CBCS SCHEME

USN 1CR19MCA91

18MCA32

Third Semester MCA Degree Examination, Dec.2019/Jan.2020
Programming using Python

Time: 3 hrs. Max. Marks: 100

Note: ***Answer FIVE full questions, choosing ONE full question from each module.***

Module-1

1 a. Write the following in python:

i) $0 < n < 1$

ii) 2^{3^2}

iii) $\dfrac{\log_2 x + \sin 45^\circ}{xy}$

iv) a is greater than any one of x, y, z

v) $a + \dfrac{b}{c + \dfrac{d}{ef}}$

vi) $a\left\{[b+c]^{3/2}\sqrt{gh}\right\}$ **(06 Marks)**

b. Define a function $f(x,y,z) = \sqrt{x+y+z}$ and use it in a program to compute the following:

$p = \sqrt{a+b+c}$

$q = \sqrt{a^2 + 4b^2 + c^4}$

$r = \dfrac{abc}{(a+b+c)^{3/2}}$

$s = \sqrt{a+b+\sqrt{a+b+c}}$ **(08 Marks)**

c. Explain how input operation is performed through keyboard. **(06 Marks)**

OR

2 a. Evaluate the following showing each step based on precedence and associativity:

i) 5//3*2 – 6/3*5%3

ii) 5*3%2 + 2**3**2

iii) 5%8*3+8%3*5

iv) 10*2 > = 5*2 and not 10 > 20 **(08 Marks)**

b. Write a program to read marks of 3 tests M_1, M_2, M_3 and find the average of best 2 tests rounding to next integer in case of fraction in average without using if statement. **(06 Marks)**

c. Explain how the following are used in strings: i) Multiline string ii) Quotes iii) Escape sequence characters. **(06 Marks)**

Module-2

3 a. Explain different forms of "if" statements with syntax and examples. **(08 Marks)**

b. Write a program to input x and y coordinates of a point. Find the lines in any of the following: i) Origin ii) x-axis iii) y-axis iv) I[st] quadrant v) II[nd] quadrant vi) III[rd] quadrant or vii) IV[th] quadrant. **(06 Marks)**

c. Define your own module "conversion" consisting of 2 methods i) to convert INR to USD ii) to convert USD to INR. Import this module and show how the Indian Rupees (INR) are converted to American Dollars (USD) and vise-versa assuming 1USD = 72INR. **(06 Marks)**

OR

4 a. Explain different forms of importing modules. **(06 Marks)**

b. Discuss the significance of docstrings. **(06 Marks)**

c. Explain the following string methods with examples:
i) split ii) strip iii) swapcase iv) count v) upper vi) lower vii) find viii) rjust. **(08 Marks)**

Module-3

5 a. Explain any 7 methods of list with examples. **(07 Marks)**

b. Consider a list x with elements [5, 7, 6, 4, 3, 9, 2]. Show how the following operations are performed on x without using built-in methods/functions:
i) Remove the first element
ii) Remove the last element
iii) Insert 10 at the beginning
iv) Insert 10 at the end
v) Remove element in position 2
vi) Insert 10 into position 3
vii) Remove all the elements in x. **(07 Marks)**

c. Write a program to find the sum of digits in a given number using while loop. **(06 Marks)**

OR

6 a. Explain in operator and range function with examples. **(06 Marks)**

b. Show how matrix a of order 4 × 4 is created using list initializing a as unit matrix. Write the python code segment to find the trace (sum of diagonal elements) of a. **(08 Marks)**

c. Write a program to compute $f(x) = \sqrt{x^2 + 2x - 1}$ for all x ranging from 2.0 to 3.0 in steps of 0.1 using for loop with range function. **(06 Marks)**

Module-4

7 a. Explain any 4 methods on each of the following storage collection types i) file ii) set iii) dictionary. Give examples. **(12 Marks)**

b. Write a program to read string in lower case and count the occurrence of each alphabet with the help of dictionary. **(08 Marks)**

OR

8 a. Compare the storage collection types strings, lists, tuples, sets and dictionary. **(12 Marks)**

b. What is inversion of dictionary? Give examples. **(08 Marks)**

Module-5

9 a. What are classes and objects? Explain encapsulation, polymorphism, inheritance in object oriented programming. **(10 Marks)**

b. Define a class distance with 2 data members feet and inches along with 3 member functions to read a distance object, print a distance object and add 2 distance objects. Use this class in a program to read and add 3 distance objects. **(10 Marks)**

OR

10 a. Explain any 10 GUI widgets with respect to tkinter. **(10 Marks)**

b. What is event driven programming? Explain any 3 event driven operations. **(10 Marks)**

* * * * *

Q1: a

Ans: (i) 0<x and x<1

(ii) (2**3)**2

(iii) (math.log/x,2)+math.sin/math.radious(45)/(x*Y)

(iv)a>x or a>y or a>z

(v) a+ b/(c+d/(e*f))

(vi) a * ((b+c)**(3/2)*math.sqrt(g*h))

Q1: b

Ans:
```
import math
    def  f(x,y,z):
        s=math.sqrt(x+y+z)
        return s
  a= float(input())
  b= float(input())
  c= float(input())
  p=f(a,b,c)
 q=f(a*a, 4*b*b,c**4)
 r=a*b*c/f(a,b,c)**(3/2)
s= f(a,b,f(a,b,c))
print(p,q,r,s)
```

Q1: c

Ans: While Python provides us with two inbuilt functions to read the input from the keyboard.

- **raw_input (prompt)**
- **input (prompt)**

raw_input ( ) : This function works in older version (like Python 2.x). This function takes exactly what is typed from the keyboard, convert it to string and then return it to the variable in which we want to store. For example –

input ( ) : This function first takes the input from the user and then evaluates the expression, which means Python automatically identifies whether user entered a string or a number or list. If the input provided is not correct then either syntax error or exception is raised by python. For example –

```
val = input("Enter your value: ")
print(val)
```

```
Enter your value: 123
123
>>>
```

v

How the input function works in Python :

- When input() function executes program flow will be stopped until the user has given an input.
- The text or message display on the output screen to ask a user to enter input value is optional i.e. the prompt, will be printed on the screen is optional.
- Whatever you enter as input, input function convert it into a string. if you enter an integer value still input() function convert it into a string. You need to explicitly convert it into an integer in your code using typecasting.

Q2 a:

Ans: (i) 5//3*2-6/3*5%3

2-1=1

(ii) 5*3%2+2**3**2

1+512=513

(iii) 5%8*3+8%3*5

15+10=25

(iv) 10*2>=5*2 and not10>20

20>=10 and not 10>20

True and not False

True and True

True

Q 2 b:

Ans:

```
m1=int(input())

m2=int(input())

m3=int(input())

s=sum([m1,m2,m3])

best=s-min(m1,m2,m3)

avg=int(best/2 +0.5)

print("Average of two better marks is", avg)
```

Q2 c:

Ans(i) Multiline String: If you create a string using single or double quotes, the whole string must fit onto a single line. Here's what happens when you try to stretch a string across multiple lines:

```
>>> 'one
Traceback (most recent call last):
File "<string>" , line 1, in <string>
Could not execute because an error occurred:
EOL while scanning single-quoted string: <string>, line 1, pos 4:
'one
```

EOL stands for "end of line," so in this error report, Python is saying that it reached the end of the line before it found the end of the string. To span multiple lines, put three single quotes or three double quotes around the string instead of one of each. The string can then span as many lines as you want:

```
>>> " 'one
... two
... three'"
' one\ntwo\nthree'
```

Notice that the string Python creates contains a \n sequence everywhere our input started a new line. In reality, each of the three major operating systems uses a different set of characters to indicate the end of a line. This set of characters is called a *newline*. On Linux, a newline is one '\n' character; on Mac OS X, it is one '\r'; and on Windows, the ends of lines are marked with both characters as '\r\n'.
Python always uses a single \n to indicate a newline, even on operating systems like Windows that do things other ways. This is called *normalizing* the string; Python does this so that you can write exactly the same program no matter what kind of machine you're running on.

(ii) Quotes: In Python, we indicate that a value is a string by putting either single or double quotes around it:

```
>>> 'Aristotle'
'Aristotle'
```

```
>>> "Isaac Newton"
'Isaac Newton'
```
The quotes must match:
```
>>> 'Charles Darwin"
File "<stdin>" , line 1
'Charles Darwin"
^
SyntaxError: EOL while scanning single-quoted string
```
We can join two strings together by putting them side by side:
```
>>> 'Albert' 'Einstein'
'AlbertEinstein'
```
Notice that the words Albert and Einstein run together. If we want a space between the words, then we can add a space either to the end of Albert or to the beginning of Einstein:
```
>>> 'Albert ' 'Einstein'
'Albert Einstein'
>>> 'Albert' ' Einstein'
'Albert Einstein'
```
It's almost always clearer to join strings with +. When + has two string operands, then it is referred to as the *concatenation operator*:
```
>>> 'Albert' + ' Einstein'
'Albert Einstein'
```

(iii) Escape Sequence Character:

Suppose you want to put a single quote inside a string. If you write it directly, Python will complain:

```
>>> 'that' s not going to work'
File "<stdin>" , line 1
'that' s not going to work'
^
SyntaxError: invalid syntax
```
The problem is that when Python sees the second quote—the one that you think of as being part of the string—it thinks the string is over. It then doesn't know what to do with all the stuff that comes after the second quote.
One simple way to fix this is to use double quotes around the string:

```
>>> "that's better"
"that's better"
```

Escape Sequence Description
\n End of line
\\ Backslash
\' Single quote
\" Double quote
\t Tab
Figure 3.1: Escape sequences
If you need to put a double quote in a string, you can use single quotes around the string. But what if you want to put both kinds of quote in

one string? You could do this:
>>> 'She said, "That' + '"' + 's hard to read.'"
Luckily, there's a better way. If you type the previous expression into Python, the result is as follows:
'She said, "That\'s hard to read."'
The combination of the backslash and the single quote is called an *escape sequence*. When Python sees a backslash inside a string, it means that the next character represents something special—in this case, a single quote, rather than the end of the string. The backslash is called an *escape character*, since it signals the start of an escape sequence.

Q3 a:

Ans: The basic form of an if statement is as follows:
if condition:
block
The *condition* is an expression, such as name != '' or x < y. Note that this doesn't have to be a Boolean expression. As we discussed in Section In particular, 0, None, the empty string '', and the empty list [ ] all are considered to false, while all other values that we have encountered are considered to be true.
If the condition is true, then the statements in the block are executed; otherwise, they are not. As with loops and functions, the block of statements must be indented to show that it belongs to the if statement. If you don't indent properly, Python might raise an error, or worse, might happily execute the code that you wrote but, because some statements were not indented properly, do something you didn't intend. We'll briefly explore both problems in this chapter.
Here is a table of solution categories based on pH level:

pH Level	**Solution Category**
0–4	Strong acid
5–6	Weak acid
7	Neutral
8–9	Weak base
10–14	Strong base

```
>>> compound = raw_input()
H2SO4
>>> if compound == "H2O" :
... print "Water"
... elif compound == "NH3" :
... print "Ammonia"
... elif compound == "CH3" :
... print "Methane"
... else:
... print "Unknown compound"
...
Unknown compound
```

>>>

Nested if Statements
An if statement's block can contain any type of Python statement, which means that it can include other if statements. An if statement inside another is called a *nested* if statement.

```
input = raw_input()
if len(input) > 0:
ph = float(input)
if ph < 7.0:
print "%s is acidic." % (ph)
elif ph > 7.0:
print "%s is basic." % (ph)
else:
print "%s is neutral." % (ph)
else:
print "No pH value was given!"
```

3 b:

Ans:

```
def quadrant(x, y):

    if (x > 0 and y > 0):
        print ("lies in First quadrant")

    elif (x < 0 and y > 0):
        print ("lies in Second quadrant")

    elif (x < 0 and y < 0):
        print ("lies in Third quadrant")

    elif (x > 0 and y < 0):
        print ("lies in Fourth quadrant")

    elif (x == 0 and y > 0):
        print ("lies at positive y axis")

    elif (x == 0 and y < 0):
        print ("lies at negative y axis")

    elif (y == 0 and x < 0):
        print ("lies at negative x axis")

    elif (y == 0 and x > 0):
        print ("lies at positive x axis")

    else:
        print ("lies at origin")


x = 1
```

```
y = 1
quadrant(x, y)
```

Output: lies in First quadrant

Q 3 c:

Ans:
```
def INR2USD(x):
        y=x/72
        return y
    def   USD2INR(x):

        y=x*72

        return y
```

/// Save above file with conversion.py

```
import conversion


USD=conversion.INR2USD(2000)

INR=conversion.USD2INR(50)

Print("Amount in dollor", USD)

Print("Amount in Rs.", INR)
```

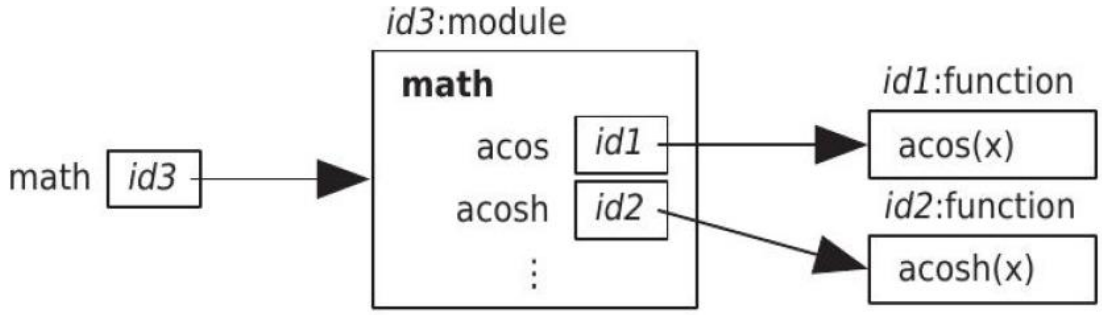
Q 4 a:

Ans: The two ways to import a module is to import the entire module using the statement:

import <module_name>

e.g. import math

In this case all the functions in the module are imported and are assigned memory.



A module variable math is created which points to module structure containg addresses of all the functions in the module. as shown in the above diagram. To invoke a function we use the format

<module name>.<function name>(args)

.E.g. math.sqrt(5).

Another way to import are specific functions and variables from a module using the format:

from <module name> import fn1,fn2

E.g. from math import math, pi

In such a case only the functions sqrt and variable pi are imported from math. The other functions are not stored . A function call be invoked by just calling the function without prepending the module name

e.g. sqrt(9)

Usually the second method is preferred since it only imports the functions and variables required. The first method causes all the methods to occupy memory irrespective of whether they are used or not.

4 b:

Ans: The term *docstring* is short for "documentation string." Docstrings are
easy to create: if the first thing in a file or a function is a string that
isn't assigned to anything, Python saves it so that help can print it later. Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods. It's specified in source code that is used, like a comment, to document a specific segment of code. Unlike conventional source code comments, the docstring should describe what the function does, not how.

What should a docstring look like?

- The doc string line should begin with a capital letter and end with a period.
- The first line should be a short description.
- If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description.
- The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

Declaring Docstrings: The docstrings are declared using """triple double quotes""" just below the class, method or function declaration. All functions should have a docstring.

Docstrings are great for the understanding the functionality of the larger part of the code, i.e., the general purpose of any class, module or function whereas the comments are used for code, statement, and expressions which tend to be small. They are a descriptive text written by a programmer mainly for

themselves to know what the line of code or expression does. It is an essential part that documenting your code is going to serve well enough for writing clean code and well-written programs. Though already mentioned there are no standard and rules for doing so.

4 c: Ans:

(i) Split():

S.split([sep [,maxsplit]]) -> list of strings

Return a list of the words in the string S, using sep as the

delimiter string. If maxsplit is given, at most maxsplit

splits are done. If sep is not specified or is None, any

whitespace string is a separator and empty strings are removed

from the result.

```
txt = "welcome to the jungle"

x = txt.split()

print(x)
```

O/P: ['welcome', 'to', 'the', 'jungle']

(ii) Strip():

S.strip([chars]) -> string or unicode

Return a copy of the string S with leading and trailing

whitespace removed.

If chars is given and not None, remove characters in chars instead.

```
txt = "     banana     "

x = txt.strip()

print("of all fruits", x, "is my favorite")
```

O/P: of all fruits banana is my favorite

(iii) Swapcase():

S.swapcase() -> string

Return a copy of the string S with uppercase characters converted to lowercase and vice versa

```
txt = "Hello My Name Is PETER"

x = txt.swapcase()

print(x)
```

O/P: hELLO mY nAME iS peter

(iv) Count():Returns the number of times a specified value occurs in a string

```
txt = "I love apples, apple are my favorite fruit"

x = txt.count("apple")

print(x)
```

O/P: 2

(v) Upper():Converts a string into upper case

```
txt = "Hello my friends"

x = txt.upper()

print(x)
```

O/P: HELLO MY FRIENDS

(vi) Lower():Converts a string into lower case

```
txt = "Hello my FRIENDS"

x = txt.lower()

print(x)
```

O/P: hello my friends

(vii) Find():Searches the string for a specified value and returns the position of where it was found

```
txt = "Hello, welcome to my world."

x = txt.find("welcome")

print(x)
```

O/P: 7

(viii) Rjust():Returns a right justified version of the string

```
txt = "banana"

x = txt.rjust(20)

print(x, "is my favorite fruit.")
```

O/P: banana is my favorite fruit.

Q5 a:

Ans:

L.append(v)	Appends value v to list L
L.insert(i, v)	Inserts value v at index i in list L, shifting following items to make room
L.remove(v)	Removes the first occurrence of value v from list L
L.reverse()	Reverses the order of the values in list L
L.sort()	Sorts the values in list L in ascending order (for strings,alphabetical order)
L.pop()	Removes and returns the last element of L (which must be nonempty)
L.clear()	Removes all items from list

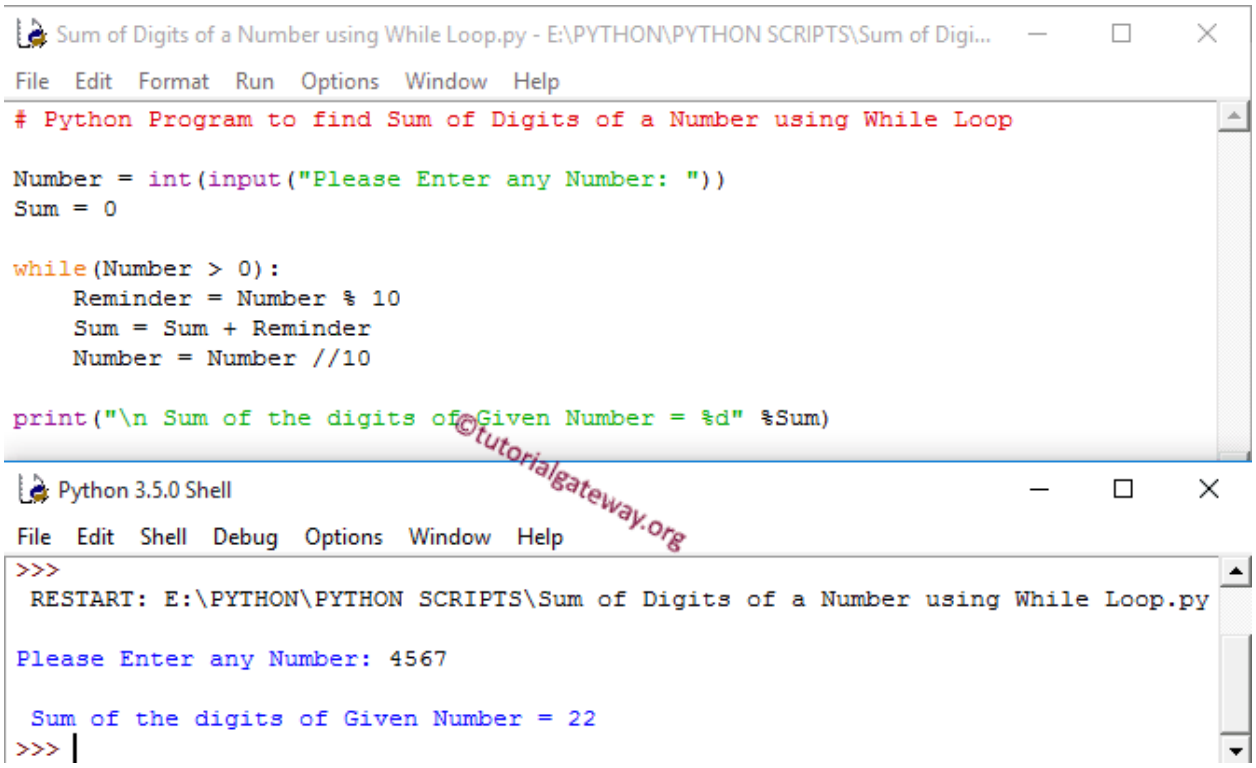
5 b:
Ans:

L= [5,7,6,4,3,9,2]

(i) L=L[1:]
(ii) L=L[:-1]
(iii) L=[10]+L
(iv) L=L+[10]
(v) L=L[:2]+L[3:]
(vi) L=L[:3]+[10]+L[3:]
(vii) L=[]

5 c:
Ans:

```
Number = int(input("Please Enter any Number: "))
Sum = 0

while(Number > 0):
   Reminder = Number % 10
   Sum = Sum + Reminder
   Number = Number //10

print("\n Sum of the digits of Given Number = %d" %Sum)
```



Q 6 a:

Ans:
in and **not in** are the membership operators; used to test whether a value or variable is in a sequence.

in	True if value is found in the sequence
not in	True if value is not found in the sequence

```
x = 'Geeks for Geeks'
y = {3:'a',4:'b'}


print('G' in x)

print('geeks' not in x)

print('Geeks' not in x)

print(3 in y)

print('b' in y)
```

Output:

```
True

True

False
```

True

False

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Syntax

range(*start*, *stop*, *step*)

Parameter	**Description**
start	Optional. An integer number specifying at which position to start. Default is 0
stop	Required. An integer number specifying at which position to end.
step	Optional. An integer number specifying the incrementation. Default is 1

Example:

```
x = range(3, 6)
for n in x:
  print(n)
```

O/P: 3
4
5

6 b:
Ans:

```
a= [[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]]
sum=0
for i in range(0,4):
        sum=sum+a[i][i]
print(sum)
```

6 c.
Ans:

```
for a in range(20,31)
        x=a/10
        fx=math.sqrt(x**2+2*x-1)
        print(x,fx)
```

Q7 a:

Ans: File:

Method	Description
close()	Closes the file
detach()	Returns the separated raw stream from the buffer
fileno()	Returns a number that represents the stream, from the operating system's perspective
flush()	Flushes the internal buffer

Set:

Method	Purpose	Example	Result
Add	Adds an element to a set	lows.add(9)	None
Clear	Removes all elements from a set	lows.clear()	None

difference	Creates a set with elements from one set, but not the other	lows.difference(odds)	set([0, 2, 4]])
intersection	Creates a set with elements that are in both sets	lows.intersection(odds)	set([1, 3]])

Dictionary:

Method	Purpose	Example	Result
clear	Empties the dictionary.	d.clear()	Returns None, but d is now empty.
get	Returns the value associated with a key, or a default value if the key is not present.	d.get('x', 99)	Returns d['x'] if "x" is in d, or 99 if it is not.
keys	Returns the dictionary's keys as a list. Entries are guaranteed to be unique.	birthday.keys()	['Turing', 'Newton', Darwin']
items	Returns a list of (key, value) pairs.	birthday.items()	[('Turing', 1912), ('Newton', 1642), ('Darwin', 1809)]

7 b.:

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))
```

Q8 a:
Ans:

There are quite a few data structures available. The builtins data structures are: lists, tuples, dictionaries, strings, sets and frozensets.

Lists, strings and tuples are ordered sequences of objects. Unlike strings that contain only characters, list and tuples can contain any type of objects. Lists and tuples are like arrays. Tuples like strings are immutables. Lists are mutables so they can be extended or reduced at will. Sets are mutable unordered sequence of unique elements whereas frozensets are immutable sets.

Lists are enclosed in brackets:

```
l = [1, 2, "a"]
```

Tuples are enclosed in parentheses:

```
t = (1, 2, "a")
```

Tuples are faster and consume less memory. See *Tuples* for more information.

Dictionaries are built with curly brackets:

```
d = {"a":1, "b":2}
```

Sets are made using the **set()** builtin function.

collection	Mutable?	Ordered?	Use when
Str	yes	Yes	You want to keep track of text
List	Yes	Yes	You want to keep track of an unordered sequence that you want to update
Tuple	No	Yes	You want to build an ordered equnce that you want to use as a key in a dictionary or as a value in a set.
Set	Yes	No	You want to keep track of values, but order does'nt matter,and you don't want to keep duplicates. The values must be immutable.
dictionary	Yes	No	You want to keep a mapping of keys to values. The keys must be immutable.

8 b:

Ans:

- We might want to print the birds in another order—in order of frequency, for example. To do this, we need to *invert* the dictionary; that is, use the values as keys and the keys as values.
- There's no guarantee that the values are unique, so we have to handle *collisions*.
- The solution is to use some sort of collection, such as a list, to store the inverted dictionary's values.

```
import sys
# Count all the birds.
count = {}
for filename in sys.argv[1:]:
infile = open(filename, 'r')
for line in infile:
name = line.strip()
count[name] = count.get(name, 0) + 1
infile.close()
# Invert the dictionary.
freq = {}
for (name, times) in count.items():
if times in freq:
freq[times].append(name)
else:
freq[times] = [name]
# Print.
for key in sorted(freq):
print key
for name in freq[key]:
print ' ', name
```

Q 9 a:
Ans:

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

To create a class, use the keyword class:

Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

Now we can use the class named MyClass to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

The __init__() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Create a class named Person, use the __init__() function to assign values for name and age:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Encapsulation in Python

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those type of variables are known as **private varibale**.
A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

In literal sense, Polymorphism means the ability to take various forms. In Python, Polymorphism allows us to define methods in the child class with the same name as defined in their parent class.

As we know, a child class inherits all the methods from the parent class. However, you will encounter situations where the method inherited from the parent class doesn't quite fit into the child class. In such cases, you will have to re-implement method in the child class. This process is known as Method Overriding.

If you have overridden a method in the child class, then the version of the method will be called based upon the type of the object used to call it. If a child class object is used to call an overridden method then the child class version of the method is called. On the other hand, if parent class object is used to call an overridden method, then the parent class version of the method is called.

9 b:
Ans:

```
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)
```

Q 10 a:
Ans:

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 :

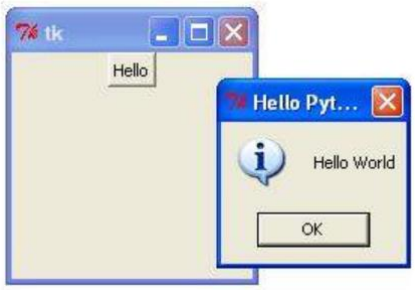
button = tkinter.Button(<<parent>>, text=<<text to be displayed on the button>>, command=<<Name of function to be called when button is clicked>>)

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.

10 b:
Ans:

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()

program:

```
# Initialization.
from Tkinter import *
# The controller.
def click():
counter.set(counter.get() + 1)
if __name__ == '__main__':
# More initialization.
window = Tk()
# The model.
counter = IntVar()
counter.set(0)
# The views.
frame = Frame(window)
frame.pack()
button = Button(frame, text="Click", command=click)
button.pack()
label = Label(frame, textvariable=counter)
label.pack()
window.mainloop()
```

PROGRAM:

```
from Tkinter import *
def cross(text):
text.insert(INSERT, 'X')
```

```
window = Tk()
frame = Frame(window)
frame.pack()
text = Text(frame, height=3, width=10)
text.pack()
button = Button(frame, text="Add", command=lambda: cross(text))
button.pack()
window.mainloop()
```