**Kathmandu University**

**Department of Computer Science and Engineering**

**Dhulikhel, Kavre**



**A lab Report 5**

**On**

**“Algorithm and Complexity”**

**[Course Code: COMP 314]**

**Submitted By:**

Sabil Shrestha (51)

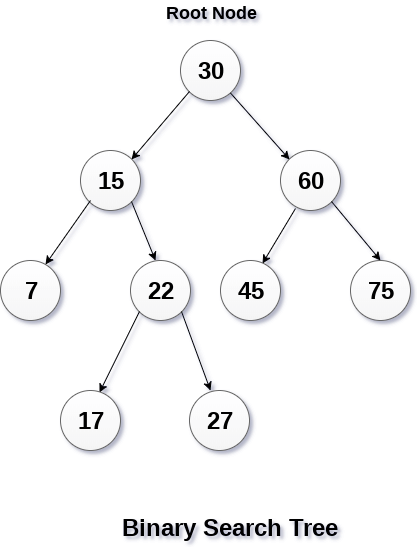
**Submitted To:**

**Rajani Chulyadyo Mam**

**Submission Date: 29th November, 2020**

# **Binary Search Tree**

Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree. In a Binary search tree, the value of all the nodes in the left sub-tree is less than the value of root. value of all the nodes in the right sub-tree is greater than or equal to the value of the root. The information represented by each node is a record rather than a single data element. One of the advantages of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient. The binary search tree is efficient data structure if compared with arrays and linked lists. BST removes half sub-tree at every step. So, it is very fast and efficient. Searching for an element in a binary search tree takes o(log2n) time. In worst case, the time it takes to search an element is 0(n) which is in linear time and efficient.



While inserting value in BST the left part should always be less than the parent and the right part should always be greater than the parent node. If this rule isn’t followed then it is not a Binary Search Tree. After deletion of value from a node. If the node is leaf node or a node with no children then no other operation should be performed to maintain the Binary Search Tree. But if the node isn’t a leaf node or node with children then either the largest element from left most subtree should be selected as new value of the node or the smallest element from right most node is selected. The insertion of values in BST, searching of BST, removal of value from BST and making of BST after the removal of value are some operations.

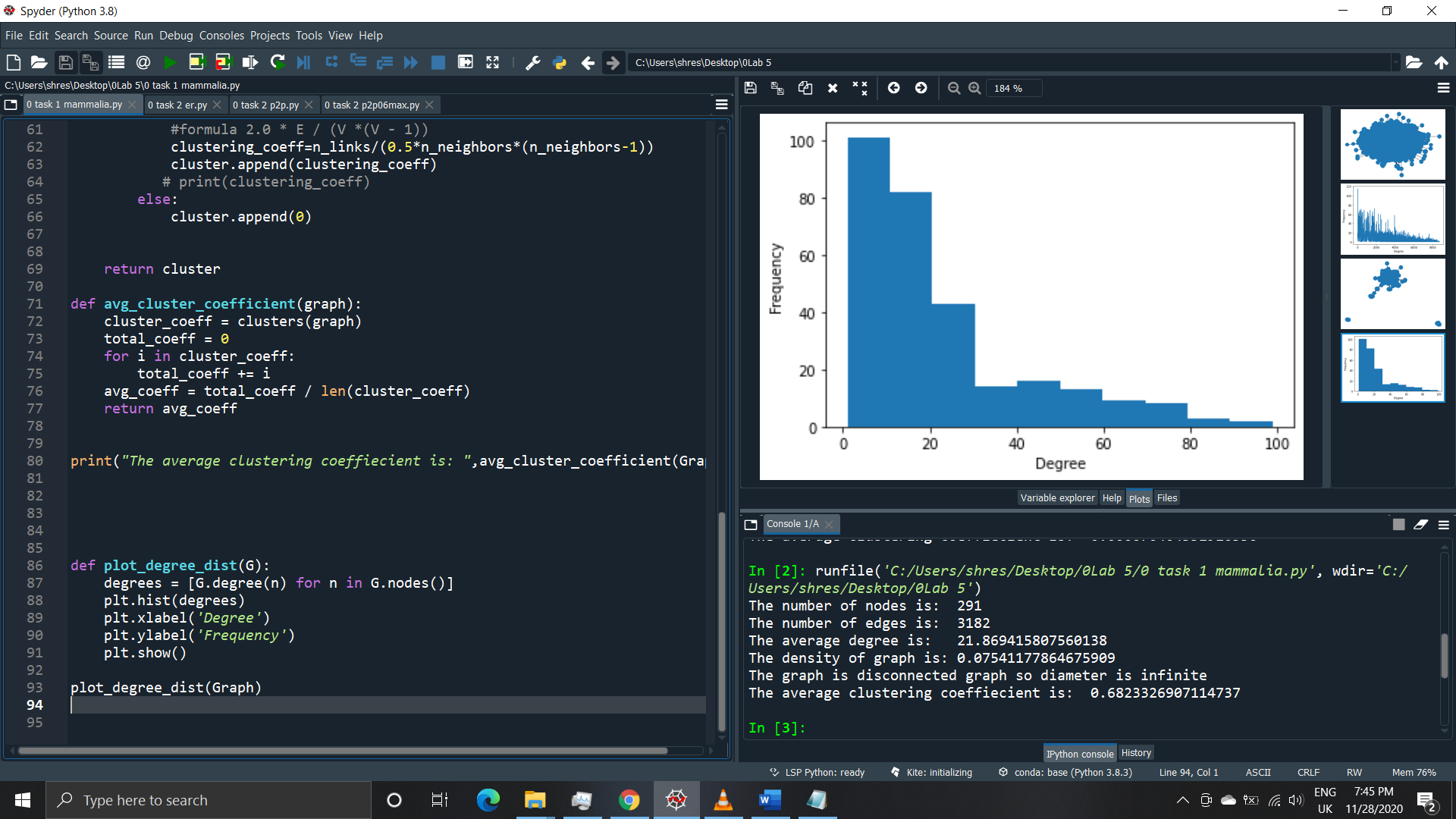
**Source Code**

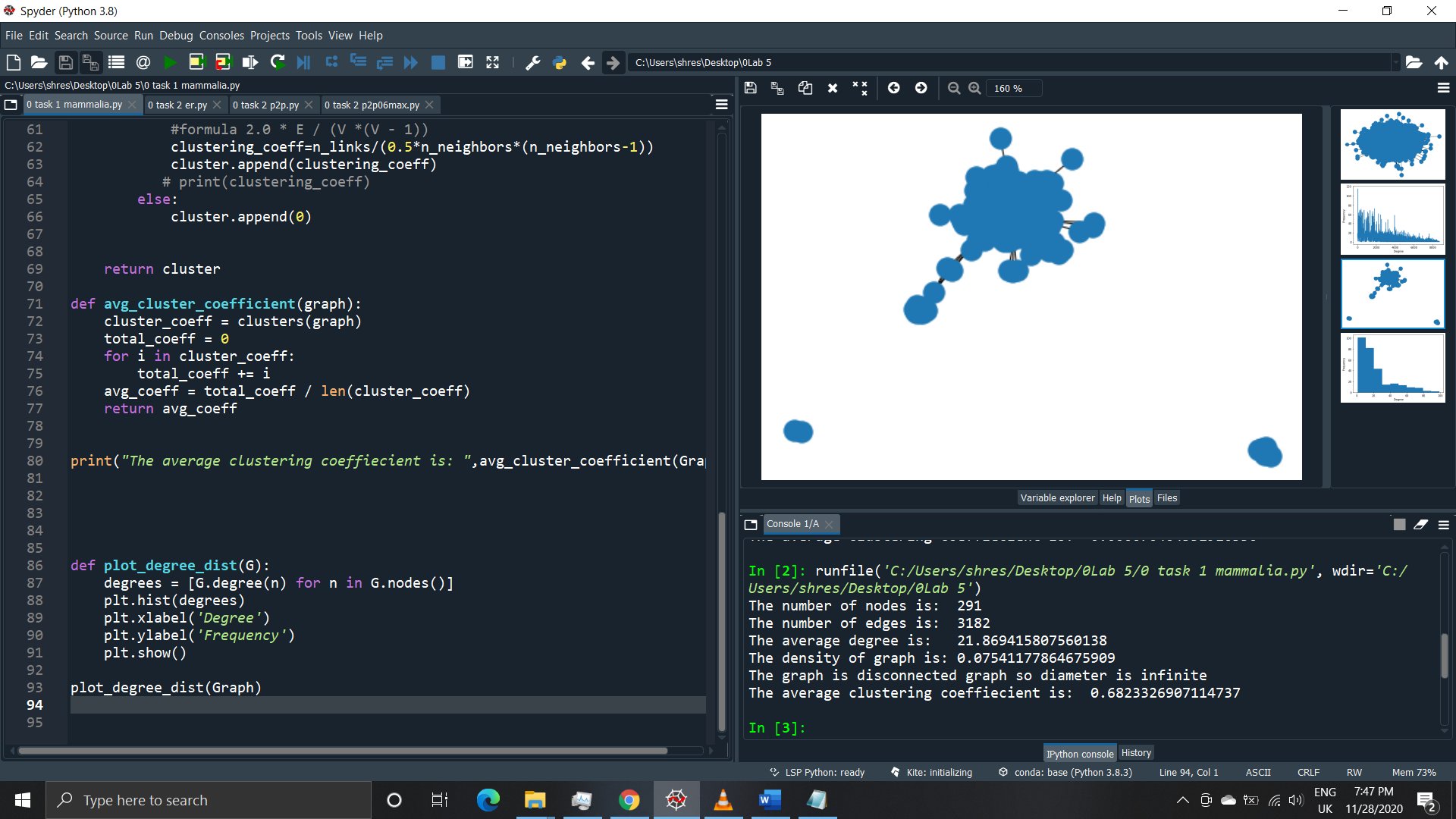
**Task 1**

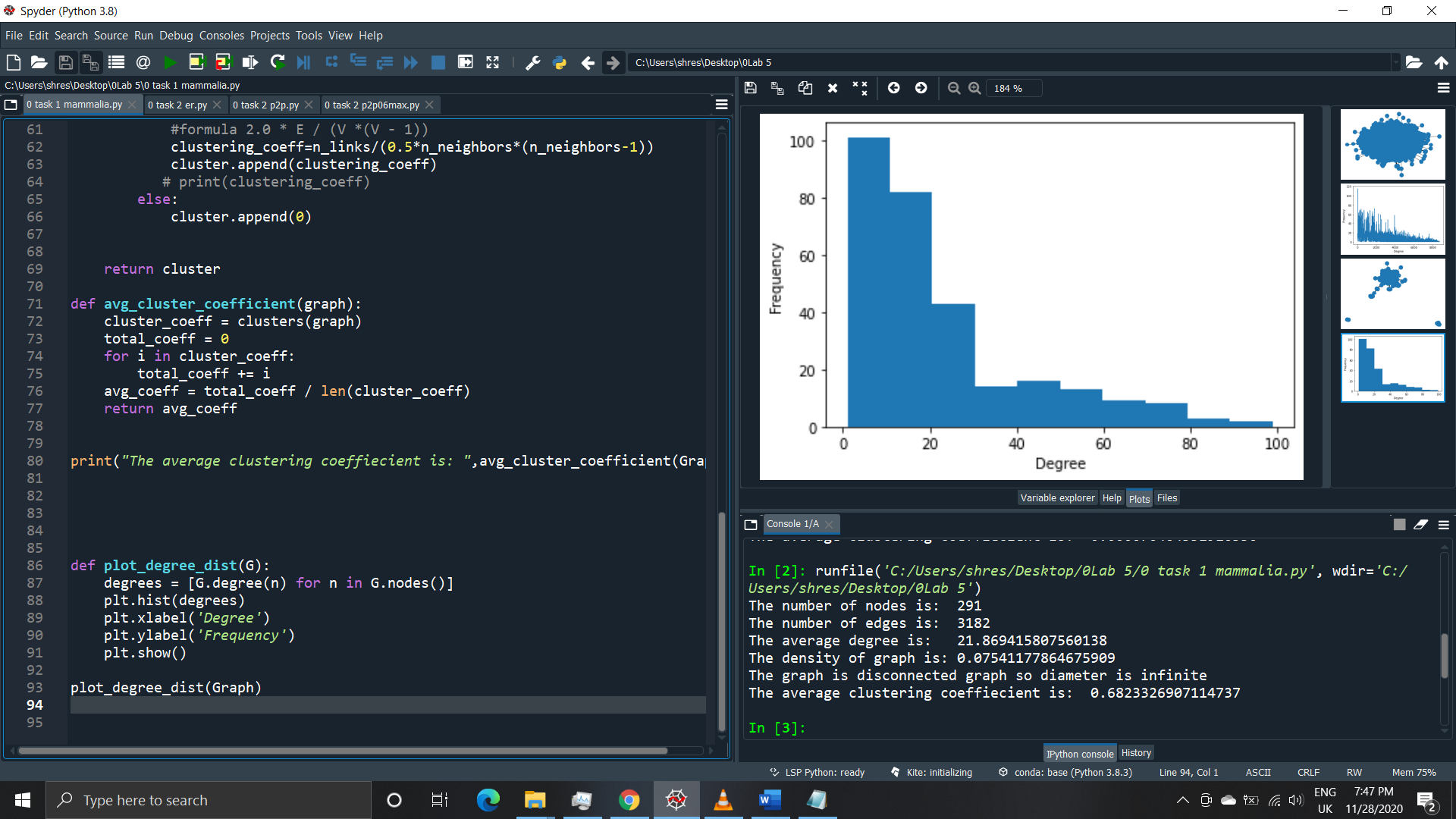
**set mammalia**

**(0 task 1 mammalia.py)**

**Output**





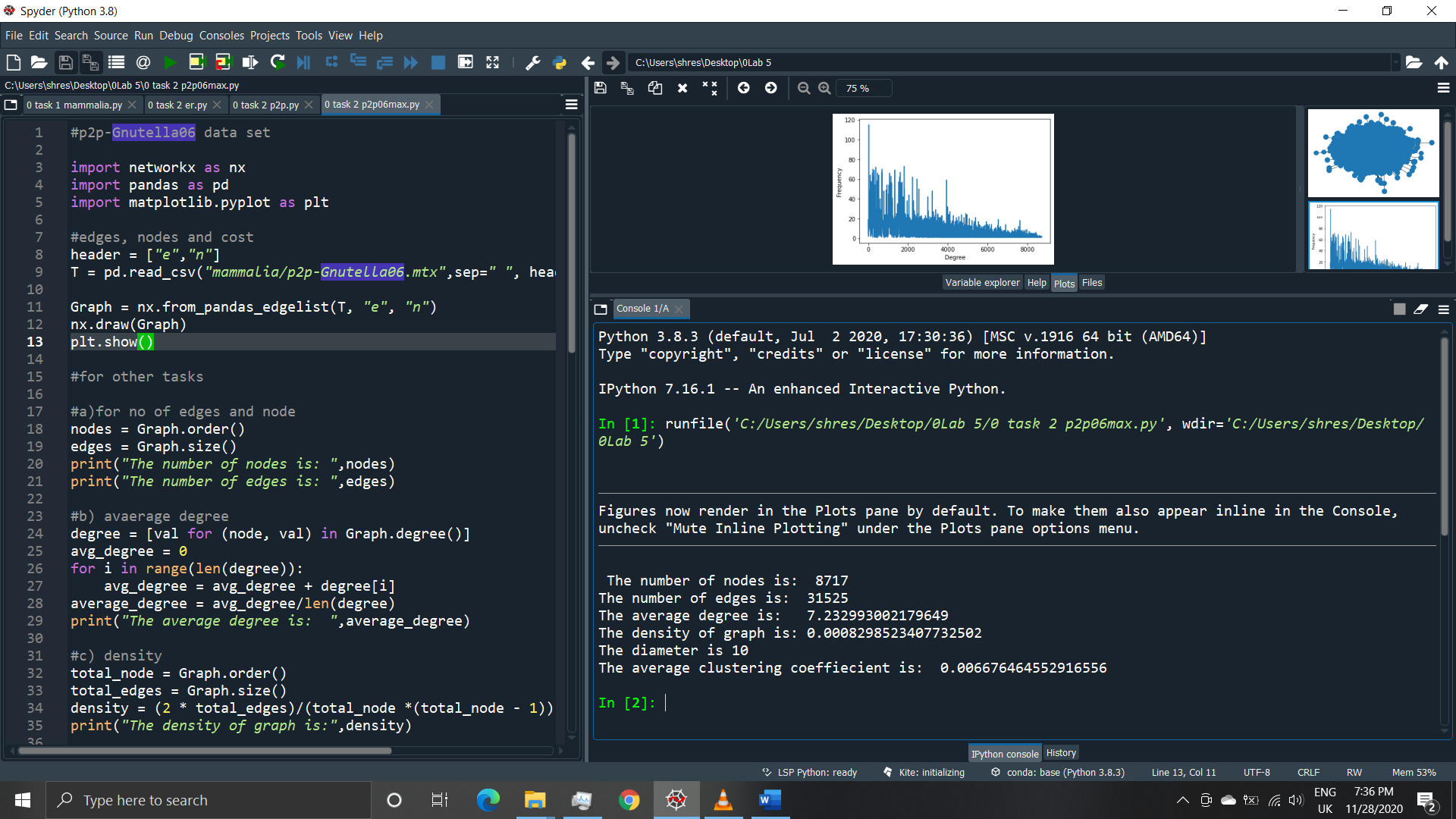


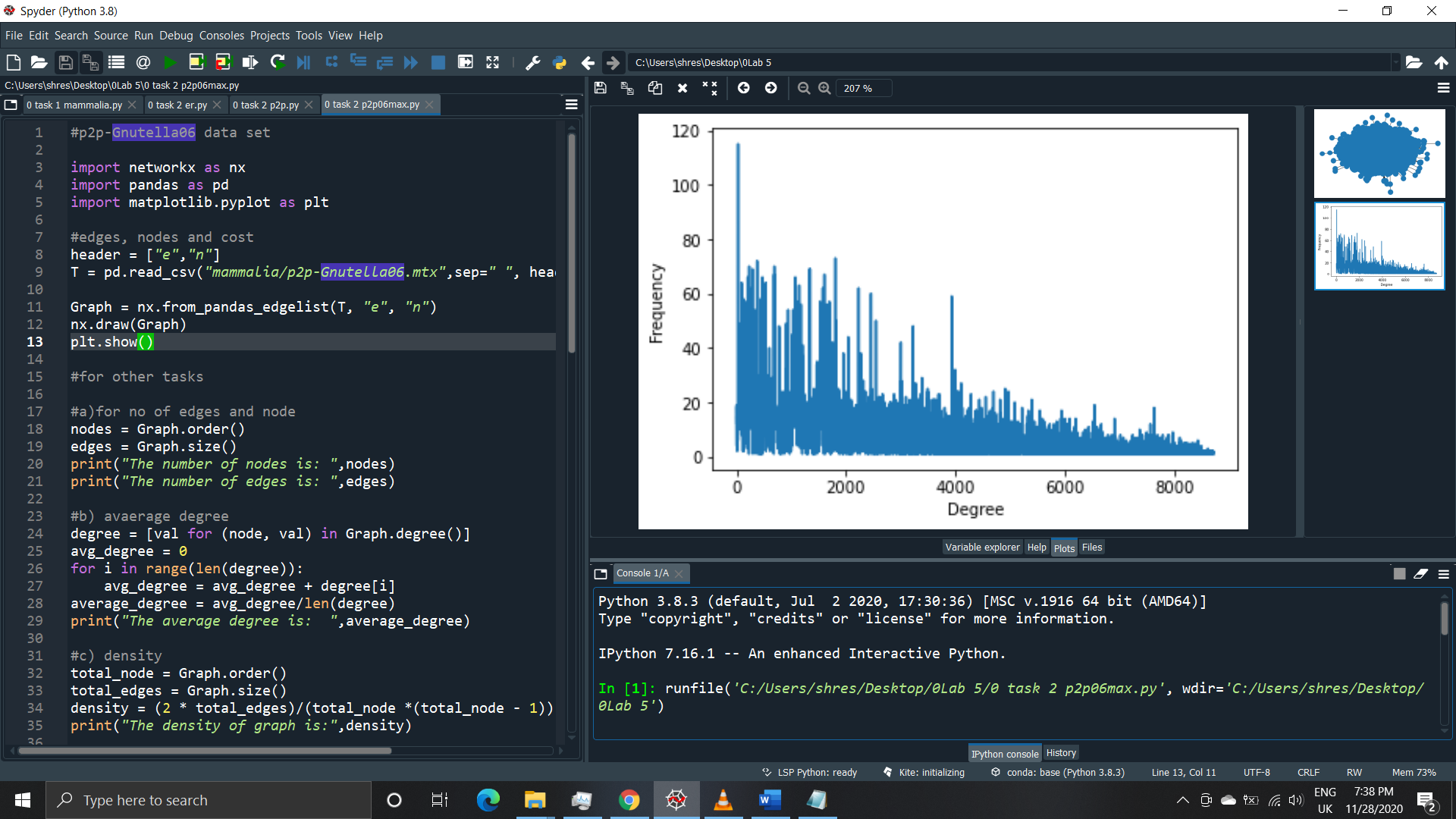
**Task 2**

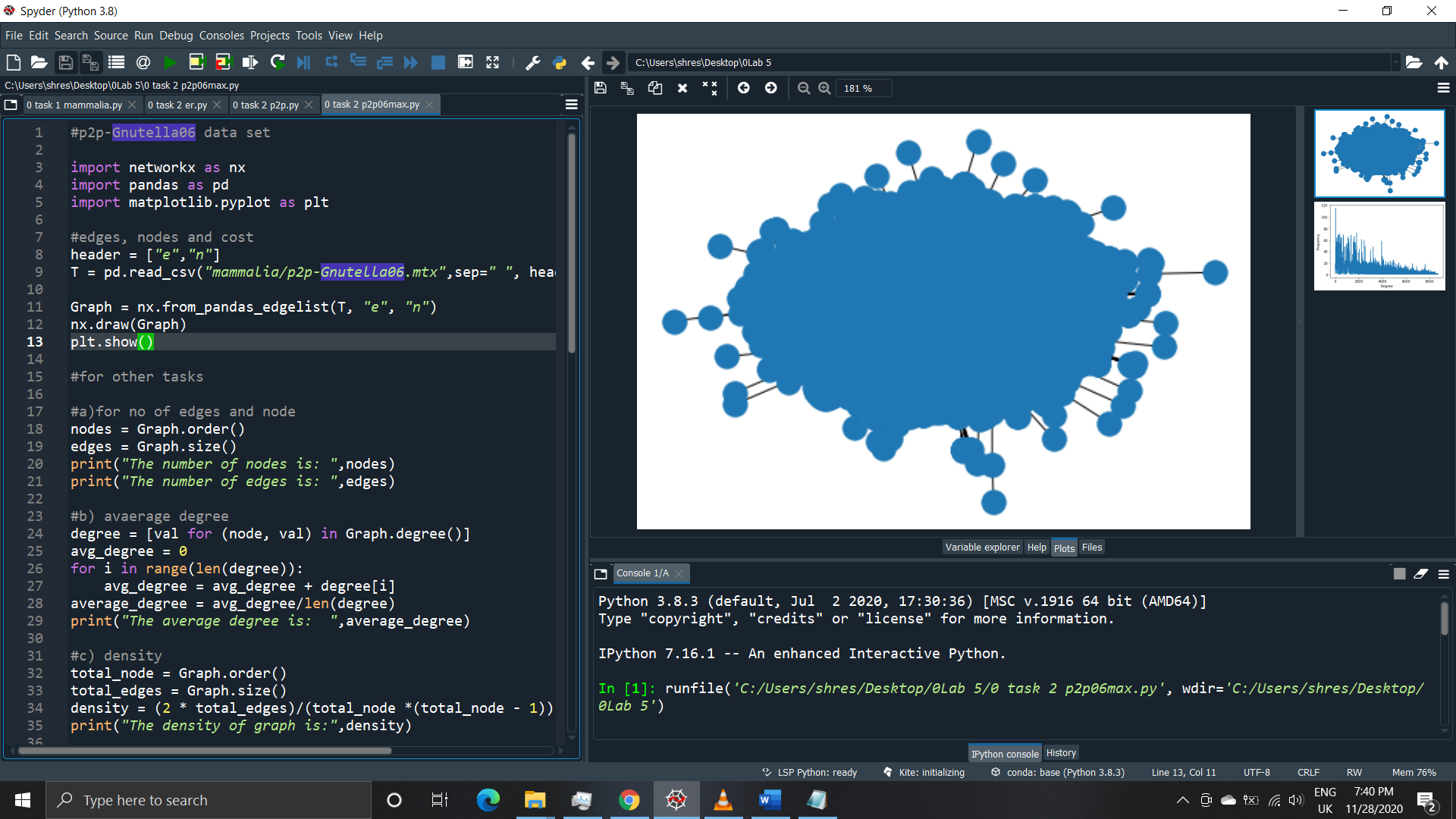
**First set p2p-nutella06**

**(01 task 2 p2p06max.py)**

**Output**





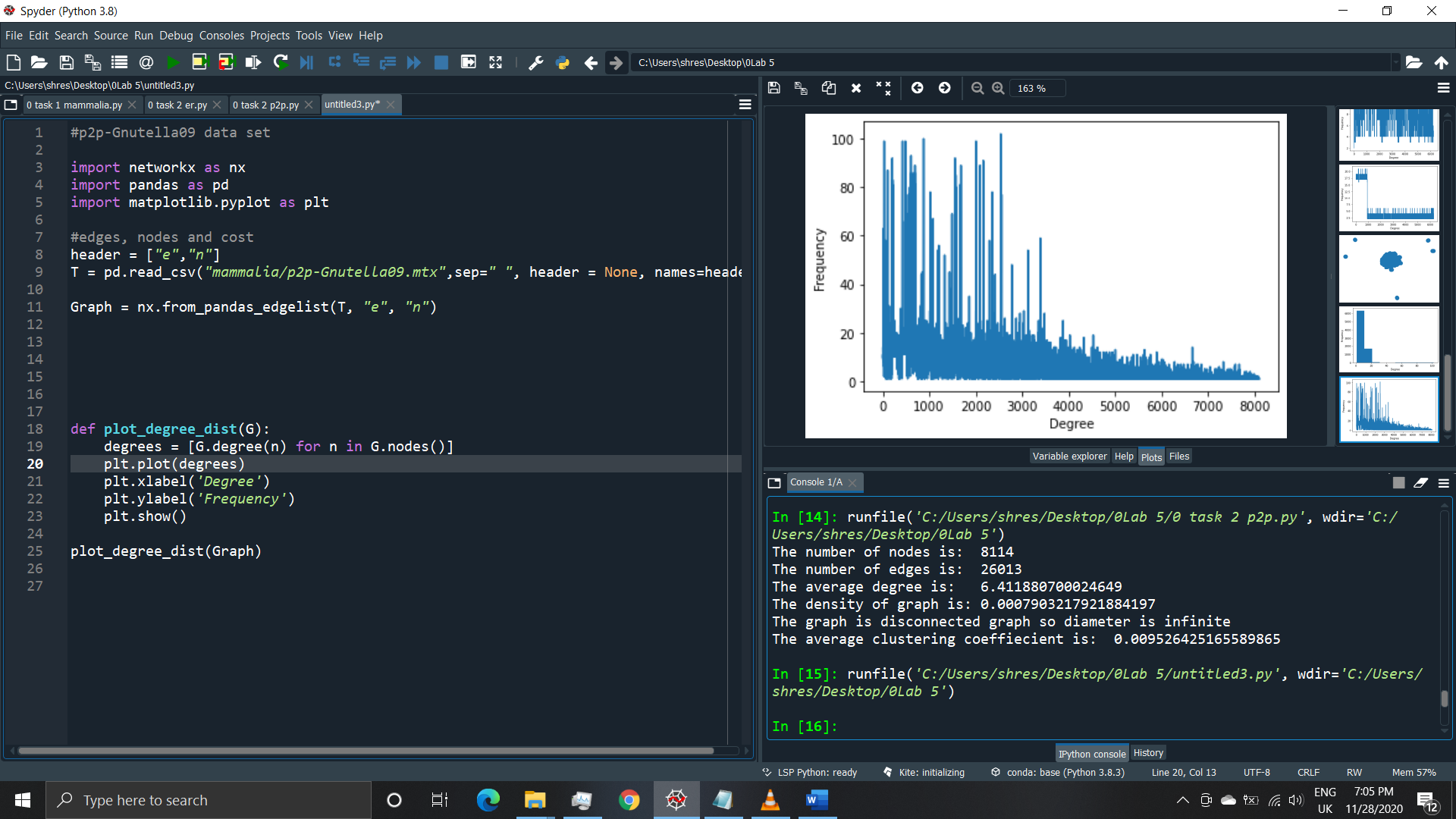


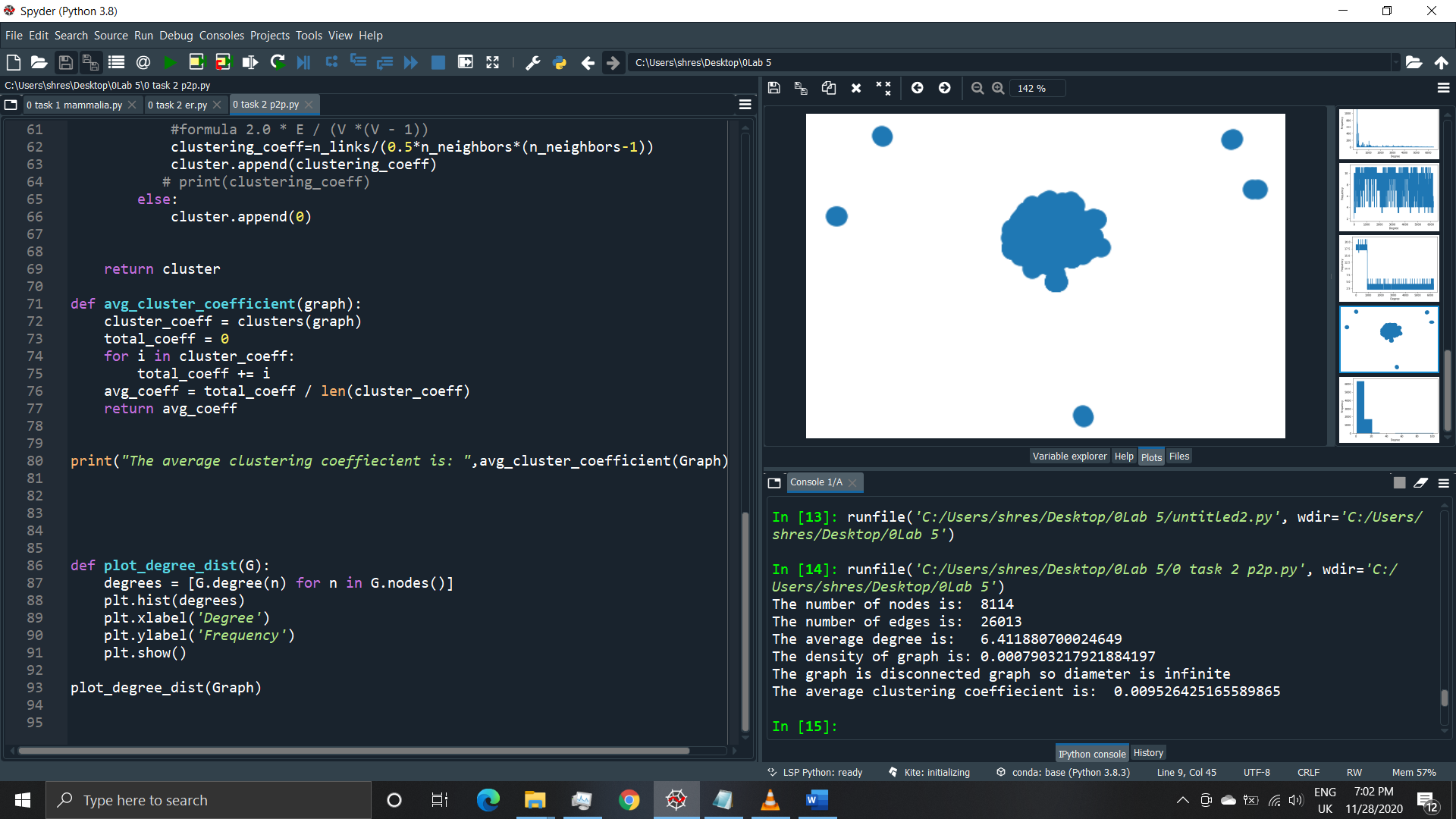
**Second set p2p-nutella09-this**

**(022 task 2 p2p.py)**

**Output**



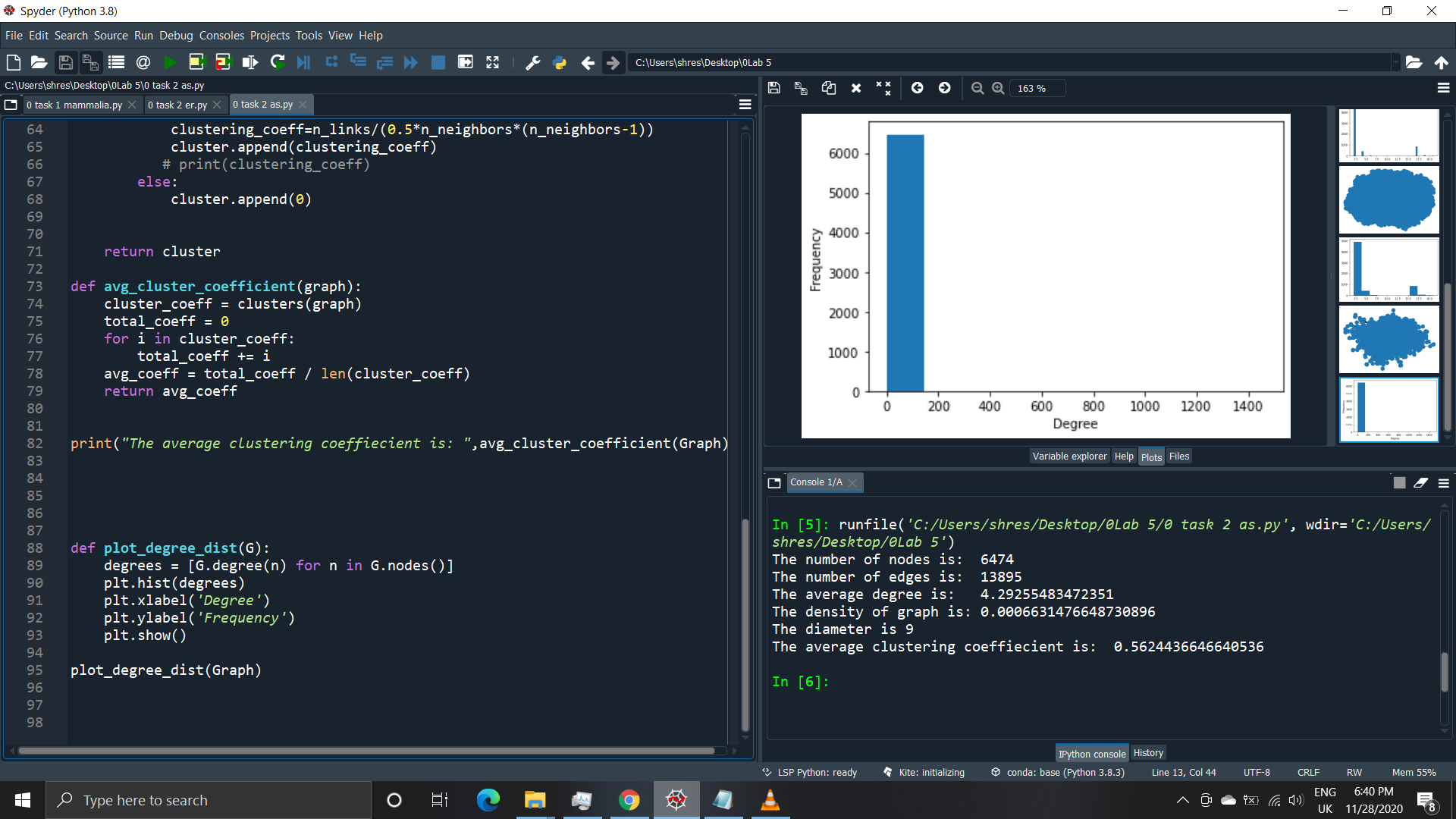


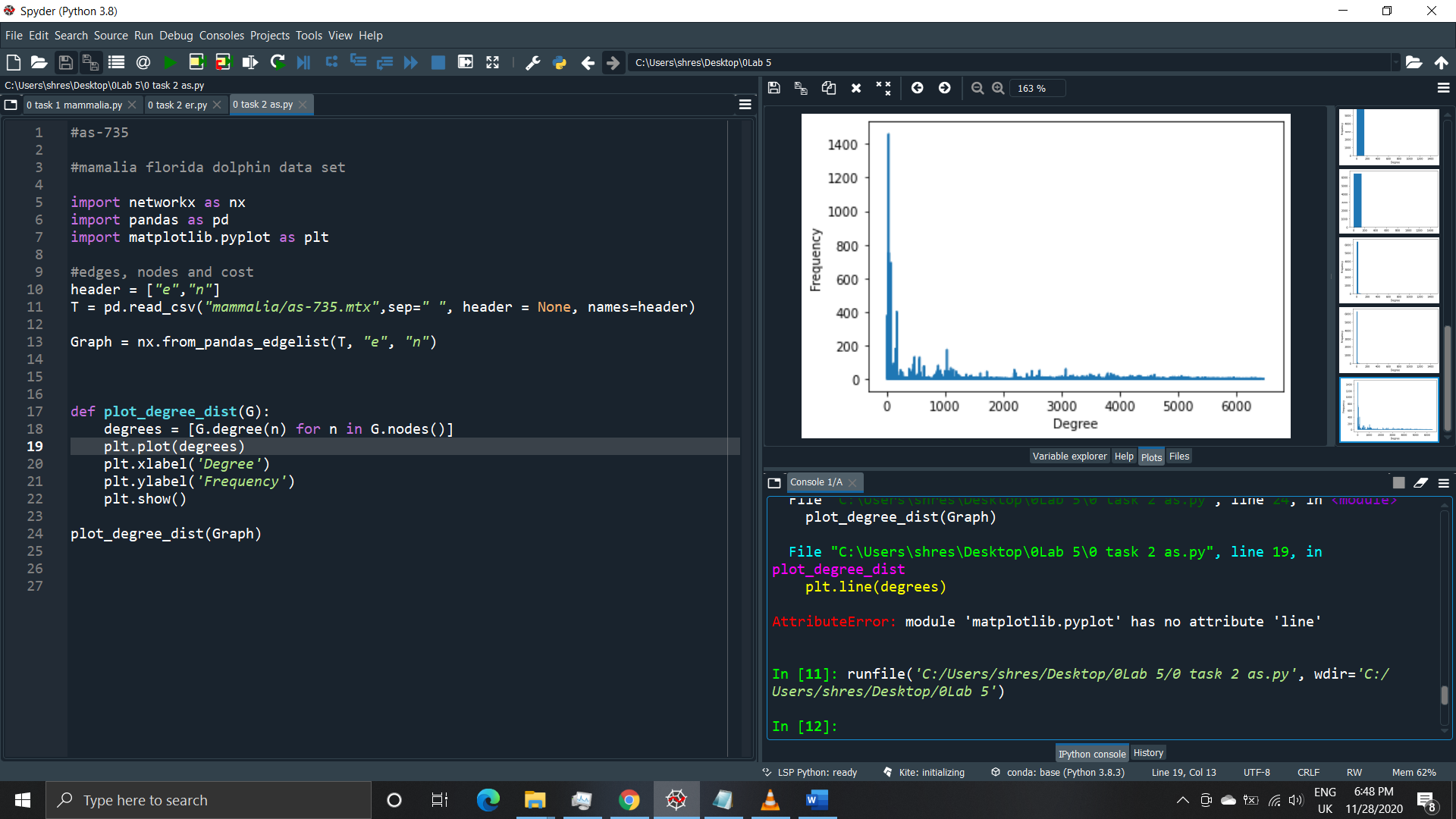


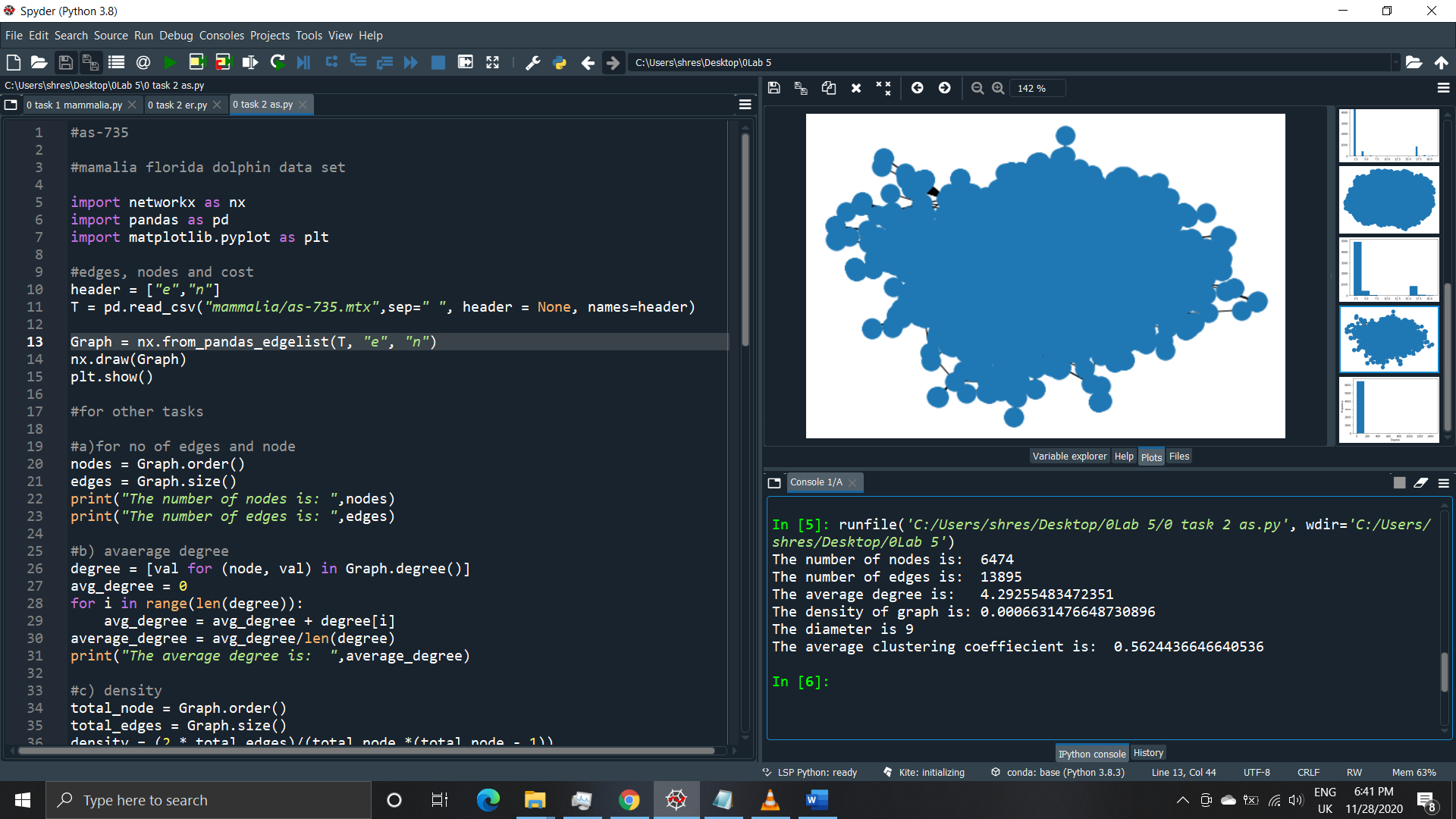
**Third set as-735**

**(03 task 2 as.py)**

**Output**



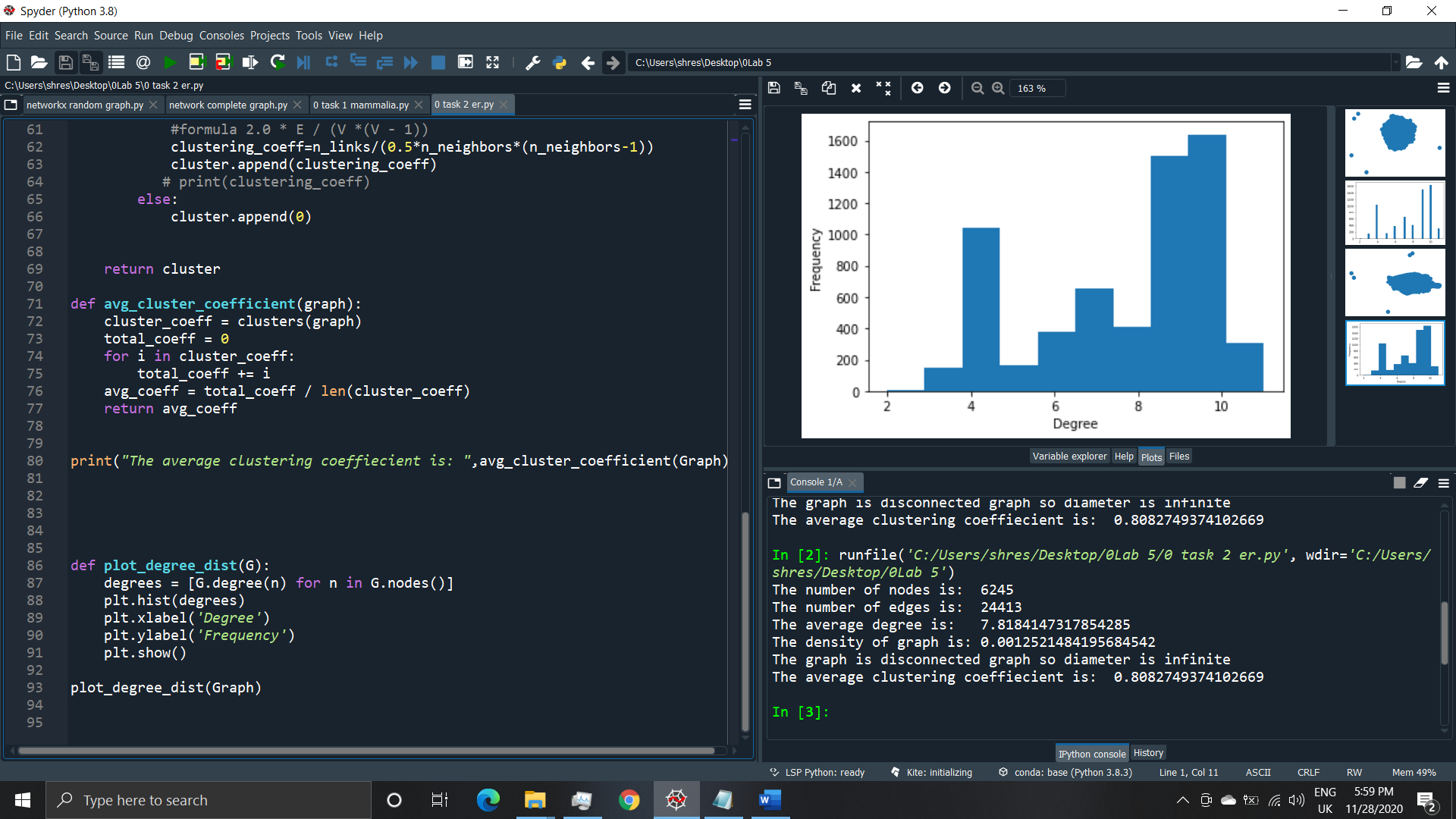


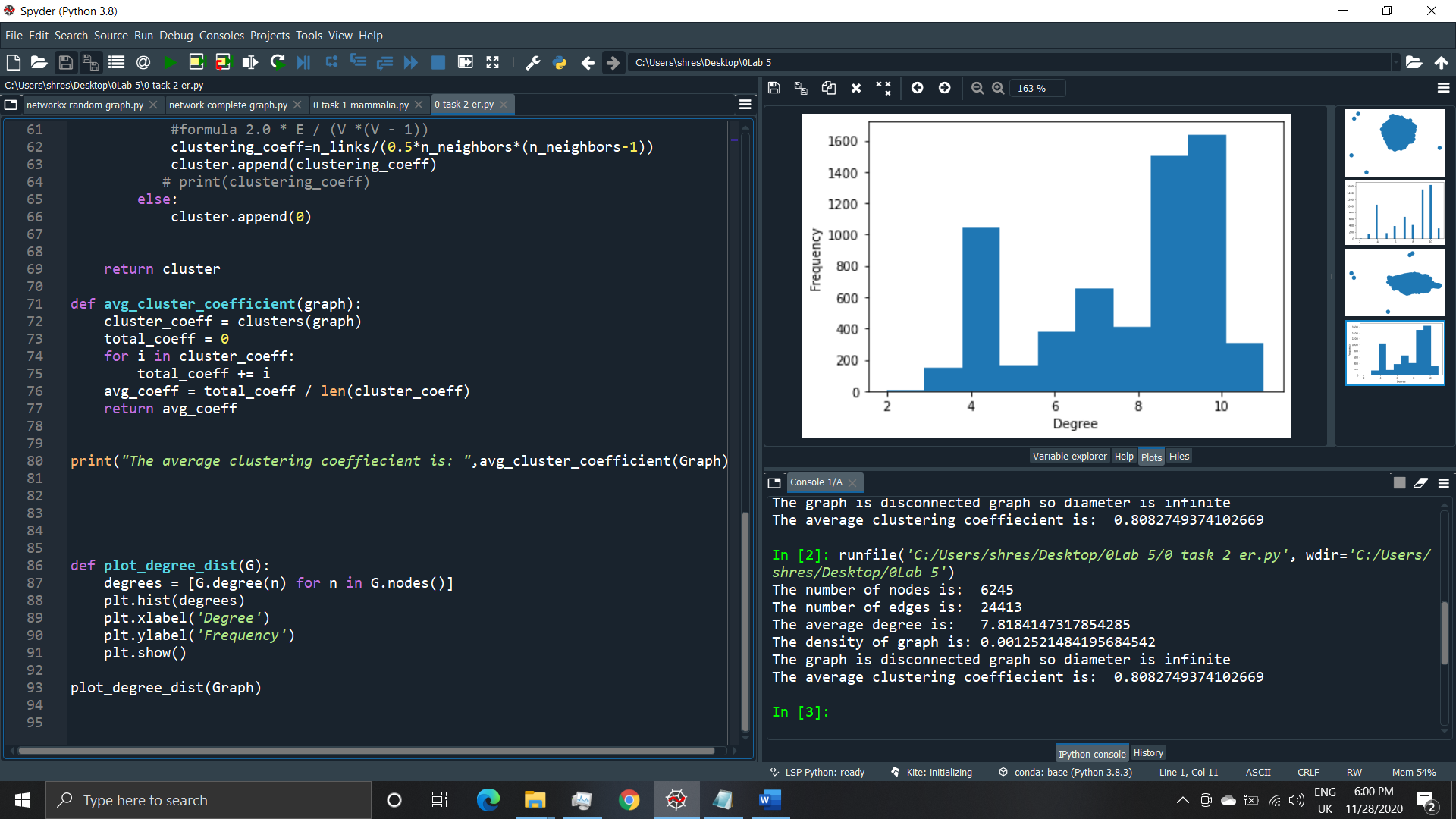


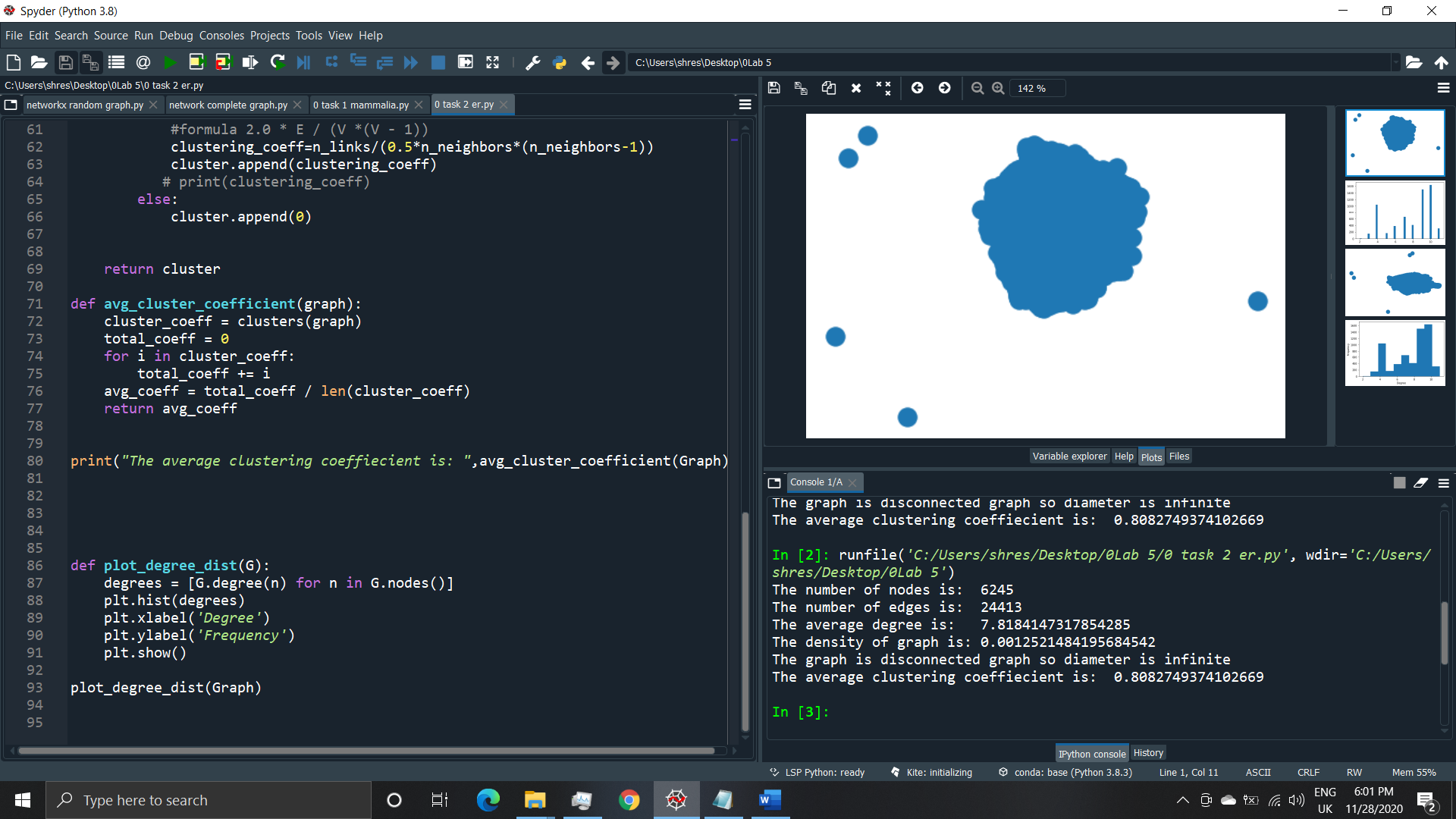
**Fourth set Alamender**

**(04 task 2 alamender.py)**

**Output**



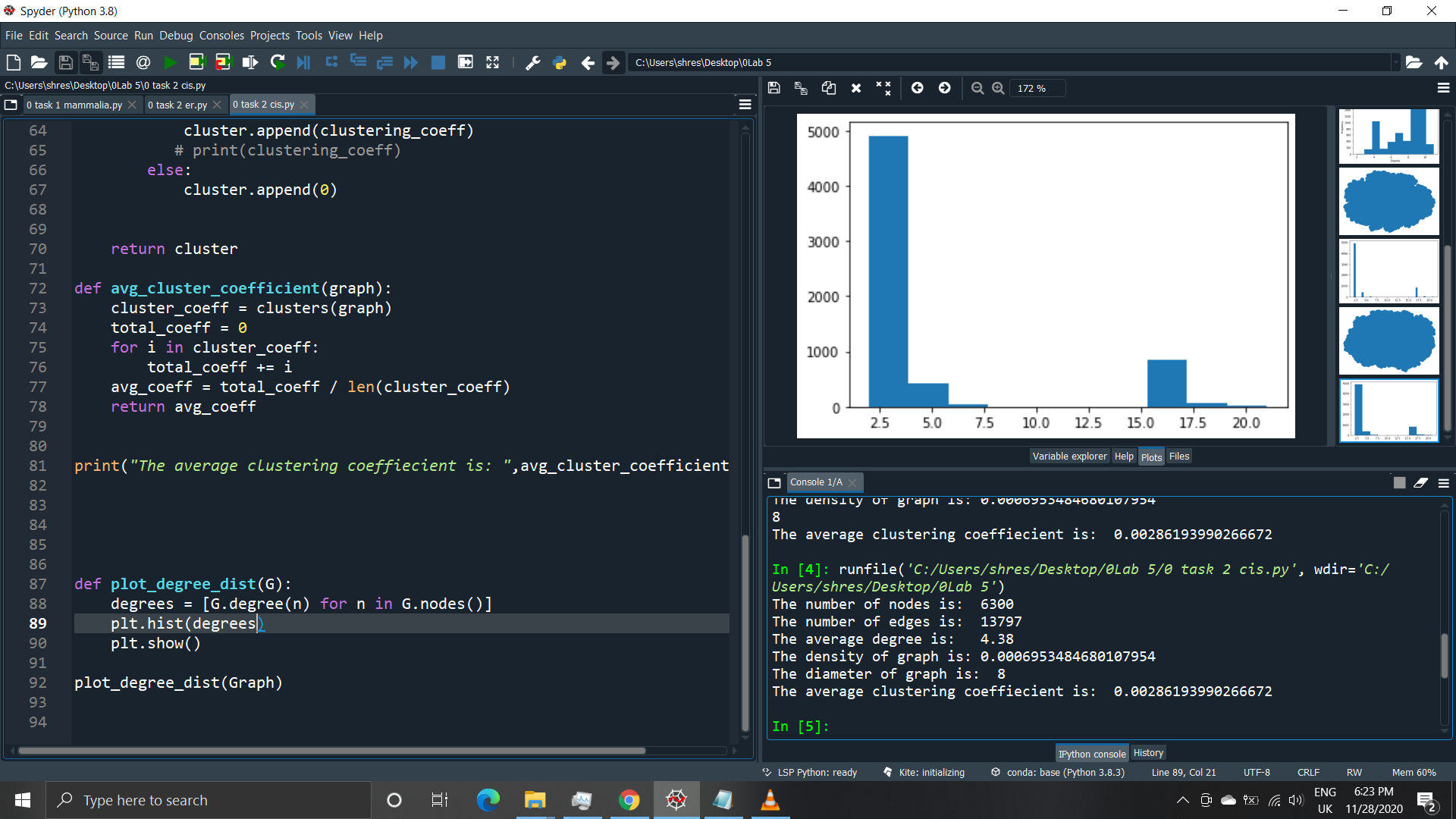


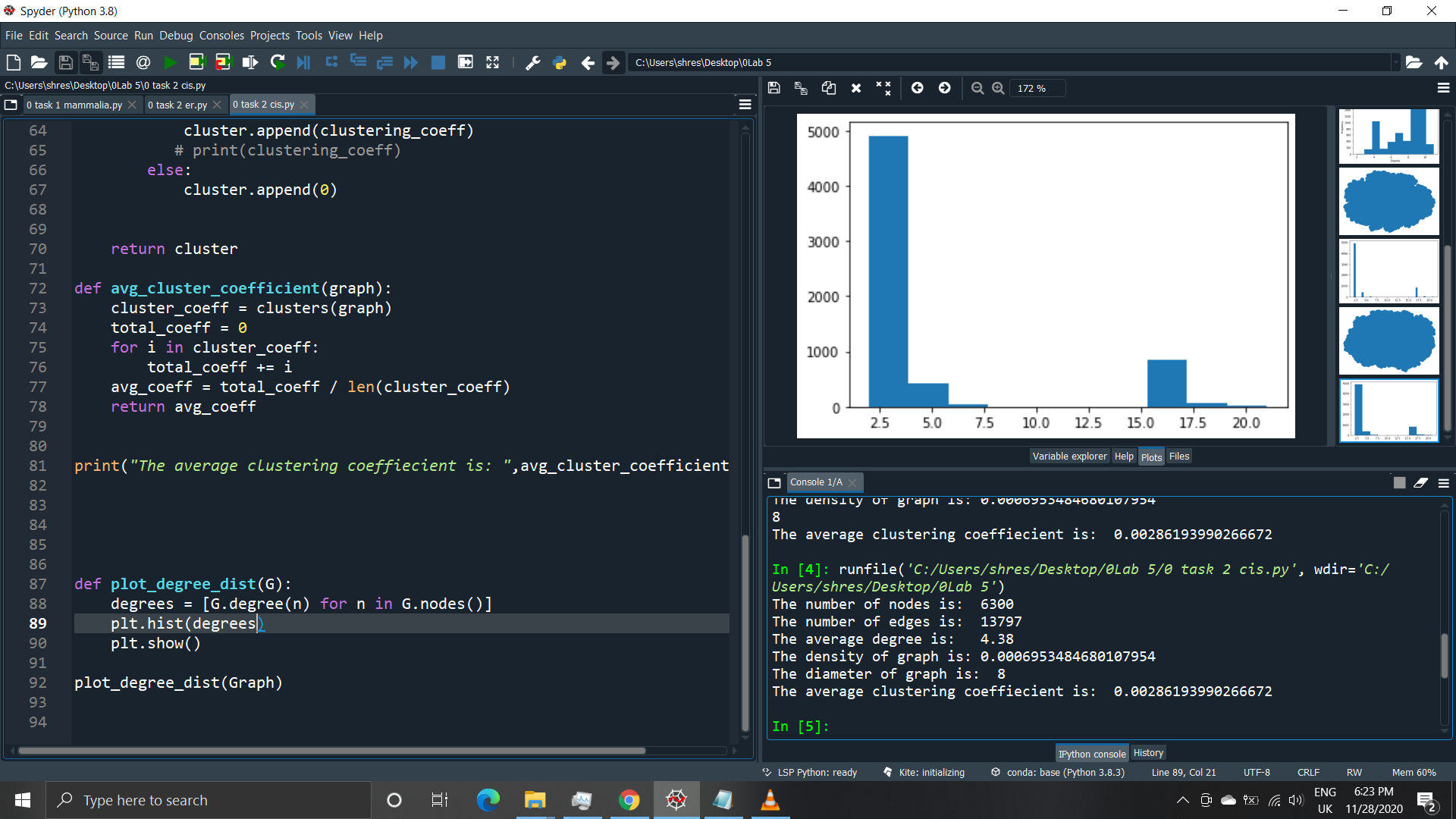


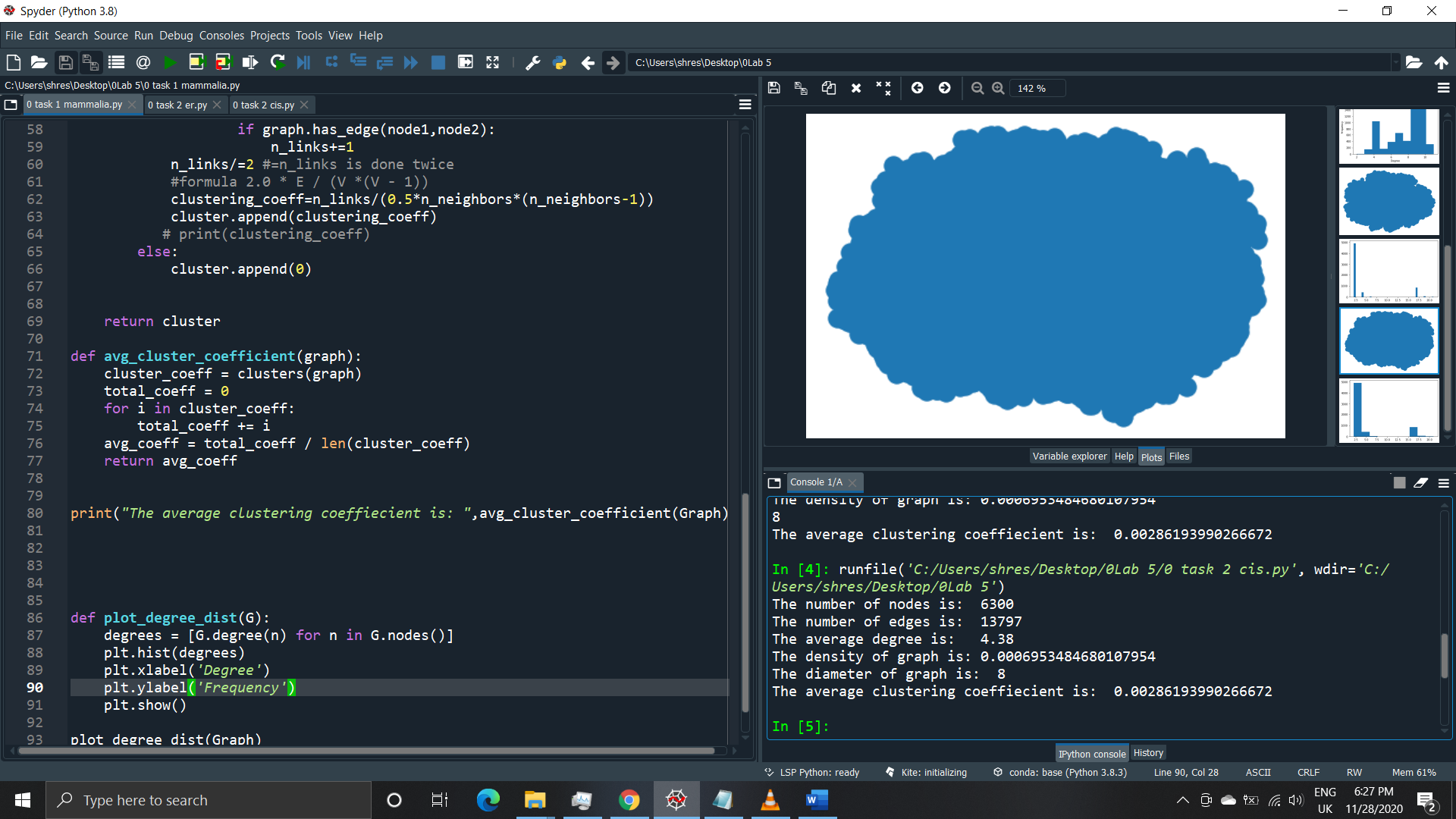
**Fifth set cis**

**(05 task 2 cis.py)**

**Output**







**Test case(test\_bst.py)**

import unittest

from bst import BinarySearchTree

class BSTTestCase(unittest.TestCase):

    def setUp(self):

        """

        Executed before each test method.

        Before each test method, create a BST with some fixed key-values.

        """

        self.bst = BinarySearchTree()

        self.bst.add(10, "Value for 10")

        self.bst.add(52, "Value for 52")

        self.bst.add(5, "Value for 5")

        self.bst.add(8, "Value for 8")

        self.bst.add(1, "Value for 1")

        self.bst.add(40, "Value for 40")

        self.bst.add(30, "Value for 30")

        self.bst.add(45, "Value for 45")

    def test\_add(self):

        """

        tests for add

        """

        # Create an instance of BinarySearchTree

        bsTree = BinarySearchTree()

        # bsTree must be empty

        self.assertEqual(bsTree.size(), 0)

        # Add a key-value pair

        bsTree.add(15, "Value for 15")

        # Size of bsTree must be 1

        self.assertEqual(bsTree.size(), 1)

        # Add another key-value pair

        bsTree.add(10, "Value for 10")

        # Size of bsTree must be 2

        self.assertEqual(bsTree.size(), 2)

        # The added keys must exist.

        self.assertEqual(bsTree.search(10), "Value for 10")

        self.assertEqual(bsTree.search(15), "Value for 15")

    def test\_inorder(self):

        """

        tests for inorder\_walk

        """

        self.assertListEqual(self.bst.inorder\_walk(), [1, 5, 8, 10, 30, 40, 45, 52])

        # Add one node

        self.bst.add(25, "Value for 25")

        # Inorder traversal must return a different sequence

        self.assertListEqual(self.bst.inorder\_walk(), [1, 5, 8, 10, 25, 30, 40, 45, 52])

    def test\_postorder(self):

        """

        tests for postorder\_walk

        """

        self.assertListEqual(self.bst.postorder\_walk(), [1, 8, 5, 30, 45, 40, 52, 10])

        # Add one node

        self.bst.add(25, "Value for 25")

        # Inorder traversal must return a different sequence

        self.assertListEqual(self.bst.postorder\_walk(), [1, 8, 5, 25, 30, 45, 40, 52, 10])

    def test\_preorder(self):

        """

        tests for preorder\_walk

        """

        self.assertListEqual(self.bst.preorder\_walk(), [10, 5, 1, 8, 52, 40, 30, 45])

        # Add one node

        self.bst.add(25, "Value for 25")

        # Inorder traversal must return a different sequence

        self.assertListEqual(self.bst.preorder\_walk(), [10, 5, 1, 8, 52, 40, 30, 25, 45])

    def test\_search(self):

        """

        tests for search

        """

        self.assertEqual(self.bst.search(40), "Value for 40")

        self.assertFalse(self.bst.search(90))

        self.bst.add(90, "Value for 90")

        self.assertEqual(self.bst.search(90), "Value for 90")

    def test\_remove(self):

        """

        tests for remove

        """

        self.bst.remove(40)

        self.assertEqual(self.bst.size(), 7)

        self.assertListEqual(self.bst.inorder\_walk(), [1, 5, 8, 10, 30, 45, 52])

        self.assertListEqual(self.bst.preorder\_walk(), [10, 5, 1, 8, 52, 45, 30])

    def test\_smallest(self):

        """

        tests for smallest

        """

        self.assertTupleEqual(self.bst.smallest(), (1, "Value for 1"))

        # Add some nodes

        self.bst.add(6, "Value for 6")

        self.bst.add(4, "Value for 4")

        self.bst.add(0, "Value for 0")

        self.bst.add(32, "Value for 32")

        # Now the smallest key is 0.

        self.assertTupleEqual(self.bst.smallest(), (0, "Value for 0"))

    def test\_largest(self):

        """

        tests for largest

        """

        self.assertTupleEqual(self.bst.largest(), (52, "Value for 52"))

        # Add some nodes

        self.bst.add(6, "Value for 6")

        self.bst.add(54, "Value for 54")

        self.bst.add(0, "Value for 0")

        self.bst.add(32, "Value for 32")

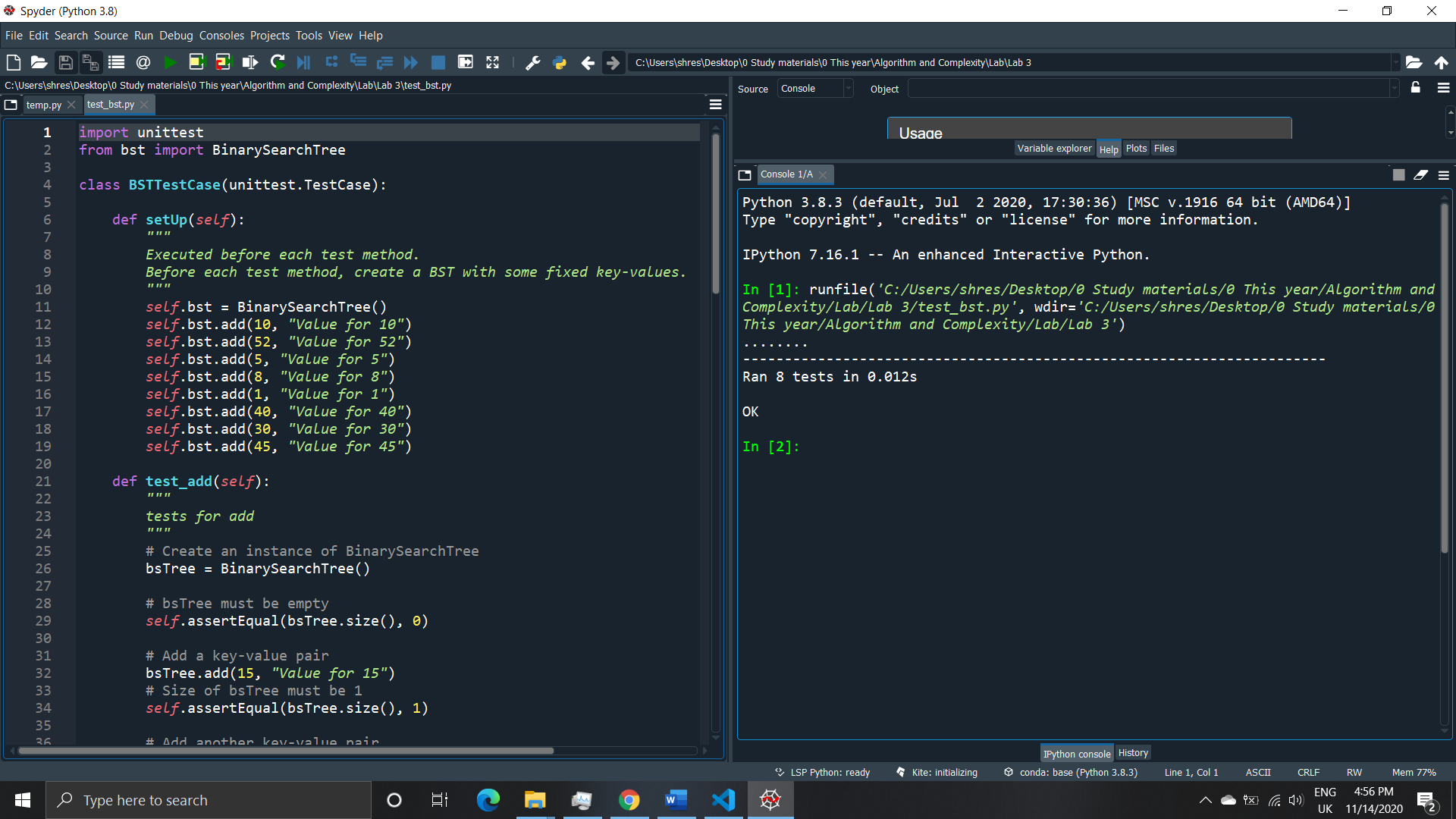
        # Now the largest key is 54

        self.assertTupleEqual(self.bst.largest(), (54, "Value for 54"))

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

**Output**



**Observation**

The operations in Binary Search Tree was observed. The insertion of values in BST, searching of BST, removal of value from BST and making of BST after the removal of value was performed using python. While inserting in BST the left part should always be less then the parent and the right part should always be greater than the parent node. If this rule isn’t followed then it is not a Binary Search Tree. After deletion of value from a node. If the node is leaf node or a node with no children then no other operation should be performed to maintain the Binary Search Tree. But if the node isn’t a leaf node or node with children then either the largest element from left most subtree should be selected as new value of the node or the smallest element from right most node is selected. The smallest value is present in the leftmost node of the tree and the largest value is present in the rightmost part of the tree. Traversal of the tree was performed. In order, preorder and post order traversal was performed in the binary search tree. Inorder traversal gives nodes in non-decreasing order. Preorder traversal is used to create a copy of the tree. Postorder traversal is used to delete the tree. In inorder traversal first left node is selected then parent and then right node is selected. In preorder traversal first the leftmost node is selected then right node and then the parent is selected. This preorder traversal is similar in concept to DFS traversal. In postorder traversal, parent is selected then left and right. The program was written in bst.py. The program was then tested with program test\_bst.py. The test\_bst.py was given by lecturer to test the algorithm we have written. All the tests were successfully completed using unittest in python.