FPP PART-B

SET-1:

2) a. 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.

1. Default Arguments

Python Copy code

```python
class Example:
    def show(self, a=None, b=None):
        if a is not None and b is not None:
            print(f"a: {a}, b: {b}")
        elif a is not None:
            print(f"a: {a}")
        else:
            print("No arguments")

obj = Example()
obj.show()
obj.show(10)
obj.show(10, 20)
```

2. Variable-Length Arguments (*args, **kwargs)

Python Copy code

```python
class Example:
    def show(self, *args):
        for arg in args:
            print(arg)

obj = Example()
obj.show(1)
obj.show(1, 2, 3)
```

2) b. Inheritance and types

🔗 Types of Inheritance in Python

Type	Description
Single Inheritance	One child class inherits from one parent class.
Multiple Inheritance	One child class inherits from **multiple** parent classes.
Multilevel Inheritance	Inheritance chain: child → parent → grandparent.
Hierarchical Inheritance	Multiple child classes inherit from a **single** parent class.
Hybrid Inheritance	Combination of two or more types of inheritance.

PROGRAM:

```
# ---------- SINGLE INHERITANCE ----------
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):  # Inherits from Animal
    def bark(self):
        print("Dog barks")

# ---------- MULTILEVEL INHERITANCE ----------
class Puppy(Dog):  # Inherits from Dog -> Dog inherits from Animal
    def weep(self):
        print("Puppy weeps")

# ---------- MULTIPLE INHERITANCE ----------
class Father:
    def skills(self):
        print("Knows gardening")

class Mother:
    def skills(self):
        print("Knows cooking")

class Child(Father, Mother):  # Inherits from both Father and Mother
    def play(self):
        print("Child plays")

# ---------- HIERARCHICAL INHERITANCE ----------
class Bird:
    def fly(self):
        print("Bird can fly")

class Sparrow(Bird):  # Inherits from Bird
    def chirp(self):
        print("Sparrow chirps")

class Eagle(Bird):  # Inherits from Bird
    def hunt(self):
        print("Eagle hunts")
```

```
# ---------- OUTPUT SECTION ----------

# Single and Multilevel Inheritance
print("----- Single and Multilevel Inheritance -----")
p = Puppy()
p.speak()   # From Animal
p.bark()    # From Dog
p.weep()    # From Puppy

# Multiple Inheritance
print("\n----- Multiple Inheritance -----")
c = Child()
c.skills()  # From Father (due to Method Resolution Order - MRO)
c.play()

# Hierarchical Inheritance
print("\n----- Hierarchical Inheritance -----")
s = Sparrow()
s.fly()
s.chirp()

e = Eagle()
e.fly()
e.hunt()
```

OUTPUT:

```
----- Single and Multilevel Inheritance -----
Animal speaks
Dog barks
Puppy weeps

----- Multiple Inheritance -----
Knows gardening
Child plays

----- Hierarchical Inheritance -----
Bird can fly
Sparrow chirps
Bird can fly
Eagle hunts
```

4) File handling functions:

i) open(): To open a file and return a file object.

Syntax: file = open("filename", "mode")

ii) write(): To write content to a file.

Syntax: file.write("Your content here")

iii) read(): To read the contents of a file.

Syntax: data = file.read()

iv) tell(): Returns the **current position** (in bytes) of the file pointer.

Syntax: position = file.tell()

v) close(): To close the file and free up system resources.

Syntax: file.close()

```
# Writing to a file
file = open("demo.txt", "w")
file.write("Welcome to file handling!")
file.close()

# Reading from the file and using tell()
file = open("demo.txt", "r")
print("File content:", file.read())
file.seek(0)
print("Current file pointer position:", file.tell())
file.close()
```

7) Program to verify email:

```
import re  # Regular Expression Library

def is_valid_email(email):
    # Improved pattern to require proper domain and TLD
    pattern = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z]{2,}$'
    return bool(re.match(pattern, email))

# Input from the user
email_address = input("Enter a string to check if it's a valid email: ")

# Check and display the result
if is_valid_email(email_address):
    print(f"{email_address} is a valid email format.")
else:
    print(f"{email_address} is NOT a valid email format.")
```

SET-2:

3) a. Key features of OOP

🌟 Four Main Principles of OOP in Python

Principle	Description
1. **Encapsulation**	Wrapping data and methods into a single unit (class).
2. **Abstraction**	Hiding internal details and showing only necessary parts.
3. **Inheritance**	One class inherits properties/methods from another.
4. **Polymorphism**	Same function or operator behaves differently based on context.

1 Encapsulation

Wraps variables and methods inside a class.

python

```python
class Student:
    def __init__(self, name, age):
        self.name = name        # public
        self.__age = age        # private (by convention using __)

    def display(self):
        print(f"Name: {self.name}, Age: {self.__age}")

s = Student("Rushitha", 20)
s.display()
# print(s.__age)  # ❌ Will raise an AttributeError
```

2 Abstraction

Hides complex logic and shows only essentials using methods or abstract classes.

python

```python
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start(self):
        pass

class Car(Vehicle):
    def start(self):
        print("Car engine started")

c = Car()
c.start()
```

3 Inheritance

Allows a class to use features of another class.

python

```python
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

d = Dog()
d.speak()
d.bark()
```

4 Polymorphism

Same method name behaves differently based on object.

python

```python
class Bird:
    def sound(self):
        print("Some generic bird sound")

class Parrot(Bird):
    def sound(self):
        print("Parrot says: Hello!")

class Sparrow(Bird):
    def sound(self):
        print("Sparrow chirps")

def make_sound(bird):
    bird.sound()

make_sound(Parrot())
make_sound(Sparrow())
```

3) b. Multiple inheritance

```python
# First parent class
class Father:
    def gardening(self):
        print("Father loves gardening.")

# Second parent class
class Mother:
    def cooking(self):
        print("Mother loves cooking.")

# Child class inherits from both Father and Mother
class Child(Father, Mother):
    def sports(self):
        print("Child loves playing sports.")

# Create an object of Child
c = Child()

# Accessing methods from both parents and the child
c.gardening()  # From Father
c.cooking()    # From Mother
c.sports()     # From Child
```

5. file functions

Same answer from set 1
(Add this to the answer)

v) seek(): Used to **move the file pointer** to a specific position.

Syntax: file.seek(position)

```
file = open("sample.txt", "r")
file.seek(0)  # Move to beginning
print(file.read(5))  # Read 5 characters from start
file.close()
```

7) Special symbols

In Python's `re` (regular expression) module, **special symbols** (also called metacharacters) are used to define search patterns. Here's a table with the **most important special symbols** and their roles, along with examples:

No.	Symbol	Meaning	Example	Output/Explanation
1	`.`	Matches any character except newline	`re.findall("a.c", "abc aac acc")`	`['abc', 'aac', 'acc']`
2	`^`	Matches start of string	`re.findall("^Hi", "Hi there")`	`['Hi']`
3	`$`	Matches end of string	`re.findall("end$", "the end")`	`['end']`
4	`*`	0 or more repetitions	`re.findall("a*", "aaab")`	`['aaa', '', '']`
5	`+`	1 or more repetitions	`re.findall("a+", "aaab")`	`['aaa']`
6	`?`	0 or 1 repetition	`re.findall("a?", "aaab")`	`['a', 'a', 'a', '']`
7	`{n}`	Exactly n repetitions	`re.findall("a{2}", "aaab")`	`['aa']`
8	`[]`	Set of characters	`re.findall("[ae]", "apple")`	`['a', 'e']`
9	` ` `	` ` `	OR operator	`re.findall("cat
10	`()`	Grouping	`re.findall("(ab)+", "abab")`	`['ab']` (first match of grouped pattern)
11	`\`	Escape character	`re.findall("\.", "a.b")`	`['.']`

These symbols allow flexible and powerful pattern matching for tasks like validation, extraction, or replacement in strings.

SET- 3:

2) User-defined exceptions

python Copy Edit

```python
# Step 1: Define a custom exception with a message
class NegativeNumberError(Exception):
    """Custom exception raised when a negative number is encountered."""
    def __init__(self, message):
        super().__init__(message)

# Step 2: Function to check number validity
def check_number(n):
    if n < 0:
        # Step 3: Raise custom exception if number is negative
        raise NegativeNumberError("Negative numbers are not allowed")
    else:
        print("Number is valid")

# Step 4: Use try-except to handle the exception
try:
    check_number(-5)
except NegativeNumberError as e:
    print("Error:", e)
```

Output:

javascript Copy Edit

```
Error: Negative numbers are not allowed
```

4) a. File operations

Same answer from set 1

4) b. program to copy contents

```
# Open the source file in read mode
source_file = open("source.txt", "r")

# Open the destination file in write mode
destination_file = open("destination.txt", "w")

# Read from source and write to destination
for line in source_file:
    destination_file.write(line)

# Close both files
source_file.close()
destination_file.close()

print("File copied successfully!")
```

6) program to verify email

Same answer from set 1