1. You are given the following uncommented code for function `evaluate_performance`:

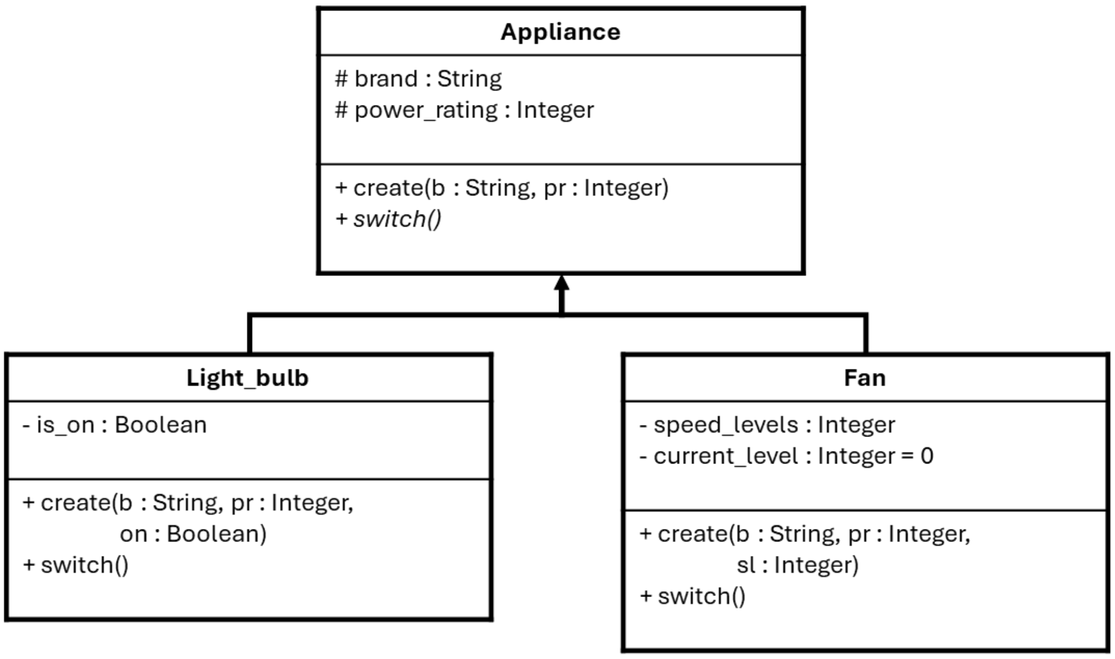
```
def evaluate_performance(scores):
    avg_score = sum(scores) / len(scores)
    if avg_score >= 75:
        print("Excellent Performance")
    elif avg_score >= 50:
        print("Satisfactory Performance")
    else:
        print("Needs Improvement")

student_name = "David"
student_grades = [80, 90, 75, 60]
```

   i) Write a new version of the code that has a class called “Student”. This class should have the scores as an instance variable and a method that has the same functionality as function `evaluate_performance`. Then write code to instantiate an object of the `Student` class.

   ii) Using a docstring, explain in your own words at the start of the class what its purpose is. It should then have clear comments in the rest of the code so that another user can understand.

2. You are given the following class diagram:


Appliance
# brand : String
# power_rating : Integer
+ create(b : String, pr : Integer)
+ *switch()*
Light_bulb
- is_on : Boolean
+ create(b : String, pr : Integer, on : Boolean)
+ switch()
Fan
- speed_levels : Integer
- current_level : Integer = 0
+ create(b : String, pr : Integer, sl : Integer)
+ switch()

i) Write the class definitions that correspond to the elements in the class diagram.

ii) Modify a method called switch for the Light_bulb class, so that it toggles the state of the lightbulb between on and off. Modify the switch method for the Fan class so that the current level is incremented by one or reset to 0 if at maximum level.

iii) Add a dunder method in the Light_bulb class that converts the object of type Light_bulb into a Boolean. It should return True if the bulb is currently turned on and False otherwise.

iv) Add comments in your code for parts i), ii) and iii). The comments should clearly explain the function of the code written so that another user can understand and follow the code.

3. Consider the following list:

```
[20, 14, 35, 9, 42, 18]
```

i) Sort the list above using heapsort algorithm. Show the state of the list contents after each loop iteration. Describe which elements are swapped when heapifying the list in the first stage of the algorithm. in the second stage of the algorithm, state which element is being extracted, describe which elements are swapped and which elements form the sorted part of the list.

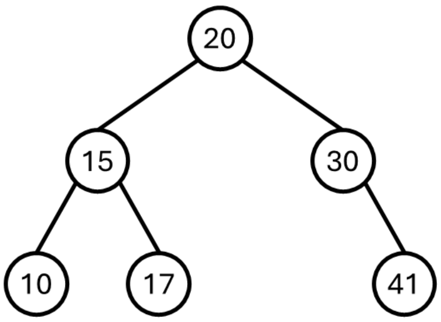
ii) Write a class called `Heap_numbers` that takes a list of integers as an argument and stores them in a heap. Validate that all inputs are integers; if not, raise a type error. Instantiate the class with the list above and apply heapsort to sort the numbers.

Note: use the function `is_int(value)` defined in Appendix D to check if a value is an integer

iii) Add a method to `Heap_numbers` that removes all numbers greater than a given integer x from the heap. If x is not an integer, raise a type error.

iv) Add comments in your code for parts ii) and iii). The comments should clearly explain the function of the code written so that another user can understand and follow the code.

4. Consider the following tree diagram and the binary tree node class in Appendix B:



i) Write code to create this tree object using the `Node` class.

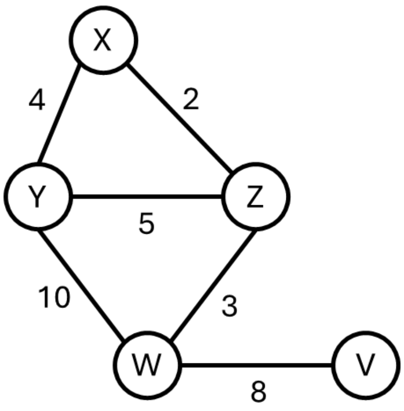
ii) Write a comment stating whether this tree is an example of a binary search tree or not. Justify your answer.

iii) Modify the `insert` method of the Search_tree class to check whether the value already exists in the search tree. If the value exists, raise a value error with the message “duplicate values are not allowed”.

iv) Add a recursive method to the node class that performs a in-order depth-first traversal of the tree and prints True if the value of that node is even, and False otherwise.

v) Add comments in your code for parts i), iii) and iv). The comments should clearly explain the function of the code written so that another user can understand and follow the code.

5. Consider the following graph diagram and the `Graph` class in Appendix E:



i) Write code to create this graph object using the `Weight_graph` class.

ii) Add a method to the graph class which takes a vertex as an argument and outputs to the console the average (mean) of the weights of all edges attached to that vertex.

iii) Add a dunder method to the `Weight_graph` class that modifies the behavior of the power operator (**). This method should raise the weights of all edges in the graph to the given power

iv) Add comments in your code for parts i), ii) and iii). The comments should clearly explain the function of the code written so that another user can understand and follow the code.

Appendix A

```python
class Heap:
    def __init__(self, data = []):
        self.data = data

    def is_empty(self):
        return len(self.data) == 0

    def parent(self, index):
        return (index - 1) // 2

    def left(self, index):
        return index * 2 + 1

    def right(self, index):
        return index * 2 + 2

    def insert(self, value):
        pos = len(self.data)
        self.data.append(0)
        while pos > 0 and self.data[self.parent(pos)] < value:
            self.data[pos] = self.data[self.parent(pos)]
            pos = self.parent(pos)
        self.data[pos] = value
```

```python
    def extract(self):
        max_value = self.data[0]
        self.data[0] = self.data[len(self.data)-1]
        self.data.pop()
        self.move_down(0, len(self.data)-1)
        return max_value

    def move_down(self, first, last):
        while self.left(first) <= last:
            larger = self.left(first)
            if (self.right(first) <= last and
                self.data[self.left(first)] < self.data[self.right(first)]):
                larger = self.right(first)
            if self.data[first] >= self.data[larger]:
                break
            temp = self.data[first]
            self.data[first] = self.data[larger]
            self.data[larger] = temp
            first = larger
```

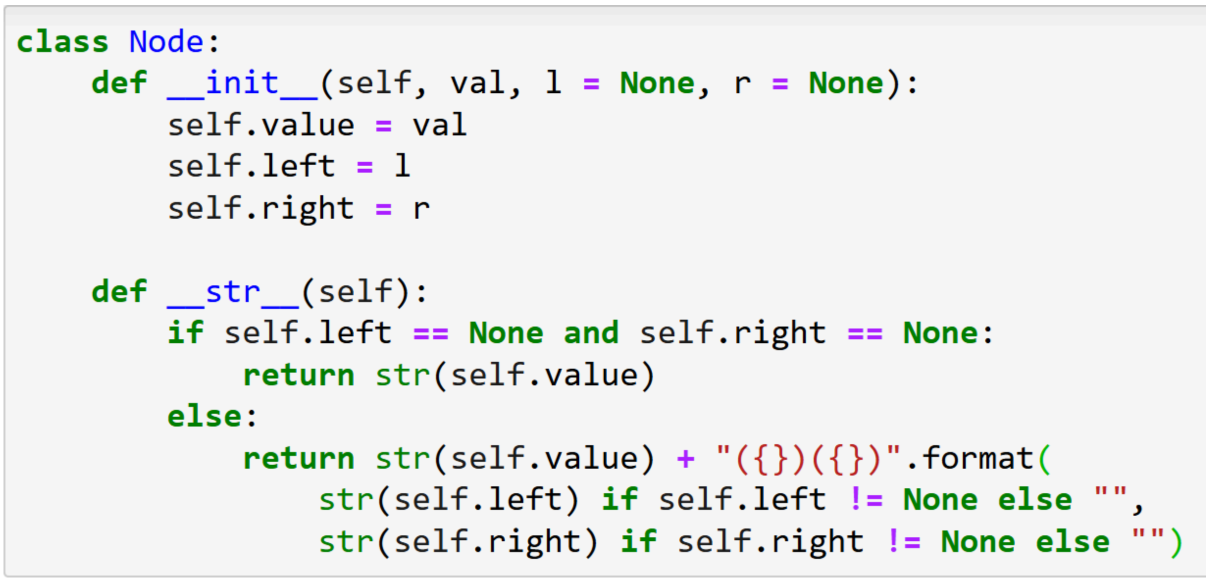
```python
def heap_sort_desc(data):
    heap = Heap(data)
    for i in range(len(heap.data)//2 - 1, -1, -1):
        heap.move_down(i, len(heap.data)-1)

    for i in range(len(heap.data) - 1, 0, -1):
        temp = heap.data[0]
        heap.data[0] = heap.data[i]
        heap.data[i] = temp
        heap.move_down(0, i-1)

    heap.data.reverse()
    return heap
```

Appendix B

Binary Tree Node class:



```python
class Node:
    def __init__(self, val, l = None, r = None):
        self.value = val
        self.left = l
        self.right = r

    def __str__(self):
        if self.left == None and self.right == None:
            return str(self.value)
        else:
            return str(self.value) + "({})({})".format(
                str(self.left) if self.left != None else "",
                str(self.right) if self.right != None else "")
```

Appendix C

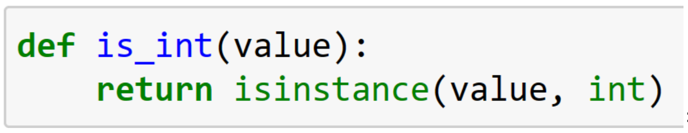
```python
from Node import Node

class Search_tree:
    def __init__(self):
        self.root = None

    def insert(self, val, node = "root"):
        if node == "root":
            self.root = self.insert(val, self.root)
        # If node is a empty, create a new node
        elif node == None:
            return Node(val)
        # Otherwise, recur through the tree
        else:
            # Insert into left subtree
            if val < node.value:
                node.left = self.insert(val, node.left)
            # Insert into right subtree
            elif val > node.value:
                node.right = self.insert(val, node.right)
            return node
```

Appendix D

The function to check if a value is an integer



```
def is_int(value):
    return isinstance(value, int)
```

#

Appendix E

Weighted graph class:

```python
from collections import defaultdict
```

```python
class Weight_graph:
    def __init__(self):
        self.vertices = set()
        self.edges = defaultdict(list)
        self.weights = {}

    def add_vertex(self, v):
        self.vertices.add(v)

    def add_edge(self, v, u, dist):
        self.edges[v].append(u)
        self.weights[(v, u)] = dist
```