|  |
| --- |
| Experiment No. 6 |
| Serialization in Python using Pickle |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 6**

**Title:** Serialization in Python using Pickle

**Aim:** To study and implement serialization using Pickle in Python

**Objective:** To introduce serialization and deserialization using Pickle module in Python

**Theory:**

Serialization and deserialization play crucial roles in data handling, especially in scenarios where data needs to be stored or transmitted efficiently. Pickle, being a built-in module in Python, simplifies this process by offering a convenient way to serialize and deserialize Python objects.

One important aspect to note about Pickle is its ability to handle complex data structures seamlessly. It can serialize and deserialize not only basic data types like strings and integers but also more complex objects like lists, dictionaries, and even user-defined classes.

Additionally, Pickle provides support for protocol versions, allowing developers to choose the appropriate protocol based on factors such as compatibility and efficiency. The protocol version determines the format of the serialized data and can impact factors like file size and serialization/deserialization speed.

It's worth mentioning that while Pickle is powerful and convenient, it's not without limitations. One notable limitation is that the serialized data is not human-readable, making it unsuitable for scenarios where human-readable data is required. Also, Pickle may not be the most efficient solution for large datasets or scenarios where interoperability with non-Python systems is a requirement.

Despite these limitations, Pickle remains a valuable tool in the Python ecosystem for many use cases, offering a quick and straightforward solution for serialization and deserialization tasks. By understanding its capabilities and limitations, developers can leverage Pickle effectively to manage data in their Python applications.

**Code:.**import pickle

class Employee:

def \_\_init\_\_(self, name, emp\_id, department):

self.name = name

self.emp\_id = emp\_id

self.department = department

def \_\_str\_\_(self):

return f"Employee: {self.name}, ID: {self.emp\_id}, Department: {self.department}"

# Example usage

employee1 = Employee("Kevin Hart", 1001, "Engineering")

employee2 = Employee("Dwayne Johnson", 1002, "Marketing")

# Serialize the objects using pickle

with open("employees.pkl", "wb") as f:

pickle.dump([employee1, employee2], f)

# Deserialize the objects

with open("employees.pkl", "rb") as f:

loaded\_employees = pickle.load(f)

for employee in loaded\_employees:

print(employee)

**Output:**



**Conclusion:**

This process showcases the practical utility of serialization for data persistence and transfer in Python programming. Through Pickle, we can easily maintain the state of objects across sessions, enhancing the versatility and efficiency of our code.