**Chapter 1:**

**R- 1.2: Write a short Python function, is even(k), that takes an integer value and returns True if k is even, and False otherwise. However, your function cannot use the multiplication, modulo, or division operators.**

k = input("Please enter a number: \n")

k = int(k)

def evennumber (k):

return (k&1 == 0)

print("Your number is even: \n")

print (evennumber(k))

**R- 1.4: Write a short Python function that takes a positive integer n and returns the sum of the squares of all the positive integers smaller than n.**

**n = input("Please enter a number you want to get the sum of the squares for: \n")**

**n = int(n)**

**def squares(n):**

**tally = 0**

**for a in range (1,n):**

**tally = tally + (a\*a)**

**return tally**

**print (squares(n))**

**C- 1.15: Write a Python function that takes a sequence of numbers and determines if all the numbers are different from each other (that is, they are distinct).**

**isdistinct = []**

**x = int(input("Please enter how many elements you wish to test and then enter said elements: \n"))**

**for i in range(0,x):**

**element = int(input())**

**isdistinct.append(element)**

**def distinct(isdistinct):**

**elements = list()**

**for i in isdistinct:**

**if i in elements:**

**return "Your elements are not distinct!"**

**#after first elements, if that number is already in list, the elements are not distinct**

**else: elements.append(i)**

**return "Your elements are distinct!"**

**print(distinct(isdistinct))**

**C-1.17 Had we implemented the scale function (page 25) as follows, does it work properly? def scale(data, factor): for val in data: val = factor Explain why or why not**

**No**

**(Say we test this using a list) We are changing a copy of the list, not the actual instance in the list. Val is a local variable which is different from the object being stored in a list. Basically we wouldn't be able to change the original Val, only a copy. We change the object through a reference to it.**

**C- 1.24: Write a short Python function that counts the number of vowels in a given character string.**

**given = input ("Please enter string for vowel checker: \n")**

**given = given.lower()**

**def checker (given, vowels = {"a", "e", "i", "o", "u", "y"}):**

**#some people count y, some people dont. Take it or leave it vowel.**

**counter = 0**

**for i in given:**

**#if the character in the user input is also in the list of vowels we add to the counter**

**if i in vowels:**

**counter = counter + 1**

**return counter**

**print (checker(given))**

**Chapter 2:**

**R-2.5: Use the techniques of Section 1.7 to revise the charge and make\_payment methods of the Credit Card class to ensure that the caller sends a number as a parameter**

**def make\_payment(self, amount):**

**try:**

**amount = float(amount)**

**except:**

**print("You have to give a number as a payment")**

**self.\_balance -= amount**

**R- 2.8: Modify the declaration of the first for loop in the CreditCard tests, from Code Fragment 2.3, so that it will eventually cause exactly one of the three credit cards to go over its credit limit. Which credit card is it?**

**for val in range(1, 58):**

**wallet[0].charge(val)**

**wallet[1].charge(2\*val)**

**wallet[2].charge(3\*val)**

After lengthy testing of every single range from 1,60; I have found that the 3rd card would go over its limit first if we ONLY changed the declaration of the for loop. Although in the charge method, we know that we cannot go over the limit, we can see that if the 3rd card was charged at range 1,58 it would be over its limit while the first and second card would not be.

**R- 2.10: Implement the \_\_ neg \_\_ method for the Vector class of section 2.3.3, so that the expression -v returns a new vector instance whose coordinates are all the negated values of the respective coordinates of v.**

**class Vector:**

**def init (self, d):**

**self. coords = [0]\*d**

**def len (self):**

**return len(self. coords)**

**def getitem (self, j):**

**return self. coords[j]**

**def setitem (self, j, val):**

**self. coords[j] = val**

**def add (self, other):**

**if len(self) != len(other): # relies on len method**

**raise ValueError("dimensions must agree")**

**result = Vector(len(self)) # start with vector of zeros**

**for j in range(len(self)):**

**result[j] = self[j] + other[j]**

**return result**

**def eq (self, other):**

**return self. coords == other. coords**

**def ne (self, other):**

**return not self == other # rely on existing eq definition**

**def str (self):**

**return "<" + str(self. coords)[1:-1] + ">" # adapt list representation**

**def \_\_neg\_\_ (self):**

**final = []**

**for i in range (len(self)):**

**final.append(-self[i])**

**return final**