EXPERIMENT NO: 11

Title: Implement a python program to demonstrate Method Overloading and Method Overriding

Prior Concepts: Basic Python expressions, Functions and classes in Python, Object-Oriented Programming concepts (OOP)

Objective: To understand and demonstrate method overloading and method overriding in Python using classes and inheritance.

Theory:

What is Inheritance?

Inheritance is an OOP concept that allows a class (called a child class or subclass) to inherit attributes and methods from another class (called the parent class or superclass). This makes code more modular, reusable, and organized.

With inheritance, you can create specialized classes based on more general classes, avoiding redundancy and enabling better organization of shared behaviors and properties.

Syntax:

```
class ParentClass:
    # Parent class attributes and methods

class ChildClass(ParentClass):
    # Child class attributes and methods
```

What is Method Overloading?

Method overloading is the ability to define multiple methods with the same name but different numbers or types of parameters. While many languages like Java and C++ directly support method overloading, Python does not support it in the traditional sense. However, you can achieve similar functionality by using default values for parameters or *args and **kwargs to handle varying arguments.

Why use Method Overloading?

- Provides flexibility in method usage.
- Allows methods to behave differently based on arguments provided.

What is Method Overriding?

Method overriding occurs when a subclass provides its own version of a method that is already defined in the parent class. This allows a subclass to implement or extend the behavior of an inherited method to suit its own needs.

Why Use Method Overriding?

- Allows specific behavior in subclasses.
- Provides flexibility to modify inherited methods without altering the parent class.
- Enables polymorphism, where objects of different classes can be treated as instances of the same class and behave differently.

Procedure:

Step 1: Setting Up the Environment

1. Open your Python IDE or text editor.
2. Create a new Python file (your_script_name.py) to write the code.

Step 2: Writing the Python Code

1. **Example of Method Overloading Using Default Arguments and *args**

```
class MathOperations:
    # Method to add numbers
    def add(self, a, b=0, c=0):
        return a + b + c

# Testing method overloading
math_op = MathOperations()
print("Addition of two numbers:", math_op.add(5, 10))
print("Addition of three numbers:", math_op.add(5, 10, 15))
```

2. **Example of Method Overloading Using *args**

```
class MathOperations:
    def add(self, *args):
        return sum(args)

# Testing method overloading with varying arguments
math_op = MathOperations()
print("Addition of two numbers:", math_op.add(5, 10))
print("Addition of three numbers:", math_op.add(5, 10, 15))
print("Addition of four numbers:", math_op.add(5, 10, 15, 20))
```

3. **Example of Method Overriding:**

```
class Animal:
    def sound(self):
        return "Animal makes sound"
```

```
class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

# Testing method overriding
animal = Animal()
dog = Dog()
cat = Cat()

print("Animal sound:", animal.sound())
print("Dog sound:", dog.sound())
print("Cat sound:", cat.sound())
```

Step 3: Running the Python Script

- Save your Python file.
- Run the script in your IDE or from the command line using:
  python method_overloading_overriding.py.
- Observe the output generated from each operation to verify the functionality of method overloading and overriding.

Conclusion: Thus, we successfully demonstrated method overloading in Python using default arguments and *args, and method overriding using inheritance in subclasses.

Questions:

1. Explain why Python does not directly support method overloading and how it can be achieved in Python.
2. How does method overriding differ from method overloading, and why is it useful in OOP?
3. Write a class Vehicle with a method move() that prints "Vehicle is moving." Override this method in subclasses Car and Bike to print "Car is moving on road" and "Bike is moving on track," respectively.

EXPERIMENT NO: 12

Title: Write a python program to handle different types of exceptions using try-except blocks, multiple except blocks and finally block.

Prior Concepts: Basic syntax of Python, Conditional statements and loops, Basic understanding of errors and exceptions in programming.

Objective: To understand how to handle different types of exceptions in Python and use try-except, multiple except blocks, and the finally block for error handling.

Theory:

Exceptions are errors that occur during the execution of a program. Python provides a way to handle these exceptions using try-except blocks, which help prevent the program from crashing when unexpected errors arise.

Try-Except Block:

- A try block contains code that may throw an exception.
- The except block handles exceptions if any occur, preventing the program from crashing.

Multiple Except Blocks:

Python allows multiple except blocks to handle different types of exceptions. Each except block can be specified for a particular exception type, allowing precise control over error handling.

Finally Block:

A finally block is used to execute code that should run regardless of whether an exception was raised. It is often used for cleanup tasks like closing files or releasing resources.

Common Exceptions in Python:

SyntaxError: Occurs when there's an error in Python syntax.

TypeError: Raised when an operation or function is applied to an object of inappropriate type.

ValueError: Occurs when a function receives an argument of the correct type but inappropriate value.

ZeroDivisionError: Raised when a division by zero is attempted.

FileNotFoundError: Occurs when trying to open a file that doesn't exist.

IndexError: Happens when trying to access an index that's out of the bounds of a list.

KeyError: Raised when trying to access a dictionary key that doesn't exist.

Procedure:

Step 1: Setting Up the Environment

- Open your Python IDE or text editor.
- Create a new Python file (e.g., exception_handling.py) to write the code.

Step 2: Writing the Python Code

The following program demonstrates exception handling using try-except, multiple except blocks, and the finally block:

```
def divide_numbers(num1, num2):
    try:
        result = num1 / num2
        print("Result:", result)
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
    except TypeError:
        print("Error: Both inputs must be numbers.")
    finally:
        print("Division operation complete.\n")

def open_file(filename):
    try:
        with open(filename, 'r') as file:
            print("File content:", file.read())
    except FileNotFoundError:
        print("Error: The file was not found.")
    finally:
        print("File operation complete.\n")

# Demonstrating exception handling
# Example 1: Handling ZeroDivisionError and TypeError
print("Example 1:")
divide_numbers(10, 2)          # Normal case
divide_numbers(10, 0)          # Will raise ZeroDivisionError
divide_numbers("10", 2)        # Will raise TypeError

# Example 2: Handling FileNotFoundError
print("Example 2:")
open_file("example.txt")       # Will raise FileNotFoundError (if file
does not exist)
open_file("existing_file.txt") # Normal case (if file exists)
```

Step 3: Running the Python Script

- Save your Python file.
- Run the script in your IDE or from the command line using: python exception_handling.py.

- Observe the outputs generated from each operation to verify that exceptions are handled correctly and that the finally blocks are executed.

Conclusion: Thus, we successfully demonstrated the handling of different types of exceptions in Python using try-except, multiple except blocks for specific errors, and a finally block to ensure the execution of final statements regardless of exceptions.

Questions:

1. Explain the purpose of using multiple except blocks and give an example of when it would be useful.
2. What is the role of the finally block, and how does it differ from the except block?
3. Modify the divide_numbers function to handle ValueError if non-numeric values are entered as inputs, in addition to the existing exception handling.

EXPERIMENT NO: 13

Title: Implement python program load a CSV file into a Pandas DataFrame and perform operations.

Prior Concepts: Basic Python programming, familiarity with data structures like lists and dictionaries, basic file handling in Python.

Objective: To load a CSV file into a Pandas DataFrame and perform data manipulation operations on the loaded DataFrame.

Theory:

Pandas is a powerful Python library for data manipulation and analysis. It provides data structures such as DataFrames and Series, which make it easy to work with structured data, like CSV files. A DataFrame is a two-dimensional, size-mutable, and labeled data structure with columns of potentially different types, making it ideal for data manipulation and analysis.

CSV (Comma-Separated Values) files are a popular format for storing and exchanging data. Pandas provides functions like pd.read_csv() to load a CSV file into a DataFrame easily. Once loaded, various operations can be performed on the DataFrame, such as:

- Viewing Data: Displaying a snapshot of the data using methods like head() and tail().
- Filtering: Selecting specific rows or columns based on conditions.
- Sorting: Organizing data in ascending or descending order.
- Summarizing: Calculating summary statistics like mean, median, and standard deviation.

Procedure:

Step 1: Setting Up the Environment

- Open your Python IDE or text editor.
- Create a new Python file, e.g., data_operations.py.

Step 2: Writing the Python Code

The code below demonstrates loading a CSV file, viewing data, and performing basic operations.

```
import pandas as pd

# Load the CSV file into a DataFrame
df = pd.read_csv('employee_data.csv')

# Display the first few rows of the DataFrame
print("First 5 rows of the data:")
print(df.head())

# Basic Information about the DataFrame
print("\nData Information:")
print(df.info())

# Statistical summary of the DataFrame
print("\nStatistical Summary:")
print(df.describe())

# Sorting data based on the 'salary' column in descending order
sorted_df = df.sort_values(by='salary', ascending=False)
print("\nData Sorted by Salary (Descending):")
print(sorted_df)

# Selecting specific columns: 'employee_name' and 'department'
selected_columns = df[['employee_name', 'department']]
print("\nSelected Columns - Employee Name and Department:")
print(selected_columns)

# Grouping data by 'department' and calculating the mean for each group
grouped_data = df.groupby('department').mean()
print("\nMean Salary and Age by Department:")
print(grouped_data)

# Check for missing values in each column
print("\nMissing Values in Each Column:")
print(df.isnull().sum())

# Fill missing values with 0
df_filled = df.fillna(0)
print("\nData after Filling Missing Values with 0:")
print(df_filled)

# Drop rows with any missing values
df_dropped = df.dropna()
print("\nData after Dropping Rows with Missing Values:")
print(df_dropped)

# Rename columns: changing 'employee_name' to 'name'
```

```
df_renamed = df.rename(columns={'employee_name': 'name'})
print("\nData with Renamed Column (employee_name to name):")
print(df_renamed.head())

# Convert 'join_date' column to datetime
df['join_date'] = pd.to_datetime(df['join_date'])
print("\nData with 'join_date' as datetime:")
print(df.head())

# Extract year, month, and day from 'join_date'
df['join_year'] = df['join_date'].dt.year
df['join_month'] = df['join_date'].dt.month
df['join_day'] = df['join_date'].dt.day
print("\nExtracted Year, Month, Day from 'join_date':")
print(df[['join_date', 'join_year', 'join_month', 'join_day']].head())

# Convert all strings in 'department' column to uppercase
df['department'] = df['department'].str.upper()
print("\nDepartment Column in Uppercase:")
print(df[['department']].head())

# Select rows where salary is greater than 60000
high_salary_df = df.loc[df['salary'] > 60000]
print("\nEmployees with Salary Greater than 60000:")
print(high_salary_df)

# Select specific rows and columns using iloc
subset_df = df.iloc[1:5, 0:3]
print("\nSubset of Rows and Columns (First 4 rows, first 3 columns):")
print(subset_df)

# Remove duplicate rows
df_no_duplicates = df.drop_duplicates()
print("\nData after Removing Duplicate Rows:")
print(df_no_duplicates)

# Remove duplicates based on 'department' and 'location' columns
df_no_duplicates_specified   =   df.drop_duplicates(subset=['department',
'location'])
print("\nData   after   Removing   Duplicates   based   on   Department   and
Location:")
print(df_no_duplicates_specified)
```

Step 3: Running the Python Script

- Save your Python file.

- Run the script in your IDE or from the command line using: python data_operations.py.
- Observe the outputs generated from each operation.

Conclusion:

Thus, we successfully loaded a CSV file into a Pandas DataFrame and performed various operations, to understand and manipulate the dataset effectively.

Questions:

1. Rename the department column to dept. Then, calculate the total salary by department and display the department with the highest total salary.

2. Filter the DataFrame to display all employees in the IT department who earn a salary greater than 65000. Display only the employee_id, employee_name, salary, and location columns.

3. Sort the DataFrame by the join_date column in ascending order. Select the top 3 employees who joined most recently and display their employee_name, department, and join_date.