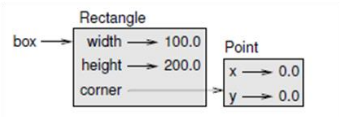
Roll No. | | | | |

Internal Assessment Test 3 – Jan. 2024

Sub:	**Introduction to Python Programming**					Sub Code:	BPLCK105B	Branch :	Chemistry Cycle
Date:	5-1-2024	Duration:	90 min's	Max Marks:	50	Sem/Sec	I / Chemistry Cycle		OBE

	Answer any FIVE FULL QUESTIONS	MARKS	CO	RBT
1 (a)	Explain the following file operations in Python with suitable example: i) Copying files and folders ii) Moving files and folders iii) Permanently deleting files and folders.	[6]	L2	CO3
(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.	[4]	L3	CO3
2 (a)	Describe logging methods used in python to categorize log messages by importance	[5]	L2	CO4
(b)	Explain five buttons available in Debug control window.	[5]	L2	CO4
3 (a)	Write a program to implement the following object diagram and its functionality as shown. Initialize an attribute through a constructor and print the same 	[5]	L3	CO4
(b)	Explain _ _init_ _ ( ) and _ _str_ _ ( ) method in detail.	[5]	L2	CO4
4 (a)	Define Pure function and Modifier function. Illustrate with an example Python program.	[5]	L3	CO4
(b)	Define a function which takes two objects representing complex numbers and returns a new complex number with an addition of two complex numbers. Define a suitable class Complex' to represent the complex number. Develop a program to read N complex numbers and to compute the addition of N complex numbers.	[5]	L3	CO4
5(a)	What is a class? How to define a class in python? How to initiate a class and how the class members are accessed?	[5]	L2	CO4
(b)	Discuss Operator overloading with an example program	[5]	L2	CO4
6 (a)	Explain Assertions with an example program of how assertions used in traffic light simulation	[5]	L2	CO3
(b)	List out the benefits of compressing files? Also explain reading of a zip file	[5]	L2	CO3
7 (a)	Explain the concept of copy.copy() and copy.deepcopy() module in class with an example object diagram.	[5]	L2	CO4
(b)	Briefly explain the printing of objects with examples.	[5]	L2	CO4

CI CCI HOD

Q.1.a

i) Copying files and folders

[Copying Files syntax and example each 1/2 mark, Copying folders syntax and example each 1/2 mark]

The shutil (or shell utilities) module has functions to let you copy, move, rename, and delete files in your Python programs. To use the shutil functions, you will first need to use import shutil.

The shutil module provides functions for copying files, as well as entire folders. Calling **shutil.copy(source, destination)** will copy the file at the path source to the folder at the path destination. (Both source and destination are strings.) If destination is a filename, it will be used as the new name of the copied file. This function returns a string of the path of the copied file.

While shutil.copy() will copy a single file, shutil.copytree() will copy an entire folder and every folder and file contained in it. Calling **shutil.copytree(source ,destination)** will copy the folder at the path source, along with all of its files and subfolders, to the folder at the path destination.

```
>>> import shutil, os
>>> os.chdir('C:\\')
>>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
'C:\\delicious\\spam.txt'
>>> shutil.copy('eggs.txt', 'C:\\delicious\\eggs2.txt')
'C:\\delicious\\eggs2.txt'
>>> shutil.copytree('C:\\bacon', 'C:\\bacon_backup')
'C:\\bacon_backup'
```

ii) Moving file & Folders

[Moving file syntax and example each 1/2 mark, Moving folder syntax and example each 1/2 mark

Calling **shutil.move(source, destination)** will move the file or folder at the path source to the path destination and will return a string of the absolute path of the new location. If destination points to a folder, the source file gets moved into destination and keeps its current filename.

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

iii) Permanently deleting file & Folders

[Permanently delete file syntax and example each 1/2 mark, Permanently delete file syntax and example each 1/2 mark]

You can delete a single file or a single empty folder with functions in the os module, whereas to delete a folder and all of its contents, you 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, and all files and folders it contains will also be deleted.

Code:

```
import os
```

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

Q. 1.b
[importing file 1 mark, code 3 marks]

```
import os
import zipfile

zf = zipfile.ZipFile("myzipfile.zip", "w")
for dirname, subdirs, files in os.walk("../Python_Programs"):
    zf.write(dirname)
    for filename in files:
        zf.write(os.path.join(dirname, filename))
zf.close()
```

Q.2.a
[each level 1 mark]

Logging levels provide a way to categorize your log messages by importance. There are five logging levels, from least to most important. Messages can be logged at each level using a different logging function.

Level	Logging Function	Description
DEBUG	`logging.debug()`	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
INFO	`logging.info()`	Used to record information on general events in your program or confirm that things are working at their point in the program.
WARNING	`logging.warning()`	Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.
ERROR	`logging.error()`	Used to record an error that caused the program to fail to do something.
CRITICAL	`logging.critical()`	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

The benefit of logging levels is that you can change what priority of logging message you want to see. Passing logging.DEBUG to the basicConfig() function's level keyword argument will show messages from all the logging levels (DEBUG being the lowest level). But after developing your program some more, you may be interested only in errors. In that case, you can set basicConfig()'s level argument to logging.ERROR. This will show only ERROR and CRITICAL messages and skip the DEBUG, INFO, and WARNING messages

Q.2.b
[Each step 1 mark]

The program will stay paused until you press one of the five buttons in the Debug Control window: **Go, Step, Over, Out, or Quit.**

Go Clicking the Go button will cause the program to execute normally until it terminates or reaches a breakpoint. If you are done debugging and want the program to continue normally, click the Go button.
Step Clicking the Step button will cause the debugger to execute the next line of code and then pause again. The Debug Control window's list of global and local variables will be updated if their values change. If the next line of code is a function call, the debugger will "step into" that function and jump to the first line of code of that function.
Over Clicking the Over button will execute the next line of code, similar to the Step button. However, if the next line of code is a function call, the Over button will "step over" the code in the function. The function's code will be executed at full speed, and the debugger will pause as soon as the function call returns.
Out Clicking the Out button will cause the debugger to execute lines of code at full speed until it returns from the current function. If you have stepped into a function call with the Step button and now simply want to keep executing instructions until you get back out, click the Out button to "step out" of the current function call.
Quit If you want to stop debugging entirely and not bother to continue executing the rest of the program, click the Quit button. The Quit button will immediately terminate the program. If you want to run your program normally again, select Debug4Debugger again to disable the debugger.

Q.3.a
[program 3 marks, initialization 1 mark, print 1 mark]

```
class Point:
     def __init__(self,x,y):
            self.x=x
            self.y=y
class Rectangle:
      def __init__(self,width,height):
            self.width=width
            self.height=height
            self.corner=Point(0.0,0.0)
def print_point(p):
       print(box.width,box.height,box.corner.x,box.corner.y)

box = Rectangle(100,200)

print_point(box)
```

```
100 200 0.0 0.0
```

Q.3.b
[init and str method each 21/2 marks]

The init method ("initialization") is a special method that gets invoked when an object is instantiated. Its full name is _ _init_ _ (two underscore characters, followed by init, and then two more underscores).

```
class Time:
    """Represents the time of day. attributes: hour, minute, second"""

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

    def print_time(t):
        print('Hour:', t.hour, '\nMinute: ', t.minute, '\nSeconds: ', t.second)

print('----------------')
time1 = Time()
time1.print_time()

print('----------------')
time2 = Time(10)
time2.print_time()

print('----------------')
time2 = Time(10,20)
time2.print_time()

print('----------------')
time2 = Time(10,20,30)
time2.print_time()
```

```
----------------
Hour: 0
Minute:  0
Seconds:  0
----------------
Hour: 10
Minute:  0
Seconds:  0
----------------
Hour: 10
Minute:  20
Seconds:  0
----------------
Hour: 10
Minute:  20
Seconds:  30
```

_ _str_ _ ( ) method

__str__ is a special method, like __init__, that is supposed to return a string representation of an object. When you print an object, Python invokes the str method

```
class Time:
    """Represents the time of day. attributes: hour, minute, second"""

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

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

print('----------------')
time1 = Time()
print(time1)

print('----------------')
time2 = Time(10, 20)
print(time2)
```

```
----------------
00:00:00
----------------
10:20:00
```

Q.4.a

[Pure function syntax and example 21/2 marks, Modifier function syntax and example 21/2 marks]

PURE FUNCTION

The function that creates a new Time object, initializes its attributes with new values and returns a reference to the new object. This is called a pure function because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

```
class Time:
    """Represents the time of day.
    attributes: hour, minute, second"""


def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum

def print_time(t):
    print('Hour:', t.hour, '\nMinute: ', t.minute, '\nSeconds: ', t.second)

start = Time()
start.hour = 9
start.minute = 45
start.second = 0

duration = Time()
duration.hour = 1
duration.minute = 35
duration.second = 0

done = add_time(start, duration)
print_time(done)
print(start.hour)
```

```
Hour: 11
Minute:  20
Seconds:  0
9
```

MODIFIER FUNCTION

Sometimes it is useful for a function to modify the objects or instances it gets as parameters. In that case, the changes are visible to the caller. Functions that work this way are called modifiers. increment() function adds a given number of seconds to a Time object or instance which is visible to the called function.

```
class Time:
    """Represents the time of day.
    attributes: hour, minute, second"""

def increment(time, seconds):
    time.second += seconds

    m = int (time.second/60)              # 5000 // 60= 83
    time.second = time.second - (m*60)    # 5000-(83*60)=20
    time.minute += m                      # 45+83 = 128

    h = int (time.minute/60)              # 128//60= 2
    time.minute = time.minute - (h*60)    # 128-2*60=128-120=8
    time.hour += h                        # 9 + 2 = 11


def print_time(t):
    print('Hour:', t.hour, '\nMinute: ', t.minute, '\nSeconds: ', t.second)

start = Time()
start.hour = 9
start.minute = 45
start.second = 0

seconds = 5000
#print_time(start)
increment(start, seconds)
print_time(start)
print(start.hour)
```

```
Hour: 11
```

```
        Minute:  8
        Seconds:  20
        11
```

Q.4.b

[complex output 2 mark, real and imaginary part each 1 mark, sum 1 mark]

```
class Complex:

    def __init__(self, tempReal, tempImaginary):
        self.real = tempReal
        self.imaginary = tempImaginary

    def addComplex(self, C1, C2):
            temp=Complex(0, 0)              # creating temporary object of class Complex
            temp.real = C1.real + C2.real
            temp.imaginary = C1.imaginary + C2.imaginary
            return temp

csum = Complex(0,0)               # csum is the final result initized to 0,0
n = int(input("Enter the number of complex numbers to be added: "))
for i in range(1, n+1):
    print("Enter real and imaginary part of a complex number %d :"%i, end=" ")
    c = input().split()
    csum = csum.addComplex(csum,Complex(int(c[0]), int(c[1])))
print("Sum of given Complex Numbers = %d + i%d"%(csum.real, csum.imaginary))
```

```
Enter the number of complex numbers to be added: 3
Enter real and imaginary part of a complex number 1 : 3 5
Enter real and imaginary part of a complex number 2 : 4 3
Enter real and imaginary part of a complex number 3 : 2 1
Sum of given Complex Numbers = 9 + i9
```

Q.5.a

[class definition 1 mark, initialization 2 marks, accessing class members 2 marks]

A programmer-defined type is called a **class**. A class definition looks like this:

class Point:

    """Represents a point in 2-D space."""

The header indicates that the new class is called Point. The body is a docstring that explains what the class is for. You can define variables and methods inside a class definition.

Defining a class named Point creates a class object.
>>> Point
<class __main__.Point>

Because Point is defined at the top level, its “full name” is __main__.Point. The class object is like a **factory for creating objects**. To create a Point, you call Point as if it were a function.
>>> blank = Point()
>>> blank

```
<__main__.Point object at 0xb7e9d3ac>
```

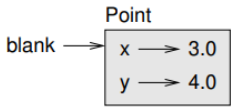
The return value is a reference to a Point object, which we assign to blank. Creating a new object is called **instantiation,** and the **object is an instance of the class**. When you print an instance, Python tells you what class it belongs to and where it is stored in **memory** (the prefix 0x means that the following

number is in hexadecimal). Every object is an instance of some class, so “object” and “instance” are interchangeable.

You can assign values to an instance using **dot notation**:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

though, we are assigning values to named elements of an object. These elements are called attributes. A state diagram that shows an object and its attributes is called an object diagram.



The variable blank refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number. You can read the value of an attribute using the same syntax

```
import math

def print_point(p):
    print('(%g, %g)' % (p.x, p.y))

def distance(q):
    d = math.sqrt((q.x)**2+(q.y)**2)
    print ('Distance is %g'% (d))

class Point:
    """Represents a point in 2-D space."""

blank = Point()
blank.x = 3.0
blank.y = 4.0

print_point(blank)
distance(blank)
```

```
(3, 4)
Distance is 5
```

Q.5.b

[Explanation 2 marks, example 3 marks]

Changing the behavior of an operator so that it works with programmer-defined types is called operator overloading. For every operator in Python there is a corresponding special method, like __add__.When you apply the + operator to Time objects, Python invokes __add__. When you print the result, Python invokes __str__.

By defining other special methods, you can specify the behavior of operators on programmer-defined types. For example, if you define a method named __add__ for the Time class, you can use the + operator on Time objects.

```
class Time:
    """Represents the time of day. attributes: hour, minute, second"""

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

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

    def __add__(self, other):
        sum = Time()
        sum.hour = self.hour + other.hour
        sum.minute = self.minute + other.minute
        sum.second = self.second + other.second

        if sum.second >= 60:
            sum.second -= 60
            sum.minute += 1

        if sum.minute >= 60:
            sum.minute -= 60
            sum.hour += 1

        return sum

start = Time(9, 45)
duration = Time(1, 35,20)
print(start + duration)
```

```
11:20:20
```

Q.6.a

[Assertion explanation 1 mark, traffic signal example 4 marks]

Assertions

An assertion is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by assert statements. If the sanity check fails, then an Assertion Error exception is raised. Assertions are for programmer errors, not user errors.

In code, an assert statement consists of the following:

- The assert keyword
- A condition (that is, an expression that evaluates to True or False)
- A comma

Using an Assertion in a Traffic Light Simulation

Say you're building a traffic light simulation program. The data structure representing the stoplights at an intersection is a dictionary keys 'ns' and 'ew', for the stoplights facing north-south and east-west, respectively. The values at these keys will be one of the strings 'green', 'yellow', or 'red'. The code would look something like this:

```
a = {'ns': 'green', 'ew': 'red'}
def switchLights(stoplight):
for key in stoplight.keys():
    if stoplight[key] == 'green':
        stoplight[key] = 'yellow'
    elif stoplight[key] == 'yellow':
        stoplight[key] = 'red'
    elif stoplight[key] == 'red':
        stoplight[key] = 'green'
switchLights(a)
```

To start the project, you want to write a switchLights() function, which will take an intersection dictionary as an argument and switch the lights. At first, you might think that

switchLights() should simply switch each light to the next color in the sequence: Any 'green' values should change to 'yellow', 'yellow' values should change to 'red', and 'red' values should change to 'green'

You may already see the problem with this code, but let's pretend you wrote the rest of the simulation code, thousands of lines long, without noticing it. When you finally do run the simulation, the program doesn't crash—but your virtual cars do! Since you've already written the rest of the program, you have no idea where the bug could be. Maybe it's in the code simulating the cars or in the code simulating the virtual drivers. It could take hours to trace the bug back to the switchLights() function. But if while writing switchLights() you had added an assertion to check that at least one of the lights is always red, you might have included the following at the bottom of the function

assert 'red' in stoplight.values(), 'Neither light is red! '

By adding this assert statement, your program would crash with this error message

Q.6.b
[Each benefit 1 mark, reading ZIP file 3 marks]

In Python Zipfile is an archive file format and a compression standard; it is a single file that holds compressed files. Compressing a file reduces its size, which is useful when transferring it over the Internet. And since a ZIP file can also contain multiple files and subfolders, it's a handy way to package several files into one. This single file, called an archive file, can then be, say, attached to an email.
Python programs can both create and open (or extract) ZIP files using functions in the zipfile module. Python Zipfile is an ideal way to group similar files and compress large files to reduce their size.

Benefits of Python Zipfiles

Bunching files into zips offer the following advantages:

1. It reduces storage requirements
Since ZIP files use compression, they can hold much more for the same amount of storage
2. It improves transfer speed over standard connections
Since it is just one file holding less storage, it transfers faster

Reading ZIP Files

To read the contents of a ZIP file, first you must create a ZipFile object (note the capital letters Z and F). To create a ZipFile object, call the zipfile.ZipFile() function, passing it a string of the .zip file's filename. Note that zipfile is the name of the Python module, and ZipFile() is the name of the function.

A ZipFile object has a namelist() method that returns a list of strings for all the files and folders contained in the ZIP file. These strings can be passed to the getinfo() ZipFile method to return a ZipInfo object about that particular file.

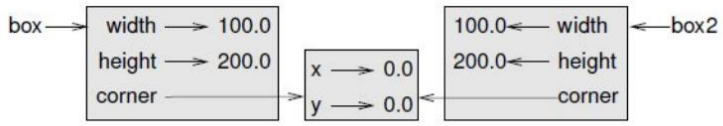
ZipInfo objects have their own attributes, such as file_size and compress_size in bytes, which hold integers of the original file size and compressed file size, respectively. While a ZipFile object represents an entire archive file, a ZipInfo object holds useful information about a single file in the archive. The command at u calculates how efficiently example.zip is compressed by dividing the original file size by the compressed file size and prints this information using a string formatted with %s.

```
>>> import zipfile, os
>>> os.chdir('C:\\')    # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
>>> exampleZip.namelist()
['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']
>>> spamInfo = exampleZip.getinfo('spam.txt')
>>> spamInfo.file_size
13908
>>> spamInfo.compress_size
3828
>>> 'Compressed file is %sx smaller!' % (round(spamInfo.file_size / spamInfo
.compress_size, 2))
'Compressed file is 3.63x smaller!'
>>> exampleZip.close()
```

Q.7.a
[copy & deep copy each 2 1/2 marks]

copy.copy( ):

The copy module contains a function called copy that can duplicate any object: If you use copy.copy to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point. This operation is called a shallow copy because it copies the object and any references it contains, but not the embedded objects.




```
import copy

class Rectangle:
    """Represents a rectangle attributes: width, height, corner."""

class Point:
    """Represents a point in 2-D space."""

box = Rectangle()

box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0

box2 = copy.copy(box)
```

```
box2 is box
```

```
False
```

```
box2.corner is box2.corner
```

```
True
```

copy.deepcopy( ):

The copy module provides a method named **deepcopy** that copies not only the object but also the objects it refers to, and the objects they refer to, and so on. This operation is called a deep copy.

```
import copy

class Rectangle:
    """Represents a rectangle attributes: width, height, corner."""

class Point:
    """Represents a point in 2-D space."""

box = Rectangle()

box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0

box3 = copy.deepcopy(box)
```

```
box3 is box
```

```
False
```

```
box3.corner is box.corner
```

```
False
```

Q.7.b
[Explanation 1 mark, code 3 marks, output 1 mark]

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values. When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of
instances.

Printing objects give us information about the objects we are working with. In Python, this can be achieved by using __repr__ or __str__ methods. __repr__ is used if we need a detailed information for debugging while __str__ is used to print a string version for the users.

Code:-

```
class Test:              # Defining a class

    def __init__(self, a, b):
            self.a = a
            self.b = b
    def __repr__(self):
            return "Test a:% s b:% s" % (self.a, self.b)
    def __str__(self):
            return "From str method of Test: a is % s,  b is % s" % (self.a, self.b)

t = Test(1234, 5678)
print(t)                 # This calls __str__()
print([t])               # This calls __repr__()
```

Output:-

From str method of Test: a is 1234, b is 5678
[Test a:1234 b:5678]