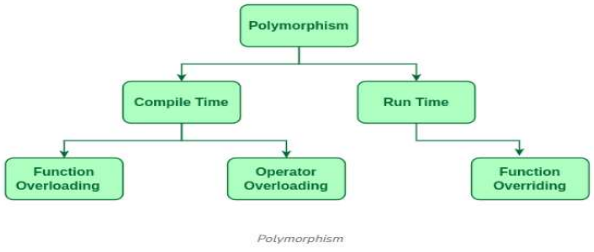
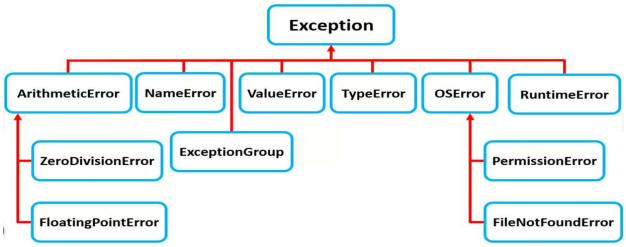
Q. No	Question (s)	Marks	BL	CO
	UNIT – 4			
1	a) Define an exception in Python. Issues occurring during execution that disrupt program flow. examples: ZeroDivisionError, Type Error, Key Error.	**1M**	L1	C224.4
	b) List the characteristics of Object Oriented Programming (OOP). Class Object Method Inheritance Polymorphism Data Abstraction Encapsulation	**1M**	L1	C224.4
	c) Identify the keywords required for handling exceptions in python. Handling Exceptions:The try...except block is used to handle exceptions in Python. Here's the syntax of try...except block: try: # code that may cause exception except: # code to run when exception occurs	**1M**	L1	C224.4
	d) Define class and object. **A class** can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. **An object** is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc.	**1M**	L1	C224.4
	e) List any four built-in exceptions in Python. **Built-in Exceptions:** Python provides many built-in exceptions like ValueError, TypeError, KeyError, IndexError, ZeroDivisionError, etc.	**1M**	L1	C224.4

2	a) Explain the need for Inheritance. The Concept of making available the properties of one class to another class is called inheritance. Inheritance is nothing but deriving a new class from existing one.	**3M**	L2	C224.4

	b) Describe how to create a custom exception in Python.
	In Python, we can define custom exceptions by creating a new class that is derived from the built-in Exception class.
	Here's the syntax to define custom exceptions,
	class CustomError(Exception):
	.....
	pass
	try:
	.....
	except CustomError:
	.....
	Here, CustomError is a user-defined error which inherits from the Exception class.
		3M	L1	C224.4

	c) Write short notes on usage of **self** in Python classes.	**3M**	L1	C224.4
	d) Explain the concept of polymorphism. **Polymorphism** is a core concept in **object-oriented programming (OOP)** that allows different types of objects to be treated as if they are objects of a common superclass. The word polymorphism comes from Greek and means "having multiple forms".  Polymorphism Here are some ways polymorphism is used in programming: **1. Code reuse:** Polymorphism allows the same lines of code to be reused multiple times.	**3M**	L1	C224.4

	2. Variable names: A single variable name can store variables of different data types, such as int, float, long, and double. **3. Class objects:** Class objects can have the same names but different behaviors. **4. Inheritance: Polymorphism** creates a relationship between classes using inheritance. **5. Method overriding:** The decision about which method implementation to execute depends on the actual object type at runtime.			
	e) Explain the process of creating classes in Python with examples. To create a class in Python, you can: 1. Use the keyword class 2. Follow the keyword with the class name and a colon 3. Indent the class's variables and methods 4. Capitalize the first letter of the class name. Here's an example of creating a class in Python: 1. Create a class named MyClass 2. Define a property named x 3. Use the class MyClass to create an object named p1 4. Print the value of x 5. Define a method named swim that prints "The shark is swimming" 6. Define a method named be_awesome that prints "The shark is being awesome" Explanation • A class is a blueprint or prototype for creating objects • The class keyword is used to start a class definition • The variables within a class are called attributes	**3M**	L2	C224.4

	• The self parameter is a reference to the current instance of the class • The __init__() method is a special method to initialize class attributes • The indented lines within a class are the class's methods Benefits of using classes in Python • You can reuse code and avoid repetition • You can encapsulate related data and behaviors in a single entity • You can abstract away the implementation details of concepts and objects You can implement a particular interface in several slightly different classes			
3	a) Describe how to handle multiple exceptions in python with an example. **Multiple Exceptions in a Single Block:** You can handle multiple exceptions in a single block by grouping them in parentheses. 	**5M**	L2	C224.4
	b) Explain Object-Oriented Programming (OOP) in in python in detail.? In Python object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of object-oriented Programming in Python is to bind the data and the	**5M**	L2	C224.4

functions that work together as a single unit so that no other part of the code can access this data. F

OOPs Concepts in Python:

Class in Python:

Create a Class
To create a class, use the keyword class:

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

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

Objects in Python:

Create Object

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)
```

Polymorphism in Python
The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

Encapsulation in Python:
Encapsulation describes the idea of wrapping data and the methods that work on data within one unit.

Inheritance in Python:
In Python object oriented Programming, Inheritance is the capability of one class to derive or inherit the properties from another class.

Data Abstraction in Python:
Data abstraction in Python is a programming concept that hides complex implementation

	details while exposing only essential information and functionalities to users.  			

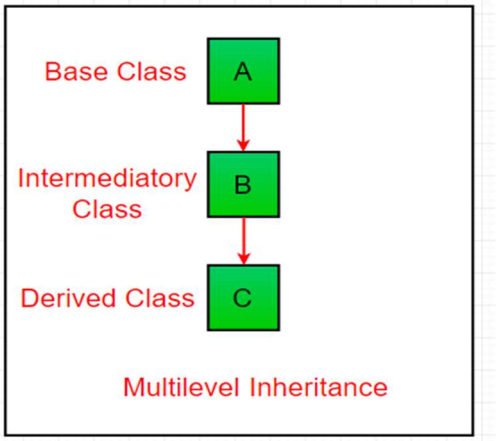
	c) Illustrate the implementation of **method overloading** in Python with an example.
	Method Overloading:
	Two or more methods have the same name but different numbers of parameters or different types of parameters, or both. These methods are called overloaded methods and this is called method overloading.
	`# Second product method`
	`# Takes three argument and print their`
	`# product`
	
	`def product(a, b, c):`
	`  p = a * b*c`
	`  print(p)`
	
	`# Uncommenting the below line shows an error`
	`# product(4, 5)`
	
	`# This line will call the second product method`
	`product(4, 5, 5)`
		5M	L1	C224.4

	d) Demonstrate implementation of multi-level inheritance in Python, with a program. **Multilevel Inheritance** in Python is a type of Inheritance in which a class inherits from a class, which itself inherits from another class. It allows a class to inherit properties and methods from **multiple parent classes**, forming a hierarchy	**5M**	L3	C224.4

similar to a family tree. It consists of two main aspects:

- **Base class:** This represents a broad concept.
- **Derived classes:** These inherit from the base class and add specific traits.

Diagram for Multilevel Inheritance in Python




```
class Base:
    # Constructor to set Data
    def __init__(self, name, roll, role):
        self.name = name
        self.roll = roll
        self.role = role

# Intermediate Class: Inherits the Base Class
class Intermediate(Base):
    # Constructor to set age
    def __init__(self, age, name, roll,
role):
        super().__init__(name, roll, role)
        self.age = age

# Derived Class: Inherits the Intermediate
Class
class Derived(Intermediate):
    # Method to Print Data
    def __init__(self, age, name, roll,
role):
        super().__init__(age, name, roll,
role)

    def Print_Data(self):
        print(f&amp;quot;The Name is :
{self.name}&amp;quot;)
        print(f&amp;quot;The Age is :
{self.age}&amp;quot;)
```

```
        print(f&quot;The role is :
{self.role}&quot;)
        print(f&quot;The Roll is :
{self.roll}&quot;)

# Creating Object of Base Class
obj = Derived(21, &quot;Lokesh Singh&quot;,
25, &quot;Software Trainer&quot;)

# Printing the data with the help of derived
class
obj.Print_Data()
```

	e) Explain in detail about **constructor** in Python with an example.
	
	In Python, a constructor is a special method that is called automatically when an object is created from a class. Its main role is to initialize the object by setting up its attributes or state.
	The method `__new__` is the constructor that creates a new instance of the class while `__init__` is the initializer that sets up the instance's attributes after creation. These methods work together to manage object creation and initialization.
	
	__new__ Method
	This method is responsible for creating a new instance of a class. It allocates memory and returns the new object. It is called before **__init__.**
	
	class ClassName:
	def __new__(cls, parameters):
	instance = super(ClassName, cls).__new__(cls)
	return instance
	
	__init__ Method
	This method initializes the newly created instance and is commonly used as a constructor in Python. It is called immediately after the object is created by **__new__** method and is responsible for initializing attributes of the instance.
		5M	L2	C224.4

	class ClassName:
	  def __init__(self, parameters):
	  self.attribute = value			

4	a) Describe how to create user defined exceptions in Python with the help of a suitable example.
	User-Defined Exception in Python
	Exceptions need to be derived from the Exception class, either directly or indirectly. Although not mandatory, most of the exceptions are named as names that end in "**Error**" similar to the naming of the standard exceptions in python. **For example,**
	
	`# A python program to create user-defined exception`
	`# class MyError is derived from super class Exception`
	`class MyError(Exception):`
	
	`  # Constructor or Initializer`
	`  def __init__(self, value):`
	`  self.value = value`
	
	`  # __str__ is to print() the value`
	`  def __str__(self):`
	`  return(repr(self.value))`
	
	`try:`
	`  raise(MyError(3*2))`
	
	`# Value of Exception is stored in error`
	`except MyError as error:`
	`  print('A New Exception occurred: ', error.value)`
		10M	L2	C224.4

	• Explain in detail about Inheritance and its types. Illustrate with one suitable example.	**10M**	L2	C224.4

Inheritance in Python:
In Python object oriented Programming, Inheritance is the capability of one class to derive or inherit the properties from another class.

Different types of Python Inheritance
There are 5 different types of inheritance in Python. They are as follows:

- **Single inheritance**: When a child class inherits from only one parent class, it is called single inheritance. We saw an example above.

```
# Python program to demonstrate
# single inheritance

# Base class
class Parent:
    def func1(self):
        print("This function is in parent
class.")

# Derived class
class Child(Parent):
    def func2(self):
        print("This function is in child
class.")
# Driver's code
object = Child()
object.func1()
object.func2()
```

- **Multiple inheritances**: When a child class inherits from multiple parent classes, it is called multiple inheritances.

```
# Python program to demonstrate
# multiple inheritance
# Base class1
class Mother:
    mothername = ""

    def mother(self):
        print(self.mothername)

# Base class2
```

```
class Father:
    fathername = ""

    def father(self):
        print(self.fathername)
# Derived class
class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)

# Driver's code
s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

- **Multilevel inheritance**: When we have a child and grandchild relationship. This means that a child class will inherit from its parent class, which in turn is inheriting from its parent class.

```
# Python program to demonstrate
# multilevel inheritance

# Base class
class Grandfather:

    def __init__(self, grandfathername):
        self.grandfathername =
grandfathername

# Intermediate class
class Father(Grandfather):
    def __init__(self, fathername,
grandfathername):
        self.fathername = fathername

        # invoking constructor of
Grandfather class
        Grandfather.__init__(self,
grandfathername)
```

```
# Derived class


class Son(Father):
    def __init__(self, sonname, fathername,
grandfathername):
        self.sonname = sonname

        # invoking constructor of Father
class
        Father.__init__(self, fathername,
grandfathername)

    def print_name(self):
        print('Grandfather name :',
self.grandfathername)
        print("Father name :",
self.fathername)
        print("Son name :", self.sonname)


#  Driver code
s1 = Son('Prince', 'Rampal', 'Lal mani')
print(s1.grandfathername)
s1.print_name()
```

- **Hierarchical inheritance** More than one derived class can be created from a single base.

```
# Python program to demonstrate
# Hierarchical inheritance


# Base class
class Parent:
    def func1(self):
        print("This function is in parent
class.")

# Derived class1


class Child1(Parent):
    def func2(self):
```

```
        print("This function is in child
1.")

# Derivied class2


class Child2(Parent):
    def func3(self):
        print("This function is in child
2.")


# Driver's code
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

- **Hybrid inheritance**: This form combines more than one form of inheritance. Basically, it is a blend of more than one type of inheritance.

```
# Python program to demonstrate
# hybrid inheritance

class School:
    def func1(self):
        print("This function is in
school.")

class Student1(School):
    def func2(self):
        print("This function is in student
1. ")

class Student2(School):
    def func3(self):
        print("This function is in student
2.")

class Student3(Student1, School):
    def func4(self):
```

```
        print("This function is in student
3.")

# Driver's code
object = Student3()
object.func1()
object.func2()
```

b) Write brief notes on python exception handling using try, except, raise and finally statements with an example. **10M** L2 C224.4

- The **try** block contains the code that may raise an exception.
- The **except** block handles the exception if one occurs.
- The **finally** block contains code that should always be executed, whether an exception occurred or not.

Here's an example:

```
try:
    # Some code that may raise an
exception
    x = 10 / 0
except ZeroDivisionError:
    # Handling specific exception
    print("Division by zero error!")
finally:
    # Code that always executes,
regardless of whether an exception
occurred or not
    print("Finally block executed")
```

Python Raise Keyword

Python raise Keyword is used to raise exceptions or errors. The raise keyword raises an error and stops the control flow of the program. It is used to bring up the current exception in an exception handler so that it can be handled further up the call stack.

Python Raise Syntax

```
raise {name_of_ the_ exception_class}
```

```
a = 5

if a % 2 != 0:
    raise Exception("The number shouldn't
be an odd integer")
```

Output:

```
Traceback (most recent call last):
  File "/home/3dd59d932258303ca09ee7c2e500e499.py", line 4, in <module>
    raise Exception("The number shouldn't be an odd integer")
Exception: The number shouldn't be an odd integer
```

(OR)

An Exception is an Unexpected Event, which occurs during the execution of the program. It is also known as a **run time error**. When that error occurs, Python generates an exception during the execution and that can be handled, which prevents your program from interrupting.

Example: In this code, The system can not divide the number with zero so an exception is raised.

```
a = 5
b = 0
print(a/b)
```

Output

```
Traceback (most recent call last):
  File
"/home/8a10be6ca075391a8b174e0987a3e7f5.py
", line 3, in <module>
    print(a/b)
ZeroDivisionError: division by zero
```

Exception handling with try, except, else, and finally

- **Try**: This block will test the excepted error to occur
- **Except**: Here you can handle the error
- **Else**: If there is no exception then this block will be executed
- **Finally**: Finally block always gets executed either exception is generated or not

Python Try, Except, else and Finally Syntax

```
try:
       # Some Code....
except:
       # optional block
       # Handling of exception (if
required)
else:
       # execute if no exception
finally:
      # Some code .....(always executed)
```

Working of 'try' and 'except'

Let's first understand how the Python try and except works

- First **try** clause is executed i.e. the code between **try** and **except** clause.
- If there is no exception, then only **try** clause will run, **except** clause will not get executed.
- If any exception occurs, the **try** clause will be skipped and **except** clause will run.
- If any exception occurs, but the **except** clause within the code doesn't handle it, it is passed on to the outer **try** statements. If the exception is left unhandled, then the execution stops.
- A **try** statement can have more than one **except** clause.

```
# Python code to illustrate working of try()
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional
        # Part as Answer
        result = x // y
```

```
        print("Yeah ! Your answer is :", result)
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero
")

# Look at parameters and note the working
of Program
divide(3, 2)
divide(3, 0)
```

Catch Multiple Exceptions in Python
Here's an example that demonstrates how to use multiple except clauses to handle different exceptions:

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
except ZeroDivisionError:
    print("You cannot divide by zero.")
except ValueError:
    print("Invalid input. Please enter a valid
number.")
except Exception as e:
    print(f"An error occurred: {e}")
```

Python finally Keyword

Python provides a keyword finally, which is **always executed** after try and except blocks. The finally block always executes after normal termination of try block or after try block terminates due to some exception. Even if you return in the except block still the finally block will execute
Example: Let's try to throw the exception in except block and Finally will execute either exception will generate or not.

```
# Python code to illustrate

# working of try()
```

```
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional
        # Part as Answer
        result = x // y
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero ")
    else:
        print("Yeah ! Your answer is :", result)
    finally:
        # this block is always executed
        # regardless of exception generation.
        print('This is always executed')


# Look at parameters and note the working of Program
divide(3, 2)
divide(3, 0)
```