D. Y. Patil College of Engineering & Technology, Kolhapur

Department of CSE(Data Science)

Class: S.Y. B. Tech

Sem.-III

Lab Manual

Python Programming Laboratory

AY 2023-2024

D. Y. Patil College of Engineering & Technology,

KolhapurDepartment of CSE (Data Science)

Name of the course: Python Programming Laboratory **Class**: SY
Course Code: 201DSP206 **Semester:** III
Faculty: Prof. J. B. Metkari

List of Experiment

Sr. No.	Document Title
1	Exploring basics of python like data types (strings, list, array, dictionaries, set, tuples).
2	Program for Implementation of control statements.
3	Creating functions, classes and objects using python. Demonstrate exception handling.
4	Program for implementation of inheritance.
5	a. Python program to display file available in current directory. b. Python program to append data to existing file and then display the entire file.
6	Python program to count number of lines, words and characters in a file.
7	Creating Calculator GUI with python.
8	Menu driven program to create a phone directory.
9	Creation of simple socket for basic information exchange between server and client.
10	Creating web application using Django web framework to demonstrate functionality of user login and its validation using regular expression

Prepared by: **Course Coordinator**
Checked by: **Module Coordinator**
Verified by: **Program Coordinator**
Approved by: **HOD DS**

Experiment No.1

Title: Exploring Python Data Types: Strings, Lists, Arrays, Dictionaries, Sets, and Tuples

Aim: The aim of this experiment is to gain a fundamental understanding of various data types in Python, including strings, lists, arrays, dictionaries, sets, and tuples. We will explore their creation, manipulation, and basic operations.

Experimental Setup:

1**. Python Environment:** Set up a Python development environment on your computer. You can use a code editor like VSCode or Jupyter Notebook to write and run Python code.

2. **Python Version:** Ensure that you have a version of Python installed (preferably Python 3.x), which includes all the necessary modules (e.g., `array`, `collections`, etc.) to work with the mentioned data types.

3. **Notebook/Script:** Create a new Jupyter Notebook or Python script to document and run your experiments.

Experimental Tasks:

1. Strings:
- Create strings using both single and double quotes.
- Perform basic string operations like concatenation, slicing, and length calculation.
- Explore string methods such as `split()`, `upper()`, `lower()`, `strip()`, etc.

2. Lists:
- Create lists with elements of different data types.
- Access list elements using indexing and slicing.
- Explore list methods like `append()`, `remove()`, `sort()`, etc.
- Learn about list comprehensions.

3. Arrays:
- Import the `array` module and create arrays with numeric elements.
- Perform basic array operations such as element access and manipulation.
- Explore mathematical operations using arrays.

4. Dictionaries:
- Create dictionaries with key-value pairs.
- Access dictionary values using keys.
- Perform dictionary operations like adding, modifying, and deleting entries.
- Understand dictionary methods like `keys()`, `values()`, etc.

5. Sets:
- Create sets with unique elements.
- Perform set operations like union, intersection, and difference.
- Learn about set methods such as `add()`, `remove()`, etc.

6. Tuples:
- Create tuples with elements of different data types.
- Access tuple elements using indexing and slicing.

- Try modifying tuple elements to observe the immutability.

Documentation and Analysis:
- As you progress through the experiment, document your observations, code snippets, and any insights gained about each data type.
- Include comments and explanations in your code to make it more understandable.
- Compare and contrast the characteristics and use cases of different data types.
- Write a summary of your findings and reflections on what you've learned about Python data types.

Conclusion:
By the end of this experiment, you will have a solid understanding of the basic data types in Python, enabling you to work with them effectively and utilize them in various programming scenarios.

Experiment No.2

Title: Implementation of Control Statements in Python

Aim: The aim of this experiment is to explore and implement various control statements in Python, including conditional statements (`if`, `elif`, `else`), loops (`for` loop, `while` loop), and control flow modifiers (`break`, `continue`).

Experimental Tasks:

1. Conditional Statements:
- Implement a basic `if` statement to check a condition and execute code accordingly.
- Utilize the `if-else` statement to handle alternate outcomes based on a condition.
- Explore the `elif` statement to handle multiple conditional cases.

2. Loops:
- Implement a `for` loop to iterate over a sequence of elements (e.g., a list, string, or range).
- Use the `while` loop to execute code repeatedly until a certain condition is met.

3. Control Flow Modifiers:
- Experiment with the `break` statement to exit a loop prematurely.
- Utilize the `continue` statement to skip the current iteration and proceed with the next iteration of a loop.

4. Nested Control Statements:
- Implement nested `if` statements to handle complex conditional scenarios.
- Explore nesting loops (e.g., a loop within a loop) to solve more intricate problems.

5. Practical Applications:
- Apply control statements to solve simple programming problems like finding prime numbers, computing factorial, checking palindrome strings, etc.

Documentation and Analysis:
- As you progress through the experiment, document your code implementations, observations, and any challenges faced while using control statements.
- Include comments and explanations in your code to make it more understandable.
- Discuss the advantages of using control statements to make code more efficient and dynamic.

Conclusion:
By the end of this experiment, you will have gained hands-on experience in implementing control statements in Python. You will be able to make decisions in your code based on conditions, create loops for iterative tasks, and use control flow modifiers to control the flow of execution. Understanding these concepts is essential for writing effective and structured Python programs.

Experiment No.3

Title: Creating functions, classes and objects using python. Demonstrate exception handling.

Aim: The objective of this experiment is to create a simple calculator program using functions, classes, and objects in Python. The calculator should be able to perform basic arithmetic operations (addition, subtraction, multiplication, and division) and handle exceptions gracefully when invalid input is provided.

Experimental Tasks:

Step 1: Creating a Calculator Class Let's create a Calculator class that will handle the arithmetic operations.

```
class Calculator:
  def add(self, num1, num2):
    return num1 + num2

  def subtract(self, num1, num2):
    return num1 - num2

  def multiply(self, num1, num2):
    return num1 * num2

  def divide(self, num1, num2):
    if num2 == 0:
      raise ValueError("Division by zero is not allowed.")
    return num1 / num2
```

Step 2: Implementing Exception Handling We'll use try-except blocks to handle exceptions. Specifically, we'll catch ValueError when dividing by zero and TypeError for invalid inputs.

```
def main():
  calculator = Calculator()

  while True:
    print("Choose an operation:")
    print("1. Add")
    print("2. Subtract")
    print("3. Multiply")
    print("4. Divide")
    print("5. Exit")

    choice = input("Enter your choice (1/2/3/4/5): ")

    if choice == '5':
      print("Exiting...")
      break

    if choice not in {'1', '2', '3', '4'}:
      print("Invalid choice. Please try again.")
```

```
        continue

    try:
        num1 = float(input("Enter the first number: "))
        num2 = float(input("Enter the second number: "))

        if choice == '1':
            result = calculator.add(num1, num2)
            print(f"Result: {result}")
        elif choice == '2':
            result = calculator.subtract(num1, num2)
            print(f"Result: {result}")
        elif choice == '3':
            result = calculator.multiply(num1, num2)
            print(f"Result: {result}")
        elif choice == '4':
            result = calculator.divide(num1, num2)
            print(f"Result: {result}")
    except ValueError as ve:
        print(f"Error: {ve}")
    except TypeError:
        print("Invalid input. Please enter numeric values.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")

if __name__ == "__main__": main()
```

Step 3: Run the Experiment Save the code in a Python file (e.g., calculator.py) and run the script. The program will present a simple text-based menu to perform arithmetic operations. It will handle exceptions gracefully and prompt the user to enter valid input in case of errors.

Keep in mind that this is a basic experiment to demonstrate the use of functions, classes, objects, and exception handling in Python. You can extend the program to include more features and advanced error handling as needed.

Conclusion:

By the end of this experiment, you will have gained hands-on experience in implementing functions, classes and objects using python and demonstrated exception handling.

Experiment No.4

Title: Program for implementation of inheritance.

Aim: The objective of this experiment is to implement inheritance in Python by creating a program that showcases the concept of base and derived classes.

Experimental Tasks:

Task 1: Create the Animal Class

1. Open your code editor and create a new Python file named "animals.py".
2. Define the base class "Animal" with some common attributes and a method.

```
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def make_sound(self):
        pass  # This method will be overridden in the derived classes
```

Task 2: Create the Derived Classes (Dog and Cat)

1. Create a derived class "Dog" that inherits from the "Animal" class.
2. Implement the "make_sound" method for the "Dog" class.

```
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name, species="Dog")
        self.breed = breed

    def make_sound(self):
        return "Woof! Woof!"
```

3. Create another derived class "Cat" that also inherits from the "Animal" class.
4. Implement the "make_sound" method for the "Cat" class

```
class Cat(Animal):
    def __init__(self, name, breed):
        super().__init__(name, species="Cat")
        self.breed = breed

    def make_sound(self):
        return "Meow!"
```

Task 3: Main Function

1. In the same "animals.py" file, create a main function to test the classes.

2. Instantiate objects of the "Dog" and "Cat" classes and display their attributes and the sound they make.

```
def main():
    dog1 = Dog("Buddy", "Labrador")
    cat1 = Cat("Whiskers", "Siamese")

    print(f"{dog1.name} is a {dog1.species} of breed {dog1.breed}.")
    print(f"{dog1.name} says: {dog1.make_sound()}")

    print(f"{cat1.name} is a {cat1.species} of breed {cat1.breed}.")
    print(f"{cat1.name} says: {cat1.make_sound()}")

if __name__ == "__main__":
    main()
```

Task 4: Run the Program

1. Save your "animals.py" script and run it using your Python environment.
2. Observe the output, which will display the attributes and sounds of the instantiated objects.

Task 5: Experiment

1. Experiment with different attributes and methods in the base and derived classes to understand how inheritance works.
2. Add more derived classes or attributes to further explore the concept of inheritance.

Conclusion: Congratulations! You have successfully implemented inheritance in Python. In this lab experiment, you learned how to create base and derived classes, inherit attributes and methods, and override methods in the derived classes. You can now apply this knowledge to build more complex class hierarchies and explore the power of inheritance in your Python projects.

Experiment No.5

Title: Implementation of file handling in Python.

Aim: The objective of this experiment is to use:

a. Python program to display file available in current directory.

b. Python program to append data to existing file and then display the entire file.

Experimental Tasks:

Task 1: Import Libraries

At the beginning of your Python script, import the "os" library to interact with the operating system.

```
import os
```

Task 2: Function to Display Files

Create a function named "display_files" that lists and displays the files available in the current working directory

```
def display_files():
    current_directory = os.getcwd()
    files = os.listdir(current_directory)

    if len(files) == 0:
        print("No files found in the current directory.")
    else:
        print("Files in the current directory:")
        for file in files:
            print(file)
```

Task 3: Main Function

Create a main function to run the file display program. In this function, call the "display_files" function to show the files in the current directory.

```
def main():
    print("Current Directory:", os.getcwd())
    display_files()

if __name__ == "__main__":
    main()
```

Task 4: Run the Program

1. Save your "file_list.py" script and run it using your Python environment.
2. The program will display the current directory and list all the files available in that directory.

Task 5: Experiment

1. Experiment with the program by adding files to the current directory or creating subdirectories with

files.

2. Run the program after making changes to observe how it displays the updated list of files.

Task 1: Create a Text File

1. Create a text file named "data.txt" in the same directory as your Python script.
2. Add some text content to the file, such as a few lines of text.

Task 2: Function to Append Data

1. Open your code editor or IDE and create a new Python file. Save it as "append_and_display.py".
2. Define a function named "append_data" that appends data to the existing "data.txt" file

```
def append_data():
    try:
        with open("data.txt", "a") as file:
            new_data = input("Enter data to append: ")
            file.write("\n" + new_data)
        print("Data appended successfully.")
    except IOError:
        print("Error: Unable to access the file.")
```

Task 3: Function to Display Entire File

Define a function named "display_file" that displays the entire content of the "data.txt" file

```
def display_file():
    try:
        with open("data.txt", "r") as file:
            content = file.read()
            print("Content of the file:")
            print(content)
    except IOError:
        print("Error: Unable to access the file.")
```

Task 4: Main Function

1. Create a main function to run the append and display program. In this function, call the "append_data" function to add new data to the file, and then call the "display_file" function to show the entire content.

```
def main():
    append_data()
    display_file()

if __name__ == "__main__":
    main()
```

Task 5: Run the Program

1. Save your "append_and_display.py" script and run it using your Python environment.

2. The program will prompt you to enter data to append to the file. After appending the data, it will display the entire content of the "data.txt" file.

Task 6: Experiment

1. Experiment with the program by running it multiple times and appending different data each time.
2. Check the "data.txt" file after each run to see the updated content.

Conclusion: Congratulations! You have successfully implemented a Python program to display files in the current directory. And also implemented a Python program to append data to an existing file and then display the entire content of the file. In this lab experiment, you learned how to use the "os" library to interact with the operating system, get the current working directory, and list the files available in it. And also learned how to use the "open" function to work with files, how to append data to a file in append mode, and how to read and display the content of a file in read mode. You can now apply this knowledge to perform various file operations in your Python projects.

Experiment No.6

Title: Python program that counts the number of lines, words, and characters in a file.

Aim: To implement a Python program that counts the number of lines, words, and characters in a file.

Experimental Tasks:

Follow these steps:

1. Create a text file with some content. You can do this manually using a text editor or generate it programmatically.
2. Write a Python program that reads the file, counts the lines, words, and characters, and then displays the results.

Here's an example of a Python program to accomplish this task:

```
def count_lines_words_chars(file_path):
    line_count = 0
    word_count = 0
    char_count = 0

    try:
        with open(file_path, 'r') as file:
            for line in file:
                line_count += 1
                words = line.split()
                word_count += len(words)
                char_count += len(line)
    except FileNotFoundError:
        print("File not found.")
        return

    print("Number of lines:", line_count)
    print("Number of words:", word_count)
    print("Number of characters:", char_count)


if __name__ == "__main__":
    file_path = "example.txt"  # Replace this with the path to your text file.
    count_lines_words_chars(file_path)
```

Replace "example.txt" in the file_path variable with the path to your text file. When you run this program, it will read the content of the file and display the number of lines, words, and characters in it.

Remember to save the Python code in a .py file in the same directory as the text file you want to analyze. Then, execute the Python script to see the results.

Conclusion:

By the end of this experiment, you will have gained hands-on experience in reading the file and display the number of lines, words, and characters in it.

Experiment No.7

Title: Creating Calculator GUI with python

Aim: To implement a Python program for Creating Calculator GUI.

Experimental Tasks:

Creating a calculator GUI involves using a graphical library like Tkinter to design the user interface and implementing the calculator functionality. Below is an example of a simple calculator GUI using Python and Tkinter:

```
import tkinter as tk

def on_click(event):
    current_text = display.get()
    button_text = event.widget.cget("text")

    if button_text == "=":
        try:
            result = eval(current_text)
            display.set(result)
        except Exception as e:
            display.set("Error")
    elif button_text == "C":
        display.set("")
    else:
        display.set(current_text + button_text)

# Create the main window
root = tk.Tk()
root.title("Calculator")
root.geometry("300x400")

# Create the display widget
display = tk.StringVar()
display.set("")
entry = tk.Entry(root, textvariable=display, font=("Arial", 20), bd=10, relief=tk.SUNKEN, justify=tk.RIGHT)
entry.pack(fill=tk.BOTH, expand=True)

# Create buttons for digits and operators
buttons = [
    ("7", "8", "9", "/"),
    ("4", "5", "6", "*"),
    ("1", "2", "3", "-"),
    ("C", "0", "=", "+"),
]

for button_row in buttons:
```

```
    frame = tk.Frame(root)
    frame.pack(fill=tk.BOTH, expand=True)

    for button_text in button_row:
        button = tk.Button(frame, text=button_text, font=("Arial", 20), relief=tk.GROOVE)
        button.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
        button.bind("<Button-1>", on_click)

# Run the main event loop
root.mainloop()
```

Save this code in a .py file and execute it. A window with a simple calculator GUI will appear, allowing you to perform basic arithmetic operations.

Please note that this is a basic calculator implementation and may lack more advanced functionalities like handling parentheses or memory operations. You can extend the code to include additional features according to your needs and requirements.

Conclusion:

By the end of this experiment, you will have gained hands-on creating a calculator GUI using a graphical library Tkinter to design the user interface and implementing the calculator functionality.

Experiment No. 8

Title: Menu driven program to create a phone directory.

Aim: To implement a Python program Menu driven program to create a phone directory.

Experimental Tasks:

Creating a menu-driven program to manage a phone directory involves implementing a user interface that allows users to add, search, and display contacts. Below is an example of a simple phone directory program using Python:

```
import os

def add_contact():
    name = input("Enter the name: ")
    phone_number = input("Enter the phone number: ")
    with open("phone_directory.txt", "a") as file:
        file.write(f"{name},{phone_number}\n")
    print("Contact added successfully.")

def search_contact():
    name = input("Enter the name to search: ")
    with open("phone_directory.txt", "r") as file:
        for line in file:
            contact_name, phone_number = line.strip().split(',')
            if name.lower() == contact_name.lower():
                print(f"Contact found - {contact_name}: {phone_number}")
                return
        print("Contact not found.")

def display_contacts():
    with open("phone_directory.txt", "r") as file:
        print("Contacts in Phone Directory:")
        for line in file:
            contact_name, phone_number = line.strip().split(',')
            print(f"{contact_name}: {phone_number}")

def menu():
    while True:
        print("\nPhone Directory Menu:")
        print("1. Add Contact")
        print("2. Search Contact")
        print("3. Display Contacts")
        print("4. Exit")

        choice = input("Enter your choice (1/2/3/4): ")

        if choice == "1":
            add_contact()
```

```
        elif choice == "2":
            search_contact()
        elif choice == "3":
            display_contacts()
        elif choice == "4":
            print("Exiting Phone Directory Program.")
            break
        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    if not os.path.exists("phone_directory.txt"):
        with open("phone_directory.txt", "w"):
            pass

    menu()
```

Save this code in a .py file and execute it. The program will display a menu with options to add contacts, search for contacts, display all contacts, and exit the program. The phone directory data will be stored in a text file named "phone_directory.txt" in the same directory as the Python script.

Please note that this is a basic implementation and may not handle edge cases or errors extensively. You can enhance the program to add features like deleting contacts, editing existing contacts, or sorting contacts alphabetically based on names, based on your requirements.

Conclusion:

By the end of this experiment, you will have gained hands-on creating a menu-driven program to manage a phone directory.

Experiment No. 9

Title: Creation of simple socket for basic information exchange between server and client.

Aim: To implement a Python program for creation of simple socket for basic information exchange between server and client.

Experimental Tasks:

Creating a simple socket-based communication between a server and client involves setting up a server socket to listen for incoming connections and a client socket to establish a connection with the server. The client can then send a message to the server, and the server can respond with a message. Below is an example of a simple server-client interaction using Python's socket module:

Server (server.py):

```
import socket

def start_server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    host = "127.0.0.1"
    port = 12345

    server_socket.bind((host, port))
    server_socket.listen(1)

    print("Server listening on {}:{}".format(host, port))

    conn, addr = server_socket.accept()
    print("Connection established with:", addr)

    message = "Welcome to the server!"
    conn.sendall(message.encode())

    data = conn.recv(1024).decode()
    print("Received from client:", data)

    conn.close()
    server_socket.close()

if __name__ == "__main__":
    start_server()
```

Client (client.py):

```
import socket

def start_client():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    host = "127.0.0.1"
```

```
    port = 12345

    client_socket.connect((host, port))

    data = client_socket.recv(1024).decode()
    print("Received from server:", data)

    message = "Hello, server! How are you?"
    client_socket.sendall(message.encode())

    client_socket.close()

if __name__ == "__main__":
    start_client()
```

How to run the experiment:

1. Save the above server code in a file named server.py and the client code in a file named client.py.
2. Open two separate terminals or command prompt windows.
3. In one terminal, run the server by executing the command: python server.py
4. In the other terminal, run the client by executing the command: python client.py

You should see the following output:

```
Server listening on 127.0.0.1:12345
Connection established with: ('127.0.0.1', <PORT>)
Received from server: Welcome to the server!
Received from client: Hello, server! How are you?
```

The client sends a message to the server, and the server responds with a welcome message. The server then receives the message from the client and prints it on the console.

Conclusion: This experiment demonstrates a basic example of socket communication between a server and a client. In practice, you can extend this further to implement more complex applications, such as real-time chat applications or data exchange between distributed systems.

Experiment No. 10

Title: Creating web application using Django web framework to demonstrate functionality of user login and its validation using regular expression

Aim: To implement a Python program for creating web application using Django web framework to demonstrate functionality of user login and its validation using regular expression

Experimental Tasks:

Creating a web application using Django to demonstrate user login and its validation using regular expressions involves setting up a Django project and app, creating the necessary views, templates, and forms. Below are the steps to create such an application:

Step 1: Install Django If you haven't installed Django, you can do so using pip:

```
pip install django
```

Step 2: Create a Django Project Create a new Django project using the following command:

```
django-admin startproject user_login_app
```

Step 3: Create a Django App Create a new Django app inside the project:

```
cd user_login_app
python manage.py startapp login_app
```

Step 4: Define URLs

In the **urls.py** file of the app, define the URL patterns:

```
# login_app/urls.py
from django.urls import path
from . import views
urlpatterns = [
    path('', views.login_view, name='login'),
    path('logout/', views.logout_view, name='logout'),
]
```

Step 5: Create Views

In the views.py file of the app, define the views to handle login and logout:

```
# login_app/views.py
from django.shortcuts import render, redirect
```

```
import re
def login_view(request):
    if request.method == 'POST':
        username = request.POST.get('username')
        password = request.POST.get('password')

        # Validate the username using regular expression
        if not re.match(r'^[a-zA-Z0-9_-]{4,16}$', username):
            return render(request, 'login.html', {'error': 'Invalid username'})

        # Your authentication logic goes here
        return redirect('login')
    return render(request, 'login.html')

def logout_view(request):
    # Your logout logic goes here
    return redirect('login')
```

Step 6: Create Templates

Create a templates folder inside the app's directory and create the login.html template:

```
<!-- login_app/templates/login.html -->
<!DOCTYPE html>
<html>
<head>
    <title>User Login</title>
</head>
<body>
    {% if error %}
    <p style="color: red;">{{ error }}</p>
    {% endif %}
    <form method="post">
        {% csrf_token %}
        <label for="username">Username:</label>
        <input type="text" name="username" required><br><br>
        <label for="password">Password:</label>
        <input type="password" name="password" required><br><br>
        <input type="submit" value="Login">
    </form>
</body>
</html>
```

Step 7: Update Settings

In the settings.py file of the project, add the app to the INSTALLED_APPS and configure the templates:

```
# user_login_app/settings.py
# ...
INSTALLED_APPS = [
    # ...
    'login_app',
]
TEMPLATES = [
    {
        # ...
        'APP_DIRS': True,
    },
]
# ...
```

Step 8: Run the Development Server

Now, run the development server:

```
python manage.py runserver
```

Visit http://127.0.0.1:8000/ in your browser to access the login page.

Conclusion: This example demonstrates a basic Django web application that allows users to log in and validates the username using a regular expression. The actual authentication and logout logic are left for you to implement, depending on your specific use case. You can further expand this application by adding user registration, password hashing, and other features as needed.