UNIT - 4

1-mark

a) **Define an exception in Python.**

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

b) **List the characteristics of Object Oriented Programming (OOP).**

OOP characteristics include encapsulation, inheritance, polymorphism, and abstraction.

c) **Identify the keywords required for handling exceptions in python.**

The keywords are try, except, finally, and raise.

d) **Define class and object.**

A class is a blueprint for creating objects, while an object is an instance of a class.

e) **List any four built-in exceptions in Python.**

TypeError, ValueError, IndexError, and KeyError.

3 MARK ANSWERS

a) **Explain the need for Inheritance.**

Inheritance allows a new class (subclass) to inherit attributes and methods from an existing class (superclass). This promotes code reusability, reduces redundancy, and establishes an "is-a" relationship between classes, improving code organization and maintainability.

b) **Describe how to create a custom exception in Python.**

To create a custom exception, define a new class that inherits from the built-in `Exception` class or one of its subclasses. You can add custom attributes or methods to the exception class. Then, use the `raise` keyword to trigger the custom exception when needed.

Example:

```
class MyCustomError(Exception):
    def __init__(self, message):
        super().__init__(message)
```

```
raise MyCustomError("This is my custom exception.")
```

c) **Write short notes on usage of self in Python classes.**

`self` is a reference to the instance of the class. It's used to access variables that belong to the class. When a method is called on an object, the object itself is automatically passed as the first argument, which is conventionally named `self`. This allows methods to access and modify the object's attributes.

d) **Explain the concept of polymorphism.**

Polymorphism means "many forms." In OOP, it allows objects of different classes to respond to the same method call in their own way. This is achieved through method overriding or method overloading (though Python does not support traditional method overloading). It enables writing code that can work with objects of different types without needing to know their specific classes.

e) **Explain the process of creating classes in Python with examples.**

To create a class, use the `class` keyword followed by the class name. Inside the class, define attributes and methods using the `def` keyword. The `__init__` method is a special method used for object initialization.

Example:

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed
    def bark(self):
        print("Woof!")
my_dog = Dog("Buddy", "Golden Retriever")
my_dog.bark()
```

5-mark

a) **Describe how to handle multiple exceptions in python with an example.**

```
try:
    num1 = int(input("Enter a numerator: "))
    num2 = int(input("Enter a denominator: "))
    result = num1 / num2
    my_list = [1, 2, 3]
    print(my_list[num1]) # potential IndexError
except ValueError:
    print("Invalid input. Please enter integers.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
except IndexError:
    print("Index out of range")
except Exception as e: #catch any other exception
    print(f"An unexpected error occurred: {e}")
else:
    print(f"Result: {result}")
finally:
    print("Execution complete.")
```

Explanation: The try block contains code that might raise exceptions. Multiple except blocks handle specific

exceptions. The else block executes if no exceptions occur. The finally block always executes.

b) **Explain Object-Oriented Programming (OOP) in detail.**

OOP is a programming paradigm that organizes code around objects, which are instances of classes. Key concepts include:

- **Encapsulation:** Bundling data (attributes) and methods that operate on the data into a single unit (class). This hides internal details and protects data.
- **Inheritance:** Allows a class (subclass) to inherit properties and methods from another class (superclass), promoting code reuse and establishing relationships.
- **Polymorphism:** Enables objects of different classes to respond to the same method call in their own way, providing flexibility and extensibility.
- **Abstraction:** Hiding complex implementation details and showing only essential features, simplifying code and improving maintainability.
- **Objects:** Instances of classes, representing real-world entities with attributes and behaviors.
- **Classes:** Blueprints for creating objects, defining the structure and behavior of objects.

c) **Illustrate the implementation of method overloading in Python with an example.**

```
class Calculator:
    def add(self, a, b=None, c=None):
        if b is not None and c is not None:
            return a + b + c
        elif b is not None:
            return a + b
        else:
            return a

calc = Calculator()
print(calc.add(5))
print(calc.add(5, 10))
print(calc.add(5, 10, 15))
```

Explanation: Python doesn't support traditional method overloading. We simulate it by using default arguments and checking if they are provided.

d) **Demonstrate implementation of multi-level inheritance in Python, with a program.**

```
class Grandfather:
```

```
    def grandfather_method(self):
        print("Grandfather's method")

class Father(Grandfather):
    def father_method(self):
        print("Father's method")

class Child(Father):
    def child_method(self):
        print("Child's method")

child_obj = Child()
child_obj.grandfather_method()
child_obj.father_method()
child_obj.child_method()
```

Explanation: Child inherits from Father, which inherits from Grandfather, forming a multi-level inheritance hierarchy.

e) **Explain in detail about constructor in Python with an example.**

A constructor is a special method used to initialize objects when they are created. In Python, the constructor is named `__init__`. It's automatically called when an object is instantiated.

Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def display(self):
        print(f"Name: {self.name}, Age: {self.age}")


person1 = Person("Alice", 30)
person1.display()
```

Explanation: The __init__ method takes name and age as parameters and initializes the object's attributes. When person1 is created, __init__ is called, and the object's attributes are set.

10-mark

a) **Describe how to create user-defined exceptions in Python with the help of a suitable example.**

Python

```
class InsufficientFundsError(Exception):
    """Custom exception raised when an account has insufficient funds."""
    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount
        super().__init__(f"Insufficient funds: Balance={balance}, Attempted withdrawal={amount}")

class BankAccount:
    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        if amount > self.balance:
            raise InsufficientFundsError(self.balance, amount)
        self.balance -= amount
        print(f"Withdrawal successful. New balance: {self.balance}")
```

```
# Example usage
account = BankAccount(1000)
try:
    account.withdraw(1500)
except InsufficientFundsError as e:
    print(f"Error: {e}")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
else:
    print("Transaction completed successfully.")
finally:
    print("Transaction processing finished.")
```

Explanation:

1. **Custom Exception Class:**
    - We define a class InsufficientFundsError that inherits from the built-in Exception class.
    - The __init__ method initializes the exception with relevant information (balance and attempted withdrawal amount).

- We use super().__init__ to call the base class constructor and set the error message.

2. **BankAccount Class:**
   - The BankAccount class has a withdraw method that checks if there are sufficient funds.
   - If the withdrawal amount exceeds the balance, it raises the InsufficientFundsError using the raise keyword.
3. **Exception Handling:**
   - The try block contains the code that might raise the custom exception.
   - The except InsufficientFundsError block handles the specific custom exception.
   - The except Exception as e block handles any other unexpected errors.
   - The else block executes when no exceptions are raised.
   - The finally block executes regardless of whether an exception occurred.

b) **Explain in detail about Inheritance and its types. Illustrate with one suitable example.**

Inheritance:

- Inheritance is a fundamental OOP concept that allows a class (subclass or derived class) to inherit properties[1] and methods from another class (superclass or base class).
- It[2] promotes code reuse, reduces redundancy, and establishes an "is-a" relationship between classes.

Types of Inheritance:

1. **Single Inheritance:**
    - A subclass inherits from a single superclass.
2. **Multiple Inheritance:**
    - A subclass[3] inherits from multiple superclasses.[4]
3. **Multi-level Inheritance:**
    - A subclass inherits from another subclass, forming a chain of inheritance.
4. **Hierarchical Inheritance:**
    - Multiple subclasses inherit from a single superclass.[5]
5. **Hybrid Inheritance:**
    - A combination of two or more types of inheritance.

Example (Hierarchical Inheritance):

Python

```
class Animal:
    def eat(self):
        print("Animal is eating")

class Dog(Animal):
    def bark(self):
        print("Dog is barking")

class Cat(Animal):
    def meow(self):
        print("Cat is meowing")

dog = Dog()
cat = Cat()

dog.eat()
dog.bark()

cat.eat()
```

cat.meow()

Explanation:

- Animal is the superclass.
- Dog and Cat are subclasses that inherit from Animal.
- Both Dog and Cat inherit the eat method from Animal.

c) **Write brief notes on python exception handling using try, except, raise and finally statements with an example.**

Exception Handling:

- Exception handling allows you to gracefully handle errors that occur during program execution.

Keywords:

1. **try:**
    - Encloses the code that might raise an exception.
2. **except:**
    - Handles specific exceptions that are raised in the try block.
3. **raise:**
    - Explicitly raises an exception.

4. **finally:**
    - Executes code that always runs, regardless of whether an exception occurred.

Example:

Python

```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
        result = None
    except TypeError:
        print("Error: Invalid input types.")
        result = None
    else:
        print("Division successful.")
    finally:
        print("Division operation completed.")
    return result
```

```
print(divide(10, 2))

print(divide(10, 0))

print(divide(10, "string"))
```

Explanation:

- The try block attempts to divide a by b.
- The except ZeroDivisionError block handles division by zero.
- The except TypeError block handles invalid datatypes.
- The else block executes when no error occurs.
- The finally block always executes.
- The function returns the result, or None if an error occurs.