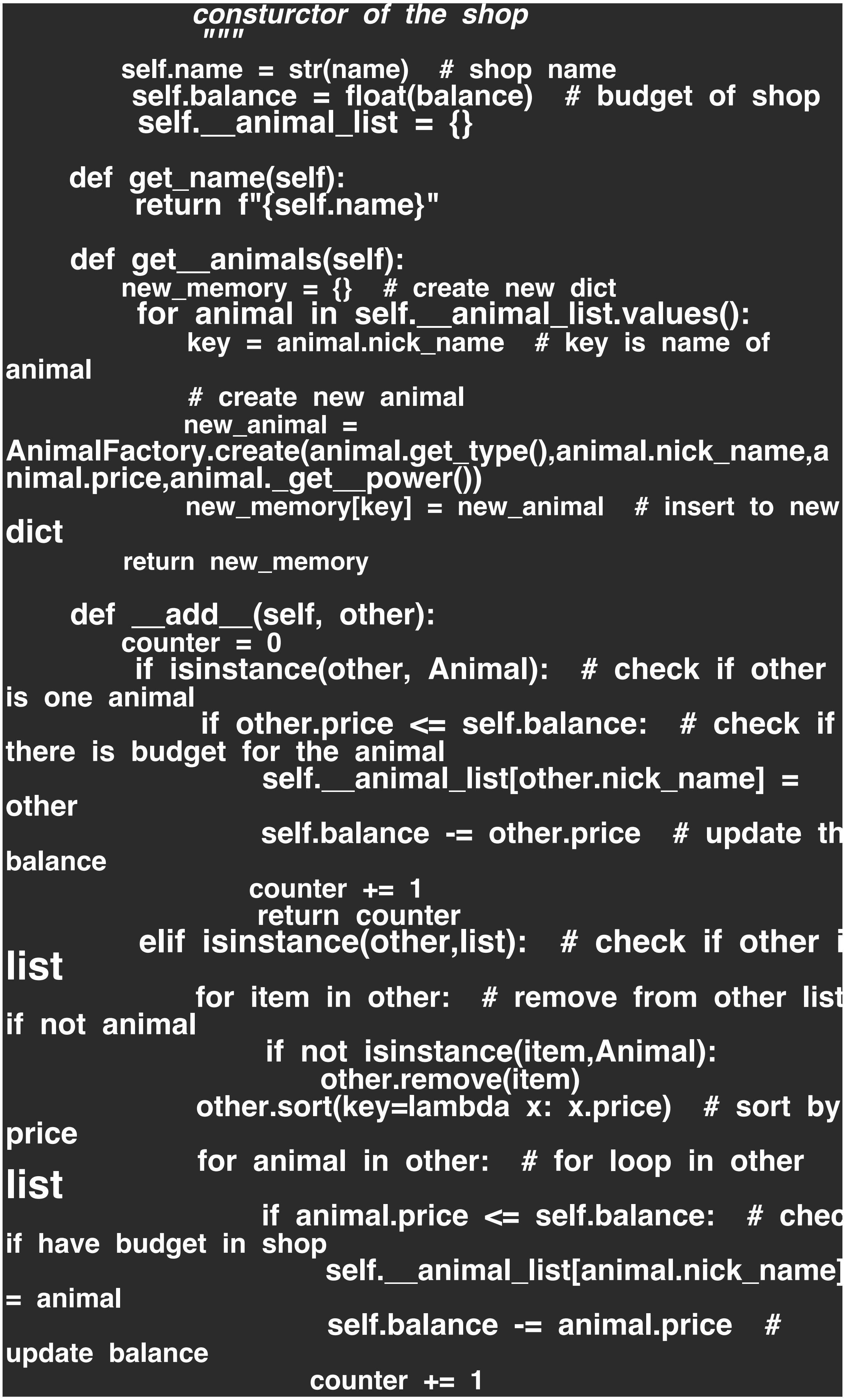
```
def binary_add(bin1,bin2):
    maxlen = max(len(bin1), len(bin2))

    #Normalize lengths
    bin1 = bin1.zfill(maxlen)
    bin2 = bin2.zfill(maxlen)

    result = ''
    carry = 0

    for i in range(maxlen-1, -1, -1):
        r = carry
        if bin1[i] == '1':
            r += 1

        if bin2[i] == '1':
            r += 1

        # r can be 0,1,2,3 (carry + x[i] + y[i])
        # for r==1 or r==3 --> result bit = 1
        if r % 2 == 1:
            result = '1' + result
        else:
            result = '0' + result

        # for r==2 or r==3 --> carry = 1
        if r < 2:
            carry = 0
        else:
            carry = 1
        #carry = 0 if r < 2 else 1

    if carry !=0 :
        result = '1' + result

    return result.zfill(maxlen)

print(binary_add('1','111'))
print(binary_add('111','111'))
print(binary_add('111','1000'))
```

## תכנות מונחה עצמים

```
class  Animal:
      def  __init__(self,nick_name,price,power,type=str):
           if  price  <=  0  or  power  <  0  or  power  >  100:
                  raise  ValueError("enter  valid  power  or
price")
             else:
                  self.nick_name  =  nick_name
                    self.price  =  float(price)
                   self.__power  =  float(power)    private
```

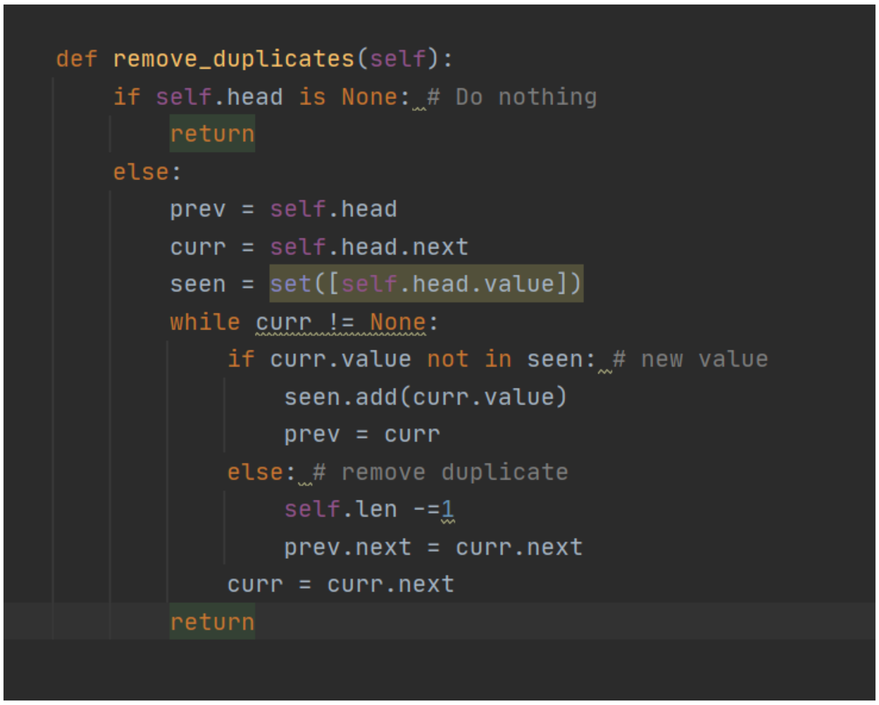


```
            self.type = type


    def __repr__(self):

        return f"Name: {self.nick_name}, Price:
{str(self.price)} NIS, Power: {str(self.__power)}"

    def _get__power(self):
        return self.__power

    def _set__power(self,new_power):
        if isinstance(new_power,int) or
isinstance(new_power, float):
            if 0 < new_power <= 100:  # check valid inpu
                self.__power = float(new_power)

    def win(self):  # method that represent the animal
win
        return self.nick_name + " winner"

    def loss(self):  # method that represent the animal
loss
        return self.nick_name + " loser"

    def __ge__(self, other):
        if not isinstance(other, Animal):  # check if
animal
            raise ValueError
        return self.__power >= other.__power

    def __eq__(self, other):
        if not isinstance(other,Animal):
                return False
        return self.nick_name == other.nick_name

    def get_type(self):  # return type of animal
        return self.type

class Mammal(Animal):
    def __init__(self, nick_name, price, power,
type=str):
        Animal.__init__(self, nick_name, price, power ,
type)

    def speak(self):
        return self.nick_name + " says"

class Shop:
    def __init__(self, name, balance):
        """
```

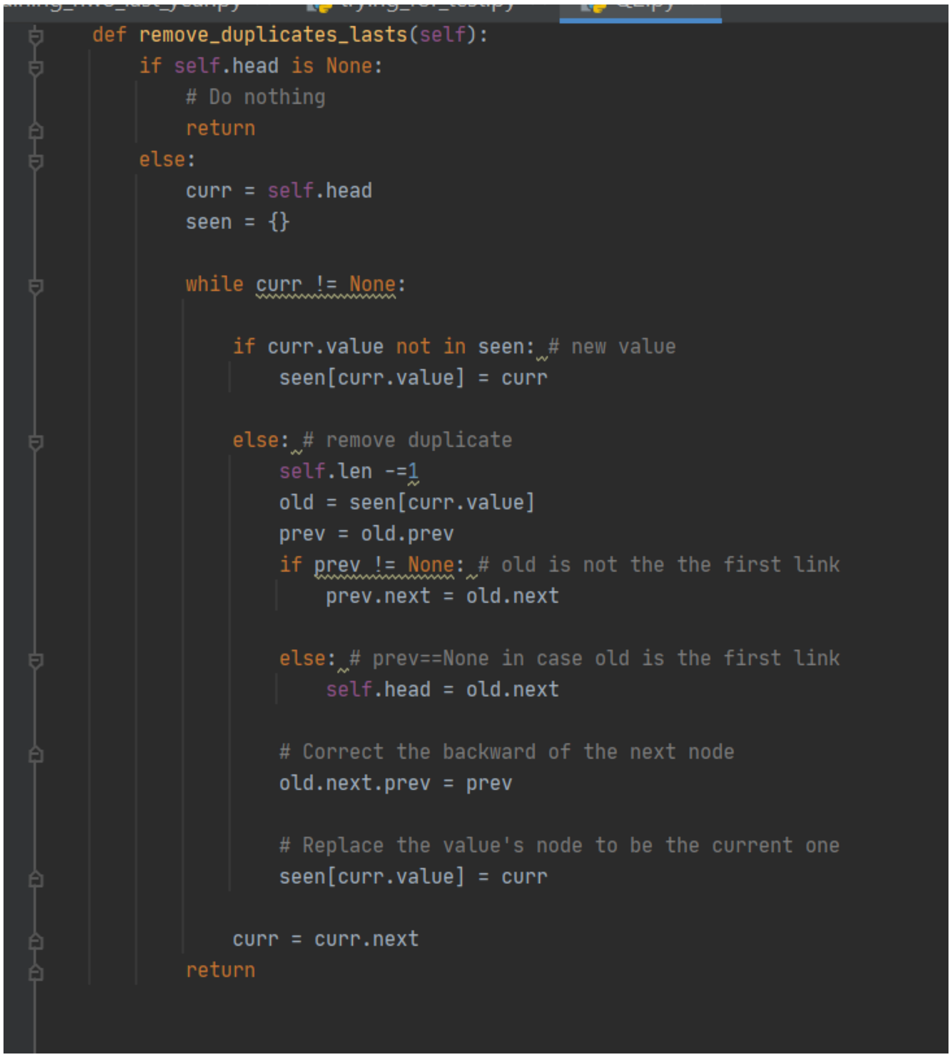


```
            consturctor of the shop
            """
        self.name = str(name)   # shop name
        self.balance = float(balance)   # budget of shop
        self.__animal_list = {}

    def get_name(self):
        return f"{self.name}"

    def get__animals(self):
        new_memory = {}   # create new dict
        for animal in self.__animal_list.values():
            key = animal.nick_name   # key is name of
animal
            # create new animal
            new_animal =
AnimalFactory.create(animal.get_type(),animal.nick_name,a
nimal.price,animal._get__power())
            new_memory[key] = new_animal   # insert to new
dict
        return new_memory

    def __add__(self, other):
        counter = 0
        if isinstance(other, Animal):   # check if other
is one animal
            if other.price <= self.balance:   # check if
there is budget for the animal
                self.__animal_list[other.nick_name] =
other
                self.balance -= other.price   # update th
balance
                counter += 1
                return counter
        elif isinstance(other,list):   # check if other i
list
            for item in other:   # remove from other list
if not animal
                if not isinstance(item,Animal):
                    other.remove(item)
            other.sort(key=lambda x: x.price)   # sort by
price
            for animal in other:   # for loop in other
list
                if animal.price <= self.balance:   # chec
if have budget in shop
                    self.__animal_list[animal.nick_name]
= animal
                    self.balance -= animal.price   #
update balance
                    counter += 1
                    return counter
            return counter  # return number of animels added

    def sell(self, nick_name):
        if nick_name in self.__animal_list:  # check if
animal in dict
            self.balance +=
self.__animal_list[nick_name].price  # add the price to
budget
            return self.__animal_list.pop(nick_name)  #
return animal removed
        else:
            return None

    def num_of_animals(self):
        return len(self.__animal_list)

    def play(self, animal_1, animal_2):
        if animal_1 not in self.__animal_list or animal_2
not in self.__animal_list:
            return False
        else:
            first_animal = self.__animal_list[animal_1]
            second_animal = self.__animal_list[animal_2]
            if first_animal >= second_animal:
                return first_animal.win() + '\n' +
second_animal.loss()
            else:
                return second_animal.win() + '\n' +
first_animal.loss()
```

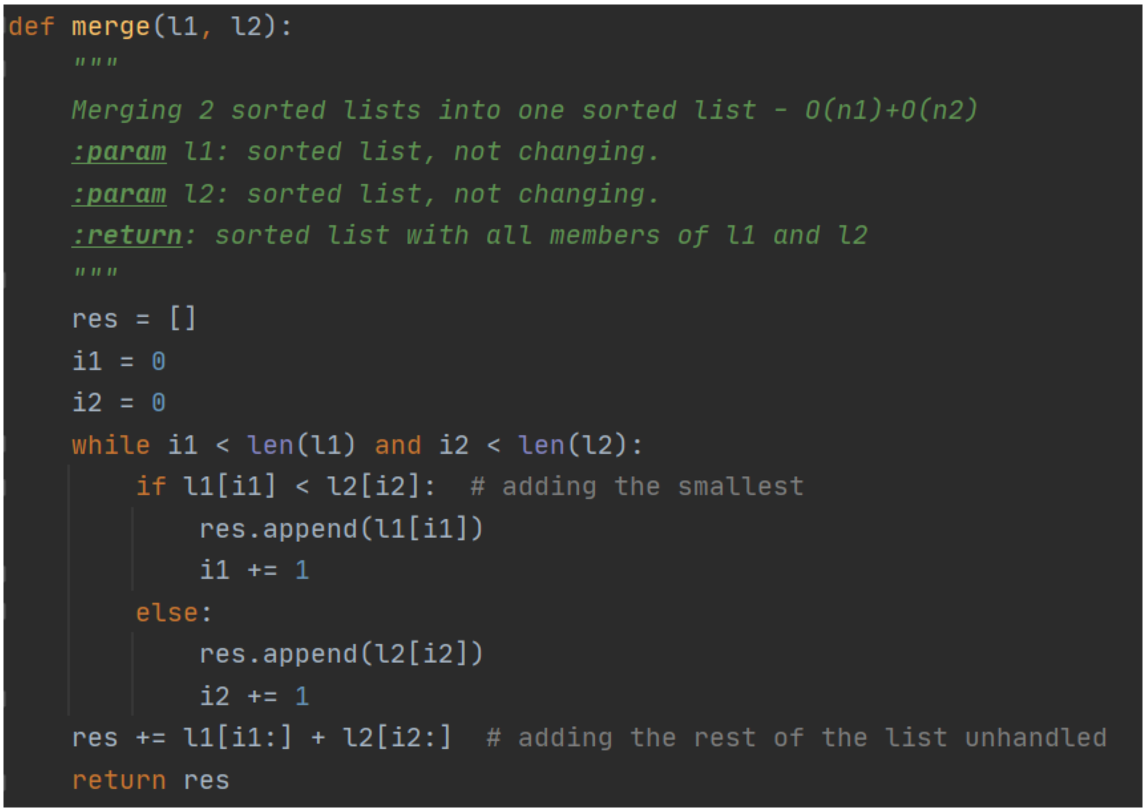
## רשימות מקושרות:



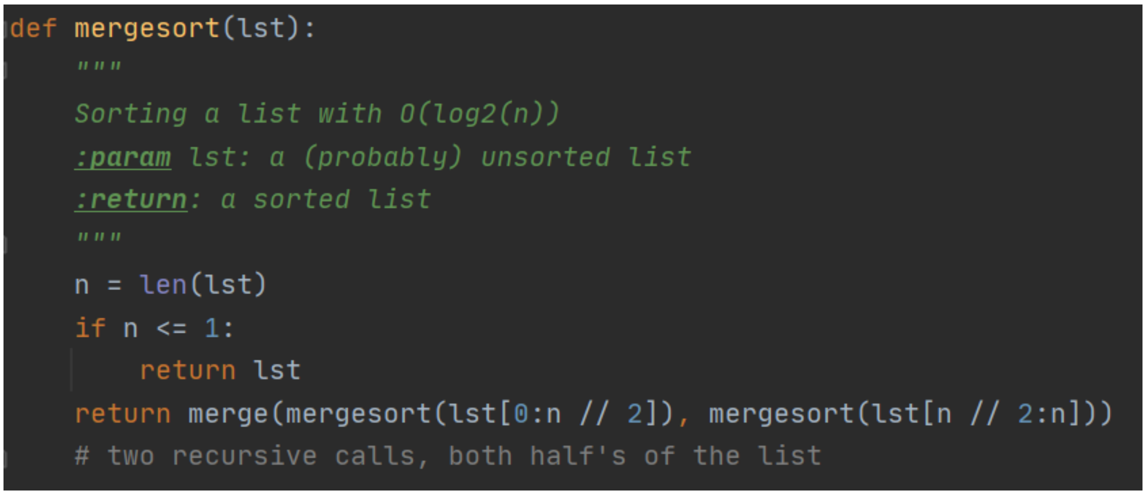
```
def remove_duplicates(self):
    if self.head is None: # Do nothing
        return
    else:
        prev = self.head
        curr = self.head.next
        seen = set([self.head.value])
        while curr != None:
            if curr.value not in seen: # new value
                seen.add(curr.value)
                prev = curr
            else: # remove duplicate
                self.len -=1
                prev.next = curr.next
            curr = curr.next
        return
```



```
def remove_duplicates_lasts(self):
    if self.head is None:
        # Do nothing
        return
    else:
        curr = self.head
        seen = {}

        while curr != None:

            if curr.value not in seen: # new value
                seen[curr.value] = curr

            else: # remove duplicate
                self.len -=1
                old = seen[curr.value]
                prev = old.prev
                if prev != None: # old is not the the first link
                    prev.next = old.next

                else: # prev==None in case old is the first link
                    self.head = old.next

                # Correct the backward of the next node
                old.next.prev = prev

                # Replace the value's node to be the current one
                seen[curr.value] = curr

            curr = curr.next
        return
```



```
def merge(l1, l2):
    """
    Merging 2 sorted lists into one sorted list - O(n1)+O(n2)
    :param l1: sorted list, not changing.
    :param l2: sorted list, not changing.
    :return: sorted list with all members of l1 and l2
    """
    res = []
    i1 = 0
    i2 = 0
    while i1 < len(l1) and i2 < len(l2):
        if l1[i1] < l2[i2]:  # adding the smallest
            res.append(l1[i1])
            i1 += 1
        else:
            res.append(l2[i2])
            i2 += 1
    res += l1[i1:] + l2[i2:]  # adding the rest of the list unhandled
    return res
```



```
def mergesort(lst):
    """
    Sorting a list with O(log2(n))
    :param lst: a (probably) unsorted list
    :return: a sorted list
    """
    n = len(lst)
    if n <= 1:
        return lst
    return merge(mergesort(lst[0:n // 2]), mergesort(lst[n // 2:n]))
    # two recursive calls, both half's of the list
```

## עצים בינארים

```
def sum_nodes(self)

    def rec_sum(tree_node):

        if tree_node is None:

            return 0
        return rec_sum(tree_node.left) +
        tree_node.val+rec_sum(tree_node.right)
    if self.root is None:
        return 0
    else:
        return rec_sum(self.root)
def is_heap(self):
    def rec_heap(tree_node):
        if tree_node is None:
            return True
        if tree_node.left is not None and tree_node.left.val
        > tree_node.val:
        return False
        if tree_node.right is not None and
        tree_node.right.val > tree_node.val:
        return False
        return rec_heap(tree_node.left) and
        rec_heap(tree_node.right)
    if self.root is None:
        return True
    else:
        return rec_heap(self.root)
```