Model Question Paper-I with effect from 2022-23 (CBCS Scheme)

USN | | | | | | | | | | |

First/Second Semester B.E. Degree Examination

Introductionto Python Programming

TIME: 03 Hours **Max. Marks: 100**

Note: 01. Answer any **FIVE** full questions, choosing at least **ONE** question from each **MODULE**.

		Module -1	*Bloom's Taxonomy Level	Marks
Q.01	a	With Python programming examples to each, explain the syntax and control flow diagrams of break and continue statements.	L2	08
	b	Explain TWO ways of importing modules into application in Python with syntax and suitable programming examples.	L2	06
	c	Write a function to calculate factorial of a number. Develop a program to compute binomialcoefficient (Given N and R).	L3	06
		OR		
Q.02	a	Explain looping control statements in Python with a syntax and example to each.	L2	06
	b	Develop a Python program to generate Fibonacci sequence of length (N). Read N from the console.	L3	04
	c	Write a function named DivExp which takes TWO parameters a, b and returns a value c (c=a/b). Write suitable assertion for a>0 in function DivExp and raise an exception for when b=0. Develop a Python program which reads two values from the console and calls a function DivExp.	L3	06
	d	Explain FOUR scope rules of variables in Python.	L2	04
		Module-2		
Q. 03	a	Explain with a programming example to each: (ii) get() (iii) setdefault()	L2	06
	b	Develop suitable Python programs with nested lists to explain copy.copy( ) and copy.deepcopy( ) methods.	L3	08
	c	Explain append() and index() functions with respect to lists in Python.	L2	06
		OR		
Q.04	a	Explain different ways to delete an element from a list with suitable Python syntax and programming examples.	L2	10
	b	Read a multi-digit number (as chars) from the console. Develop a program to print the frequency of each digit with suitable message.	L3	06
	c	Tuples are immutable. Explain with Python programming example.	L2	04
		Module-3		
Q. 05	a	Explain Python string handling methods with examples: split(),endswith(), ljust(), center(), lstrip()	L2	10
	b	Explain reading and saving python program variables using shelve module with suitable Python program.	L2	06
	c	Develop a Python program to read and print the contents of a text file.	L3	04
		OR		
Q. 06	a	Explain Python string handling methods with examples: join(), startswith(),rjust(),strip(),rstrip()	L2	10
	b	Explain with suitable Python program segments: (i) os.path.basename() (ii) os.path.join().	L2	05
	c	Develop a Python program find the total size of all the files in the given	L3	05

		directory.		
		Module-4		
Q. 07	a	Explain permanent delete and safe delete with a suitable Python programming example to each.	L2	08
	b	Develop a program to backing Up a given Folder (Folder in a current working directory) into a ZIP File by using relevant modules and suitable methods.	L3	06
	c	Explain the role of Assertions in Python with a suitable program.	L2	06
		OR		
Q. 08	a	Explain the functions with examples: (i) shutil.copytree() (ii) shutil.move() (iii) shutil.rmtree().	L3	06
	b	Develop a Python program to traverse the current directory by listing sub-folders and files.	L2	06
	c	Explain the support for Logging with logging module in Python.	L2	08
		Module-5		
Q. 09	a	Explain the methods __init__ and __str__ with suitable code example to each.	L2	06
	b	Explain the program development concept 'prototype and patch' with suitable example.	L2	06
	c	Define a function which takes TWO objects representing complex numbers and returns new complex number with a addition of two complex numbers. Define a suitable class 'Complex' to represent the complex number. Develop a program to read N (N >=2) complex numbers and to compute the addition of N complex numbers.	L3	08
		OR		
Q. 10	a	Explain the following with syntax and suitable code snippet: i) Class definition ii) instantiation iii) passing an instance (or objects) as an argument iv) instances as return values.	L2	10
	b	Define pure function and modifier. Explain the role of pure functions and modifiers in application development with suitable python programs.	L2	10

*Bloom's Taxonomy Level: Indicate as L1, L2, L3, L4, etc. It is also desirable to indicate the COs and POs to be attained by every bit of questions.

Q1)

a) Break and Continue statements →

Break statement →

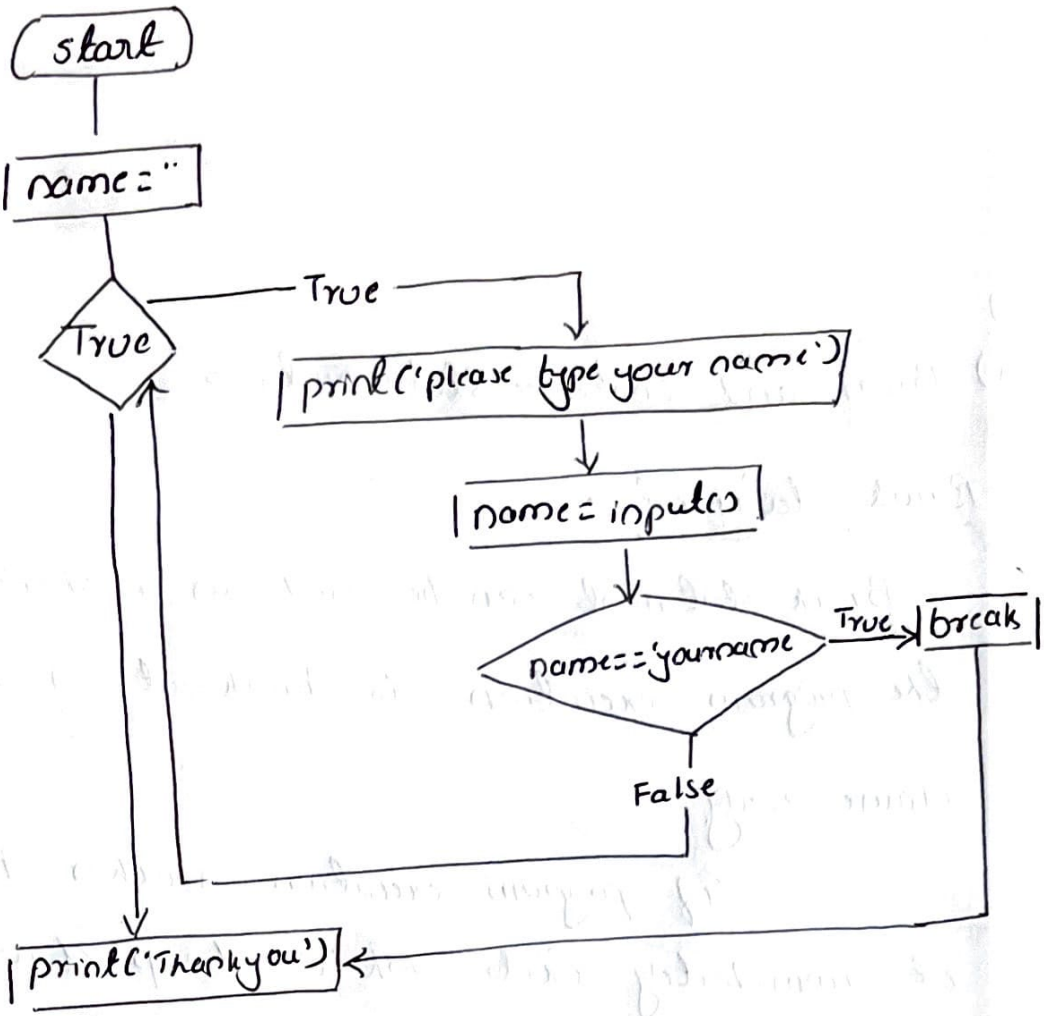
Break statement can be used as a shortcut to getting the program execution to break out of while loops clause early.

If program execution reaches break statement, it immidiately exits while loop clause. In code, a break statement simply contains break keyword.

ex→

```
While True:
    print('please type your name.')
    name = input()
    if name == 'your name':
        break
print('thank you')
```

The above program execution will always enter loop & will exit it only when break statement is executed.



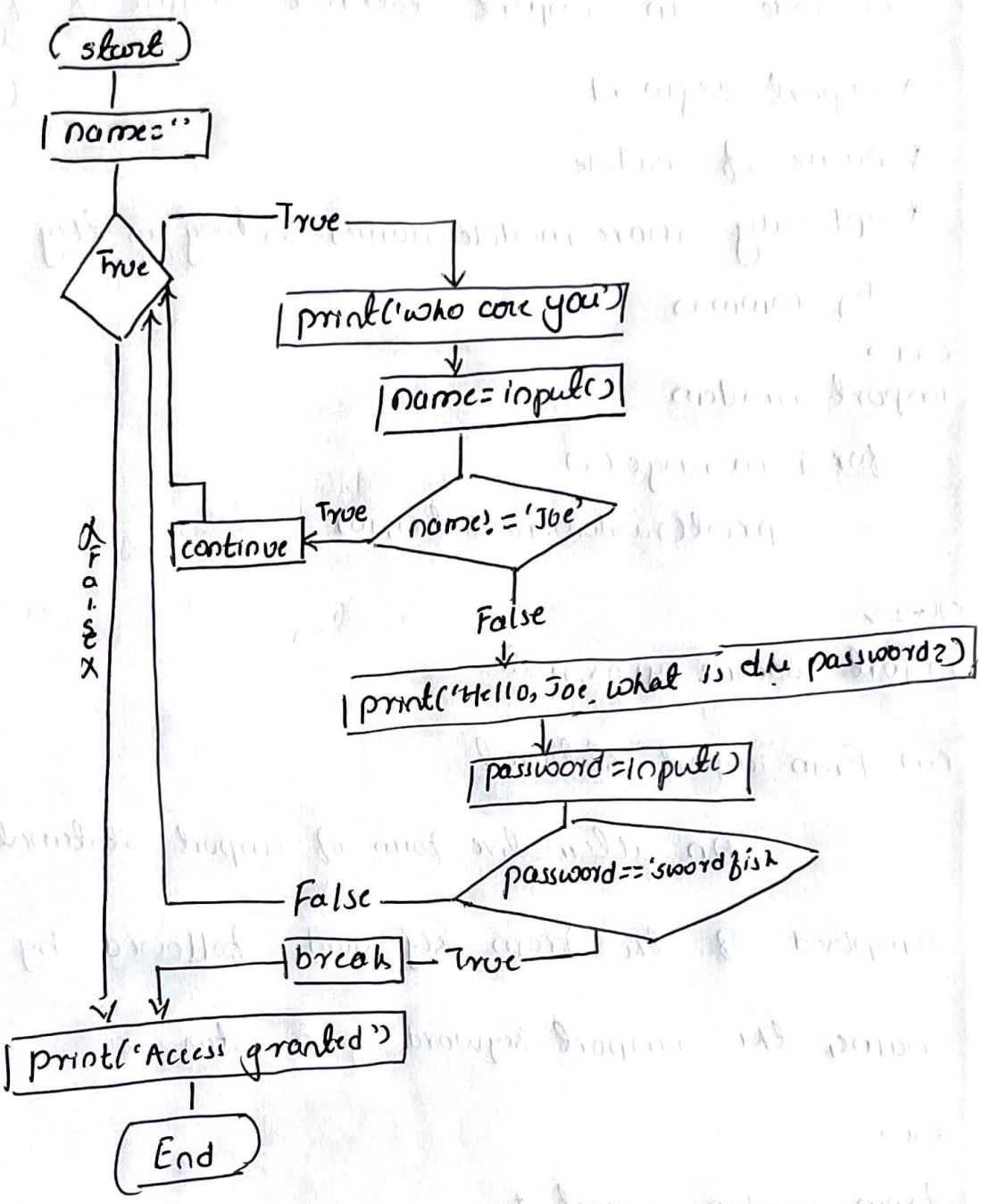
Continue statement:

Like break statements, continue statements are used inside loops. When program execution reaches a continue statement, the program execution immediately jumps back to the start of loop & reevaluates the loops condition.

```
while True:
    print('Who are you')
    name = input()
    if name != 'Joe':
        continue
    print('Hello, Joe') what is the password?')
    password = input()
    if password == 'swordfish'
```

```
        break()
print('Access granted')
```

If user enters any name besides Joe the continue statement causes the program execution to jump back to the start of the loop.



Q1)

b) Two ways of importing modules into application in python →

(i) Import statement →

In code, an import statement consist of following

* import keyword
* name of module
* optionally, more module names as long as they are separated by commas

ex1→

```
import random
    for i in range(5)
        print(random.randint(1, 10)
```

ex-2→

```
import random, sys, os, math
```

(ii) from import statement

An alternative form of import statement is composed of the from key word, followed by module name, the import keyword & a star

ex→

```
from random import *
```

with this form of import statement, calls to functions in random will not need the random. prefix.

Q1)

C)

```
import math

def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)

print('enter a nomber to compule factorial')
n=int(input())
res=factorial(n)
print('factorial is ', res)
print('enter n value')
n=int(input())
print('enter r value')
r=int(input())
res=math.comb(n,r)
print('binomial coefficient ', res)
```

(O/P) →

```
enter a number to compute factorial
4
factorial is 24
enter n value
4
enter r value
2
binomial co-efficient 6
```

Q 2)

a) While Loop →

While Loop make a block of code execute over & over again. The code in while clause will be executed as long as while statement's condition is True

while statement consists of

* The while keyword
* A condition
* A colon

ex →

```
spam = 0
While spam < 5:
    print('Hello world')
    spam = spam + 1
```

for Loop → with range function

for loop with range function will execute a code of block only a certain number of times. a for statement consist of -

* for keyword
* A variable name
* The in keyword
* A call to range()
* A colon

ex →

```
for i in range(5):
    print(i)
```

Q2)

b)

```
print("Enter the value of 'n':")
n=int(input())
f1=0
f2=1
print("Fibonacci Serries;")
print(f1)
print(f2)
for x in range(2,n)
    f3=f1+f2
    print(f3)
    f1=f2
    f2=f3
```

O/P →

```
Enter the value of 'n':
5
Fibonacci Serries:
0
1
1
2
3
```

Q2)

c)

```
def DivExp(a, b):
    assert a>0, "Assertion failed: a must be greater than 0"
    if b==0:
        raise ZeroDivisionError("Division by zero is not allowed")
    c = a/b
    return c

try:
    a = float(input("Enter the value of 'a':"))
    b = float(input("Enter the value of 'b':"))
    result = DivExp(a,b)
    print("Result:", result)
Except ValueError:
    print("Invalid input. please enter valid number for a&b")
Except AssertionError as e:
    print(e)
Except ZeroDivision Error as e:
    print(e)
```

Q2)

d) Four scope rules of variables in python

Parameters & variables that are assigned in a called function are said to exist in that functions local scope.

Variable that are assigned outside all functions are said exist in global scope. A variable that exist in local scope is called local variable, while a variable that exist in global scope is called global variable

Scope rules of varriable in Python are →

* Code in global scope cannot use any local variable
* A local scope can access global variables.
* Code in functions local scope cannot use any other local scope
* The same name for different variables can be used if they are in different scopes.

Q3)

a)

The get() method →

It is tedious to check whether a key exist in dictionary before accessing key value, for this dictionaries have get() method that takes two arguments: the key of value to retrieve & fallback value to return if key does not exist

ex→

```
>>> picItems={'apples':'5', 'cups':'2'}
>>>  print(str(picItems.get('cups',0)))
'2'
>>> picItems = {'apples':'5', 'cups':'2'}
>>> print(str(picItems.get('orange',0)))
0
```

The setdefault() method →

This method will be called when it is required to set a value in a dictionary for a certain key only if that key doesnot already have a value.

The first argument passed to this method is the key to check for & second argument is the value to set at that key if the key does not exist. If the key does exist, the setdefault() method returns key value.

ex->

```
>>> spam={'name:'abc', 'age':'5'}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'name: abc', 'age': '5'}
>>> spam.set default('color', 'white')
'black'
>>> spam
{'color': 'black', 'name': 'abc', 'age': '5'}
```

Q3)

b) copy.copy() & copy.deepcopy() methods ->

copy.copy() method:-

This method creates a shallow copy of an object. For lists this means that a new list is created, but elements themselves are not deeply copied. If original list contains references to other objects (such as nested lists), those references are still shared between original list & copied list.

ex->

```
import copy
original_list = [[1, 2, 3], [4, 5, 6]]
shallow_list = copy.copy(original_list)
original_list[0][0] = 0
print("original List", original_list)
print("shallow copied List", shallow_list)
```

O/P ->

original list: [[0, 2, 3], [4, 5, 6]]

shallow copied list:
[[0, 2, 3], [4, 5, 6]]

copy.deepcopy() method:

This method creates a deepcopy of an object. For lists this means that a new list is created, along with new copies of all nested elements within original list.

ex→

```
import copy
original_list = [[1, 2, 3], [4, 5, 6]]
deep_list = copy.deepcopy(original_list)
original_list[0][0] = 0
print("original list:", original_list)
print("deep copied list:", deep_list)
```

O/P →

original list: [[0, 2, 3], [4, 5, 6]]

deep copied list: [[1, 2, 3], [4, 5, 6]]

Q3)

(c) append() & index() functions →

append() method is used to add new values to a list. append() method adds the argument to the end of the list

ex→

```
>>> spam = ['a', 'b', 'c']
>>> spam.append('d')
>>> spam
['a', 'b', 'c', 'd']
```

List values have an index() method that can be passed a value, and if that value exist in list, the index of value is returned. If the value isn't in list, then Python produces a ValueError.

```
>>> spam=['a', 'b', 'c', 'd']
>>> spam.index('a')
0
>>> spam.index('c')
2
```

Q4)

a) Different ways to delete an element from list →

(i) The remove() method is passed the value to be removed from list it is called on.

```
>>> spam=['a', 'b', 'c', 'd']
>>> spam.remove('b')
>>> spam
['a', 'c', 'd']
```

(ii) Using del statement:

The del statement can be used to delete an element from a list using its index.

```
>>> num=[1, 2, 3, 4, 5]
>>> del num[2]
>>> num
[1, 2, 4, 5]
```

(iii) pop() method →

The pop() method removes & returns an element from specific index in the list, if no index is provided, it removes & returns the last element.

```
>>>colors = ['red', 'green', 'blue', 'yellow']
>>> rem_color = colors.pop(1)
>>> colors
['red', 'blue', 'yellow']
```

(iv) Slicing Method →

Slicing can be used to remove a range of elements from a list.

```
>>> char = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del char[1:4]
>>> char
['a', 'e', 'f']
```

(v) Using list comprehension →

List comprehension can be used to create a new list without the elements one wants to delete

```
>>> numb=[1, 2, 3, 4, 5]
>>>elem_del = 3
>>>num = [num for num in numb if num != elem_del]
>>> numb
[1, 2, 4, 5]
```

Q4)

b)

```
number=int(input("Enter any Number"))
print("Digit \t Frequancy")
for i in range(0,10)
    count=0;
    temp=number
    while temp>0
        digit=temp%10
        if digit==i
            count=count+1
        temp=temp//10
    if count>0:
        print(i,"\t",count)
```

O/P →

```
Enter any Number
1223898078
```

Digit	Frequancy
0	2
1	1
2	2
3	1
7	1
8	3
9	1

Q4)

c) Tuples Data type →

The tuple data type is almost identical to list data type except that tuples cannot have their values modified, appended or removed.

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[0]
'hello'
>>> eggs[1] = 99
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    eggs[1]=99
TypeError: 'tuple' object does not support item assignment.
```

Q5)

a)

split() → This method is called on a string value & returns a list of strings

ex→

```
>>> 'my name is Simon'.split
['my', 'name', 'is', 'simon']
```

By default the string is split wherever whitespace characters such as space, tab, or newline characters are found.

endswith() → This method will return True if string value they are called on ends with the string value passed to method, otherwise they return False.

ex →

```
>>> "Hello world!".endswith('world!')
True
```

rjust() → This method return a padded version of string they are called on, with spaces inserted to right justify the text.

ex →

```
>>> 'Hello'.rjust(10)
'     Hello'
>>> 'Hello'.rjust(20)
'               Hello'
```

center() → The center string method centeres the text rather than justifying it to left or right.

ex →

```
>>> 'Hello'.center(20)
'       Hello        '
```

lstrip() → This method will remove whitespace characters from left end.

ex →

```
>>> spam = '    Hello World    '
>>> spam.lstrip()
'Helloworld    '
```

Q5)

b) Reading & Saving Python program variables using shelve module →

Variables can be saved in python program to binary shelf files using shelve module. This way program can restore data to variables from hard drive

The shelve module adds save & open feature to the program

```
>>> import shelve
>>> shelffile = shelve.open('mydata')
>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> shelffile['cats'] = cats
>>> shelffile.close()
```

After importing shelve module, calling shelve.open() & and passing it a file name will store the returned shelf value in a variable

In the above example cats list is created. & writing shelffile['cats'] = cats will store the list in shelf file as a value associated with key 'cats'

Q5)

c) Python program to read & print the contents of text file →

```
file_path = input("Enter the path of the text file:")

try:
    with open(file_path, 'r') as file:
        contents = file.read()
        print(contents)
except FileNotFoundError:
    print("File not found")
```

Q6)

a)

join() method → This method is called on when a list of strings that need to be joined into single string value.

ex→

```
>>> ','.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
```

startswith() → This method returns True if string value they are called on begins with string passed to method, otherwise it returns False.

ex→

```
>>> 'Hello world!'.startswith('Hello')
True
```

rjust() → This method return a padded version of string they are called on, with spaces inserted to the right-justify

(ex) →

```
>>> 'Hello'.rjust(10)
'     Hello'
```

strip() → This string method will return a new string without any whitespace characters at the beginning or end

(ex) →

```
>>> spam = '  Hello World!  '
>>> spam.strip()
'Hello World!'
```

rstrip() → This method will remove whitespace characters from right end

```
>>> spam = '  Hello World  '
>>> spam.rstrip()
'  Hello World'
```

Q6)

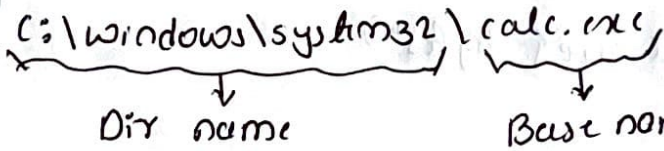
b) os.path.basename() →

Calling os.path.basename(path) will return a string of everything that comes after the last slash in path argument

ex →

```
>>> path = c:\\windows\\system32\\calc.exe
>>> os.path.basename(path)
'calc.exe'
```

The base name is the last component of path, which is typically the name of file or directory



os.path.join() → when the string values of individual file and folder name in path are passed in to os.path.join(), it will return a string with a file path using the correct path seperators

ex →

```
>>> import os
>>> os.path.join('usr', 'bin', 'spam')
'usr\\bin\\spam'
```

This function is useful in creating strings for file names

Q6)

c) Python program to find total size of all files in the given directory →

```
import os
dir_path = input("Enter the directory path: ")
if os.path.exists(dir_path) and os.path.isdir(dir_path):
    total_size = sum(entry.stat().st_size for entry in
                     os.scandir(dir_path) if entry.is_file())
    print(f"Total size of all files in {dir_path}:
          {total_size} bytes")
else:
    print("Invalid directory path")
```

Q7)

a)

Permanent delete & Safe Delete →

Permanent delete →

Using os module one can delete a single file, or single empty folder. but for a folder with all of its contents to delete one needs to use the shutil module

* calling os.unlink(path) will delete the file at path
* Calling os.rmdir(path) will delete the folder at path This folder must be empty of any files or folders.
* Calling shutil.rmtree(path) will remove the folder at path & all files & folder it contains will also be deleted.

```
>>> import os
>>> for filename in os.listdir():
        if filename.endswith('.txt')
            os.unlink(filename)
```

Safe delete →

A much better way to delete files & folders is with the third party send2trash module.

Using send2trash is much safer than python's regular delete function because it will send folders & files to computer's trash or recycle bin instead of permanently deleting them.

(ex) →

```
>>> import send2trash
>>> baconFile = open("bacon.txt", 'a')
>>> baconFile.write('bacon is not a vegetable')
25
>>> baconFile.close()
>>> send2trash.send2trash('bacon.txt')
```

Q7)

b) Backing up given folder into a zip file →

```
import zipfile, os

def backupToZip(folder):
    folder = os.path.abspath(folder)
    number = 1
    while True:
        zipFilename = os.path.basename(folder) + '_' + str(number)
                                                + '.zip'
        if not os.path.exists(zipFilename):
            break
        number = number + 1

    print('creating %s...' %(zipFilename))
    backupZip = zipfile.ZipFile(zipFilename, 'w')
    print('Done')
    for foldername, subfolders, filenames in os.walk(folder):
        backupZip.write(foldername)
        for filename in filenames:
            newbase = os.path.basename(folder) + '_'
            if filename.startswith(newBase) and filename.endswith
                                                ('.zip')
                continue
            backupZip.write(os.path.join(foldername, filename))
    backupZip.close
    print('Done')
backupToZip('C:\\Delicious')
```

Q 7)

c) Role of Assertions →

An assertion is a sanity check to make sure code isn't doing something obviously wrong. These sanity checks are performed by assert statements. If sanity check fails, then an AssertionError exception is raised.

In code an assert statement consists of

* The assert keyword
* A condition
* A comma
* A string to display when condition is false

Assertions are for programmer errors, not user errors. For errors that can be recovered from (such as file not being found or the user entering invalid data), raise an exception instead of detecting it with an assert statement.

Ex →

```
def devide(a, b):
    assert b!=0, "Division by zero is not allowed"
    return a/b
a= float(input())
b = float(input())
res = devide(a, b)
print(res)
```

Q8)

a)

(i) shutil.copytree() →

The shutil.copytree() call will copy an entire folder & every folder and file contained in it.

calling shutil.copytree(source, destination) will copy the folder at path source along with all of its files & subfolders to the folder at file path destination.

ex →

```
>>> import shutil, os
>>> os.chdir('C:\\')
>>> shutil.copytree('C:\\bacon', 'C:\\bacon_back-up')
'C:\\baconback-up
```

(ii) shutil.move() → Calling shutil.move(source, destination) will move the file or folder at path source to the path destination and will return a string of absolute path of new location.

ex →

```
>>> import shutil
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs\\bacon.txt'
```

(iii) shutil.rmtree() →

shutil.rmtree will remove the folder at path, and all files and folders it contains will also be deleted.

ex →

```
>>> import shutil
>>> shutil.rmtree('C:\\eggs')
```

Q8)

b) program →

```
import os

def list(path):

    for item in os.listdir(path):

        item_path = os.path.join(path, item)

        if os.path.isdir(item_path):

            print('Folder:', item)

            list(item_path)

        else:

            print("File:", item)

current_directory = os.getcwd()

print("Traversing directory:", current_directory)

list(current_directory)
```

Q8)

c) Logging :-

Logging is an essential aspect of software development that helps in understand the behavior of code, diagnose issues and monitor its performance

The logging module in python provides a flexible and powerful framework for generating log messages from application. It offers various logging levels, customizable formatting and ability to direct log messages to different outputs

Following are the main components and concepts of logging module →

(i) Logger → The central component that can be used to interact with logging system. One can create and configure logger instances to generate log messages.

(ii) Log levels → Python's logging module define several predefined log levels that indicate severity of log message These level include → 'DEBUG', 'INFO', 'WARNING', 'ERROR', & 'CRITICAL'.

(iii) Handlers → Handlers determine where log messages are sent

(iv) Formatters → Formatters define the format of log messages, including details such as timestamps, log levels and actual log content.

Ex →

```
import logging

logging.basicConfig(level=logging.DEBUG, format='%(asctime)s -
                    %(levelname)s - %(message)s')

logger = logging.getLogger('my-app')

logger.debug('This is debug message')
logger.infor ('This is info message')
logger.warning('This is warning message')
logger.error('This is a error message')
logger.critical('This is a critical message')
```

Q9)

a) __init__ and __str__ methods →

The_init_method → It is a special method that gets invoked when an object is initialed.

An init method for time class might look like this

```
def __init__ (self, hour=0, minut=0, second=0)
        self.hour = hour
        self.minute= minute
        self. second = second
```

It is common for parameters of __init__ to have same name as attributes. The statement self.hour=hour stores the value of parameter hour as an attribute of self.

The __str__ method :-

__str__ is a special method like __init__ that is supposed to return a string representation of an object

(ex)→

```
def __str__(self)
        return '%.2d:%.2d:%.2d' %(self.hour, self.minute,
                                  self.second)
```

When one prints an object, python invokes str method

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

Q9)

b) Prototype and Patch:-

In prototype & patch approach initially one creates prototypes of function that perform basic calculations. These prototypes are then tested and any errors encountered are patched as they are identified during testing phase.

This approach is useful when developer doesnt have deep understanding of problem and needs to iteratively refine the code. However it can lead to unnecessary complicated and error prone code due to addressing various special cases incrementally

The alternative approach is 'designed development', which relies on high level insights into major problem while prototype and patch can be effective for rapid iterative development

it can result in complex & error prone code. on the other hand designed development leverages high level insights to create a more elegant & reliable solution by utilizing known arithmatic operations

Q9)

```
c) class complex():
    def initComplex(self):
        self.realpart = int(input('Enter real part:'))
        self.imgpart = int(input("Enter imaginary part:"))
    def display(self):
        print(self.realPart, "+", self.imgpart, "i", sep="")
    def sum(self, c1, c2):
        self.realpart = c1.realPart + c2.realpart
        self.imgpart = c1.imgpart + c2.imgpart
c1=complex()
c2= complex()
c3= complex()
print('Enter first complex Number')
c1.initComplex()
print('Enter complex Number:', end ="")
c1.display()
print('Enter second complex number')
c2.initComplex()
print("second complex number:", end ="")
c2.display
print('sum of 2 complex number is", end="")
c3.sum(c1, c2)
c3 display()
```

O/P →

Enter first Complex Number

Enter the real part: 8

Enter the imaginary part: 4

First complex Number: 8+4i

Enter Second complex Number

Enter the real part: 9

Enter the imaginary part: 7

second complex Number: 9+7i

sum of two complex numbers is: 17+11i

Q10)

a)

(i) Class definition →

A class is a blueprint for creating objects. It defines the attributes and behaviors that the object of that class will have

syntax for class →

```
class ClassName:
    # class attributes & methods.
```

Ex →

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

    def introduce(self):
        print(f"my name is {self.name} and I am {self.age}
              years old.")
```

(ii) Instantiation →

Instantiation is process of creating an object (instance) of class. It involves calling the class constructor to initialize the attributes of object.

syntax →

```
obj-name = ClassName(arguments)
```

Ex →

```
person1 = Person("Alice", 25)
person2 = Person("Bob", 30)
```

(iii) Passing an instance as an Argument →

One can pass instances of class as arguments to function or methods, allowing those functions to work with attributes and behaviors of instances.

syntax →

```
def fun_name(instance)
    # code that uses instance attributes

function_name(instance_name)
```

Ex →

```
def cel_bd(person):
    person.age += 1
    print(f"Happy Birthday, {person.name}!")

cel_bd(person1)
```

(iv) Instances as Return Values →

Functions and methods can return instance of a class, allowing to create & return customized objects

syntax →

```
def fun_name(arg):
    # code block
    return instance
ret_inst = fun_name(arg)
```

Ex →

```
def Person(name, age)
    np = Person(name, age)
    return pp
np = Person("Bob", 28)
```

Q10)

b) Pure function & modifier →

Pure function →

A pure function is a type of function in programming that produces an output based solely on its input and has no side effects.

In otherwords, a pure function's return value is determined only by its arguments, and it doesnot modify any external state or variables This property makes pure fuctions predictable, easy to reason ~~out~~ about and allows for better code maintainability and testing

Ex →

```
def add(a,b)
    return(a+b)
result = add(3+5)
print(result)
```

In the above example add function is pure because it only takes input arguments and returns sum of those arguments. It doesnot modify any external state

Modifier →

A modifier is a function that not only computes an output based on its inputs but also has side effects, such as modifying global variables, modifying input arguments or interacting with external resources (like writing to files or database)

Modifier can have unpredictable behavior, make code harder to test & debug and can lead to unexpected results.

ex →

```
total = 0

def add ( a, b)
    global total
    result = a+b
    total += result
    return result

result = add (3, 5)
print (result)
print ( total)
```

o/p →

```
8
8
```

In the above example add function modifies global variable total in addition to computing and

returning the sum of input arguments. This makes it an modifier.

pure functions support parallel & concurrent programming since they dont modify shared resources. & also reduce unexpected side effects while modifiers modifiers introduce complexity & make code harder to maintain. But modifiers can be useful when one needs to change the state of a program or system.

(Prof. Naveen. S. Hiremath)

HOD
Mechanical Engineering
KLS. Vishwanathrao Deshpande
Institute of Technology
Haliyal-581329