CMR
INSTITUTE OF
TECHNOLOGY

Internal Assesment Test – III, December 2020

Sub:	Programming Using Python			Code:	18MCA32
Date: 12-12-2020	Duration: 90 mins	Max Marks: 50	Sem: III	Branch:	MCA

	Answer Any 5 **QUESTION**s	Marks	OBE CO	OBE RBT
1a)	Demonstrate the process of grouping widgets with frame types.	05	CO6	L2
b)	Write a class for complex numbers and add methods to add and multiply two complex number.	05	CO6	L3
2	Explain the MVC design with the help of Tkinter program.	10	CO6	L2
3	Demonstrate the creation of ANY 5 widgets using Tkinter.	10	CO6	L2
4	Demonstrate the creation of GUI using object-oriented methods. What is the use of mutable variables with the widgets in Tkinter?	10	CO6	L3,L2
5	What is event driving programming? Explain any one event driven operation.	10	CO6	L2,L3
6 a)	List out and explain the phases involved in Object oriented programming.	05	CO5	L2
b)	Explain the process of writing a method in Class Book.	05	CO5	L2
7	Define a class distance with two data members feet and inches along with three member functions to read a distance object, print a distance object, and add two distance objects. Use this class in a program to read and add 3 distance object.	10	CO5	L4
8	Write a program to read a string in lower case and count the occurrence of each alphabet with the help of dictionary.	10	CO3	L3

Answer 1a) A tkinter Frame is a container. Frames are not directly visible on the screen; instead, they are used to organize other widgets. The following code creates a frame, puts it in the root window, and then adds three Labels to the frame:

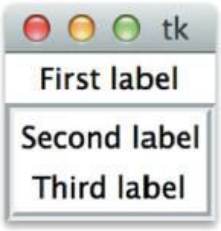
```
import tkinter
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
first = tkinter.Label(frame, text='First label')
first.pack()
second = tkinter.Label(frame, text='Second label')
second.pack()
third = tkinter.Label(frame, text='Third label')
third.pack()
window.mainloop()
```

Here is the resulting GUI:

Below is an example with the same three Labels but with two frames. The second frame has a visual border around it:

```
import tkinter
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
frame2 = tkinter.Frame(window, borderwidth=4, relief=tkinter.GROOVE)
frame2.pack()
first = tkinter.Label(frame, text='First label')
first.pack()
second = tkinter.Label(frame2, text='Second label')
second.pack()
third = tkinter.Label(frame2, text='Third label')
third.pack()
window.mainloop()
```

Here is the resulting GUI:



b) Write a class for complex numbers and add methods to add and multiply two complex number.

Ans:

```
class complex:
    def __init__(self,r,c):
        self.real=r
        self.img=c

    def add(self, c1):
        return complex(self.real+c1.real,self.img+c1.img)

    def mult(self, c1):
        return complex(self.real*c1.real-self.img*c1.img,
                       self.real*c1.img+self.img*c1.real)
    def display(self):
        print str.format('{0}+i{1}',self.real,self.img)

c1=complex(2,3)
c2=complex(4,5)
c3=c1.add(c2)
c4=c1.mult(c2)
c3.display()
```

c4.display()

Output
6+i8
-7+i22

2) The MVC(Model View, Controllers) is a GUI design method that helps separate the parts of an application, which will make the application easier to understand and modify. The main goal of this design is to keep the representation of the data separate from the parts of the program that the user interacts with; that way, it is easier to make changes to the GUI code without affecting the code that manipulates the data.

A GUI program under MVC consists of three parts:

i) View : Component that displays information to the user, e.g. Label, Entry(also accept input). But they do not do processing or storage.

ii) Models: store data, e.g. piece of text or the cost of an object etc. They also don't do computations but keep track of the application's current state and to save that state to a file or database and reload it later). E.g. counter variable keeps track of how many times a button is clicked

iii) Controllers : convert user input into calls on functions which manipulate the data in model. Controllers may update an application's models, which in turn can trigger changes to its views.Example the function registered to be triggered on click of a button.

For example the following piece of code:

```
import tkinter
# The controller.
def click():
counter.set(counter.get() + 1)
if __name__ == '__main__':
window = tkinter.Tk()
# The model.
counter = tkinter.IntVar()
counter.set(0)
# The views.
frame = tkinter.Frame(window)
frame.pack()
button = tkinter.Button(frame,  text='Click', command=click)
button.pack()
label = tkinter.Label(frame, textvariable=counter)
label.pack()
# Start the machinery!
window.mainloop()
```

Here the model is kept track of by variable counter, which refers to an IntVar so that the view will update itself automatically. The controller is function click, which updates the model whenever a button is clicked. Four objects make up the view: the root window, a Frame, a Label that shows the current value of counter, and a button that the user can click to increment the counter's value.

3) A tkinter program is a collection of widgets along with their GUI styles and their layout. Some of the widgets available with tkinter are

i) Button : A clickable button
ii) Checkbutton : A clickable box that can be selected or unselected
iii) Entry: A single-line text field that the user can type in
iv) Frame :A container for widgets
v) Label : A single-line display for text
vi) Menu : A drop-down menu
vii) Text : A multiline text field that the user can type in

Label

Labels are widgets that are used to display short pieces of text. Here we create a Label that belongs to the root window—its *parent widget*—and we specify the text to be displayed by assigning it to the Label's text parameter.The format for creating a label is

label = tkinter.Label(<<parent>>, text=*<<Text to be displayed in label>>*)

where <<parent>> is the container in which to put the label.

Frame

As described in Q3

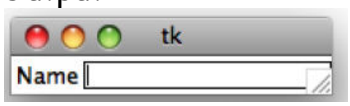
Entry

Entry is a widget which let users enter a single line of text. If we associate a StringVar with the Entry, then whenever a user types anything into that Entry, the StringVar's value will automatically be updated to the contents of the Entry.
The format for creating an Entry is
entry = tkinter.Entry(<<parent>>, textvariable=<<variable name>>)

The below example covers label and Entry:

```
from Tkinter import *
window = Tk()
frame = Frame(window)
frame.pack()
label = Label(frame, text="Name")
label.pack(side="left")
entry = Entry(frame)
entry.pack(side="left")
window.mainloop()
```

Output:



Button

Button is a clickable widget with which can act as a trigger when clicked. The format for creating a button is :

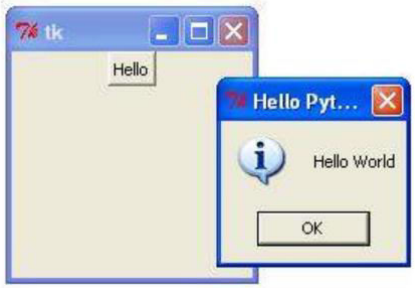
button = tkinter.Button(<<parent>>, text=<<text to be displayed on the button>>, command=<<Name of function to be called when button is clicked>>)

The third, command=<<function>>, tells it to call function <<function>> each time the user presses the button. This makes use of the fact that in Python a function is just another kind of object and can be passed as an argument like anything else.

For example the following code

```
import Tkinter
import tkMessageBox

top = Tkinter.Tk()

def helloCallBack():
   tkMessageBox.showinfo( "Hello Python", "Hello World")

B = Tkinter.Button(top, text ="Hello", command = helloCallBack)

B.pack()
top.mainloop()
```

Output:



Text

Text is a widget which is used to take multiple lines of text as input. The format of for creation of Text widget is

text = tkinter.Text(<<parent>>, height=<<h>>, width=<<w>>)

where <<parent>> is the parent frame/window, <<h>> is the number of rows and <<w>> is the number of columns.

The insert method of Text allows to enter text at the end of the text area. The format is:

text.insert(tkinter.INSERT, <<text to be inserted>>)

Text provides a much richer set of methods than the other widgets. We can embed images in the text area, put in tags, select particular lines, and so on.

For example

```
from Tkinter import *

root = Tk()
T = Text(root, height=2, width=30)
T.pack()
T.insert(END, "Just a text Widget\nin two lines\n")
mainloop()
```

The output would be

Checkbuttons:

Checkbuttons/*checkboxes*, have two states: on and off. When a user clicks a checkbutton, the state changes. We can use tkinter mutable variable to keep track of the user's selection. An IntVar variable can be used and the values 1 and 0 indicate on and off, respectively.

```
from Tkinter import *

master = Tk()

var = IntVar()

c = Checkbutton(master, text="Expand", variable=var)
c.pack()

mainloop()
```

In the above program a checkbutton 'c' is created and put in the master window and an Intvar 'var' is associated with the current state of the checkbutton.

Menu

This widget is used to display all kinds of menus used by an application. Toplevel menus are displayed just under the title bar of the root or any other toplevel windows. To create a toplevel menu, create a new Menu instance, and use **add** methods to add commands and other menu entries to it.

```
from Tkinter import *

def first():
   print "First"

def second():
   print "Second"

window=Tk()
menubar1=Menu(window)
menubar=Menu(window)
menubar.add_command(label='First',command=first)
menubar.add_command(label='Second',command=second)
menubar1.add_cascade(label='File',menu=menubar)
window.config(menu=menubar1)

window.mainloop()
```

In the above program two menu objects are created - menubar and menubar1. Items are added to the menu using the add_command method. The first argument specifies the label to be displayed and the second specifies the function that needs to be invoked on clicking on the menu option. 'menubar' object is added as a submenu of 'menubar1' using the add_cascade method invocation. The line ' window.config(menu=menubar1)' specifies that menunar1 is the main menu for the window.

4. Ans) GUI Programs written in non Object Oriented fashion are not very well structured since most of the code is not modularized into functions. Also they rely greatly on global variables even if functions are used.This becomes a challenge when building large applications for understanding and debugging.
Hence all real GUI are built using classes and objects that package models, views and controllers into one unit. An example of this is shown in the program below for displaying the contents of a counter which increases with every click of a button

```
class Counter:
    """A simple counter GUI using object-oriented programming."""
    def __init__(self, parent):

        """Create the GUI."""

        # Framework.
        self.parent = parent
        self.frame = tkinter.Frame(parent)
        self.frame.pack()

        # Model.
        self.state = tkinter.IntVar()
        self.state.set(1)

        # Label displaying current state.
        self.label = tkinter.Label(self.frame, textvariable=self.state)
        self.label.pack()

        # Buttons to control application.
        self.up = tkinter.Button(self.frame, text='up', command=self.up_click)
        self.up.pack(side='left')

        self.right = tkinter.Button(self.frame, text='quit',
                                    command=self.quit_click)
        self.right.pack(side='left')

    def up_click(self):
        """Handle click on 'up' button."""

        self.state.set(self.state.get() + 1)

    def quit_click(self):
        """Handle click on 'quit' button."""

        self.parent.destroy()
if __name__ == '__main__':

    window = tkinter.Tk()
    myapp = Counter(window)
    window.mainloop()
```

Note here that all the variables required for the application i.e. frame, state, label and button are class members. Hence they are contained within the class and are not accessible from outside. Specifically self.state which stores the counter variable is not global but still can be accessible from the function up_click().

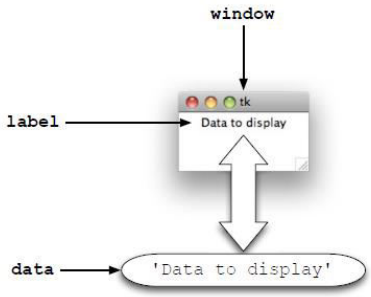
Mutable variables provide a good way to manage the interactions between a program's GUI and its variables. Suppose a string needs to be displayed in several places in a GUI—the application's status bar, some dialog boxes, and so on. Assigning a new value to each widget each time the string changes isn't the best solution because it may be possible that some of such updates are left out accidentally. The requirement is for a variable which would update widgets using it automatically.

Since Python's strings, integers, doubles, and Booleans are immutable, Tkinter provides types of its own that can be updated in place and that can notify widgets whenever their values

change. Using the Tkinter provided StringVar instead of str, will in notifying widgets it has been assigned to that its time to update, whenever a new value is assigned to that StringVar. The values in Tkinter mutable types are set and retrieved using the set and get methods. The following code snippet shows an example:

```
from tkinter import *
window = Tk()
data = StringVar()
data.set("Data to display")
label = Label(window, textvariable=data)
label.pack()
window.mainloop()
```

Here the StringVar data is associated with the 'label' and whenever it changes the label is updated automatically as depicted in the figure below:



A StringVar or any other mutable variable cannot be created until the Tk() function is called to create the top-level window. Similar to StringVar other mutable types provided by tkkinter are IntVar , BooleanVar and FloatVar.

5.) Event-driven programming

Anything that happens in a user interface is an *event*. We say that an event is *fired* whenever the user does something – for example, clicks on a button or types a keyboard shortcut. Some events could also be triggered by occurrences which are not controlled by the user – for example, a background task might complete, or a network connection might be established or lost.

Our application needs to monitor, or *listen* for, all the events that we find interesting, and respond to them in some way if they occur. To do this, we usually associate certain functions with particular events. We call a function which performs an action in response to an event an *event handler* – we *bind* handlers to events.

Program1:

```
from Tkinter import *
 window = Tk()
label = Label(window, text="This is our label.")
label.pack()
```

- ✓ The last line of this little program is crucial.
- ✓ Like other widgets, Label has a method called pack that places it in its parent and then tells the parent to resize itself as necessary. If we forget to call this method, the child widget (in this case, the label) won't be displayed or will be displayed improperly.

✓
✓ Program2:
✓ **from** Tkinter **import** *
✓ **import** time
✓ window = Tk()
✓ label = Label(window, text="*First label.*")
✓ label.pack()
✓ time.sleep(2)
✓ label.config(text="*Second labe*
✓
✓
✓
✓ Program3:
✓ **from** Tkinter **import** *
✓ window = Tk()
✓ data = StringVar()
✓ data.set("*Data to display*")
✓ label = Label(window, textvariable=data)
✓ label.pack()
✓ window.mainloop()

6 a) Object-oriented programming involves at least these phases:
1. *Understanding the problem domain.* A crucial step in problem solving is to understand the requirements.
2. *Figuring out what type(s) you might want.* Reading the problem description to decide the data types to be used may be done by identifying nouns/noun phrases.
3. *Figuring out what features you want your type to have.* The next step is to decide the methods that use the data types used in the previous step.
4. *Writing a class that represents this type: This involves describing the type* which involves - writing a class, including a set of methods inside that class.
5. *Testing your code.* This involves testing the methods separately and also the ways in which the various methods will interact.

b) Ans:
1. The first step in designing a Book class is to decide the data members/features which would be required to store details of the book. In our case we would like to store the isbn (string), authors(list of strings), publisher(string), price(float), title(string).
These features can be listed in the constructor or the __init__ method as shown below:

```
class Book:
    """Information about a book, including title, list of authors,
    publisher, ISBN, and price.
```

```
    """
    def __init__(self, title, authors, publisher, isbn, price):
        """ (Book, str, list of str, str, str, number)  -> NoneType
        Create a new book entitled title, written by the people in authors,
        published by publisher, with ISBN isbn and costing price dollars'"
        self.title = title
        # Copy the authors list in case the caller modifies that list later.
        self.authors = authors[:]
        self.publisher = publisher
        self.ISBN = isbn
        self.price = price
```

Method __init__ is called whenever a Book object is created. Its purpose is toinitialize the new object; Here
are the steps that Python follows when creating an object:
1. It creates an object at a particular memory address.
2. It calls method __init__, passing in the new object into the parameter self.
3. It produces that object's memory address.

1. After this any number of methods as per need can be added to the class. Note: the method always takes the object itself(self) as the first argument. For instance to know the number of authors we can add a method num_authors.

```
    def num_authors(self):
        return len(self.authors)
```

This method returns the number of authors for the book.

7.)
```
class dist:
    __ft__=0
    __inch__=0
    def __init__(self,a,b):
        Self.__ft__=a
        Self.__inch__=b
    def __str__(self):
        return(str(self.__ft__)+  str(self.++inch__))
    def __add__(self,other)
        d3=dist(0,0)
        d3.__inch__=self.__inch__+other.__inch__
    if d3.__inch__  >=12:
        d3.__inch__=d3.__inch__-12
        d3.__ft=1
    d3.__ft__=d3.__ft__+self.__ft__+other.__ft__
    return d3
#main  program
d1=dist(5,3)
d2=dist(6,8)
d3=dist(4,5)
d4=d1+d2
d5=d4+d3
print("d1=",  d1, "d2=", d2, "d3=", d3)
```

print(“d4=d1+d2”, d4)
print(“d5=d4+d3”, d5)

8.) Ans: `all_freq = {}`

```
test_str=input().lower()

for i in test_str:
    if i in all_freq:
        all_freq[i] += 1
    else:
        all_freq[i] = 1


print ("Count of all characters in string is :\n "
                                        +  str(all_freq))
```